# EXPERIENCE-HANDOFF REINFORCEMENT LEARNING FOR RECURSIVE SELF-IMPROVEMENT

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

It is common to train Large Language Models (LLMs) for agentic tasks with Reinforcement Learning (RL) from scalar rewards. However, this throws away the fine-grained information about the environment that the agent experienced during the trial (e.g., error logs). To utilize this rich information, recent works wrap the agent in a harness that refines its own inputs through reflection or compaction and hands past experience to future rollouts, but the construction of the handed-off artifacts (e.g., a playbook) is itself not learned. We want the best of both worlds: gradient updates from a scalar reward, while also utilizing environment information to hand off experience. As a concrete testbed, we fix the Agentic Context Engineering (ACE) harness and train one model to act, reflect, and curate task-local playbooks. Our method, ACE-RL, evaluates candidate experience-handoff artifacts by their downstream effect on future actor rollouts, and co-optimizes the role-specific behaviors (action generator, insight reflector, and playbook curator) under a fixed scaffold. Across different agentic benchmarks, ACE-RL shows faster convergence and higher converged performance than existing baselines, including outcome-only GRPO. The results suggest that agent training should optimize not only the actions in the current rollout, but also the model-written experience that future rollouts inherit.

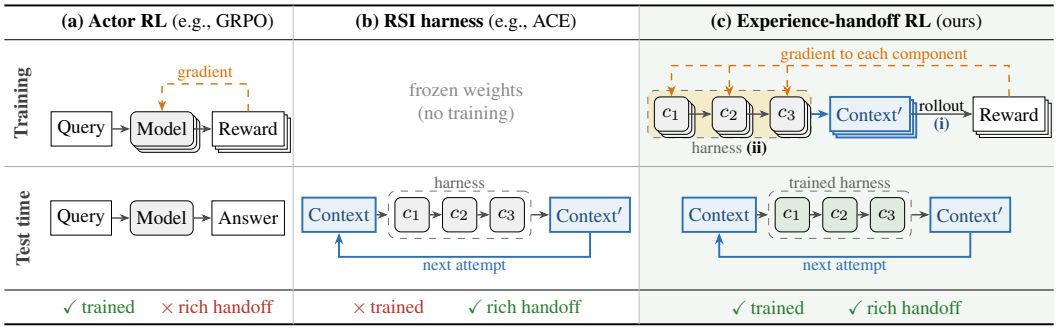



Figure 1: **Experience-Handoff RL.** (a) Outcome-only RL compresses each trial into a scalar reward. (b) A harness runs several calls to the same model ($c_1, c_2, c_3$; in ACE, act, reflect, and curate) and preserves experience as a model-written context, such as a playbook, handed to the next attempt (blue), but the model never learns to write better playbooks. (c) Experience-handoff RL trains the model for the harness it runs in, from task outcomes alone. ACE-RL rewards each handoff artifact for what it does to a future rollout: (i) each new context is scored by a fresh rollout, and (ii) each role is compared within its own group (stacked boxes), which lowers the variance of its credit.

## 1 INTRODUCTION

An agent that learns from its own trials needs *experience handoff*: what one attempt revealed about the environment has to reach the attempts that follow. There are two ways to hand off experience.



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The first is through scalar rewards that compress the trial into a single number, which the learner absorbs into its weights. The second is through rich experience artifacts, such as reflections, notes, or playbooks, that preserve the context of the trial and are passed to future rollouts in the model's input.

Reinforcement learning (RL) is the standard instance of the first way. During training, the agent hands off experience to its future self only through the policy update: each rollout is scored, the score becomes an advantage, and the gradient step shifts the policy toward rewarded behaviour (Williams, 1992; Sutton & Barto, 2018; Schulman et al., 2017). For language models, reinforcement learning with verifiable rewards (RLVR) reliably improves the *actor* across model families and scales (Shao et al., 2024; Guo et al., 2025; Yu et al., 2025b). However, outcome-only RL compresses a whole trial into one number, discarding the error logs, tool outputs, and discovered constraints that made the trial informative. The agent sees this environment feedback while it acts, but it cannot hand any of it off to future rollouts except through an abstract scalar, and RL offers no recipe for the reflection and note-writing behaviours that would carry it forward.

The second way is through *harnesses*: scaffolds that decide when the model acts, what it sees, and what it carries from one attempt to the next. Harnesses keep experience as text, such as verbal reflections (Shinn et al., 2023), skill libraries and reusable workflows (Wang et al., 2023; 2024), cheatsheets and playbooks (Suzgun et al., 2026; Zhang et al., 2025b), evolved prompts (Agrawal et al., 2025), or archives of evaluated programs in test-time discovery loops (Novikov et al., 2025). With a capable frontier model inside, such scaffolds exhibit inference-time *recursive self-improvement* (RSI): a rollout fails, the model reflects on the trace, rewrites its own notes or context, and the next rollout, conditioned on those notes, does better. Throughout this paper, RSI refers only to this loop, in which a model improves its own future rollouts by rewriting the context it will inherit, under fixed weights and a fixed harness.[1] However, the harness only preserves experience; the quality of what it hands off is bounded by the frozen model that writes it. The calls that reflect and edit are never trained for the effect of their text on future rollouts, and methods that optimize the harness itself (Agrawal et al., 2025; Lee et al., 2026) change the prompts and code around the model rather than the model's ability to write useful experience. Small open models, in particular, write artifacts that do not help their next attempt (Figure 2).

Each way therefore has what the other lacks: RL updates weights from a verifiable signal but hands off only a scalar, and harnesses hand off rich experience but never learn to write it. This paper asks:

*Can a model learn, from task outcomes alone, both to act and to write the experience that its future self will inherit?*

We refer to this kind of training as *experience-handoff RL* (EHRL). As a testbed we fix the ACE harness (Zhang et al., 2025b) and train one model for its three roles: the *actor* that solves the task, the *reflector* that extracts insights from a trajectory, and the *curator* that edits a task-local *playbook* for the next attempt. One property makes this training work: **the final playbook is verifiable**. Handing a candidate playbook to a fresh actor rollout and reading the task verifier scores it without a learned reward model, following the success behind RLVR. Our method, ACE-RL, pairs this reward with **a comparison group for every role**, so a reflection is compared with other reflections of the same trajectory instead of being credited for the trajectory it happened to read; a sparse sampling tree supplies these groups without enumerating every downstream branch. On four agentic benchmarks, ACE-RL improves held-out reward over both actor-only RL and end-to-end harness training with a learned critic (Section 5).

In summary, our contributions are:

- We view agent training as experience handoff through two channels, scalar rewards and model-written artifacts, and formulate RL that trains both the actions and the handed-off experience from task outcomes alone (Section 2).
- We identify what makes the handoff trainable with RL, a verifiable reward for the final playbook obtained by executing it, and instantiate it as ACE-RL, which estimates the resulting gradient for the actor, reflector, and curator of a fixed harness with a separate comparison group per role, using only the task verifier (Section 3).

[1] We focus on this form because it is a clean setting for the second kind of experience handoff: the only thing passed from one attempt to the next is an artifact the model wrote. The use of RSI in the field more broadly also covers agents that rewrite their own code or scaffold and models that update their own weights; see Appendix A.

- On four benchmarks we show that actor-only RL does not make the harness useful, show that the verifiable playbook reward is what makes training through the harness work where a learned critic does not, and characterize RSI@$K$ scaling against pass@$K$ (Sections 4–5).

## 2 Experience-Handoff Reinforcement Learning

This section defines the harness and states the learning problem: the model is trained for the outcome of the harness it runs in, not only for the outcome of its actor.

### 2.1 A Fixed Task-Local Harness

We adopt a task-local variant of Agentic Context Engineering (ACE) (Zhang et al., 2025b). One model $\pi_\theta$ serves three roles under fixed role instructions $\iota_{\text{act}}, \iota_{\text{ref}}, \iota_{\text{cur}}$. Let $q$ be a task instance and $P$ its current *playbook*, a short list of natural-language tips that is prepended to the actor's input. One *handoff iteration* consists of

$$\tau \sim \pi_\theta(\cdot \mid \iota_{\text{act}}(q, P)), \qquad \text{(act)} \tag{1}$$

$$z \sim \pi_\theta(\cdot \mid \iota_{\text{ref}},\ \Phi(q, \tau, R_q(\tau))), \qquad \text{(reflect)} \tag{2}$$

$$u \sim \pi_\theta(\cdot \mid \iota_{\text{cur}}(P, z)), \qquad P' = \text{Apply}(P, u). \qquad \text{(curate)} \tag{3}$$

The actor trajectory $\tau$ interleaves generated actions with environment observations, and $R_q(\tau) \in [0, 1]$ is its reward from the benchmark's task verifier. The reflector reads a benchmark adapter's formatting $\Phi$ of the trajectory and of whatever the environment revealed (test failures, tool errors, termination reason) and writes an *insight* $z$. The curator reads the incumbent playbook and the insight and emits structured edit operations $u$ (add, update, or remove a tip); Apply executes valid operations to produce a candidate playbook $P'$. The playbook is local to $q$: a fresh environment episode is started for every actor call, and the only state carried across attempts on the same task is $P$. Prompts for all roles are listed in Appendix C.

**Inference-time recursive self-improvement.** At test time the harness runs $K$ handoff iterations per task with frozen weights, replacing $P$ by $P'$ whenever the candidate scores at least as well as the rollout from the current playbook. We call this protocol *RSI@$K$*: a task counts as solved if any of the $K$ iterations solves it. Section 4.2 specifies the protocol and its rollout accounting.

### 2.2 The Learning Problem

Outcome-only agentic RL optimizes the actor alone, maximizing $J_{\text{act}}(\theta) = \mathbb{E}_q \, \mathbb{E}_{\tau \sim \pi_\theta(\cdot \mid \iota_{\text{act}}(q, P_0))}[R_q(\tau)]$, where $P_0$ is the empty (or initial) playbook. A model trained this way is then deployed inside a harness, on the expectation that a better actor also makes a better model inside the harness. When the harness does not help, the usual protocol is to change the harness rather than the model: prompts, control flow, and memory formats are engineered by hand or searched over around a fixed model (Khattab et al., 2023; Agrawal et al., 2025; Lee et al., 2026), and the result may or may not suit the model it wraps. In neither case is the model trained for what its reflections and edits do to later attempts.

