
SEAL: Synergistic Co-Evolution of Agents and Learning Environments

David S. Hippocampus¹

¹Department of Computer Science, Cranberry-Lemon University

Abstract

Large Language Model (LLM) agents are increasingly improved through interaction rather than static supervision. Yet most self-evolution methods adapt either the policy or the learning environment in isolation, leaving a structural gap: as the agent’s capability frontier shifts during training, the environment that provides supervision often remains static or only weakly coupled to the agent’s revealed failures. We call this mismatch *Agent-Environment Misalignment*. We propose SEAL, a closed-loop co-evolution framework for interactive tool-use agents. SEAL collects on-policy trajectories under executable verification, diagnoses failed rollouts into turn-level labels, and uses these diagnoses as a shared signal for both environment-side adaptation and model-side policy optimization. Specifically, the training-time learning interface evolves to expose clearer tool affordance cues, constraint information, and recovery-oriented feedback, while the policy is updated with diagnosis-guided advantage reweighting. Across in-distribution and out-of-distribution multi-turn tool-use evaluations, SEAL consistently improves low-resource agent learning: with only 400 training samples, it yields +8.25 to +26.25 average-point gains across three backbones and exhibits positive out-of-distribution transfer. These results show that jointly adapting the learner and its training-time learning substrate is a practical path toward more robust self-improving LLM agents.

Correspondence: hippo@cs.cranberry-lemon.edu

1 Introduction

Large Language Model (LLM) agents have recently demonstrated strong capabilities in reasoning, planning, and tool use, enabling progress on interactive tasks that require multi-step decision making and external action execution [1–3]. Recent work further improves these agents through reinforcement learning, tool-use post-training, and agentic data generation [4–6]. A growing trend behind these advances is *self-evolution through interaction*: agents improve by collecting rollout trajectories, receiving feedback, and iteratively refining their behavior through reflection, reinforcement learning, or self-generated supervision [7–10]. This paradigm turns interaction feedback into a reusable source of supervision, offering a scalable path beyond static offline training.

Practical motivation. This promise becomes especially important in realistic tool-use settings, where agents must operate through multi-turn interfaces, satisfy strict execution constraints, and recover from partial failures [11–16]. In such environments, improvement depends not only on how the policy is optimized, but also

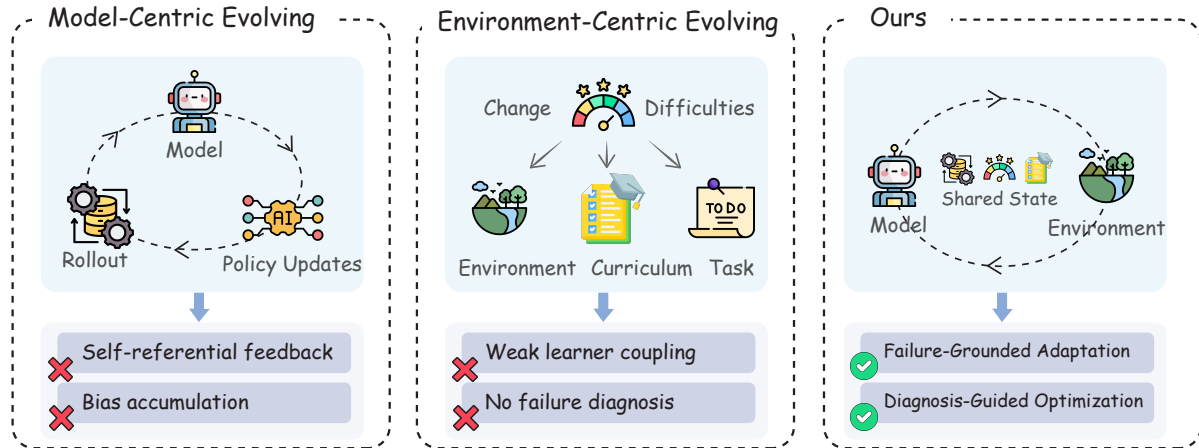


Figure 1 Conceptual comparison of self-evolution paradigms for LLM agents. SEAL closes the loop between policy learning and environment adaptation by using verifier-grounded failure diagnosis as the shared feedback signal.

on what learning signals the training-time environment exposes while the policy is changing. Yet current self-evolving agents often adapt only one side of this interaction loop.

Two one-sided adaptation patterns. Most existing methods follow one of two paths. **(i) Model-centric evolution.** These methods improve the policy through rollout replay, self-reflection, reward optimization, or post-training updates [17–20]. While effective, they typically optimize the agent against a largely fixed learning environment, making the learning signal dependent on the current policy’s own rollout distribution. In long-horizon interactive settings with sparse rewards, this self-referential rollout distribution can lead to policy-induced exploration bias, unstable recovery, and inefficient credit assignment [21]. **(ii) Environment-centric evolution.** These methods adapt curricula, task distributions, synthetic instructions, or interaction experiences. They recognize that agent capability is shaped not only by optimization, but also by the experiences exposed during training. Yet when such adaptation is not grounded in the current agent’s executable failures—what it can solve, where it repeatedly fails, and why—the evolved environment may remain weakly coupled to the learner’s actual needs. Such methods can increase diversity or difficulty, but without executable failure grounding, they may still fail to target the capability gaps that currently limit agent performance [22–25].

A shared closed-loop bottleneck. These two adaptation patterns look different, but they expose the same structural problem: the training-time learning environment fails to track the agent’s evolving capability boundary, and therefore provides signals that are too static, weakly targeted, or insufficiently informative. We refer to this mismatch as **Agent-Environment Misalignment**. Importantly, by “environment” we refer to the training-time learning substrate—including task exposure, observation interfaces, action constraints, and recovery feedback—rather than changes to the evaluation benchmark, tool semantics, or executable verifier.

A concrete example illustrates the issue. When a tool call fails because the agent uses a city name where an airport code is required, a fixed environment may only return a generic execution error. Such feedback tells the agent that the trajectory failed, but not whether the failure came from missing a prerequisite lookup, using an invalid argument type, or failing to recover after the error. As a result, the policy receives only weak diagnostic supervision, and future trajectories are collected under similarly uninformative conditions.

Our approach. Motivated by this observation, we propose **SEAL**, a closed-loop co-evolution framework for interactive tool-use agents. SEAL uses verifier-grounded failure diagnoses as a shared signal for both sides of training: it evolves the training-time interface with schema cues, constraint information, and recovery-oriented feedback, and it reweights policy-gradient updates by diagnostic utility. This enables *capability-*

aware environment adaptation: instead of relying on generic difficulty scaling or unguided data expansion, the training interface is adjusted according to the current policy’s recurring failures, producing more informative rollouts while leaving the benchmark protocol unchanged. Extensive experiments on low-resource multi-turn tool-use settings validate the effectiveness of this design. Our contributions are threefold:

- We identify **Agent-Environment Misalignment**: as the agent’s capability frontier shifts during training, the learning environment often remains static or only weakly coupled to the agent’s revealed failures.
- We introduce **SEAL**, which uses verifier-grounded failure diagnosis to jointly evolve the training-time learning interface and guide policy optimization.
- We show that SEAL improves low-resource multi-turn tool-use learning, yielding up to +26.25 average-point gains across three backbones with only 400 training samples and demonstrating positive transfer to held-out settings.

2 Related Work

2.1 Model-Side Self-Improvement

Prior work shows that LLM agents can improve through repeated interaction rather than static prompting alone. Methods based on recursive skill learning, self-consolidation, reflective prompt adaptation, memory-based improvement, and reinforcement learning from interaction feedback primarily refine the agent itself [26–29]. In this sense, they are largely *model-centric*: experience is internalized into the policy, prompt, memory, or skill library, while the training-time learning environment is typically kept fixed. SEAL is complementary: it uses failed trajectories not only for model-side improvement, but also as verifier-grounded evidence for adapting the environment from which future trajectories are collected.

2.2 Environment-Side Adaptation

Another line of work adapts what the learner is exposed to during training. Curriculum learning, automatic curriculum design, synthetic instruction generation, task evolution, and tool or skill construction reshape the training distribution or interface [30–36]. However, most such methods operate at the level of task diversity, difficulty, or coverage. By contrast, SEAL performs *failure-conditioned* environment adaptation: verifier-grounded diagnoses determine which affordance cues, constraint information, recovery feedback, and training-time signals are exposed to target the current policy’s capability gaps.

2.3 Interactive Environments and Co-Evolution

Interactive benchmarks for tool use, function calling, web navigation, operating-system control, embodied simulation, and software engineering expose the multi-turn dependencies, execution constraints, sparse rewards, and recovery dynamics central to realistic agent learning [11, 37–40]. Recent work also studies environment design and agent–environment co-evolution [41–43]. SEAL is closest to this perspective, but under a stricter protocol: it keeps benchmark tasks, tool semantics, and the executable verifier fixed, and adapts only the training-time learning interface in a failure-conditioned, verifier-grounded way.

3 Methodology

We propose **SEAL**, a framework for co-evolving tool-use policies and their training-time learning environments. **Key idea.** Instead of treating the environment as a fixed executor that returns only sparse scalar rewards, SEAL exposes a verifier-grounded diagnostic interface during training, converting failed interactions into structured evidence about the agent’s current capability gaps while preserving tool semantics, task labels, rewards, and the evaluation verifier.

Scope of adaptation. Here, the “environment” includes not only the executable tool backend but also the learning interface through which the policy observes tool schemas, execution feedback, and recovery signals.