We instead train the model for the harness it runs in. Let $\omega \sim H[\pi_\theta](q)$ denote a complete execution of harness $H$ on task $q$ and $o(\omega)$ its final task outcome. We define the EHRL objective as

$$\max_\theta \quad J_H(\theta) := \mathbb{E}_q \, \mathbb{E}_{\omega \sim H[\pi_\theta](q)} \big[ R_q(o(\omega)) \big]. \tag{4}$$

The harness fixes the prompts, tools, and control flow; the model is trained for the outcome of their combined execution, so every call it makes, not only the actor, is an object of training. For the ACE handoff in Eqs. (1)–(3), we can specify this objective to the actor's outcome *after* the playbook update:

$$\max_\theta \quad J_{\text{ACE}}(\theta) := \mathbb{E}_{(q,P) \sim \mu} \, \mathbb{E}_{\tau, z, u, \tilde{\tau} \sim \pi_\theta} \big[ R_q(\tilde{\tau}) \big]. \tag{5}$$

Here $\tilde{\tau} \sim \pi_\theta(\cdot \mid \iota_{\text{act}}(q, P'))$ is a fresh attempt using the proposed playbook, and $\mu$ is the distribution of tasks and incumbent playbooks at training encounters, held fixed for a local update. The playbook here is generally an inherited $P_i$, with $P_0$ used at initialization and scheduled resets. Equation (5)

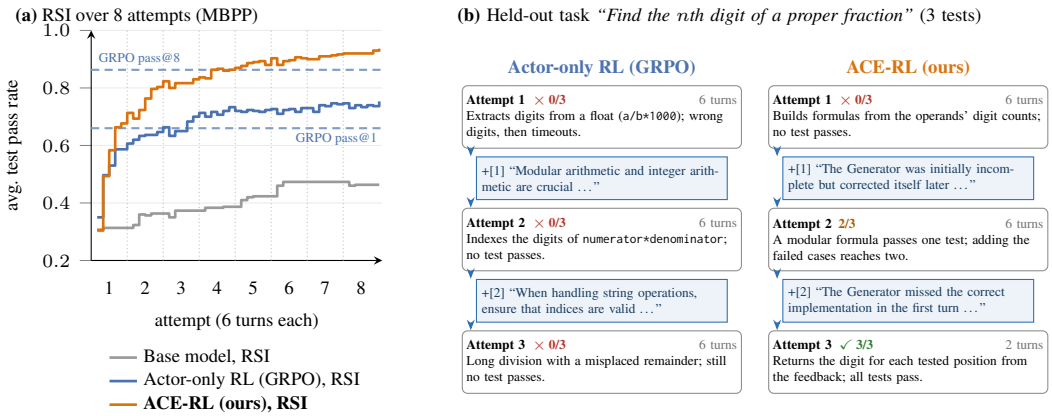



Figure 2: Rollout of Qwen3-0.6B (or checkpoints trained from it) on MBPP, with eight consecutive attempts per held-out task in the ACE harness. GRPO and ACE-RL checkpoints are taken at the same training step. **(a)** Average test pass rate by turn. Dashed lines: the same GRPO checkpoint without the harness, with one sample (pass@1) and the best of eight independent samples (pass@8). **(b)** One held-out task through the first three attempts of each checkpoint, with the playbook tip its curator adds after each attempt.

rewards only the later attempt: the earlier trajectory supplies evidence from which the model writes the handoff. A low later reward can reflect poor evidence, a missed insight, an ineffective edit, or failure to use the updated playbook.

Figure 2 gives a preview of how Section 5 proceeds. With frozen weights, the model trained only for $J_{\text{act}}$ starts above the base model but soon stops improving across attempts, and ends below what the trained checkpoint reaches by sampling independently without the harness: what its reflector and curator hand forward is worth less than simply trying again. The model trained for Eq. (5) keeps improving with every attempt, and on the task in panel (b) its own playbook edits take it from failing every test to passing all of them. Improving from one's own attempts is a behavior of the model in every role, and training the actor alone leaves it to chance. Training the other roles needs a signal for text whose value is only visible one rollout later. Section 3 constructs that signal as a component-wise GRPO surrogate for Eq. (5); Appendix D.1 derives its delayed credit and the approximations used to make it practical.

# 3 ACE-RL

ACE-RL trains all three roles with a component-wise GRPO surrogate for Eq. (5). A full sampling tree would evaluate downstream actor outcomes and propagate their rewards to the trajectories, reflections, and edits that produced them. We make this construction practical by treating the sampled seed trajectory as fixed evidence for handoff learning and training the actor on inherited playbooks. Section 3.1 defines the verifiable reward that makes the handoff trainable, and Section 3.2 the sampling tree that gives each role its own comparison group. Appendix D.1 gives the full-tree derivation.

## 3.1 Verifiable Rewards for Handoff Artifacts

Let $P$ be the current playbook of task $q$ and $\pi_{\theta_{\text{old}}}$ the collection policy shared by all roles. A curator output $u$ yields a candidate $P' = \text{Apply}(P, u)$. We define its *handoff value* by executing it:

$$\tilde{\tau} \sim \pi_{\theta_{\text{old}}}(\cdot \mid \iota_{\text{act}}(q, P')), \qquad \widehat{V}(P') = R_q(\tilde{\tau}). \tag{6}$$

This is a one-rollout estimate of the conditional downstream value in Eq. (5), under the collection policy. The task verifier supplies the reward, so no learned reward model is needed. A curator output receives the value of its playbook; a reflection receives the mean value (single reward if there's only one) of the admissible playbooks curated from it; the actor receives its task reward under the

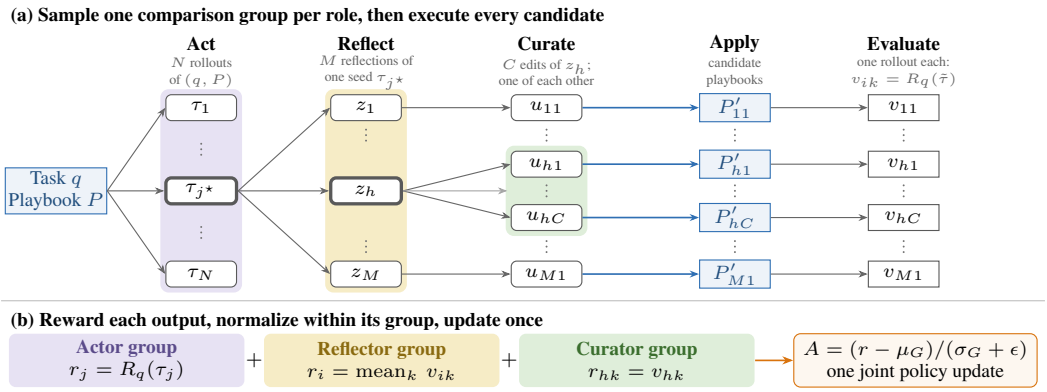



Figure 3: **Sampling tree of ACE-RL** with group sizes $N$, $M$, $C$ for actor, reflector, and curator. (a) Each shaded box is one comparison group whose members share an input: $N$ rollouts of $(q, P)$, $M$ reflections of a uniformly chosen seed $\tau_{j^\star}$, and $C$ edits of a uniformly chosen reflection $z_h$; every other reflection receives one edit. Each edit is applied to a copy of $P$ and executed once. (b) Candidate values reward the edit that produced them and, averaged, its reflection; advantages are normalized within each group and all roles share one update.

incumbent playbook:

$$r^{\mathrm{cur}}(u) = \widehat{V}(\mathrm{Apply}(P, u)), \qquad r^{\mathrm{ref}}(z) = \frac{1}{|\mathcal{C}(z)|} \sum_{u \in \mathcal{C}(z)} \widehat{V}(\mathrm{Apply}(P, u)), \qquad r^{\mathrm{act}}(\tau) = R_q(\tau), \tag{7}$$

where $\mathcal{C}(z)$ is the set of admissible sampled edits derived from $z$, with $r^{\mathrm{ref}}(z) = 0$ when this set is empty; inadmissible edits receive reward 0 (See Appendix D for detail).

**Learning to read the inherited playbook.** The actor is trained under its incumbent playbook, which carries experience retained from earlier encounters, so it also learns to read a retained handoff; Appendix D.1 states the approximation this makes.

### 3.2 A Comparison Group for Every Role

For each role, group-relative advantages need several outputs sampled from the same input: a reflection should be compared with other reflections of the same trajectory, not with reflections of trajectories that happened to be better or worse. Figure 3 shows how ACE-RL obtains such groups for a task $q$ with playbook $P$.

1. **Act.** Sample $N$ actor rollouts $\tau_1, \ldots, \tau_N$ from $(q, P)$. They form the actor group and keep their own rewards.
2. **Reflect.** Choose one seed $j^\star \sim \mathrm{Unif}[N]$ and sample $M$ reflections $z_1, \ldots, z_M$ of $\tau_{j^\star}$. They share an input and form the reflector group. Because the seed is chosen uniformly, successful and failed trajectories are reflected upon in proportion to how often the actor produces them.
3. **Curate.** Choose one usable reflection uniformly and sample $C$ curator edits of $(P, z_h)$; these form the curator group. Every other usable reflection receives a single edit so that its downstream value can be estimated in Eq. (7).
4. **Evaluate.** Apply every admissible edit to a copy of $P$ and score each resulting candidate with one fresh actor rollout (Eq. 6).

**Per-role normalization.** Advantages are the group-normalized rewards of GRPO (Shao et al., 2024), but each role is normalized only within its own group[2]: actions against the other $N - 1$ actions

[2]We also investigate an alternative without per-role groups, which compares reflections and edits across the chains of different actor rollouts. It estimates the same update, but each reflection's advantage then also carries the quality of the trajectory it read (Appendix B).

Table 1: **Compared methods** and their evaluation mode. Section 5.1 compares the bottom row, Section 5.2 the right column.

| | Learned critic | Verifiable reward |
|---|---|---|
| Actor only | PPO (Schulman et al., 2017) (vanilla) | GRPO (Shao et al., 2024) (vanilla) |
| Through the harness | Delayed-joint PPO (Li et al., 2026) (RSI@3) | **ACE-RL (ours)** (RSI@3) |

of $(q, P)$, reflections against the other reflections of $\tau_{j^\star}$, and edits against the other edits of $(P, z_h)$. A reflection is therefore reinforced when its downstream value exceeds that of the other reflections of the *same* trajectory, so a failed seed can still yield a positively rewarded reflection, and an edit is reinforced relative to other edits of the *same* insight. Groups whose rewards are constant carry no comparison and are dropped; singleton curator branches supply reflector rewards but no curator comparison. The outputs of all three roles are then optimized in one joint update of the shared model (Appendix D).

Appendix D.1 derives this scheme as a reduction of the full tree: one evaluation per candidate preserves its expected verifier value while increasing estimation noise, and the finite-sample normalization need not preserve the full tree's expected gradient.

**Playbook selection and reset.** Which playbook a task carries into its next encounter is decided separately from training credit: the highest-valued candidate is kept if it scores at least the seed's reward, and otherwise $P$ is kept. Each task's playbook is reset to $P_0$ every $F$ encounters so that edits accepted early cannot persist indefinitely. Appendix D.2 gives the rule, and Algorithm 1 summarizes one training step.

## 4 Experimental Setup

We train Qwen3-0.6B in non-thinking mode (Qwen Team, 2025) on Mastermind and MBPP (Austin et al., 2021), and Qwen3-4B-Instruct-2507 on AppWorld (Trivedi et al., 2024) and MiniSWEBench (Yang et al., 2024; Jimenez et al., 2023); Appendix E describes the tasks, splits, and reward design. Every method trains on 8 tasks per step with $N = 8$ actor rollouts per task. ACE-RL additionally samples $M = 4$ reflections of one seed trajectory and $C = 4$ edits of one reflection, plus one edit of each other reflection, scores each candidate playbook with one fresh rollout, and resets each task's playbook every $F = 3$ encounters. Delayed-joint PPO samples one reflection and one edit per actor rollout. All methods share the harness code, prompts, verifiers, and optimizer settings; Appendix D lists the remaining training details.

The experiments make two comparisons, both against ACE-RL (Table 1). The first isolates how the handoff roles should be credited once they are trained through the harness: with a learned critic (delayed-joint PPO) or with the verifiable playbook reward (ACE-RL) (Section 5.1). The second tests whether the harness has to be trained at all, comparing ACE-RL with actor-only RL, PPO and GRPO (Section 5.2). Section 5.3 then compares how the trained handoff and independent sampling convert additional attempts into reward.

### 4.1 Compared Methods

**Actor-only RL.** PPO (Schulman et al., 2017; 2015) and GRPO (Shao et al., 2024) train the actor on plain rollouts and never see the harness, which is how agents are usually post-trained before being deployed inside a scaffold. GRPO computes advantages relative to other rollouts of the same task and is the standard RLVR recipe. We include PPO because recent long-horizon agentic RL has returned to critic-based optimization: SAO trains GLM-5.2's agentic RL pipeline with single-rollout and PPO-style value model, arguing that group-wise sampling fits long, asynchronous agent rollouts poorly (Hou et al., 2026), and CompactionRL credits execution and summary segments with a learned critic (Li et al., 2026). PPO is therefore a practical alternative to GRPO in exactly the regime we study, not only a historical baseline.

Table 2: **Test reward at the final checkpoint** (Mastermind and MBPP step 290, AppWorld 400, MiniSWEBench 60; PPO 299, 299, 115, 30; delayed-joint PPO 30, 190, 80, 50), mean ± 95% CI over held-out tasks. Each method is evaluated in the mode it was trained for; GRPO is also evaluated in the harness.

| | | Qwen3-0.6B | | Qwen3-4B-Instruct | |
|---|---|---|---|---|---|
| Method | Eval | Mastermind | MBPP | AppWorld | MiniSWEBench |
| Base model | vanilla | 0.306±0.027 | 0.240±0.062 | 0.173±0.074 | 0.048±0.027 |
| Base model | RSI@3 | 0.114±0.040 | 0.430±0.087 | 0.211±0.107 | 0.110±0.062 |
| PPO (Schulman et al., 2017) | vanilla | 0.664±0.013 | 0.868±0.041 | 0.174±0.078 | 0.007±0.016 |
| GRPO (Shao et al., 2024) | vanilla | 0.643±0.015 | 0.668±0.067 | 0.529±0.105 | 0.120±0.052 |
| GRPO (Shao et al., 2024) | RSI@3 | 0.599±0.050 | 0.847±0.061 | 0.561±0.130 | 0.180±0.076 |
| Delayed-joint PPO (Li et al., 2026) | RSI@3 | 0.004±0.004 | 0.000±0.000 | 0.000±0.000 | 0.051±0.043 |
| **ACE-RL** (ours) | RSI@3 | **0.736**±**0.025** | **0.973**±**0.026** | **0.649**±**0.125** | **0.227**±**0.083** |

**Training through the harness.** *Delayed-joint PPO* is our adaptation of CompactionRL (Li et al., 2026) to the ACE harness. It branches only at the actor: each of the $N$ actor rollouts receives one reflection, one edit, and one evaluation of the resulting playbook, and all three roles are trained on these chains with PPO. A critic initialized from the policy estimates the return of actor, reflector, and curator segments, where reflector and curator segments are credited with their candidate's verifier reward, and the policy is trained on the rollouts of the accepted playbook (Appendix D). This is the credit structure of end-to-end methods that train a model whose own summaries become its future context (Li et al., 2026; Chan et al., 2026). It does not reproduce CompactionRL's context replacement or cross-segment advantage correction, which have no counterpart in ACE. ACE-RL trains the same three roles with the verifiable playbook reward and the per-role groups of Section 3.

### 4.2 Evaluation Protocols

Every checkpoint is evaluated with frozen weights on held-out tasks at temperature $0.7$, in one of two modes. *Vanilla* evaluation draws 1 independent actor episode per task without any harness, and pass@$K$ is the expected best reward of $K$ of them. *Harnessed* evaluation runs a number of sequential visits per task: each visit is one actor episode under the current playbook, followed by one reflection and one curator edit that updates the playbook for the next visit, with no separate candidate-scoring rollout. RSI@$K$ is the best reward among the first $K$ visits. The test reward of a harness method is $K = 3$. Each method is evaluated in the mode it was trained for (Table 1), with additional report on GRPO in the harness, which is how actor-only models are deployed under harness.

### 4.3 Metrics and Reporting

We report logged training rewards and completed evaluations on a 10-step schedule (5 on MiniSWEBench), plus final checkpoints when available, with the base model at step 0. The reporting budgets are 300 steps on Mastermind and MBPP, 400 on AppWorld, and 60 on MiniSWEBench; poorly performing PPO-based runs may end earlier. Results are from one training run per method, with uncertainty over held-out tasks (Appendix F.1).

## 5 Results

Table 2 reports the test reward of every method at its final checkpoint, and Figure 4 its test reward over training (training reward in Appendix F.2). Results are from one training run per method, with 95% confidence intervals over held-out tasks (Section 4.3).

### 5.1 Crediting the Handoff: Verifiable Reward versus a Learned Critic

Delayed-joint PPO and ACE-RL both train the actor, reflector, and curator through the harness; they differ in how the handoff roles are credited (Table 1, bottom row). Delayed-joint PPO performs poorly

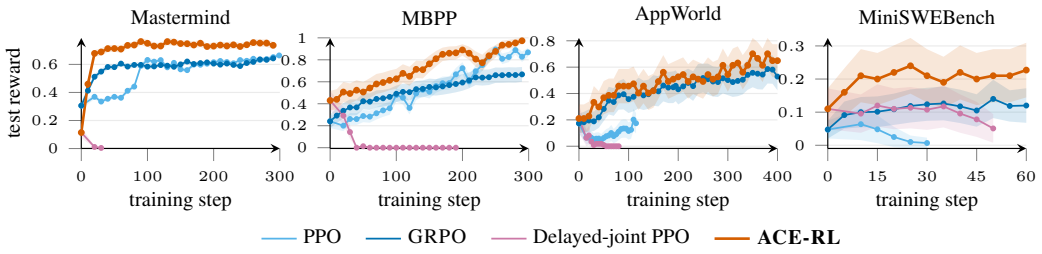
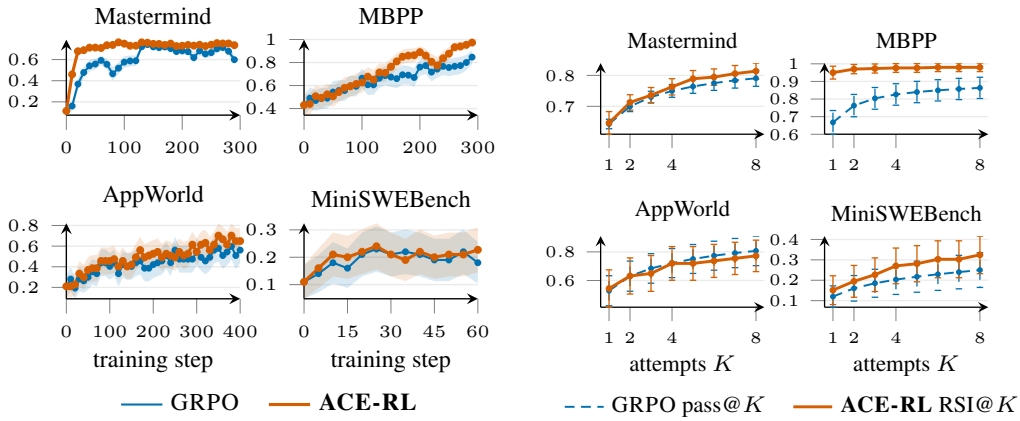



Figure 4: **Test reward on held-out tasks**, evaluated every 10 steps (every 5 on MiniSWEBench) in each method's own mode: vanilla pass@1 for PPO and GRPO, RSI@3 for delayed-joint PPO and ACE-RL. Bands: 95% CI over tasks.




(a) Test reward in the harness (RSI@3) over training. (b) Best reward after $K$ attempts at the final checkpoint.

Figure 5: **GRPO and ACE-RL in the harness. (a)** Both checkpoints are evaluated with the ACE harness every 10 steps (every 5 on MiniSWEBench). **(b)** Dashed: $K$ independent samples of the GRPO checkpoint without the harness (pass@$K$). Solid: $K$ sequential visits of the ACE-RL checkpoint in the harness (RSI@$K$). Both use $K$ actor episodes, at the final checkpoint. Error bars/bands: 95% CI over tasks.

in these runs. On MBPP its smoothed training reward falls below 0.05 by step 32 and stays there; its test reward is 0.00 from step 60 to its last checkpoint at step 190, below the base model's 0.43, while ACE-RL reaches 0.71–0.81 at steps 140–160. On Mastermind, AppWorld, and MiniSWEBench its mean training rewards are 0.004, 0.03, and 0.12 through steps 36, 80, and 50, against 0.49, 0.40, and 0.28 for ACE-RL on the same steps (Figure 9). Its MiniSWEBench test reward at step 20 is 0.11, equal to the base model. The critic's explained variance nevertheless ends near one on all four benchmarks (Appendix F.5). ACE-RL instead executes candidate playbooks and compares siblings written from shared evidence. The results favor this implementation, but do not isolate the cause of the critic-based baseline's failure.

### 5.2 Actor-Only RL Does Not Train the Harness

The common recipe post-trains the actor with outcome-only RL and deploys it inside a harness, expecting a better actor to make a better harnessed agent. Under their own evaluation, PPO and GRPO reach test rewards of 0.66 and 0.64 on Mastermind and 0.87 and 0.67 on MBPP, and GRPO reaches 0.53 on AppWorld and 0.12 on MiniSWEBench (Table 2). ACE-RL is higher on all four benchmarks, at 0.74, 0.97, 0.65, and 0.23.

Deploying the actor-only checkpoint in the harness does not close the gap (Figure 5a). ACE-RL's RSI@3 exceeds GRPO's at 29 of 29 evaluated checkpoints on Mastermind, 24 of 29 on MBPP, and

36 of 40 on AppWorld. GRPO's RSI@3, the best of three harnessed attempts, falls below its own vanilla pass@1 at 12 of these 29 Mastermind checkpoints and 19 of 40 AppWorld checkpoints. At Mastermind step 290 it is $0.60$ against $0.64$; the drop begins at the first visit, before any handoff ($0.35$ against $0.64$), consistent with the change from the plain training prompt to the harness prompt. MiniSWEBench is the exception: at step 60, GRPO and ACE-RL are within noise in the harness ($0.18$ and $0.23$), both above vanilla GRPO ($0.12$).

### 5.3 Scaling Inference: Pass@$K$ versus RSI@$K$

Independent sampling scales by drawing more attempts: pass@$K$ is the best of $K$ parallel rollouts (Brown et al., 2024). A harness scales along a different axis: RSI@$K$ is the best of $K$ sequential visits, each conditioned on experience written after the previous one. Figure 5b compares the two at the same number of actor episodes. On MBPP a single ACE-RL visit already scores higher than eight independent GRPO samples ($0.95$ against $0.86$). This gain precedes any handoff, so it reflects an actor trained to work inside the harness; with the first visit near the ceiling, further visits have little room to add ($0.98$ at $K = 8$), so the flat curve is not evidence against sequential scaling. On Mastermind the two curves stay within $0.03$ at every $K$. On AppWorld, ACE-RL is lower at $K = 3$–$8$, ending at $0.77$ against $0.81$. On MiniSWEBench ACE-RL is ahead by $0.03$–$0.07$ at every $K$, within the task CI. The inspected playbooks offer context (Appendix G.1): AppWorld and MiniSWEBench tips are task-specific and can precede repeated successes, whereas the two 0.6B curators' edits carry little task information. Sequential handoffs therefore match or exceed independent attempts on three of four benchmarks; AppWorld, where later visits add less than fresh samples, is the exception.