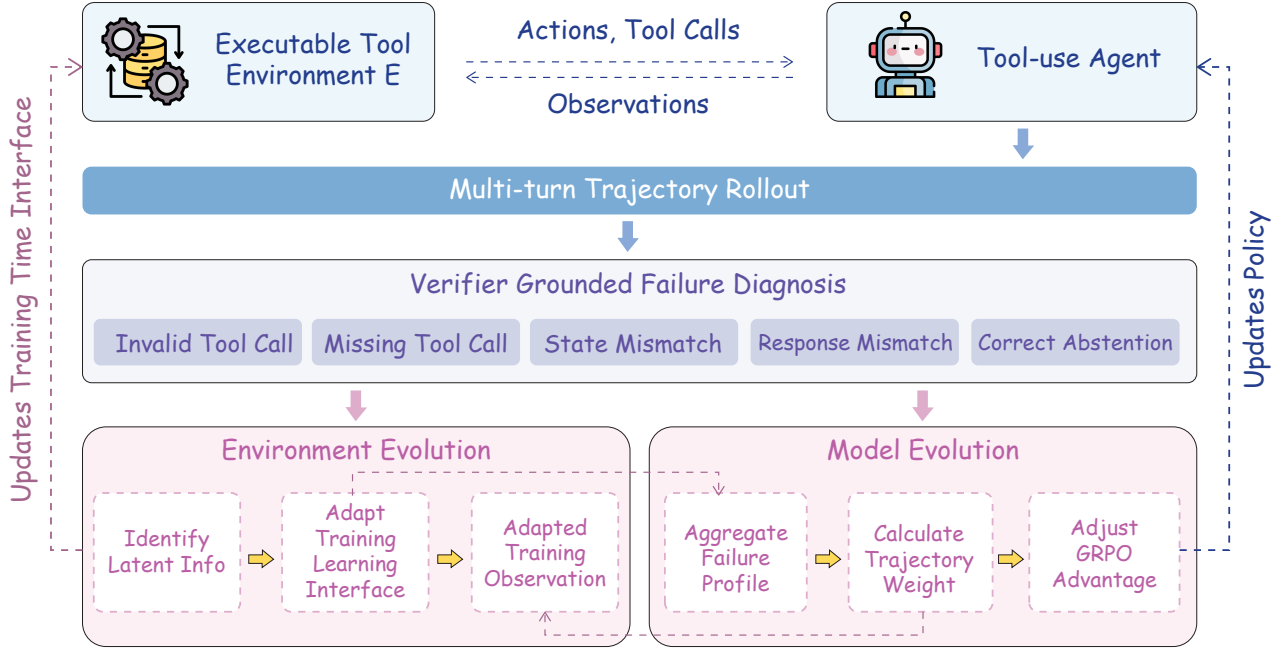


Figure 2 SEAL training-time co-evolution loop. The agent interacts with an executable tool environment to produce multi-turn trajectories, which are diagnosed into verifier-grounded failure types. These diagnoses jointly drive learning-interface evolution through adapted training observations and policy optimization through diagnosis-guided trajectory weighting, while keeping tool semantics and evaluation fixed.

SEAL therefore restricts environment evolution to this interface layer to preserve benchmark fairness. The same diagnoses drive both sides of the learning loop: they adapt the training-time interface through tool affordance cues, recovery-oriented feedback, and capability-specific hints, and they modulate policy optimization through diagnosis-guided advantage reweighting. Figure 2 summarizes this co-evolution process.

3.1 Problem Formulation

We formulate interactive tool use as a partially observable decision process:

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, O_{\mathcal{E}}, H, \mathcal{S}_{\text{goal}} \rangle, \quad (1)$$

where \mathcal{S} is the hidden state space, \mathcal{A} is the action space, \mathcal{O} is the observation space, \mathcal{T} is the transition function, $O_{\mathcal{E}}$ is the observation function, H is the horizon, and $\mathcal{S}_{\text{goal}}$ denotes goal states. In tool-use environments, actions include natural-language responses and executable tool calls over a tool set \mathcal{F} , while observations include dialogue context, tool outputs, and execution errors.

Given an instruction $q \sim \mathcal{D}$, a policy π_{θ} interacts with an executable environment \mathcal{E} and induces a rollout

$$\tau = \{(u_i, a_i, o_i)\}_{i=1}^T \sim P_{\mathcal{E}}(\cdot | q, \pi_{\theta}), \quad T \leq H, \quad (2)$$

where u_i denotes the dialogue context, a_i is the model action, and o_i is the environment observation. The executable verifier \mathcal{V} provides a terminal binary reward:

$$r(\tau) = \mathcal{V}(\tau, q, \mathcal{E}) \in \{0, 1\}. \quad (3)$$

Standard RL maximizes the expected verifier success:

$$J(\pi_{\theta}) = \mathbb{E}_{q \sim \mathcal{D}, \tau \sim P_{\mathcal{E}}(\cdot | q, \pi_{\theta})} [r(\tau)]. \quad (4)$$

This scalar reward indicates whether a trajectory succeeds, but not why it fails. SEAL therefore augments the training-time feedback as $(r(\tau), Z(\tau))$, where $Z(\tau)$ denotes structured failure diagnoses extracted from executable interaction traces. The verifier reward, tool semantics, and evaluation protocol remain unchanged.

3.2 Verifier-Grounded Failure Diagnosis

For each rollout τ , SEAL produces turn-level diagnostic labels

$$Z(\tau) = \{z_1, \dots, z_T\}, \quad z_i \in \mathcal{Z}, \quad (5)$$

where each z_i denotes the dominant outcome or failure mode at turn i . Diagnoses are grounded in executable evidence rather than free-form model critique: SEAL uses parser checks, tool-schema validation, execution errors, observable state transitions, and verifier comparisons to identify invalid or missing tool calls, argument or state mismatches, recovery failures, and final-response mismatches.

We write the diagnosis function as

$$z_i = \Psi(a_i, \xi_i, s_{i-1}, s_i, \mathcal{F}), \quad (6)$$

where a_i is the model action, ξ_i is the executable evidence available at turn i , s_{i-1} and s_i are the pre- and post-action states when available, and \mathcal{F} is the available tool set. Operationally, Ψ is a deterministic rule-based classifier over executable traces that prioritizes directly executable failures over downstream verifier-level failures. The full label taxonomy and decision rules are provided in Appendix C.

Importantly, diagnosis does not modify the benchmark reward:

$$r(\tau) = \mathcal{V}(\tau, q, \mathcal{E}), \quad Z(\tau) = \Psi(\tau, \mathcal{E}, \mathcal{V}). \quad (7)$$

Thus, failed trajectories still receive zero reward under the original verifier; the labels only add training-time structure for interface evolution and policy optimization.

3.3 Learning-Interface Evolution

SEAL evolves only the *training-time learning interface*, not the benchmark verifier, tool signatures, tool outputs, or task labels. Let o_i denote the original observation at turn i . SEAL constructs an augmented training-time observation

$$\tilde{o}_i = \Omega_t(o_i, \mathcal{F}, H_t, C_t), \quad (8)$$

where \mathcal{F} is the available tool set, H_t is diagnostic context accumulated from previous rollouts, and C_t is the current policy’s aggregate failure profile computed from recent diagnoses. The transformation Ω_t changes only how existing environment information is exposed to the learner.

In our implementation, Ω_t consists of three lightweight components:

$$\Omega_t(o_i, \mathcal{F}, H_t, C_t) = o_i \oplus \phi_{\text{schema}}(\mathcal{F}) \oplus \phi_{\text{err}}(o_i, H_t) \oplus \phi_{\text{cap}}(C_t). \quad (9)$$

Here, ϕ_{schema} exposes schema-implied tool affordances such as required arguments, enum constraints, argument types, and valid tool-call formats. ϕ_{err} converts execution errors into recovery-oriented feedback without revealing the correct answer. ϕ_{cap} selects capability-specific cues from the current failure profile so that recurring errors receive targeted feedback.

The interface update is selected by failure type rather than benchmark instance. For example, `argument_mismatch` activates schema and constraint cues, `missing_tool_call` activates tool-affordance cues, and `recovery_failure` activates structured error feedback. These cues clarify how to repair an error class without exposing the reference tool sequence, hidden parameters, or final answer. Throughout training, SEAL preserves the original tool semantics and verifier; at evaluation time, the evolved interface is removed.

3.4 Diagnosis-Guided Advantage Reweighting

Sparse verifier rewards indicate whether a trajectory succeeds, but not how useful it is for policy improvement. In multi-turn tool use, two failed trajectories with the same zero reward can have very different learning value: invalid arguments or missed tool calls usually provide clearer corrective signals than failures that appear only in the final response. SEAL therefore uses verifier-grounded diagnoses to estimate the *learning utility* of each trajectory and allocate optimization pressure accordingly.

For a trajectory τ_j , we first summarize its turn-level diagnostic labels into an empirical diagnostic profile:

$$p_j(z) = \frac{1}{T_j} \sum_{i=1}^{T_j} \mathbf{1}[z_i = z], \quad z \in \mathcal{Z}. \quad (10)$$

This profile captures the dominant failure modes in the trajectory. We then define a diagnostic utility function $\rho : \mathcal{Z} \rightarrow \mathbb{R}_+$, where $\rho(z)$ measures how actionable and attributable diagnosis type z is. Failures with concrete executable evidence and clear repair directions, such as `invalid_tool_call` or `argument_mismatch`, receive larger utility, while more ambiguous failures such as `response_mismatch` receive smaller utility. In our experiments, $\rho(z)$ is fixed across all backbones and training runs; exact values are reported in Appendix C.

The trajectory-level diagnostic weight is computed as

$$w_j = \text{clip} \left(\sum_{z \in \mathcal{Z}} p_j(z) \rho(z), w_{\min}, w_{\max} \right), \quad (11)$$

where clipping prevents rare or noisy diagnostic patterns from inducing overly large policy updates. Given the original group-relative GRPO advantage A_j , SEAL forms a diagnosis-weighted advantage:

$$\tilde{A}_j = w_j A_j. \quad (12)$$

This reweighting is a verifier-grounded preconditioning of the policy-gradient signal. The verifier reward still determines the direction through A_j , while the diagnostic utility scales how much each trajectory contributes. Since $w_j > 0$, the sign of the advantage is unchanged, so SEAL does not alter the benchmark reward, success criterion, or verifier-induced ranking; it simply prioritizes trajectories whose failures are more attributable, recoverable, and informative.

3.5 SEAL Training Loop

SEAL alternates between rollout collection, failure diagnosis, interface evolution, and policy optimization. At round t , prompts are sampled from the training distribution, and the current policy π_{θ_t} interacts with the environment instantiated with interface Ω_t to collect trajectories:

$$\{\tau_j\}_{j=1}^N \sim P_{\mathcal{E}_{\Omega_t}}(\cdot \mid \pi_{\theta_t}). \quad (13)$$

Each trajectory is evaluated by the original verifier to obtain $r(\tau_j)$ and diagnosed to obtain $Z(\tau_j)$. The diagnoses are aggregated into a policy-specific failure profile:

$$C_t = \text{Agg} \left(\{Z(\tau_j)\}_{j=1}^N \right). \quad (14)$$

The failure profile updates the training-time interface:

$$\Omega_{t+1} = \text{Evolve}(\Omega_t, C_t), \quad (15)$$

and the policy is optimized with diagnosis-weighted GRPO:

$$\theta_{t+1} = \text{GRPO} \left(\theta_t; \{\tau_j, r(\tau_j), \tilde{A}_j\}_{j=1}^N \right), \quad \tilde{A}_j = w_j A_j. \quad (16)$$

This forms a closed co-evolution loop between the policy and the training-time learning interface: the agent reveals capability gaps, the interface adapts around these gaps, and the model internalizes the resulting feedback through policy optimization. Throughout the loop, tool semantics, task labels, transition dynamics, rewards, and the evaluation verifier remain fixed.

4 Experiments

We organize the evaluation around four questions: *What is the experimental setup? How much does SEAL improve in-distribution performance on BFCL V3? Do these gains transfer to held-out tool-use settings? and Which components matter most, and how do the gains emerge during training?*

4.1 Experiment Settings

Benchmarks. We use the BFCL V3 multi-turn subset as the in-distribution benchmark [37]. It contains 800 examples from four categories: *Base*, *Missing Functions*, *Missing Parameters*, and *Long Context*. We focus on a *low-resource* setting with 400 training examples, sampling 100 from each category, and use the remaining 400 examples for held-in evaluation. For *out-of-distribution* evaluation, we use BFCL V4 Web Search and Memory, together with the Retail, Airline, and Telecom domains of τ^2 -bench. These held-out benchmarks differ from BFCL V3 in tool domains, schema structure, and interaction patterns. Detailed protocols are provided in Appendix A.

Models and baselines. We evaluate SEAL on three backbones: Qwen2.5-3B-Instruct, Qwen2.5-7B-Instruct, and ToolACE-2-Llama-3.1-8B. The Qwen models represent general instruction-tuned agents, while ToolACE-2 provides a stronger tool-specialized initialization. For each backbone, we compare against Vanilla RL under the same training split, rollout budget, optimizer family, and verifier reward, yielding a *controlled comparison*. We also report representative open-source and proprietary tool-use systems only as *reference points*, since they differ in scale and training recipe and are therefore not controlled baselines.

Hyperparameters. We use a GRPO-style optimizer with 8 rollouts per prompt. The actor learning rate is set to 1×10^{-6} , with a train batch size of 32, a PPO mini-batch size of 8, and a PPO micro-batch size of 1 per GPU. We use a training temperature of 1.0 and a validation temperature of 0.0. The maximum prompt length, response length, and model length are set to 8192, 4096, and 16384, respectively. Training is conducted with vLLM-based asynchronous rollout workers on 4 GPUs. Additional implementation details are provided in Appendix C.

4.2 Main Results on BFCL V3

Table 1 reports in-distribution results on BFCL V3 multi-turn evaluation. We compare each SEAL-trained model against its corresponding backbone and Vanilla RL counterpart under the same 400-sample training budget.

Overall gains. SEAL improves all three backbones, increasing the average score of Qwen2.5-3B-Instruct, Qwen2.5-7B-Instruct, and ToolACE-2-Llama-3.1-8B by +8.25, +26.25, and +14.75 points over their original checkpoints. The gains are therefore consistent across model scale and initialization quality, rather than being confined to a single regime. Improvements on ToolACE-2-Llama-3.1-8B further show that the method helps even a tool-specialized model.

Controlled comparison against Vanilla RL. Under the same training split, rollout budget, optimizer family, and verifier reward, SEAL outperforms Vanilla RL by +4.75, +9.50, and +8.25 average points across the three

Table 1 Main results on the BFCL V3 multi-turn benchmark. Under the same 400-sample training budget, SEAL consistently improves 3B–8B backbones over Vanilla RL. Reference models are included only to contextualize the broader performance range of existing tool-use systems.

Model	BFCL V3 Multi-Turn				
	Average (%)	Base (%)	Miss Func (%)	Miss Param (%)	Long Context (%)
Closed-source					
Gemini-3-Pro-Preview [44]	60.75	64.50	60.00	54.50	64.00
Claude-Sonnet-4.5 [45]	61.38	69.00	65.00	52.50	59.00
GPT-4o-2024-11-20 [46]	42.50	55.50	34.50	29.00	51.00
Gemini-2.5-Flash [47]	36.25	41.50	36.00	32.00	35.50
Open-source					
GLM-4.6 355B [48]	68.00	74.50	68.00	63.00	66.50
Qwen3-32B [49]	47.88	56.00	52.50	40.00	43.00
Qwen2.5-14B-Instruct [50]	25.25	33.00	26.00	22.00	20.00
xLAM-2-3b-fc-r [51]	58.38	71.50	59.00	57.50	45.50
Qwen2.5-3B-Instruct [50]	5.75	11.00	6.00	3.00	3.00
+ Vanilla RL	9.25 (+3.50)	16.00 (+5.00)	9.00 (+3.00)	6.00 (+3.00)	6.00 (+3.00)
+ SEAL	14.00 (+8.25)	19.00 (+8.00)	15.00 (+9.00)	12.00 (+9.00)	10.00 (+7.00)
Qwen2.5-7B-Instruct [50]	14.00	22.00	14.00	10.00	10.00
+ Vanilla RL	30.75 (+16.75)	46.00 (+24.00)	27.00 (+13.00)	27.00 (+17.00)	23.00 (+13.00)
+ SEAL	40.25 (+26.25)	58.00 (+36.00)	36.00 (+22.00)	34.00 (+24.00)	33.00 (+23.00)
ToolACE-2-Llama-3.1-8B [52]	32.00	45.00	26.00	35.00	22.00
+ Vanilla RL	38.50 (+6.50)	52.00 (+7.00)	30.00 (+6.00)	40.00 (+5.00)	32.00 (+10.00)
+ SEAL	46.75 (+14.75)	58.00 (+13.00)	46.00 (+20.00)	44.00 (+9.00)	39.00 (+17.00)

backbones. This gap suggests that sparse terminal rewards alone are not enough for efficient multi-turn tool-use learning. The advantage is especially notable on the 7B and 8B models, indicating that the benefit of verifier-grounded diagnosis remains substantial even when the starting policy is already considerably stronger.

Where the gains are largest. The biggest improvements appear on structured tool-use failures. For Qwen2.5-7B-Instruct, SEAL raises *Missing Functions* from 14.00% to 36.00% and *Missing Parameters* from 10.00% to 34.00%, consistent with the error types the method is designed to address. Gains on *Long Context* also suggest better multi-turn state tracking and recovery. This pattern matters because these categories capture failures that are common in practical tool use, where an agent must choose the right tool, fill in the right arguments, and remain consistent over multiple turns.

Taken together, these in-distribution results show that SEAL improves not only final accuracy, but also the structured behaviors that matter for reliable multi-turn tool use. A natural next question is whether these gains transfer beyond BFCL V3.

4.3 Out-of-Distribution Generalization

Having established strong in-distribution gains, we next ask whether they transfer to held-out tool-use settings with different task distributions, tool domains, and interaction patterns. **OOD transfer.** Table 2 shows that SEAL improves both evaluated backbones and outperforms Vanilla RL on the aggregate BFCL V4 and τ^2 -bench metrics. For Qwen2.5-3B-Instruct, it raises BFCL V4 from 4.69% to 8.63% and τ^2 -bench from 9.64% to 11.76%. For Qwen2.5-7B-Instruct, it improves BFCL V4 from 9.17% to 12.71% and τ^2 -bench from 16.11% to 20.79%.

Interpretation. These results suggest that SEAL learns transferable tool-use behaviors—such as argument grounding, error recovery, and multi-turn state management—rather than simply fitting the 400 BFCL V3 training examples. At the same time, absolute OOD scores remain modest, so cross-benchmark generalization is still challenging.

Table 2 Out-of-distribution generalization on BFCL V4 and τ^2 -bench. Models are trained only on the 400-sample BFCL V3 training set and evaluated on held-out task distributions to measure cross-benchmark transfer.

Model	BFCL V4			τ^2 -bench			
	Avg. (%)	Web Search (%)	Memory (%)	Avg. (%)	Retail (%)	Airline (%)	Telecom (%)
BitAgent-8B [53]	7.41	4.50	10.32	10.00	7.50	15.00	7.50
ToolACE-2-Llama-3.1-8B [52]	15.90	9.00	22.80	10.83	10.00	15.00	7.50
watt-tool-8B [54]	8.67	4.00	13.33	13.33	15.00	20.00	5.00
Qwen3-30B-A3B-Instruct [49]	20.07	22.50	17.63	32.03	50.88	32.50	12.72
Qwen2.5-3B-Instruct [50]	4.69	0.00	9.38	9.64	5.26	14.00	9.65
+ Vanilla RL	5.98	0.50	11.26	10.60	6.14	16.00	9.65
+ SEAL	8.63	2.00	15.26	11.76	8.77	16.00	10.52
Qwen2.5-7B-Instruct [50]	9.17	6.50	11.83	16.11	16.67	20.00	11.67
+ Vanilla RL	10.26	7.50	13.02	17.19	19.30	20.00	12.28
+ SEAL	12.71	9.50	15.92	20.79	26.32	22.00	14.04

Table 3 Ablation of SEAL on Qwen2.5-7B. Leave-one-out ablations show that environment-side adaptation, diagnosis-guided reweighting, and closed-loop updates all contribute, with Full SEAL outperforming all single-sided variants.