## 6 Conclusion

Harnesses are becoming the operating systems of language agents, and outcome-only RL trains only the process that acts inside them. We formulated Experience-Handoff RL, the problem of training a model to write the experience its future rollouts inherit especially under ACE harness. We further showed that ACE-RL makes the training practical with the ingredient behind actor RLVR, a verifiable reward, applied to the final playbook by executing it, and estimates the resulting gradient with group-relative comparisons for every role in the harness, obtained from a sparse sampling tree. Test rewards favor it over actor-only RL on all four benchmarks. At the same number of actor episodes, sequential self-improvement matches or exceeds independent sampling on three of four benchmarks, and on MBPP a single harnessed attempt already exceeds eight independent samples. The broader lesson is that agent training should optimize not only the actions of the current rollout but also the model-written experience that future rollouts inherit. Appendix A discusses related work and Appendix H the limitations of this study.

### AI use statement

We used generative AI tools to assist with writing and debugging the experiment and plotting code, and to help shorten and edit the paper text to fit the page limit. All research ideas, experimental design, and claims are the authors' own. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

### Ethics statement

This work trains open-weight language models on publicly available benchmarks (Mastermind, MBPP, AppWorld, and MiniSWEBench) in sandboxed environments; it involves no human subjects, personal data, or crowdsourced annotation. Training models to write the experience that their future attempts inherit could make autonomous agents improve with less human oversight. In our setting this is bounded: playbooks are task-local, are scored only by the task verifier, and are reset periodically, and every experiment runs in a sandbox without access to external systems. Deploying such training in open-ended software or interface agents would call for additional scoping and monitoring, which we discuss in Appendix H.

### Reproducibility statement

The full training pipeline and hyperparameters are given in Appendix D, benchmark settings, splits, and reward design in Appendix E, statistical protocols in Appendix F.1, and all prompts in Appendix C. All experiments use publicly available models (Qwen3-0.6B and Qwen3-4B-Instruct-2507) and benchmarks. We will release the training and evaluation code upon publication.

## References

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, 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, 2025. arXiv preprint arXiv:2507.19457.

Huan ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, Hongru Wang, Han Xiao, Yuhang Zhou, Shaokun Zhang, Jiayi Zhang, Jinyu Xiang, Yixiong Fang, Qiwen Zhao, Dongrui Liu, Qihan Ren, Cheng Qian, Zhenhailong Wang, Minda Hu, Huazheng Wang, Qingyun Wu, Heng Ji, and Mengdi Wang. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence, 2025. arXiv preprint arXiv:2507.21046.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732. arXiv preprint arXiv:2108.07732.

Shelly Bensal, Umar Jamil, Christopher Bryant, Melisa Russak, Kiran Kamble, Dmytro Mozolevskyi, Muayad Ali, and Waseem AlShikh. Reflect, retry, reward: Self-improving llms via reinforcement learning. *arXiv preprint arXiv:2505.24726*, 2025.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. arXiv preprint arXiv:2407.21787.

Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, Yong Mao, Ke Li, and Xing Sun. Training-free group relative policy optimization, 2025. URL https://arxiv.org/abs/2510.08191. arXiv preprint arXiv:2510.08191.

Aaron Chan, Ahmed Shalaby, Alexander Wettig, et al. Composer 2 technical report, 2026. URL https://arxiv.org/abs/2603.24477. arXiv preprint arXiv:2603.24477.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for LLM agent training, 2025. URL https://arxiv.org/abs/2505.10978.

GLM-4.5 Team. GLM-4.5: Agentic, reasoning, and coding (ARC) foundation models, 2025. arXiv preprint arXiv:2508.06471.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Zhenyu Hou, Yujiang Li, Jie Tang, and Yuxiao Dong. Single-rollout asynchronous optimization for agentic reinforcement learning, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2023. URL https://arxiv.org/abs/2310.06770. arXiv preprint arXiv:2310.06770.

Andrej Karpathy. autoresearch. https://github.com/karpathy/autoresearch, 2026. GitHub repository.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Kimi Team. Kimi K2: Open agentic intelligence, 2025. arXiv preprint arXiv:2507.20534.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052. arXiv preprint arXiv:2603.28052.

Yujiang Li, Zhenyu Hou, Yi Jing, Jie Tang, and Yuxiao Dong. CompactionRL: Reinforcement learning with context compaction for long-horizon agents, 2026. URL https://arxiv.org/abs/2607.05378.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-Zero-like training: A critical perspective, 2025. arXiv preprint arXiv:2503.20783.

Miao Lu, Weiwei Sun, Weihua Du, Zhan Ling, Xuesong Yao, Kang Liu, and Jiecao Chen. Scaling LLM multi-turn RL with end-to-end summarization-based context management, 2025. URL https://arxiv.org/abs/2510.06727.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131. arXiv preprint arXiv:2506.13131.

Qwen Team. Qwen3 technical report, 2025. arXiv preprint arXiv:2505.09388.

Jürgen Schmidhuber. Gödel machines: Self-referential universal problem solvers making provably optimal self-improvements, 2003. arXiv preprint cs/0309048.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015. URL https://arxiv.org/abs/1506.02438.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

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

Taiwei Shi, Sihao Chen, Bowen Jiang, Linxin Song, Longqi Yang, and Jieyu Zhao. Experiential reinforcement learning, 2026. URL https://arxiv.org/abs/2602.13949. arXiv preprint arXiv:2602.13949.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, 2026.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents, 2024. URL https://arxiv.org/abs/2407.18901. arXiv preprint arXiv:2407.18901.

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, 2023. URL https://arxiv.org/abs/2305.16291.

Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. Openclaw-rl: Train any agent simply by talking, 2026a. URL https://arxiv.org/abs/2603.10165. arXiv preprint arXiv:2603.10165.

Yinjie Wang, Tianbao Xie, Ke Shen, Mengdi Wang, and Ling Yang. Rlanything: Forge environment, policy, and reward model in completely dynamic rl system, 2026b. URL https://arxiv.org/abs/2602.02488. arXiv preprint arXiv:2602.02488.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning, 2025. arXiv preprint arXiv:2504.20073.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory, 2024. arXiv preprint arXiv:2409.07429.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4):229–256, 1992.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z. Pan, Hinrich Schütze, Volker Tresp, and Yunpu Ma. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning, 2025. URL https://arxiv.org/abs/2508.19828.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024. arXiv preprint arXiv:2405.15793.

Sherry Yang, Joy He-Yueya, and Percy Liang. Reinforcement learning for machine learning engineering agents, 2025. URL https://arxiv.org/abs/2509.01684. arXiv preprint arXiv:2509.01684.

Shicheng Ye and Chao Yu. Joint learning of experiential rules and policies for large language model agents, 2026. URL https://arxiv.org/abs/2606.27136.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursive self-improvement, 2024. arXiv preprint arXiv:2410.04444.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. MemAgent: Reshaping long-context LLM with multi-conv RL-based memory agent, 2025a. URL https://arxiv.org/abs/2507.02259.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale, 2025b. arXiv preprint arXiv:2503.14476.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models, 2024. arXiv preprint arXiv:2401.10020.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time, 2026. URL https://arxiv.org/abs/2601.16175. arXiv preprint arXiv:2601.16175.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation, 2023. arXiv preprint arXiv:2310.02304.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2025a. URL https://arxiv.org/abs/2505.22954. arXiv preprint arXiv:2505.22954.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models, 2025b. URL https://arxiv.org/abs/2510.04618. arXiv preprint arXiv:2510.04618.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024. arXiv preprint arXiv:2308.10144.

Yujie Zhao, Lanxiang Hu, Yang Wang, Minmin Hou, Hao Zhang, Ke Ding, and Jishen Zhao. Stronger-MAS: Multi-agent reinforcement learning for collaborative LLMs, 2025. URL https://arxiv.org/abs/2510.11062.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. ArCHer: Training language model agents via hierarchical multi-turn RL, 2024. arXiv preprint arXiv:2402.19446.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. MEM1: Learning to synergize memory and reasoning for efficient long-horizon agents, 2025. arXiv preprint arXiv:2506.15841.

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

## A Related Work

**Harnesses and inference-time self-improvement.** Reflexion, Voyager, ExpeL, Dynamic Cheatsheet, Agent Workflow Memory, and ACE let a frozen model improve later attempts by writing reflections, skills, or playbooks that its future calls consume (Shinn et al., 2023; Wang et al., 2023; Zhao et al., 2024; Suzgun et al., 2026; Wang et al., 2024; Zhang et al., 2025b). This is the form of recursive self-improvement (RSI) we study. RSI more broadly also covers agents that rewrite their own code or scaffold (Schmidhuber, 2003; Zelikman et al., 2023; Yin et al., 2024; Zhang et al., 2025a), agents that iterate on programs or research code against an automated evaluator (Novikov et al., 2025; Karpathy, 2026), and models that update their own weights from self-generated data or test-time RL (Yuan et al., 2024; Zweiger et al., 2025; Yuksekgonul et al., 2026); see ang Gao et al. (2025) for a survey. We restrict RSI to rewriting context under fixed weights because that loop is an experience handoff between attempts. These systems assume a model that already writes useful experience; Figure 2 shows that small open models do not. A complementary line optimizes the harness around a fixed model, by compiling prompts, evolving them reflectively, or searching over harness programs (Khattab et al., 2023; Agrawal et al., 2025; Lee et al., 2026; Lin et al., 2026). ACE-RL fixes the harness and trains the model.

**Agentic RL with verifiable rewards.** GRPO and its refinements train actors from group-relative advantages on verifiable outcomes (Shao et al., 2024; Guo et al., 2025; Yu et al., 2025b; Liu et al., 2025), and multi-turn agentic RL extends this to tool use and long horizons (Wang et al., 2025; Zhou et al., 2024; Yang et al., 2025; Wang et al., 2026b). Recent open agentic models report large-scale RL pipelines with learned or rubric-based rewards for intermediate agent behaviour (GLM-4.5 Team, 2025; Kimi Team, 2025). ACE-RL keeps the verifiable-reward ingredient and extends grouping from the actor to every model call in the harness, using downstream execution rather than a learned reward model to score intermediate text.

**Training the model to a fixed harness.** OpenClaw-RL trains a model on rollouts collected inside a production harness (Wang et al., 2026a). Composer 2, SUPO, CompactionRL, MEM1, and MemAgent train a policy whose own summaries or compactions become its future context, propagating the final reward through execution and summary segments with sequence-level advantages or a critic (Chan et al., 2026; Lu et al., 2025; Li et al., 2026; Zhou et al., 2025; Yu et al., 2025a). OpenClaw-RL has no model-written handoff to credit, and the compaction methods generate summaries sequentially within each rollout and credit them through the rollout's outcome or a learned critic, without branching into alternative summaries from the same intermediate context. Our delayed-joint PPO baseline follows this structure (Section 4.1); Appendix B shows that, with an executed playbook as the reward, the chain-wise structure estimates the same gradient as ACE-RL, so same-input groups change the variance of the credit rather than its target.

**Learning reflections and memories from downstream outcomes.** Reflect–Retry–Reward rewards a reflection by the success of the retry it guides (Bensal et al., 2025); ERL adds internalization so that the guided behaviour is reproduced without the reflection (Shi et al., 2026); Memory-R1 trains memory operations by downstream answer correctness, freezing the answer agent during memory-manager training and training it separately (Yan et al., 2025); JERP learns experiential rules alongside an RL-trained actor with a frozen editor (Ye & Yu, 2026). Training-Free GRPO aggregates insights from several rollouts into a shared guidebook without weight updates (Cai et al., 2025). These are the closest precedents for verifiable credit on intermediate text. ACE-RL differs in training one policy for a two-stage handoff (reflect, then curate) with a persistent, editable artifact, and in constructing a comparison group for each stage.

**Finer-grained grouping.** GiGPO groups actions at repeated environment states and AT-GRPO forms agent- and turn-wise groups under shared inputs (Feng et al., 2025; Zhao et al., 2025). ACE-RL applies the same principle to the dependency tree of a harness, and its sparse tree is a specific way to form such groups in a tree whose leaves must be evaluated by rollouts.

# B CHAIN-WISE SAMPLING: DELAYED-JOINT ACE-RL

The sampling tree of Section 3.2 is one way to estimate the gradient of Eq. (5) from executed playbooks. This appendix analyzes and evaluates the simplest alternative, which we call *delayed-joint* ACE-RL: it keeps the verifiable playbook reward of Section 3.1 but forms one chain per actor rollout instead of one comparison group per role. Comparing it with ACE-RL isolates the effect of per-role groups with the reward held fixed.

## B.1 DESIGN

Delayed-joint ACE-RL samples $N$ actor rollouts of $(q, P)$ as ACE-RL does. Each rollout $\tau_i$ then receives one reflection $z_i$, one curator edit $u_i$, and, if the edit is admissible, one fresh actor rollout that scores the candidate $P'_i = \mathrm{Apply}(P, u_i)$. The reflection and the edit of chain $i$ both receive that candidate's value $Y_i$, or 0 if the edit is inadmissible, and the actor keeps its own reward. The $N$ reflections of a task form one group and the $N$ edits another, so each reflection is compared with reflections of *other* trajectories. Playbook selection, resets, and the joint update are those of Section 3. Table 3 contrasts the two designs.

Table 3: **Sampling per task and training step** with $N = 8$ and $(M, C) = (4, 4)$.

| | ACE-RL | Delayed-joint ACE-RL |
|---|---|---|
| Actor outputs | 8 rollouts of $(q, P)$ | 8 rollouts of $(q, P)$ |
| Reflector outputs | 4 reflections of one seed | 1 reflection of each rollout (8) |
| Curator outputs | 4 edits of one reflection, 1 of each other (7) | 1 edit of each reflection (8) |
| Candidate evaluations | up to 7 | up to 8 |
| Reflector group | reflections of the same seed | reflections of the 8 rollouts |
| Curator group | edits of the same reflection | edits of the 8 reflections |
| Reflector reward | mean value of its admissible edits | value of its single candidate |
| Actor credit | own reward, seed rollouts | own reward, seed rollouts |

## B.2 RELATION TO THE EHRL GRADIENT

Delayed-joint ACE-RL is the full-tree estimator of Eq. (13) with $M = C = L = 1$: each initial trajectory has one reflection, one edit, and one evaluation, so $Y_i$ is an unbiased estimate of $Q_{\mathrm{r}}$ and $Q_{\mathrm{c}}$ on its chain. Chains of the same task are conditionally independent given $s = (q, P)$. A leave-one-out mean $b_{-i} = \frac{1}{N-1} \sum_{j \neq i} Y_j$ therefore depends only on outputs that are independent of $z_i$ and $u_i$ given $s$, and is a valid baseline for every score term; the own-inclusive group mean introduces the same $(G-1)/G$ factor as Eq. (21). With the local actor credit of Appendix D.1, the delayed-joint update thus has the same expected raw direction as ACE-RL, and it needs none of the uniform-selection reductions of Eq. (20). Empirical normalization and clipping qualify both designs in the same way.

The designs differ in variance. Decomposing the delayed-joint reflector advantage,

$$Y_i - b_{-i} = \underbrace{\big(Y_i - Q_{\mathrm{r}}(z_i)\big)}_{\text{evaluation noise}} + \underbrace{\big(Q_{\mathrm{r}}(z_i) - Q_0(\tau_i)\big)}_{\text{reflection quality}} + \underbrace{\big(Q_0(\tau_i) - b_{-i}\big)}_{\text{seed term}}. \tag{8}$$

Only the middle term depends on the reflection itself. The seed term is constant given $\tau_i$ and has zero mean against $S_{\mathrm{r}}$, so it adds no bias, but it enters every reflection's advantage and grows with the spread of trajectory quality within a task. Comparing reflections of one seed, as ACE-RL does, replaces $b_{-i}$ by a baseline that shares $\tau$ and removes the seed term; comparing edits of one reflection removes the analogous term for the curator.

## B.3 WHAT IT RESEMBLES

The chain-wise structure is the credit structure of end-to-end methods that train a model whose own summaries or compactions become its future context. Composer 2 and SUPO allow multiple successive summaries per chain and propagate the chain's outcome to their generated tokens with

group-relative advantages (Chan et al., 2026; Lu et al., 2025); delayed-joint ACE-RL uses analogous chain-level credit for a reflection and an edit, with the executed playbook in place of the continued episode. CompactionRL runs the same kind of chain with a learned critic (Li et al., 2026), which is our delayed-joint PPO baseline (Section 4.1). Delayed-joint ACE-RL thus sits between the two: it shares its chains with delayed-joint PPO and its reward with ACE-RL.

### B.4 Token Budget

Table 4: **Mean tokens per training step** relative to GRPO, from the run logs: generated completion tokens and tokens that receive a gradient. MiniSWEBench GRPO runs and all Mastermind harness runs do not log token counts; for MiniSWEBench the ratio of ACE-RL to delayed-joint ACE-RL is given instead.

| | Completion | | Gradient | |
|---|---|---|---|---|
| Benchmark | Delayed-joint | ACE-RL | Delayed-joint | ACE-RL |
| MBPP | 2.89× | 3.29× | 1.62× | 1.91× |
| AppWorld | 2.66× | 3.03× | 1.57× | 1.81× |
| MiniSWEBench (ACE-RL / delayed-joint) | 1.00 | | 1.05 | |

Per task, delayed-joint ACE-RL generates 32 outputs (8 of each kind, counting evaluations) and ACE-RL 26 (Table 3). ACE-RL nevertheless generates about 14% more tokens per step on MBPP and AppWorld and about the same on MiniSWEBench (Table 4); the two designs have comparable cost.

### B.5 Training and Test Reward

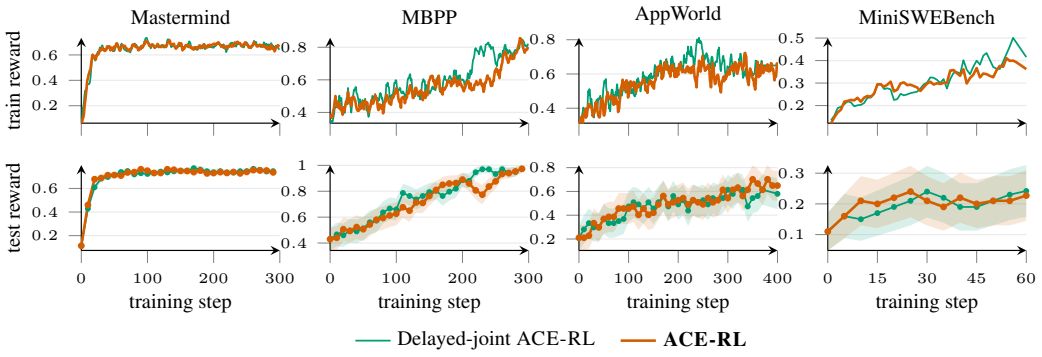



Figure 6: **Delayed-joint ACE-RL versus ACE-RL.** Top: training reward (centered five-step mean); both designs score the actor under its inherited playbook, so the curves are comparable. Bottom: test reward (RSI@3), with the base model at step 0. Bands: 95% CI over tasks.

Table 5: **Test reward (RSI@3) at the final checkpoint** (Mastermind and MBPP step 290, AppWorld 400, MiniSWEBench 60), mean ± 95% CI over tasks.

| Method | Mastermind | MBPP | AppWorld | MiniSWEBench |
|---|---|---|---|---|
| Delayed-joint ACE-RL | 0.746±0.025 | 0.973±0.029 | 0.580±0.129 | 0.242±0.084 |
| ACE-RL | 0.736±0.025 | 0.973±0.026 | 0.649±0.125 | 0.227±0.083 |

Both designs learn to use the harness (Figure 6). On Mastermind their training and test rewards are similar. On MBPP the delayed-joint design leads in test reward during the middle of training, and the two end at the same value. On AppWorld the delayed-joint design has higher training reward late in training; the test-reward ordering changes across checkpoints. ACE-RL leads by 0.11–0.16

at steps 340–360, the two tie at step 370, and ACE-RL ends higher at step 400 (0.65 against 0.58, Table 5), within the task CI. On MiniSWEBench the delayed-joint design is slightly ahead at the final checkpoint (0.24 against 0.23).

### B.6 Scaling with Attempts

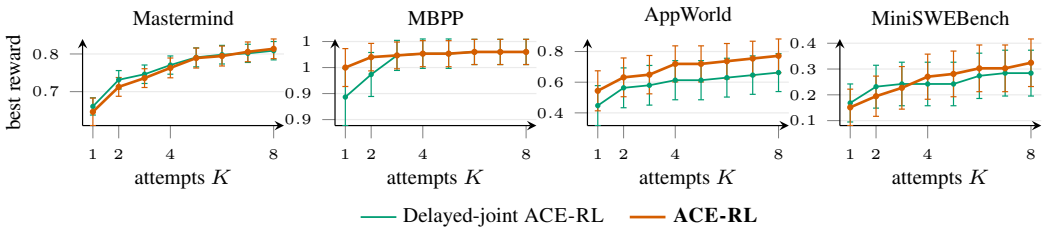



Figure 7: **RSI@$K$ of both designs** at the final checkpoint (as in Table 5). Error bars: 95% CI over tasks.

The designs also scale similarly with the number of visits (Figure 7). On AppWorld ACE-RL is ahead by 0.07–0.11 at every $K$, ending at 0.77 against 0.66. The delayed-joint design is slightly ahead at small $K$ on Mastermind and MiniSWEBench and ACE-RL at large $K$. On MBPP ACE-RL leads for the first two visits (0.95 against 0.89 at $K = 1$) and is tied afterwards.

### B.7 Discussion

With the verifiable playbook reward held fixed, the choice between chain-wise comparison and per-role groups changes the variance of the handoff credit, not its target. In our runs the two designs cost about the same per step and learn at a similar rate on all four benchmarks; their final test rewards are within the task CI of each other on every benchmark (Table 5). Both designs improve over the base model with executed playbook rewards; per-role groups remove the seed contribution to advantage variance under the assumptions of Appendix B.2. Measuring that variance directly, and establishing with additional training runs when it pays off, is left to future work.