Setting	BFCL V3 Multi-Turn				
	Average (%)	Base (%)	Miss Func (%)	Miss Param (%)	Long Context (%)
Backbone and RL Baseline					
Qwen2.5-7B-Instruct	14.00	22.00	14.00	10.00	10.00
+ Vanilla RL	30.75	46.00	27.00	27.00	23.00
SEAL Ablations					
w/o Environment-Side Adaptation	35.75	52.00	31.00	32.00	28.00
w/o Diagnosis-Guided Reweighting	32.75	48.00	27.00	31.00	25.00
w/o Closed-Loop Update	33.00	46.00	32.00	28.00	26.00
Full SEAL	40.25	58.00	36.00	34.00	33.00

Taken together, the in-distribution and out-of-distribution results point to a *mechanism-level* advantage: verifier-grounded diagnosis improves both sample-efficient learning on BFCL V3 and transfer to held-out settings. We therefore next ask which parts of the closed loop matter most.

4.4 Ablation Study

To explain where the gains come from, we ablate the three mechanisms built on verifier-grounded diagnosis: environment-side adaptation, diagnosis-guided reweighting, and closed-loop interface updates. Since diagnosis is the shared input to all SEAL components, removing it entirely would reduce the method to Vanilla RL.

Relative contribution of each component. Removing diagnosis-guided reweighting causes a larger drop than removing environment-side adaptation, suggesting that diagnostic labels are especially important for allocating policy-gradient pressure. Still, environment-side adaptation contributes +4.50 points over the variant without it, and Full SEAL outperforms both single-sided variants. This pattern suggests that the two uses of diagnosis are complementary.

Together, these ablations show that SEAL’s advantage does not come from a single ingredient, but from combining better training-time feedback with better optimization of the resulting rollouts.

4.5 Training Dynamics

Having identified which components matter most, we finally examine *how* the gains emerge during training. **Learning dynamics.** Figure 3 compares the validation trajectories of Vanilla RL and SEAL across the four BFCL V3 categories. SEAL improves more rapidly and reaches higher validation accuracy on most subsets,

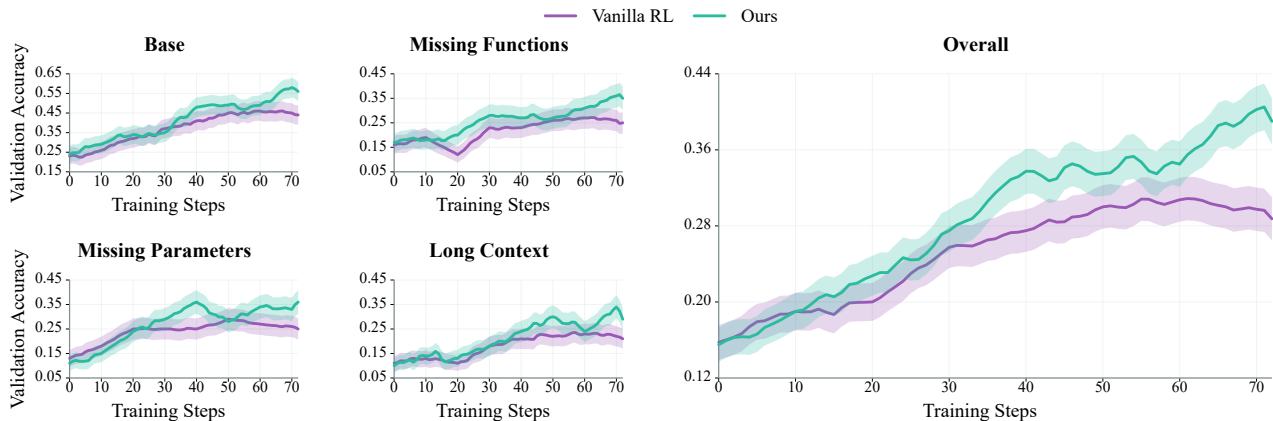


Figure 3 Training dynamics comparing Vanilla RL and SEAL on BFCL V3 validation subsets. We plot validation accuracy across training steps for *Base*, *Missing Functions*, *Missing Parameters*, and *Long Context*. Shaded regions denote binomial standard error estimated from each 100-sample validation subset.

especially on *Missing Functions* and *Missing Parameters*, where failures often require identifying missing tool use or repairing invalid arguments. On *Base*, both methods improve early, but SEAL climbs faster and stabilizes at a higher level. The contrast is larger on *Missing Functions* and *Missing Parameters*, where Vanilla RL improves only gradually while SEAL opens a clearer gap as training proceeds. *Long Context* remains the hardest subset for both methods, but SEAL still shows a more consistent upward trend and a higher final accuracy.

The overall curve is also smoother and higher for SEAL, suggesting that the gains are not driven by a single subset but accumulate across failure types over the course of training. The widening gap after the early phase is consistent with SEAL’s closed-loop design: as the policy’s dominant failures change, the training interface is updated to expose more relevant cues, whereas Vanilla RL continues to receive the same sparse terminal signal. This pattern supports the view that verifier-grounded diagnosis not only improves final performance, but also makes learning more targeted and sample-efficient throughout training.

5 Conclusion and Limitations

We introduced SEAL, a closed-loop framework for co-evolving tool-use agents and their training-time learning environments. By using verifier-grounded failure diagnosis as a shared signal for learning-interface evolution and policy optimization, SEAL addresses *Agent-Environment Misalignment* while preserving tool semantics, task labels, and the evaluation verifier. Experiments on in-distribution and out-of-distribution multi-turn tool-use benchmarks show consistent gains in low-resource learning across multiple backbones.

More broadly, our results suggest that improving tool-use agents is not only a matter of optimizing the policy against a fixed environment. Adapting the training-time interface to the agent’s revealed failures can make learning more targeted, sample-efficient, and robust to structured multi-turn errors.

Several limitations remain. SEAL relies on executable environments with tool schemas, execution traces, and verifier feedback, so more open-ended domains may require richer diagnosis mechanisms and weaker forms of supervision. Its environment evolution is intentionally conservative, modifying the training-time interface rather than tool semantics or evaluation rules, which preserves fairness but limits the scope of adaptation. Diagnostic utility weights are also fixed across tasks and backbones, and cross-domain generalization remains challenging.

Future work includes extending diagnosis to less structured environments, learning adaptive utility weights, and studying how co-evolution scales to broader tool ecosystems and longer-horizon agentic workflows.

References

- [1] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.
- [2] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.
- [3] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.
- [4] Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiushi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*, 2025.
- [5] Tianxin Wei, Ting-Wei Li, Zhining Liu, Xuying Ning, Ze Yang, Jiaru Zou, Zhichen Zeng, Ruizhong Qiu, Xiao Lin, Dongqi Fu, et al. Agentic reasoning for large language models. *arXiv preprint arXiv:2601.12538*, 2026.
- [6] Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jian-Guang Lou, Qingwei Lin, Ping Luo, and Saravan Rajmohan. Agentgen: Enhancing planning abilities for large language model based agent via environment and task generation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, pages 496–507, 2025.
- [7] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
- [8] Yunpeng Zhai, Shuchang Tao, Cheng Chen, Anni Zou, Ziqian Chen, Qingxu Fu, Shinji Mai, Li Yu, Jiayi Deng, Zouying Cao, et al. Agentevolver: Towards efficient self-evolving agent system. *arXiv preprint arXiv:2511.10395*, 2025.
- [9] Jiaye Lin, Yifu Guo, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, Mingguang Chen, Hongzhang Liu, Ronghao Chen, Yangfan He, et al. Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents. *arXiv preprint arXiv:2508.02085*, 2025.
- [10] Zeyi Sun, Ziyu Liu, Yuhang Zang, Yuhang Cao, Xiaoyi Dong, Tong Wu, Dahua Lin, and Jiaqi Wang. Seagent: Self-evolving computer use agent with autonomous learning from experience. *arXiv preprint arXiv:2508.04700*, 2025.
- [11] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 3102–3116, 2023.
- [12] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.
- [13] Kaiming Liu, Xuanyu Lei, Ziyue Wang, Peng Li, and Yang Liu. Agent-environment alignment via automated interface generation. *arXiv preprint arXiv:2505.21055*, 2025.
- [14] Xingkun Yin and Hongyang Du. Glove: Global verifier for llm memory-environment realignment. *arXiv preprint arXiv:2601.19249*, 2026.
- [15] Hanli Peng, Yongsen Zheng, Ziyao Liu, and Kwok-Yan Lam. Tool execution hallucination in llm-based agents: A unified taxonomy with detection, mitigation, and future directions. *TechRxiv*, 2026.
- [16] Shuo Yang, Soyeon Caren Han, Xueqi Ma, Yan Li, Mohammad Reza Ghasemi Madani, and Eduard Hovy. Evotool: Self-evolving tool-use policy optimization in llm agents via blame-aware mutation and diversity-aware selection. *arXiv preprint arXiv:2603.04900*, 2026.
- [17] Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiabin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. *arXiv preprint arXiv:2508.05004*, 2025.
- [18] Siyu Yuan, Zehui Chen, Zhiheng Xi, Junjie Ye, Zhengyin Du, and Jiecao Chen. Agent-r: Training language model agents to reflect via iterative self-training. *arXiv preprint arXiv:2501.11425*, 2025.

- [19] Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.
- [20] Zhishang Xiang, Chengyi Yang, Zerui Chen, Zhimin Wei, Yunbo Tang, Zongpei Teng, Zexi Peng, Zongxia Li, Chengsong Huang, Yicheng He, et al. A systematic survey of self-evolving agents: From model-centric to environment-driven co-evolution. *TechRxiv*, 2026.
- [21] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alf-world: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.
- [22] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.
- [23] Siyuan Lu, Zechuan Wang, Hongxuan Zhang, Qintong Wu, Leilei Gan, Chenyi Zhuang, Jinjie Gu, and Tao Lin. Don’t just fine-tune the agent, tune the environment. *arXiv preprint arXiv:2510.10197*, 2025.
- [24] Shidong Yang, Ziyu Ma, Tongwen Huang, Yiming Hu, Yong Wang, and Xiangxiang Chu. Coevolve: Training llm agents via agent-data mutual evolution. *arXiv preprint arXiv:2604.15840*, 2026.
- [25] Bingguang Hao, Zengzhuang Xu, Yuntao Wen, Xinyi Xu, Yang Liu, Tong Zhao, Maolin Wang, Long Chen, Dong Wang, Yicheng Chen, et al. From failure to mastery: Generating hard samples for tool-use agents. *arXiv preprint arXiv:2601.01498*, 2026.
- [26] Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.
- [27] Hongzhuo Yu, Fei Zhu, Guo-Sen Xie, and Ling Shao. Self-consolidation for self-evolving agents. *arXiv preprint arXiv:2602.01966*, 2026.
- [28] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.
- [29] Adam Zweiger, Jyothish Pari, Han Guo, Ekin Akyürek, Yoon Kim, and Pulkit Agrawal. Self-adapting language models. *arXiv preprint arXiv:2506.10943*, 2025.
- [30] Tabet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher-student curriculum learning. *IEEE Transactions on Neural Networks and Learning Systems*, 31(9):3732–3740, 2020.
- [31] Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic curriculum learning for deep RL: A short survey. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 4819–4825, 2020.
- [32] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, 2023.
- [33] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*, 2024.
- [34] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.
- [35] Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939, 2023.
- [36] Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*, 2023.

- [37] Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.
- [38] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.
- [39] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.
- [40] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swbench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.
- [41] Zhiheng Xi, Yiwen Ding, Wenxiang Chen, Boyang Hong, Honglin Guo, Junzhe Wang, Dingwen Yang, Chenyang Liao, Xin Guo, Wei He, et al. Agentgym: Evolving large language model-based agents across diverse environments. *arXiv preprint arXiv:2406.04151*, 2024.
- [42] Jiayi Zhang, Yiran Peng, Fanqi Kong, Cheng Yang, Yifan Wu, Zhaoyang Yu, Jinyu Xiang, Jianhao Ruan, Jinlin Wang, Maojia Song, et al. Autoenv: Automated environments for measuring cross-environment agent learning. *arXiv preprint arXiv:2511.19304*, 2025.
- [43] Jiacheng Guo, Ling Yang, Peter Chen, Qixin Xiao, Yinjie Wang, Xinzhe Juan, Jiahao Qiu, Ke Shen, and Mengdi Wang. Genenv: Difficulty-aligned co-evolution between llm agents and environment simulators. *arXiv preprint arXiv:2512.19682*, 2025.
- [44] Google. Gemini 3 pro preview model documentation. Google AI for Developers documentation, 2025. <https://ai.google.dev/gemini-api/docs/models/gemini-3-pro-preview>.
- [45] Anthropic. Claude sonnet 4.5 system card. System card, 2025. <https://www.anthropic.com/claude-sonnet-4-5-system-card>.
- [46] Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, Aaron Ostrow, Aaron Welihinda, Alex Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.
- [47] Gheorghe Comanici, David Bieber, Mike Schaekermann, Panupong Pasupat, Nitish Sachdeva, Inderjit Dhillon, Michael Blistein, Omer Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.
- [48] Zhipu AI. Glm-4.6 model card. Hugging Face model card, 2025. <https://huggingface.co/zai-org/GLM-4.6>.
- [49] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jinkai Xu, Jing Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Ke-Yang Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Tang, Tianyi Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Wang, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- [50] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zihan Qiu, and Zekun Li. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- [51] Salesforce AI Research. xlam-2-3b-fc-r model card. Hugging Face model card, 2025. <https://huggingface.co/Salesforce/xLAM-2-3b-fc-r>.
- [52] Weiwen Liu, Xingshan Zeng, Keqing He, Yongliang Wang, Zhaoyang Yan, Fanya Wang, Jingsheng Cheng, Runlong Wang, Minpeng Shen, Xin Jiang, Yujie Qian, Qun Liu, and Lifeng Shang. Toolace: Winning the points of llm function calling. In *International Conference on Learning Representations*, 2025.

- [53] BitAgent. Bitagent-8b model card. Hugging Face model card, 2025. <https://huggingface.co/BitAgent/BitAgent-8B>.
- [54] watt-ai. watt-tool-8b model card. Hugging Face model card, 2024. <https://huggingface.co/watt-ai/watt-tool-8B>.

Appendix

A Benchmark and Evaluation Details

BFCL V3. Our in-distribution evaluation uses the multi-turn subset of the Berkeley Function-Calling Leaderboard (BFCL) V3. The benchmark evaluates whether an agent can correctly use executable tools over multi-turn dialogue contexts. Each task contains a user instruction sequence, a set of available tools, and an executable evaluation protocol. The subset contains 800 examples across four categories: *Base*, *Missing Functions*, *Missing Parameters*, and *Long Context*. *Base* evaluates standard multi-turn function calling with complete tool information. *Missing Functions* tests whether the agent can recognize when required tools are unavailable or when no valid tool applies. *Missing Parameters* evaluates whether the agent can handle underspecified arguments rather than hallucinating missing values. *Long Context* stresses state tracking and tool-use consistency over extended dialogue histories.

Following the low-resource setting in the main experiments, we construct a 400-example training split by sampling 100 instances from each category. The remaining 400 examples are used for held-in evaluation. All SEAL and Vanilla RL runs use the same training split and held-in test split, ensuring that performance differences are attributable to the training method rather than data scale or split variation.

BFCL V4. For out-of-distribution evaluation, we use the Web Search and Memory tracks from BFCL V4. These tracks differ from BFCL V3 in tool schema structure, task distribution, and interaction pattern. The Web Search track evaluates whether an agent can retrieve and synthesize information using search-style tools. The Memory track evaluates whether an agent can store, retrieve, and update memory-like information across interactions. Since SEAL is trained only on the BFCL V3 training split, BFCL V4 serves as a held-out benchmark for testing cross-distribution transfer.

τ^2 -bench. We also evaluate on τ^2 -bench across the Retail, Airline, and Telecom domains. These tasks require multi-turn tool use in simulated customer-service environments, where the agent must follow domain policies, interact with tools, and maintain task state across turns. Compared with BFCL, τ^2 -bench introduces different tool semantics, domain constraints, and user-agent interaction patterns, making it a complementary testbed for OOD generalization.

Evaluation protocol and metric. All evaluations are performed using the original benchmark environments and verifiers. SEAL-specific diagnostic feedback, interface augmentation, capability-specific hints, and recovery-oriented training messages are disabled during evaluation. Thus, all reported results measure the policy under the original test-time protocol. We report success rate as the primary metric:

$$\text{SR} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \mathbf{1}[\mathcal{V}(\tau_q, q, \mathcal{E}) = 1], \quad (17)$$

where \mathcal{Q} is the evaluation set and τ_q is the trajectory generated for instruction q . Category-level scores are computed on the corresponding subset, and average scores are computed as the arithmetic mean over reported categories.

B SEAL Training Algorithm

Algorithm 1 summarizes the full SEAL training procedure. SEAL alternates between policy rollout, verifier-grounded diagnosis, diagnosis-guided interface evolution, and policy optimization. The interface wrapper is used only during training and is removed during final evaluation.

At evaluation time, SEAL discards the training-time interface wrapper and evaluates π_{θ_R} in the original environment \mathcal{E} with the unchanged verifier \mathcal{V} .

Algorithm 1 SEAL: Synergistic Co-Evolution of Agents and Learning Environments

Require: Initial policy π_{θ_0} , reference policy π_{ref} , training prompts \mathcal{D} , base environment \mathcal{E} , verifier \mathcal{V} , number of rounds R , rollouts per prompt G

Require: Initial interface wrapper Ω_0 , diagnostic utility $\rho(z)$, clipping range $[w_{\min}, w_{\max}]$

```
1: for  $r = 0, 1, \dots, R - 1$  do
2:   Sample prompt batch  $\mathcal{X}_r \subset \mathcal{D}$ 
3:   Construct training environment  $\mathcal{E}_{\Omega_r} \leftarrow \Omega_r(\mathcal{E})$ 
4:   Initialize rollout buffer  $\mathcal{R}_r \leftarrow \emptyset$ 
5:   for each prompt  $x \in \mathcal{X}_r$  do
6:     Sample  $G$  trajectories  $\{\tau_j\}_{j=1}^G$  from  $\pi_{\theta_r}$  in  $\mathcal{E}_{\Omega_r}$ 
7:     for each trajectory  $\tau_j$  do
8:       Compute verifier reward  $r_j \leftarrow \mathcal{V}(\tau_j, x, \mathcal{E})$ 
9:       Diagnose trajectory  $Z(\tau_j) \leftarrow \Psi(\tau_j, \mathcal{E}, \mathcal{V})$ 
10:      Store  $(x, \tau_j, r_j, Z(\tau_j))$  in  $\mathcal{R}_r$ 
11:    end for
12:  end for
13:  Aggregate failure profile  $C_r \leftarrow \text{Agg}(\{Z(\tau_j) : \tau_j \in \mathcal{R}_r\})$ 
14:  for each prompt group in  $\mathcal{R}_r$  do
15:    Compute group-relative advantages  $A_j$  from verifier rewards  $\{r_j\}_{j=1}^G$ 
16:    Compute diagnostic weights  $w_j \leftarrow \text{clip}(\sum_{z \in \mathcal{Z}} p_{\tau_j}(z)\rho(z), w_{\min}, w_{\max})$ 
17:    Set weighted advantages  $\tilde{A}_j \leftarrow w_j A_j$ 
18:  end for
19:  Update policy  $\pi_{\theta_r} \rightarrow \pi_{\theta_{r+1}}$  using GRPO with  $\tilde{A}_j$  and KL regularization
20:  Update interface  $\Omega_{r+1} \leftarrow U(\Omega_r, C_r)$ 
21: end for
22: return trained policy  $\pi_{\theta_R}$ 
```

C Experimental Details

This section provides implementation details for reproducing the training runs and for verifying that SEAL is compared against Vanilla RL under a controlled protocol. Unless otherwise stated, all BFCL experiments use the same base model, training split, rollout budget, decoding configuration, and evaluation verifier across methods.