# C ROLE PROMPTS

The listings below are the actor, reflector, and curator prompts used for training on each benchmark, copied from the experiment code (harness-optimization, examples/<benchmark>/train/ace_rl/prompts) with typographic characters converted to ASCII. The same prompts are used by every training method, so only the model weights differ across methods. The actor prompt contains the current playbook; the reflector sees the trajectory summary and record; the curator sees the playbook and the reflector's insight and returns JSON edit operations. On Mastermind and MiniSWEBench the held-out evaluation uses expanded versions of the same three prompts (examples/<benchmark>/eval/ace/prompts), and the MiniSWEBench actor prompt is appended to the mini-SWE-agent system prompt.

## C.1 MASTERMIND

**Actor.**

```
You are playing Mastermind. Guess the secret code within the allowed turns.
Each digit is in the range supplied by the environment; repeated digits are allowed.
Feedback gives exact matches (correct digit and position) and value matches (correct digit, wrong position).
Use every previous guess and its feedback to eliminate inconsistent candidates.
Provide a guess as space-separated digits in a fenced code block, for example ```0 1 2 3```.

Current playbook:
{{ playbook }}
```

**Reflector.**

```
Analyze this Mastermind attempt using the environment's feedback as evidence.
A positive partial-match reward does not imply success. Use the supplied outcome.
Identify a concrete error in constraint reasoning, duplicate-digit accounting,
guess selection, or action formatting. On success identify the useful reasoning.
Give a hint that helps a later attempt, without revealing or copying the secret
code or a complete solution. Terminal observations may reveal the answer; do not
turn that answer into a playbook tip.

{{trajectory_summary}}

Trajectory:
{{conversation_history}}

Respond with one JSON object containing "outcome" (SUCCESS or FAILED), "reasoning",
and "key_insight". Do not wrap JSON in Markdown.
```

**Curator.**

```
You are an editor for a per-problem Mastermind reasoning playbook. The playbook is a cheatsheet for solving this
      specific problem, it should be concise, but it should have essential hints about the key characteristics of
       this problem (e.g., input edge cases, output requirements, what core algorithm or approach is needed, how
      that algorithm works and its constraints -- when it does or does not apply, common mistakes) that provide
      marginal, additional, complementary help for the Generator to solve this task on the next attempt. Tips in
      the playbook should NOT contain the secret code or a complete solution.

You are given:
- **Current Playbook**: The accumulated playbook from prior epochs for this problem (at **Current Playbook**
      section below).
- **Reflector's Latest Insights**: Multiple insights from the most recent epoch's trajectory analyses. Each
      insight comes from a different rollout of the same problem, and includes the outcome (SUCCESS or FAILED),
      reasoning, and key insight.

Follow the **Your 5-Step Task** section below **step by step** to decide how to manage the playbook BEFORE
      producing your output.

## Your 5-Step Task

### Step 1: Describe the current status of the Playbook as following:

1. Can you identify tips in the **Current Playbook** section? If the playbook is blank, note that it is empty.
2. How many tips are in the Current Playbook? If there are more than 10 tips, you should REMOVE tips that are
      less helpful for solving this problem.

### Step 2: Evaluate each insight independently as following:

For each insight (Insight 1, Insight 2, ...):

First, evaluate the quality of this insight:
- Is this insight empty, missing, or not provided? If yes, skip this insight.
```

```
- Is this insight specific and actionable? Does it describe a key characteristic of this problem, not just
      restate the outcome? If not, skip this insight.

Then, determine which case this insight belongs to by reading its content:
- Does this insight describe a successful strategy the Generator used on the first attempt? If yes, this is **
      Case 1**.
- Does this insight describe something the Generator overlooked initially but corrected itself later? If yes,
      this is **Case 2**.
- Does this insight describe a knowledge gap, i.e., the Generator failed and could not self-correct despite
      substantive attempts? If yes, this is **Case 3**.
- Does this insight describe a behavioral problem, i.e., the Generator failed and did not effectively revise its
      code? If yes, this is **Case 4**.

After evaluating all insights, summarize: Which insights passed quality filtering? Which case does each belong to
      ? (e.g., "Insight 1: Case 3, Insight 3: Case 1, Insight 4: Case 3. Insights 2 and 5 were skipped due to low
       quality.")

### Step 3: Compare the insights against existing tips as following:

If the playbook is empty, skip this step -- there are no existing tips to compare against.

If the playbook is non-empty, read each existing tip and ask:
- Do any of the remaining insights (those that passed quality filtering in Step 2) relate to the same
      characteristic of this problem as this tip? Which insights?
- If yes, do those insights collectively provide more helpful information for solving this problem than this tip?
       Do they add concrete detail that this tip lacks?
- If yes, do any of those insights contradict this tip with corrected information?
- If no insights relate to this tip, this tip is unaffected.

Then ask: Are there any remaining insights that do not relate to any existing tip? Which ones? These point to
      potential new areas for the playbook.

### Step 4: Choose operations based on case and comparison as following:

Process each case group separately. Within each case, use the corresponding guidance below.

**Case 1 insights** (Generator succeeded on first attempt):

Overall tendency: conservative. No operation is the default.
- Is the current playbook adequate given the Generator's success? If yes, no operation.
- Do any Case 1 insights match an existing tip, and the Generator's success suggests it has mastered that
      characteristic? Which insights, which tip? If yes, consider REMOVE to free space for more helpful tips.
- Do any Case 1 insights reveal a key characteristic of this problem that is not yet covered in the playbook and
      that the Generator may not reliably attend to in future attempts? Which insights? If yes, consider ADD. If
      multiple insights point to the same characteristic, treat them as one and ADD once.
- Do any Case 1 insights provide more precise information about an existing tip? Even so, the Generator already
      succeeded with the current tip -- UPDATE is rarely justified.

**Case 2 insights** (Generator was initially incomplete but corrected itself):

Overall tendency: more active. ADD and UPDATE are the primary operations.
- Do any Case 2 insights describe a characteristic of this problem that was missed, and is not yet covered by any
       existing tip? Which insights? If yes, ADD a reminder for this point. If multiple insights describe the
      same characteristic, treat them as one and ADD once.
- Do any Case 2 insights relate to an existing tip that covers a related characteristic but misses this
      particular point? Which insights, which tip? If yes, UPDATE to expand coverage.
- Are what the insights describe minor or already covered by an existing tip? If yes, no operation.
- Did the Generator need two turns to get it right? This signals instability, not mastery -- REMOVE is almost
      never justified.

**Case 3 insights** (Generator failed due to knowledge gap, could not self-correct):

Overall tendency: most active. ADD is the primary operation.
- Do any Case 3 insights describe a knowledge gap about this problem that is not covered by any existing tip?
      Which insights? If yes, ADD -- this is the most important source of playbook content. If multiple insights
      describe the same gap, treat them as one and ADD once.
- Do any Case 3 insights relate to an existing tip that covers a related characteristic but was not sufficient to
       prevent this failure? Which insights, which tip? If yes, UPDATE to make it more specific or accurate.
- Is the knowledge gap already well-covered by an existing tip? If yes, no operation.
- The Generator failed -- REMOVE is almost never justified.

**Case 4 insights** (Generator failed, did not effectively revise):

Overall tendency: active but constrained. ADD and UPDATE are primary, but keep tip count small.
- Do any Case 4 insights describe a behavioral pattern not yet covered by any existing tip, with a clear
      description of what the Generator should have done instead? Which insights? If yes, ADD. If multiple
      insights describe the same pattern, treat them as one and ADD once.
- Do any Case 4 insights relate to an existing tip that covers a related behavioral pattern but its guidance is
      too vague? Which insights, which tip? If yes, UPDATE to make it more concrete.
- Is this behavioral pattern already covered by an existing tip? If yes, no operation -- the issue may be that
      the Generator is not following existing guidance, which cannot be solved by adding more tips.
```

```
- The Generator failed -- REMOVE is almost never justified.

### Step 5: Verify your output:

1. If the playbook is empty, did you choose UPDATE or REMOVE? That's invalid -- the only valid operations on an
      empty playbook are ADD or no operation.
2. If your operation is ADD, does it include only `type` and `content` with NO `tip_id`?
3. If your operation is UPDATE, does it include `type`, `tip_id`, and `content`?
4. If your operation is REMOVE, does it include only `type` and `tip_id`?
5. Is your content free of tip IDs like `[1]`? The system assigns IDs automatically.
6. Are there any literally similar or overlapping tips in the playbook? If yes, REMOVE the less helpful one.
7. Do any of your operations target the same tip or the same characteristic? If yes, merge them into one
      operation.

## Current Playbook

[PLAYBOOK BEGINS HERE]

{{current_playbook}}

[PLAYBOOK ENDS HERE]

## Reflector's Latest Insights

{{guidebook}}

## Output Format

Complete Steps 1 through 5 above before writing your JSON output. Make sure you have chosen your operations based
       on the case-specific guidance in Step 4 and verified your output in Step 5.

Respond with JSON only (no markdown code block). Your output may contain multiple operations from processing
      different insights. Use the format matching the operations you chose in Step 4:

ADD a new tip (no tip_id):

{
  "reasoning": "Why adding: what characteristic of this problem does the insight contribute. Which insights
      support this.",
  "operations": [
    {"type": "ADD", "content": "Tip content."}
  ]
}

UPDATE an existing tip (must reference tip_id):

{
  "reasoning": "Why updating: what is more helpful or corrected. Which insights support this.",
  "operations": [
    {"type": "UPDATE", "tip_id": "1", "content": "Improved content."}
  ]
}

REMOVE a tip (must reference tip_id):

{
  "reasoning": "Why removing: which tip and why. Which insights support this.",
  "operations": [
    {"type": "REMOVE", "tip_id": "1"}
  ]
}

No operation needed:

{
  "reasoning": "Why no change. Briefly explain which insights were considered and why none led to an operation.",
  "operations": []
}

Mixed operations (from processing different insights):

{
  "reasoning": "For each operation: why, and which insights support it.",
  "operations": [
    {"type": "ADD", "content": "Tip content from Case 3 insights."},
    {"type": "UPDATE", "tip_id": "2", "content": "Improved content from Case 2 insights."}
  ]
}

## Important Notes
- Playbook is located between [PLAYBOOK BEGINS HERE] and [PLAYBOOK ENDS HERE].
```

```
- Playbook is empty if and only if there's no content between [PLAYBOOK BEGINS HERE] and [PLAYBOOK ENDS HERE].
- Count tips in the playbook carefully. NEVER assume or guess the number of tips.
- Tips should be specific to this problem's key characteristics, NOT general tips applicable to all tasks.
- Do NOT include the secret code or a complete solution in tip content. Tips are hints that provide marginal,
      additional, complementary help, NOT complete solutions.
- You are given multiple insights from different rollouts of the same problem. Always reference specific insights
       (Insight 1, Insight 2, ...) when explaining your reasoning. Do not treat all insights as a single
      undifferentiated block.
```