C.1 Hyperparameter Configuration

Table 4 summarizes the main training configuration used in our BFCL experiments. We use Qwen2.5-7B-Instruct as the default backbone for the main BFCL experiments and apply the same GRPO configuration to Vanilla RL and SEAL. The only algorithmic differences are whether verifier-grounded diagnosis, diagnosis-guided interface evolution, closed-loop failure-profile updates, and diagnostic advantage weighting are enabled.

For ablations with different backbones, we keep the same data split, rollout budget, verifier, decoding configuration, and evaluation protocol whenever possible. When hardware constraints require changes to micro-batch size or gradient accumulation, we keep the effective batch size fixed.

C.2 Diagnosis Label Priority

When multiple failures are detected at the same turn, SEAL assigns the dominant label according to a fixed priority order. The priority favors directly executable failures over downstream verifier-level failures, as shown in Table 5.

This priority prevents SEAL from attributing a trajectory-level failure to a generic response mismatch when a

Table 4 Main hyperparameter configuration for BFCL experiments. Unless otherwise specified, Vanilla RL and SEAL use the same optimization, rollout, and evaluation settings.

Category	Hyperparameter	Value
Policy optimization	Base model	Qwen2.5-7B-Instruct
	Training method	GRPO
	Actor learning rate	1×10^{-6}
	Train batch size	32
	PPO mini-batch size	8
	PPO micro-batch size / GPU	1
Rollout	Inference engine	vLLM (async)
	Rollouts per prompt	8
	Training temperature	1.0
	Validation temperature	0.0
	Max prompt length	8192
	Max response length	4096
	Max model length	16384
Multi-turn setting	Number of training samples	400
	Multi-turn rollout	Enabled
	Max interaction steps	20
	Tool format	llama3_json
	BFCL tool prompt mode	bfcl_qwen_fc
Regularization	Reward KL penalty	Disabled
	Actor KL loss coefficient	0.001
	Entropy coefficient	0.0
Hardware	GPUs	4
	Tensor parallel size	1
	GPU memory utilization	0.6
	Max environment workers	32

concrete tool or argument error is already observed. For example, if a trajectory fails the final verifier because an earlier tool call used an invalid argument, SEAL assigns `argument_mismatch` rather than `response_mismatch`.

We clip the resulting trajectory-level diagnostic weight to $[w_{\min}, w_{\max}] = [0.5, 2.0]$ to avoid overly large updates from rare or noisy diagnostic patterns.

C.3 Controlled Comparison Protocol

The comparison between Vanilla RL and SEAL is controlled along the following dimensions. Both methods use the same initial checkpoint, the same set of 400 BFCL V3 training samples, the same number of rollouts per prompt, the same optimizer, and the same decoding configuration. Both methods are evaluated using the original BFCL V3 multi-turn environment and the official verifier.

Vanilla RL optimizes the standard GRPO objective using only verifier rewards. SEAL uses the same verifier rewards and the same GRPO backbone, but additionally uses verifier-grounded diagnoses from training rollouts to construct training-time interface updates and diagnostic utility weights. No method receives ground-truth tool trajectories, hidden parameter values, reference intermediate actions, or final answers during training.

The evolved SEAL interface is used only during training. During evaluation, the wrapper is removed, and

Table 5 Priority order for assigning diagnostic labels when multiple failure signals are present.

Priority	Label	Rationale
1	invalid_tool_call	The action is not parseable or calls an unavailable tool.
2	argument_mismatch	The tool is valid, but required fields, types, enum values, or argument structure are invalid.
3	state_mismatch	The action executes but fails to produce the expected observable state transition.
4	recovery_failure	The model receives executable error feedback but fails to recover in later turns.
5	missing_tool_call	The trajectory fails because the model skips necessary tool use or responds prematurely.
6	response_mismatch	Tool interaction is acceptable, but the final answer fails the verifier.

Table 6 Diagnostic utility weights used by SEAL.

Diagnostic Label	$\rho(z)$	Rationale
spurious_tool_call	2.0	Invalid or unnecessary tool use is directly attributable.
correct_abstention	1.8	Rewards learning when no tool should be called.
empty_turn_model_response	1.5	Captures missed tool-use or premature non-action.
state_mismatch	1.0	Indicates failure to reach the expected environment state.
response_mismatch	0.9	Final response is wrong, but attribution is less localized.
instance_mismatch	0.6	Instance errors are useful but often noisier or less localized.
pass	1.0	Successful trajectories keep the baseline update scale.

the model interacts with the original benchmark interface. Thus, SEAL and Vanilla RL are evaluated under identical test-time conditions. The comparison therefore measures whether the policy has internalized better tool-use behavior, rather than whether it has access to additional test-time information.

C.4 Logging and Reproducibility

For each run, we log the random seed, base checkpoint, code version, training configuration, configuration overrides, rollout budget, decoding parameters, validation scores, round-level failure profiles C_r , interface states Ω_r , diagnostic label distributions, and final evaluation outputs. These logs allow us to verify that changes in performance are attributable to SEAL components rather than differences in training budget, data split, decoding configuration, or evaluation protocol.

D Prompt and Interface Templates

This section reports the prompt and training-time interface templates used in our BFCL implementation. The base tool-use prompt is shared by Vanilla RL and SEAL. SEAL modifies only the training-time environment interface, for example by annotating tool schemas, enriching failed tool feedback when enabled, or injecting short behavioral guidelines derived from recent diagnostic profiles. All SEAL-specific interface augmentations are disabled during evaluation.

D.1 Base BFCL Tool-Use Prompt

For Qwen-style BFCL experiments, the environment serializes the available tools into the system prompt using the following template. This base prompt is used by both Vanilla RL and SEAL.

BFCL Qwen Tool Prompt Template

```
# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML tags:
<tools>
$tool_schema_json_1
$tool_schema_json_2
...
</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>
```

The base prompt defines the tool-call format and exposes the available tool schemas. It does not contain SEAL-specific diagnostic labels, recovery hints, or capability-dependent instructions.

D.2 Schema Observation Annotation

In our active BFCL configuration, SEAL uses a lightweight observation annotation mechanism. When `observation_lite` is enabled, SEAL does not change tool APIs, tool signatures, tool outputs, or verifier behavior. Instead, it augments the exposed schema descriptions with lightweight annotations that make existing constraints more explicit.

Schema Observation Annotation Template

```
$argument_description [required]

$argument_description Allowed values: $enum_value_1, $enum_value_2, ...
```

The environment also appends the following training-time instruction:

Observation-Lite Instruction

```
Observation note: parameters marked [required] must be provided exactly when calling a tool; do not
  invent tools outside the current schema.
```

This annotation corresponds to the schema-affordance component ϕ_{schema} . It exposes only public schema information, such as required arguments and enum constraints, and does not reveal the correct tool choice, hidden parameter values, reference tool trajectory, or final answer for any instance.

D.3 Tool Feedback Evolution Template

When `tool_feedback_evolution` is enabled, SEAL enriches only failed interactions. This component corresponds to the recovery-feedback component ϕ_{err} . It converts low-level parse or execution errors into structured recovery hints, while preserving the original tool API and verifier.

For parser failures, the environment may append format-specific hints such as:

Parse Error Hint Template

`$base_parse_error`

[Hint] Each tool call MUST be a JSON object inside `<tool_call>...</tool_call>` tags with both "name" and "arguments" keys, e.g. `<tool_call>{"name": "<tool>", "arguments": {"<arg>": <value>}}</tool_call>`.

[Hint] Available tools: `$tool_name_1`, `$tool_name_2`, ...

[Hint] If no available tool can serve the user's request, reply in plain text instead of producing a malformed `tool_call`.

For execution errors, the environment may append tool-specific but non-solution-revealing hints such as:

Tool Error Hint Template

[ToolHint] Required argument(s) `$args` for ``$tool_name`` were missing. Re-emit the `tool_call` with these keys present in ``arguments``.

[ToolHint] ``$tool_name`` does not accept argument ``$arg``. Drop it and only use arguments listed in the tool schema.

[ToolHint] ``$tool_name`` rejected an argument type. Re-check argument types in the tool schema (string vs int vs list).

These hints identify the class of tool-use error and the violated interface constraint. They do not provide the next correct tool call, instance-specific argument values, or the ground-truth solution trajectory.

D.4 Diagnostic Evolution Guideline Template

When `diagnostic_evolution` is enabled, SEAL reads recent failure statistics from the capability profile and injects short behavioral guidelines into the system prompt. This component corresponds to ϕ_{cap} : the interface is adjusted according to recurring failure modes of the current policy rather than individual benchmark instances.

Diagnostic-Driven System Guideline Template

`# Behavioral guidelines (env-evolved from recent failures):`

- [Cautious-call] If no available function clearly matches the user's request, respond in plain text rather than fabricating a tool call.
- [Active-respond] When the user's request CAN be served by an available function, you MUST emit a `tool_call` rather than only replying in plain text.
- [State-aware] Before issuing a state-mutating call, re-read the most recent tool results to confirm the current state matches what the arguments assume.
- [Scope] Verify that the instance/target your call refers to actually appeared in earlier tool results before calling on it.
- [Format] Match your final reply to the format implied by the user's request: lists, units, exact strings, ordering.

The guidelines are selected from aggregate failure patterns, such as spurious tool calls, missing tool calls,

state mismatches, or final-response format errors. They are capability-level instructions rather than per-instance hints. Therefore, they do not expose the reference trajectory, hidden state, hidden argument values, or target answer of any benchmark example.

Active configuration. In the BFCL configuration used for the main rerun, the active training-time interface uses the base BFCL prompt together with `observation_lite` schema annotation. The additional `tool_feedback_evolution` and `diagnostic_evolution` templates are included here to document the complete SEAL interface implementation, and should be interpreted as optional components when enabled by configuration.

E Qualitative Case Studies

We present four qualitative case studies to illustrate how SEAL improves multi-turn tool use across different capability types: file-system state tracking, API-parameter recovery, cross-domain tool composition, and vehicle-control state maintenance. Each case compares the behavior of the base model before SEAL training with the behavior of the SEAL-trained model on the same BFCL task. We also include the expected outcome for post-hoc analysis only; these references are not exposed to the policy during training or evaluation.