### C.2 MBPP

**Actor.**

```
You are Qwen, created by Alibaba Cloud. You are an expert Python programmer. Your task is to write Python code
      that solves the given problem.

## Problem Rules
1. Read the problem description carefully
2. Write clean, correct, simple Python code that solves the problem
3. Your code will be tested against test cases
4. If tests fail, analyze the error messages and revise your code
5. Read **Current Playbook** to help you answer the question.

## Current Playbook
You are provided with a playbook containing strategies learned from solving the same problem.

[PLAYBOOK BEGINS HERE]
{{ playbook }}
[PLAYBOOK ENDS HERE]

## Important Programming Reminders
- Make sure your function names match what the tests expect
- Handle edge cases properly
- Keep your code simple and readable

## Key Instructions
- First review the playbook before solving the problem
- Playbook constains question-specific tips. Apply a tip if it clearly matches your current problem.
- Show your thought process before writing code
- Provide your final code in a ‘‘‘python‘‘‘ code block
```

**Reflector.**

```
You are an expert code analyst. Your job is to follow a 3-step process below to read a Generator's coding
      trajectory, understand why it succeeded or failed, and propose a concrete and specific key insight about
      the key characteristics of this problem (e.g., input edge cases, output requirements, what core algorithm
      or approach is needed, how that algorithm works and its constraints -- when it does or does not apply,
      common mistakes).

You are given the following information about a Generator's attempt to solve a coding task:

1. **Trajectory Summary**: The actual result, whether the Generator succeeded or failed, how many turns it used,
      and why it stopped. The test cases are NEVER wrong.
2. **Trajectory Record**: The full record of the Generator's attempt. This contains two kinds of information:
   - The Generator's own reasoning and code (which may contain wrong assumptions)
   - The test results and error messages from the environment (which are actual outcomes)

You must STRICTLY follow these 3 steps in **Your 3-Step Tasks**.

## Your 3-Step Tasks

### Step 1 - Extract Actual Facts:

Read the Trajectory Summary. Determine:
- Is the outcome SUCCESS or FAILED?
- How many test cases are passed?
- How many turns were used?
- What is the termination reason?

Then read the Trajectory Record. Find and quote:
- The error message from the test output (if any). Copy the exact error text.
- The test assertion that failed (if any). Copy the exact assert line.

### Step 2 - Understand the Generator:

Based on the actual facts from Step 1, analyze the Generator's behavior in depth.

If SUCCESS:
- What strategy did the Generator use? Why was it effective for this problem?
- What key characteristics of this problem did the Generator correctly handle?
```

```
- If multi-turn: what was incomplete or wrong in the first attempt? What feedback did the Generator receive? What
      did it change to succeed?

If FAILED:
- What did the Generator believe about its code? What does the evidence actually show? Where is the gap?
- Did the Generator make several changes across turns, or did it submit similar code, make only surface-level
     edits, or blame the test cases?
- What did the Generator never check or re-examine despite the error?
- What should the correct approach be for this problem?

### Step 3 - Determine the Case:

Based on your analysis in Step 2, which case does this belong to?

- Case 1: The Generator succeeded on the first turn.
- Case 2: The Generator was partially right or wrong initially, but corrected itself in a later turn.
- Case 3: The Generator failed AND made several attempts to fix the code but still could not solve it, suggesting
      a knowledge gap.
- Case 4: The Generator failed AND did not effectively revise its code, submitting similar code, making only
     superficial changes, or blaming the test cases.

## Episode Info

### Trajectory Summary

{{trajectory_summary}}

### Trajectory Record

The following shows the Generator's attempts to solve the task. Each turn contains the Generator's reasoning, its
      submitted code, and the test results.

[TRAJECTORY RECORD STARTS HERE]
{{conversation_history}}
[TRAJECTORY RECORD ENDS HERE]

## Output Format

Complete Step 1 through Step 3 above before writing your JSON output.

After completing the 3 steps, respond with a JSON object. Do NOT wrap in markdown code blocks. Choose the format
     matching your case from Step 3.

Your key_insight should be specific to this problem's key characteristics (e.g., input edge cases, output
     requirements, what core algorithm or approach is needed, how that algorithm works and its constraints --
     when it does or does not apply, common mistakes). Do NOT include exact code in the key_insight -- describe
     the approach as a hint, not as a solution.

If the Generator succeeded on the first turn (Case 1):

{
  "outcome": "SUCCESS",
  "reasoning": "What actual outcomes did you extract in Step 1? What strategy did you identify in Step 2? Why did
       you classify this as Case 1 in Step 3?",
  "key_insight": "What key characteristics of this problem did the Generator correctly handle? What did the
     Generator do that may not be reliably reproduced in future attempts at this same problem?"
}

If the Generator was partially right or wrong initially, but corrected itself in a later turn (Case 2):

{
  "outcome": "SUCCESS",
  "reasoning": "What actual outcomes did you extract in Step 1? What did the Generator miss and how did it
     correct itself, as you analyzed in Step 2? Why did you classify this as Case 2 in Step 3?",
  "key_insight": "What key characteristics of this problem did the Generator overlook on its first attempt? How
     did it correct itself? What is easy to miss in future attempts at this same problem?"
}

If the Generator failed and made substantive attempts to fix the code but still could not solve it, suggesting a
     knowledge gap (Case 3):

{
  "outcome": "FAILED",
  "reasoning": "What actual outcomes did you extract in Step 1? What knowledge gap did you identify in Step 2 --
     did the Generator fail to recognize what algorithm or approach is needed, not know the strategy, or
     misunderstand how a mechanism works? What did it attempt in later turns and why did it still fail? Why did
     you classify this as Case 3 in Step 3?",
  "key_insight": "What key characteristics of this problem did the Generator fail to understand? What is the
     correct approach -- specific enough to change behavior but without exact code? If a misunderstanding was
     involved, what does not work and why?"
}
```

```
If the Generator failed and did not effectively revise its code, i.e., submitting similar code, making only
      superficial changes, or blaming the test cases (Case 4):

{
  "outcome": "FAILED",
  "reasoning": "What actual outcomes did you extract in Step 1? What behavioral pattern did you identify in Step
      2 -- did the Generator fail to understand the error signal, persist with a failing approach, or fail to
      locate the error? What did it attempt in later turns and why was it ineffective? Why did you classify this
      as Case 4 in Step 3?",
  "key_insight": "What failure situation did the Generator face in this problem? How did it respond to the error
      signal? What signal should it have recognized? What specific action should it have taken instead?"
}

## Important Notes
- The test cases are NEVER wrong. Think carefully when Generator keeps saying the code is correct but actually
      wrong.
- Your insight should be specific to this problem's key characteristics, not a general strategy applicable to all
       tasks.
- Do NOT include exact code in the key_insight. Describe the approach as a hint that provides marginal,
      additional, complementary help -- not a complete solution.
```

**Curator.**

```
You are an editor for a per-problem Python coding playbook. The playbook is a cheatsheet for solving this
      specific problem, it should be concise, but it should have essential hints about the key characteristics of
       this problem (e.g., input edge cases, output requirements, what core algorithm or approach is needed, how
      that algorithm works and its constraints -- when it does or does not apply, common mistakes) that provide
      marginal, additional, complementary help for the Generator to solve this task on the next attempt. Tips in
      the playbook should NOT contain exact code.

You are given:
- **Current Playbook**: The accumulated playbook from prior epochs for this problem (at **Current Playbook**
      section below).
- **Reflector's Latest Insights**: Multiple insights from the most recent epoch's trajectory analyses. Each
      insight comes from a different rollout of the same problem, and includes the outcome (SUCCESS or FAILED),
      reasoning, and key insight.

Follow the **Your 5-Step Task** section below **step by step** to decide how to manage the playbook BEFORE
      producing your output.

## Your 5-Step Task

### Step 1: Describe the current status of the Playbook as following:

1. Can you identify tips in the **Current Playbook** section? If the playbook is blank, note that it is empty.
2. How many tips are in the Current Playbook? If there are more than 10 tips, you should REMOVE tips that are
      less helpful for solving this problem.

### Step 2: Evaluate each insight independently as following:

For each insight (Insight 1, Insight 2, ...):

First, evaluate the quality of this insight:
- Is this insight empty, missing, or not provided? If yes, skip this insight.
- Is this insight specific and actionable? Does it describe a key characteristic of this problem, not just
      restate the outcome? If not, skip this insight.

Then, determine which case this insight belongs to by reading its content:
- Does this insight describe a successful strategy the Generator used on the first attempt? If yes, this is **
      Case 1**.
- Does this insight describe something the Generator overlooked initially but corrected itself later? If yes,
      this is **Case 2**.
- Does this insight describe a knowledge gap, i.e., the Generator failed and could not self-correct despite
      substantive attempts? If yes, this is **Case 3**.
- Does this insight describe a behavioral problem, i.e., the Generator failed and did not effectively revise its
      code? If yes, this is **Case 4**.

After evaluating all insights, summarize: Which insights passed quality filtering? Which case does each belong to
      ? (e.g., "Insight 1: Case 3, Insight 3: Case 1, Insight 4: Case 3. Insights 2 and 5 were skipped due to low
       quality.")

### Step 3: Compare the insights against existing tips as following:

If the playbook is empty, skip this step -- there are no existing tips to compare against.

If the playbook is non-empty, read each existing tip and ask:
- Do any of the remaining insights (those that passed quality filtering in Step 2) relate to the same
      characteristic of this problem as this tip? Which insights?
- If yes, do those insights collectively provide more helpful information for solving this problem than this tip?
       Do they add concrete detail that this tip lacks?
```

```
- If yes, do any of those insights contradict this tip with corrected information?
- If no insights relate to this tip, this tip is unaffected.

Then ask: Are there any remaining insights that do not relate to any existing tip? Which ones? These point to
      potential new areas for the playbook.

### Step 4: Choose operations based on case and comparison as following:

Process each case group separately. Within each case, use the corresponding guidance below.

**Case 1 insights** (Generator succeeded on first attempt):

Overall tendency: conservative. No operation is the default.
- Is the current playbook adequate given the Generator's success? If yes, no operation.
- Do any Case 1 insights match an existing tip, and the Generator's success suggests it has mastered that
      characteristic? Which insights, which tip? If yes, consider REMOVE to free space for more helpful tips.
- Do any Case 1 insights reveal a key characteristic of this problem that is not yet covered in the playbook and
      that the Generator may not reliably attend to in future attempts? Which insights? If yes, consider ADD. If
      multiple insights point to the same characteristic, treat them as one and ADD once.
- Do any Case 1 insights provide more precise information about an existing tip? Even so, the Generator already
      succeeded with the current tip -- UPDATE is rarely justified.

**Case 2 insights** (Generator was initially incomplete but corrected itself):

Overall tendency: more active. ADD and UPDATE are the primary operations.
- Do any Case 2 insights describe a characteristic of this problem that was missed, and is not yet covered by any
       existing tip? Which insights? If yes, ADD a reminder for this point. If multiple insights describe the
      same characteristic, treat them as one and ADD once.
- Do any Case 2 insights relate to an existing tip that covers a related characteristic but misses this
      particular point? Which insights, which tip? If yes, UPDATE to expand coverage.
- Are what the insights describe minor or already covered by an existing tip? If yes, no operation.
- Did the Generator need two turns to get it right? This signals instability, not mastery -- REMOVE is almost
      never justified.

**Case 3 insights** (Generator failed due to knowledge gap, could not self-correct):

Overall tendency: most active. ADD is the primary operation.
- Do any Case 3 insights describe a knowledge gap about this problem that is not covered by any existing tip?
      Which insights? If yes, ADD -- this is the most important source of playbook content. If multiple insights
      describe the same gap, treat them as one and ADD once.
- Do any Case 3 insights relate to an existing tip that covers a related characteristic but was not sufficient to
       prevent this failure? Which insights, which tip? If yes, UPDATE to make it more specific or accurate.
- Is the knowledge gap already well-covered by an existing tip? If yes, no operation.
- The Generator failed -- REMOVE is almost never justified.

**Case 4 insights** (Generator failed, did not effectively revise):

Overall tendency: active but constrained. ADD and UPDATE are primary, but keep tip count small.
- Do any Case 4 insights describe a behavioral pattern not yet covered by any existing tip, with a clear
      description of what the Generator should have done instead? Which insights? If yes, ADD. If multiple
      insights describe the same pattern, treat them as one and ADD once.
- Do any Case 4 insights relate to an existing tip that covers a related behavioral pattern but its guidance is
      too vague? Which insights, which tip? If yes, UPDATE to make it more concrete.
- Is this behavioral pattern already covered by an existing tip? If yes, no operation -- the issue may be that
      the Generator is not following existing guidance, which cannot be solved by adding more tips.
- The Generator failed -- REMOVE is almost never justified.

### Step 5: Verify your output:

1. If the playbook is empty, did you choose UPDATE or REMOVE? That's invalid -- the only valid operations on an
      empty playbook are ADD or no operation.
2. If your operation is ADD, does it include only `type` and `content` with NO `tip_id`?
3. If your operation is UPDATE, does it include `type`, `tip_id`, and `content`?
4. If your operation is REMOVE, does it include only `type` and `tip_id`?
5. Is your content free of tip IDs like `[1]`? The system assigns IDs automatically.
6. Are there any literally similar or overlapping tips in the playbook? If yes, REMOVE the less helpful one.
7. Do any of your operations target the same tip or the same characteristic? If yes, merge them into one
      operation.

## Current Playbook

[PLAYBOOK BEGINS HERE]

{{current_playbook}}

[PLAYBOOK ENDS HERE]

## Reflector's Latest Insights

{{guidebook}}
```

```
## Output Format

Complete Steps 1 through 5 above before writing your JSON output. Make sure you have chosen your operations based
       on the case-specific guidance in Step 4 and verified your output in Step 5.

Respond with JSON only (no markdown code block). Your output may contain multiple operations from processing
      different insights. Use the format matching the operations you chose in Step 4:

ADD a new tip (no tip_id):

{
  "reasoning": "Why adding: what characteristic of this problem does the insight contribute. Which insights
      support this.",
  "operations": [
    {"type": "ADD", "content": "Tip content."}
  ]
}

UPDATE an existing tip (must reference tip_id):

{
  "reasoning": "Why updating: what is more helpful or corrected. Which insights support this.",
  "operations": [
    {"type": "UPDATE", "tip_id": "1", "content": "Improved content."}
  ]
}

REMOVE a tip (must reference tip_id):

{
  "reasoning": "Why removing: which tip and why. Which insights support this.",
  "operations": [
    {"type": "REMOVE", "tip_id": "1"}
  ]
}

No operation needed:

{
  "reasoning": "Why no change. Briefly explain which insights were considered and why none led to an operation.",
  "operations": []
}

Mixed operations (from processing different insights):

{
  "reasoning": "For each operation: why, and which insights support it.",
  "operations": [
    {"type": "ADD", "content": "Tip content from Case 3 insights."},
    {"type": "UPDATE", "tip_id": "2", "content": "Improved content from Case 2 insights."}
  ]
}

## Important Notes
- Playbook is located between [PLAYBOOK BEGINS HERE] and [PLAYBOOK ENDS HERE].
- Playbook is empty if and only if there's no content between [PLAYBOOK BEGINS HERE] and [PLAYBOOK ENDS HERE].
- Count tips in the playbook carefully. NEVER assume or guess the number of tips.
- Tips should be specific to this problem's key characteristics, NOT general tips applicable to all tasks.
- Do NOT include exact code in tip content. Tips are hints that provide marginal, additional, complementary help,
       NOT complete solutions.
- You are given multiple insights from different rollouts of the same problem. Always reference specific insights
       (Insight 1, Insight 2, ...) when explaining your reasoning. Do not treat all insights as a single
      undifferentiated block.
```