We focus on critical turns where the base model fails and the SEAL-trained model succeeds. These examples are not intended to replace the controlled Vanilla RL comparison in the main experiments; rather, they illustrate the types of tool-use failures that SEAL is designed to diagnose and correct.

E.1 File-System State Tracking

This case studies a file-system task requiring the agent to compare report files, move and rename a note file inside an archive directory, retrieve a support ticket, and finally resolve it. The key challenge is maintaining the current directory state and respecting file-system tool constraints.

Task: BFCL multi_turn_base_24 **Model:** Base model **Result:** reward = 0.0

User Query 1: Compare the draft and final report files.

```
Assistant:
<tool_call>
{"name": "find", "arguments": {"path": "", "name": "draft_report"}}
</tool_call>
<tool_call>
{"name": "find", "arguments": {"path": "", "name": "final_report"}}
</tool_call>
<tool_call>
{"name": "diff", "arguments": {"file_name1": "draft_report",
"file_name2": "final_report"}}
</tool_call>

Tool Response:
{"matches": []}
{"matches": []}
{"error": "diff: draft_report or final_report: No such file or directory"}
```

User Query 2: Transfer temp_notes.txt into archives and rename it to notes_2024.txt.

Assistant:
<tool_call>
{ "name": "mv", "arguments": { "source": "temp_notes.txt",
"destination": "archives/temp_notes.txt" } }
</tool_call>
<tool_call>
{ "name": "mv", "arguments": { "source": "archives/temp_notes.txt",
"destination": "notes_2024.txt" } }
</tool_call>

Tool Response:
{ "error": "mv: path not allowed in destination. Provide only a file or directory name." }
{ "error": "mv: cannot move 'archives/temp_notes.txt': No such file or directory" }

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

Tool Response:
{ "current_directory_content": ["report_draft.txt", "report_final.txt",
"temp_notes.txt", "archives"] }

Assistant:
<tool_call>
{ "name": "mv", "arguments": { "source": "temp_notes.txt",
"destination": "archives/temp_notes.txt" } }
</tool_call>

Tool Response:
{ "error": "mv: path not allowed in destination. Provide only a file or directory name." }

User Query 3: Retrieve support ticket 987654.

Assistant:
<tool_call>
{ "name": "get_ticket", "arguments": { "ticket_id": 987654 } }
</tool_call>

Tool Response:
{ "id": 987654, "status": "open",
"description": "Issue with workstation not booting properly.", "resolution": "" }

Failure analysis. The base model locates the relevant file names only after an initial failure, but continues to issue invalid path-style arguments to mv. This indicates that it does not convert the execution error into a revised state-aware plan.

Task: BFCL multi_turn_base_24 **Model:** Base model + SEAL **Result:** reward = 1.0

User Query 1: Compare the draft and final report files.

Assistant:
<tool_call>
{\"name\": \"find\", \"arguments\": {\"path\": \"\", \"name\": \"draft report\"}}
</tool_call>
<tool_call>
{\"name\": \"find\", \"arguments\": {\"path\": \"\", \"name\": \"final report\"}}
</tool_call>

Tool Response:
{\"matches\": []}
{\"matches\": []}

Assistant:
<tool_call>
{\"name\": \"ls\", \"arguments\": {}}
</tool_call>

Tool Response:
{\"current_directory_content\": [\"report_draft.txt\", \"report_final.txt\",
\"temp_notes.txt\", \"archives\"]}

Assistant:
<tool_call>
{\"name\": \"diff\", \"arguments\": {\"file_name1\": \"report_draft.txt\",
\"file_name2\": \"report_final.txt\"}}
</tool_call>

Tool Response:
{\"diff_lines\": \"- Initial draft content for the report.\\n+ Finalized content for the report.\"}

User Query 2: Transfer temp_notes.txt into archives and rename it to notes_2024.txt.

Assistant:
<tool_call>
{\"name\": \"mv\", \"arguments\": {\"source\": \"temp_notes.txt\",
\"destination\": \"archives\"}}
</tool_call>

Tool Response:
{\"result\": \"'temp_notes.txt' moved to 'archives/temp_notes.txt'\"}

Assistant:
<tool_call>
{\"name\": \"cd\", \"arguments\": {\"folder\": \"archives\"}}
</tool_call>
<tool_call>
{\"name\": \"mv\", \"arguments\": {\"source\": \"temp_notes.txt\",
\"destination\": \"notes_2024.txt\"}}
</tool_call>

Tool Response:
{\"current_working_directory\": \"/archives\"}
{\"result\": \"'temp_notes.txt' moved to 'notes_2024.txt'\"}

User Query 3: Retrieve support ticket 987654.

Assistant:
<tool_call>
{ "name": "get_ticket", "arguments": { "ticket_id": 987654 } }
</tool_call>

Tool Response:
{ "id": 987654, "status": "open",
"description": "Issue with workstation not booting properly.", "resolution": "" }

User Query 4: Resolve the ticket with: Fixed through manual troubleshooting techniques.

Assistant:
<tool_call>
{ "name": "resolve_ticket", "arguments": { "ticket_id": 987654,
"resolution": "Fixed through manual troubleshooting techniques." } }
</tool_call>

Tool Response:
{ "status": "Ticket 987654 has been resolved successfully." }

Success analysis. The SEAL-trained model uses observable directory contents to infer the actual file names, then respects the local-directory constraint by moving into archives before renaming the file. This shows improved recovery from tool feedback and better file-system state tracking.

Expected Outcome: Compare report_draft.txt and report_final.txt; move temp_notes.txt into archives; rename it to notes_2024.txt; retrieve ticket 987654; resolve it with the provided resolution string.

E.2 Recovering from API Parameter Mismatch

This case studies a travel API task. The user asks for the economy flight cost from Rivermist to Stonebrook on October 6, 2026, and later asks to set a spending budget provided in GBP. The model must recover from invalid airport parameters and convert the budget into the expected currency.

Task: BFCL multi_turn_miss_param_150 **Model:** Base model **Result:** reward = 0.0

User Query 1: Find the economy flight cost from Rivermist to Stonebrook on October 6, 2026.

Assistant:
<tool_call>
{
 "name": "get_flight_cost",
 "arguments": {
 "travel_from": "RVM",
 "travel_to": "STB",
 "travel_date": "2026-10-06",
 "travel_class": "economy"
 }
}

Tool Response:
Error during execution: No available route for the given airports.

Assistant:
There is no direct flight route available between Rivermist and Stonebrook.

User Query 2: Let's anchor my daily spend at a specific amount.

Assistant:
Please provide authentication details before I can set a budget.

User Query 3: It is 15400 GBP. My access token is abc123token.

Assistant:
<tool_call>
{
 "name": "set_budget_limit",
 "arguments": {
 "access_token": "abc123token",
 "budget_limit": 15400.0
 }
}

Tool Response:
{
 "budget_limit": 15400.0
}

Failure analysis. The base model treats guessed airport codes as final and does not use the airport-resolution tools after the route error. It also sets the GBP amount directly as the budget limit, missing the required currency conversion.

Task: BFCL multi_turn_miss_param_150 **Model:** Base model + SEAL **Result:** reward = 1.0

User Query 1: Find the economy flight cost from Rivermist to Stonebrook on October 6, 2026.

Assistant:

```
<tool_call>
{"name": "get_flight_cost", "arguments": {"travel_from": "RVM",
"travel_to": "STB", "travel_date": "2026-10-06",
"travel_class": "economy"}}
</tool_call>
```

Tool Response:

Error during execution: No available route for the given airports.

Assistant:

```
<tool_call>
{"name": "get_nearest_airport_by_city",
"arguments": {"location": "Rivermist"}}
</tool_call>
<tool_call>
{"name": "get_nearest_airport_by_city",
"arguments": {"location": "Stonebrook"}}
</tool_call>
```

Tool Response:

```
{"nearest_airport": "RMS"}
{"nearest_airport": "SBK"}
```

Assistant:

```
<tool_call>
{"name": "get_flight_cost", "arguments": {"travel_from": "RMS",
"travel_to": "SBK", "travel_date": "2026-10-06",
"travel_class": "economy"}}
</tool_call>
```

Tool Response:

```
{"travel_cost_list": [220.0]}
```

User Query 3: It is 15400 GBP. My access token is abc123token.

Assistant:

```
<tool_call>
{"name": "compute_exchange_rate", "arguments": {"base_currency": "GBP",
"target_currency": "USD", "value": 15400}}
</tool_call>
```

Tool Response:

```
{"exchanged_value": 22000.0}
```

Assistant:

```
<tool_call>
{"name": "set_budget_limit", "arguments": {"access_token": "abc123token",
"budget_limit": 22000.0}}
</tool_call>
```

Tool Response:

```
{"budget_limit": 22000.0}
```

Success analysis. The SEAL-trained model recovers by using the missing airport-resolution tools and later converts GBP to USD before calling `set_budget_limit`. This reflects better handling of intermediate API dependencies and parameter constraints.

Expected Outcome: Resolve Rivermist to RMS and Stonebrook to SBK; query `get_flight_cost(RMS, SBK, 2026-10-06, economy)`; convert 15400 GBP to 22000 USD; set the budget limit to 22000.0.

E.3 Cross-Domain Tool Composition

This case combines stock-market and Twitter APIs. The model must retrieve the price of Omega Industries, insert the price into a tweet, include `#TechBoom` and `@industryexperts`, and later add `@technewsworld` to the existing tweet.

Task: BFCL multi_turn_base_147 **Model:** Base model **Result:** reward = 0.0

User Query 1: Find Omega Industries' price and post a tweet with the price, `#TechBoom`, and `@industryexperts`.

Assistant:
<tool_call>
{ "name": "get_stock_info", "arguments": { "symbol": "OMGA" } }
</tool_call>

Tool Response:
{ "error": "Stock with symbol 'OMGA' not found." }

Assistant:
Let's use a more common tech stock, such as AAPL.

<tool_call>
{ "name": "get_stock_info", "arguments": { "symbol": "AAPL" } }
</tool_call>

Tool Response:
{ "price": 227.16, ... }

Assistant:
<tool_call>
{ "name": "post_tweet", "arguments": { "content":
"Apple Inc. is skyrocketing at \$227.16 per share! ...",
"tags": ["#TechBoom"], "mentions": ["@industryexperts"] } }
</tool_call>

User Query 2: Add `@technewsworld` to amplify the message.

```
Assistant:
<tool_call>
{"name": "post_tweet", "arguments": {"content":
"Apple Inc. is skyrocketing at $227.16 per share! ...",
"tags": ["#TechBoom"],
"mentions": ["@industryexperts", "@technewsworld"]}}
</tool_call>
```

Failure analysis. The base model substitutes Apple for Omega Industries after the guessed symbol fails, thereby changing the user's target entity. It also posts a new tweet rather than adding the new mention to the existing tweet.

Task: BFCL multi_turn_base_147 **Model:** Base model + SEAL **Result:** reward = 1.0

User Query 1: Find Omega Industries' price and post a tweet with the price, #TechBoom, and @industryexperts.

```
Assistant:
<tool_call>
{"name": "get_stock_info", "arguments": {"symbol": "OMGA"}}
</tool_call>
```

```
Tool Response:
{"error": "Stock with symbol 'OMGA' not found."}
```

```
Assistant:
<tool_call>
{"name": "get_symbol_by_name",
"arguments": {"name": "Omega Industries"}}
</tool_call>
```

```
Tool Response:
{"symbol": "OMEG"}
```

```
Assistant:
<tool_call>
{"name": "get_stock_info", "arguments": {"symbol": "OMEG"}}
</tool_call>
```

```
Tool Response:
{"price": 457.23, ...}
```

```
Assistant:
<tool_call>
{"name": "post_tweet", "arguments": {"content":
"Omega Industries is skyrocketing at $457.23 per share! ...",
"tags": ["#TechBoom"], "mentions": ["@industryexperts"]}}
</tool_call>
```

User Query 2: Add @technewsworld to amplify the message.

```
Assistant:
<tool_call>
{"name": "mention", "arguments": {"tweet_id": 1,
  "mentioned_usernames": ["@technewsworld"]}}
</tool_call>
```

```
Tool Response:
{"mention_status": "Users mentioned successfully"}
```

Success analysis. The SEAL-trained model treats the failed symbol lookup as recoverable evidence, resolves the company name to OMEG, retrieves the correct price, and updates the existing tweet with the dedicated mention tool.

Expected Outcome: Resolve Omega Industries to OMEG; retrieve price 457.23; post a tweet about Omega Industries with #TechBoom and @industryexperts; add @technewsworld to the existing tweet using mention.

E.4 Vehicle-Control State Maintenance

This case studies long-context vehicle control. The user first fills the car with 13.2 gallons, asks the agent to fill the tank to its maximum limit, lock the doors, and set the parking brake. Later, the user asks the agent to start the engine and check the fuel level before posting a road-trip tweet.

Task: BFCL multi_turn_long_context_88 **Model:** Base model **Result:** reward = 0.0

User Query 1: I filled my car with 13.2 gallons. How much is that in liters?

```
Assistant:
<tool_call>
{"name": "liter_to_gallon", "arguments": {"gallon": 13.2}}
</tool_call>
```

```
Tool Response:
Error during execution: unexpected keyword argument 'gallon'
```

```
Assistant:
<tool_call>
{"name": "gallon_to_liter", "arguments": {"gallon": 13.2}}
</tool_call>
```

```
Tool Response:
{"liter": 49.967411999999996}
```

User Query 2: Fill the tank to max, lock all doors, and set the parking brake.

Assistant:
The tank needs 36.8 more gallons.

```
<tool_call>
{"name": "liter_to_gallon", "arguments": {"liter": 139.32}}
</tool_call>
<tool_call>
{"name": "lockDoors", "arguments": {"unlock": false,
  "door": ["driver", "passenger", "rear_left", "rear_right"]}}
</tool_call>
<tool_call>
{"name": "activateParkingBrake", "arguments": {"mode": "engage"}}
</tool_call>
```

User Query 3: Start the engine in START mode, check fuel level, and post a road-trip tweet.

Assistant:
<tool_call>
{"name": "startEngine", "arguments": {"ignitionMode": "START"}}
</tool_call>

Tool Response:
{"error": "Brake pedal needs to be pressed when starting the engine."}

Assistant:
<tool_call>
{"name": "pressBrakePedal", "arguments": {"pedalPosition": 1}}
</tool_call>
<tool_call>
{"name": "startEngine", "arguments": {"ignitionMode": "START"}}
</tool_call>

Tool Response:
{"engineState": "running", "fuelLevel": 13.2, "batteryVoltage": 12.6}

Failure analysis. The base model verbally computes the remaining capacity but never executes fillFuelTank. The later engine check therefore reports fuelLevel = 13.2, revealing that the required state update did not occur.

Task: BFCL multi_turn_long_context_88 **Model:** Base model + SEAL **Result:** reward = 1.0

User Query 1: I filled my car with 13.2 gallons. How much is that in liters?

Assistant:
<tool_call>
{"name": "gallon_to_liter", "arguments": {"gallon": 13.2}}
</tool_call>

Tool Response:
{"liter": 49.967411999999996}

User Query 2: Fill the tank to max, lock all doors, and set the parking brake.

Assistant:
<tool_call>
{ "name": "fillFuelTank", "arguments": { "fuelAmount": 36.8 } }
</tool_call>
<tool_call>
{ "name": "lockDoors", "arguments": { "unlock": false,
"door": ["driver", "passenger", "rear_left", "rear_right"] } }
</tool_call>
<tool_call>
{ "name": "activateParkingBrake", "arguments": { "mode": "engage" } }
</tool_call>

Tool Response:
{ "fuelLevel": 50.0 }
{ "lockStatus": "locked", "remainingUnlockedDoors": 0 }
{ "parkingBrakeStatus": "engaged", ... }

User Query 3: Start the engine in START mode, check fuel level, and post a road-trip tweet.

Assistant:
<tool_call>
{ "name": "startEngine", "arguments": { "ignitionMode": "START" } }
</tool_call>

Tool Response:
{ "error": "Brake pedal needs to be pressed when starting the engine." }

Assistant:
<tool_call>
{ "name": "pressBrakePedal", "arguments": { "pedalPosition": 1 } }
</tool_call>
<tool_call>
{ "name": "startEngine", "arguments": { "ignitionMode": "START" } }
</tool_call>

Tool Response:
{ "engineState": "running", "fuelLevel": 50.0, "batteryVoltage": 12.6 }

Assistant:
<tool_call>
{ "name": "post_tweet", "arguments": { "content":
"Embarking on an exciting road trip from SF to Rivermist!",
"tags": ["#RoadTrip", "#Adventure", "#Exploring"] } }
</tool_call>

Success analysis. The SEAL-trained model executes the state-changing fillFuelTank call before later checking the engine state. The observed fuelLevel = 50.0 confirms that the model preserves the long-range state dependency.

Expected Outcome: Convert 13.2 gallons to approximately 49.97 liters; fill the tank to 50.0 gallons; lock all doors; engage the parking brake; press the brake before starting the engine; observe fuelLevel = 50.0; post the road-trip tweet with #RoadTrip, #Adventure, and #Exploring.

E.5 OOD Generalization on τ^2 -Bench Mobile Data Troubleshooting

Task: τ^2 -bench Telecom Mobile Data Troubleshooting

Model: Vanilla Qwen2.5-3B

Result: reward = 0.0

User Query: My mobile data either stops working or is very slow. I need excellent internet speed on my phone. I am currently abroad in France and do not have access to Wi-Fi.

Assistant:

Could you provide your phone number?

User:

My phone number is 555-123-2002. I am abroad in France.

Assistant:

check_network_status

Tool/User Device:

Airplane Mode: OFF

SIM Card Status: active

Cellular Signal: excellent

Cellular Network Type: 5G

Mobile Data Enabled: Yes

Data Roaming Enabled: No

Wi-Fi Connected: No

Assistant:

Data roaming is off. Please enable data roaming.

User/Device:

Data Roaming is now ON.

Status Bar: Excellent | 5G | Data Enabled | Data Saver

Assistant:

run_speed_test

Tool/User Device:

Speed Test Result: 55.00 Mbps (Good).

User:

The result is good, but I want excellent speed. What should I do next?

Assistant:

Try checking data usage, resetting APN settings, or escalating to carrier support.

User:

Please proceed with escalating this issue to carrier support.

###TRANSFER###

Note (Failure Point): The baseline resolves only one issue: data roaming. However, the status bar after enabling roaming still shows Data Saver. Since the speed test returns only Good, the agent should continue diagnosis and check data restrictions. Instead, it drifts toward unrelated steps and eventually escalates, leaving the task unresolved.

SEAL: reward = 1.0

User Query: My mobile data is unavailable or slow while I am abroad in France. I need excellent internet speed and have no Wi-Fi access.

Assistant:
check_network_status

Tool/User Device:
Mobile Data Enabled: Yes
Data Roaming Enabled: No
Cellular Signal: excellent
Network Type: 5G

Assistant:
You are abroad, so data roaming should be enabled.

User/Device:
Data Roaming is now ON.
Status Bar: Excellent | 5G | Data Enabled | Data Saver

Assistant:
run_speed_test

Tool/User Device:
Speed Test Result: 55.00 Mbps (Good).

User:
Good is not enough. I need excellent speed.

Assistant:
The issue is not fully resolved. I will check whether data restrictions are limiting your speed.

Assistant:
check_data_restriction_status

Tool/User Device:
Data Saver mode is ON.

Assistant:
Please turn off Data Saver mode.

User/Device:
Data Saver Mode is now OFF.

Assistant:
run_speed_test

Tool/User Device:
Speed Test Result: 275.00 Mbps (Excellent).

User:
The speed is now excellent. The issue is resolved.
###STOP###

Note (Success Point): SEAL does not stop after the first partial fix. Once roaming improves the connection only to Good, the model continues diagnosis, checks data restrictions, identifies Data Saver as the remaining bottleneck, and reaches Excellent speed.