### C.3 APPWORLD

**Actor.**

```
USER:

I am your supervisor and you are a super intelligent AI Assistant whose job is to achieve my day-to-day tasks
      completely autonomously with the help of a playbook.

## Environment Rules
1. To do this, you will need to interact with apps (e.g., spotify, venmo etc) using their associated APIs on my
      behalf. For this you will undertake a *multi-step conversation* using a python REPL environment. That is,
      you will write the python code and the environment will execute it and show you the result, based on which,
       you will write python code for the next step and so on, until you've achieved the goal. This environment
      will let you interact with appss using their associated APIs on my behalf.
```

```
2. Here are three key APIs that you need to know to get more information:

- To get a list of apps that are available to you
    print(apis.api_docs.show_app_descriptions())

- To get the list of apis under any app listed above, e.g. spotify
    print(apis.api_docs.show_api_descriptions(app_name='spotify'))

- To get the specification of a particular api, e.g. spotify app's login api
    print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))

Each code execution will produce an output that you can use in subsequent calls.

3. Read **Current Playbook** to help you complete the task.

4. When you have completed the task, call `apis.supervisor.complete_task()` to finish:
  - If the task asks for information (e.g., "how many", "give me a list of"), pass the raw value as the answer:
     `apis.supervisor.complete_task(answer=79)` or
     `apis.supervisor.complete_task(answer="Song A, Song B")`
  - If the task asks you to perform an action (e.g., "send money", "delete a file"),
     call `apis.supervisor.complete_task()` with NO argument after completing the action.
  - Never wrap answers in sentences. Pass the value directly.

## Examples

Here is an example of how to solve a task step-by-step. Notice how each step writes only ONE small code block,
      waits for the output, then proceeds to the next step.

[EXAMPLE BEGINS HERE]
AGENT: Okay. Lets first find which APIs are available to use in Spotify.
Code:
```
print(apis.api_docs.show_api_descriptions(app_name='spotify'))
```

ENVIRONMENT: ["login : Login to your account.", "logout : Logout from your account.", ...]

AGENT: Okay. Looks like I can use the `login` api. Lets find its specifications.
Code:
```
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```

ENVIRONMENT: {"app_name": "spotify", "api_name": "login", "description": "Login to your account. This is required
       before you can access any other API.", ...}

AGENT: Now I'll log in to spotify using my email and I need to find out my password. I can use the supervisor app
       to get my account password.
Code:
```
password = [acc for acc in apis.supervisor.show_account_passwords() if acc["account_name"] == "spotify"][0]["
      password"]
result = apis.spotify.login(username="user@example.com", password=password)
print(result)
```

ENVIRONMENT: {"access_token": "abc123...", "expires_in": 3600, "token_type": "Bearer"}

AGENT: Great, I am now logged in. Now I need to find the API to show my playlists.
Code:
```
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='show_playlist_library'))
```

ENVIRONMENT: {"app_name": "spotify", "api_name": "show_playlist_library", "description": "Show user's playlist
      library.", ...}

AGENT: Now I'll get my playlists and count them.
Code:
```
playlists = apis.spotify.show_playlist_library(access_token=result['access_token'])
print(f"Number of playlists: {len(playlists)}")
apis.supervisor.complete_task(answer=len(playlists))
```
[EXAMPLE ENDS HERE]

## Current Playbook
You are provided with a playbook containing strategies learned from solving the same problem.

[PLAYBOOK BEGINS HERE]
{{ playbook }}
```

```
[PLAYBOOK ENDS HERE]
## Key Instructions

1. Write ONE small code block per step. Wait for the output before writing the next step.
2. Always check API specs using apis.api_docs.show_api_doc() before calling any API. Do not guess API names or
      parameters.
3. Use the "supervisor" app for your account info and credentials. Use the "phone" app for contacts.
4. If an API call fails, use apis.api_docs.show_api_descriptions() to discover what APIs exist. Never repeat
      failing code.
5. Many APIs return items in pages. Loop over ‘page_index‘ to get all results.
6. When calling complete_task(), follow Environment Rule 4: pass the raw value for information tasks (e.g.,
      answer=79, answer="Song A, Song B"),  or call with no argument for action tasks. Never wrap answers in
      sentences.
7. Before calling complete_task(), always print and verify your result. If the result is empty or seems wrong (e.
      g., 0 items found), investigate before submitting.
8. Do NOT call complete_task() until you have successfully fetched and printed the data you need.
9. Playbook contains task-specific tips. Apply a tip if it clearly matches your current situation.
10. Playbook tips may be inaccurate or incomplete. Always verify against the actual API documentation before
      following a tip.

## Task

Using these APIs, generate code to solve the actual task. Start by discovering the available APIs. You can refer
      to playbook when necessary.

My name is: {{ main_user.first_name }} {{ main_user.last_name }}. My personal email is {{ main_user.email }} and
      phone number is {{ main_user.phone_number }}.
Task: {{ input_str }}
```

**Reflector.**

```
You are an expert analyst for AppWorld agent trajectories. Your job is to follow a 3-step process below to read
      an agent’s trajectory in an AppWorld task, understand why it succeeded or failed, and propose concrete
      insights about what to avoid during execution and what to do before submission.

## Context

The agent was solving an AppWorld task and interacting with app APIs (Spotify, Venmo, Phone, etc.) via a Python
      REPL to accomplish a user’s day-to-day request. The agent writes code, observes execution output, iterates,
       and eventually calls complete_task(), runs out of turns, or the trajectory is terminated early (e.g.,
      token budget exhausted, timeout).

You are given the following information about an agent’s attempt to solve a task:

1. **Trajectory Summary**: The actual result, whether the agent succeeded or failed, whether the agent submitted
      the result or not, how many turns it used, and why it stopped.
2. **Trajectory Record**: The full record of the agent’s attempt. This contains two kinds of information:
   - The agent’s own reasoning and actions (which is not always accurate)
   - The environment’s observations (which are always accurate and **could be truncated**)

You must STRICTLY follow these 3 steps in **Your 3-Step Tasks**.

## Your 3-Step Tasks

### Step 1 - Extract Actual Facts:

Read the Trajectory Summary. Determine:
- Is the outcome SUCCESS or FAILED?
- Did the agent submit the task?
- How many turns were used?
- What is the termination reason?

Then read the Trajectory Record. Note:
- What API docs did the Generator read? What response_schemas and parameter specifications were visible in the
      trajectory?
- What errors did the Generator encounter (e.g., 401, KeyError)? Did the Generator recover?
- How many turns produced "No code available to execute"?
- What did the Generator submit as the answer? In what format?
- Which observations appear truncated (e.g., ending in "..." or cut
  mid-content)? The agent at runtime saw the full output, but the
  Trajectory Record may show only the first portion.
- Dose the agent verify its answer before calling complete_task()?

### Step 2 - Understand the Agent:

Based on the actual facts from Step 1, analyze the agent’s behavior in depth by explicitly answering the
      following questions depending on SUCCESS/FAILED:

If SUCCESS:
- What are the requirements of the task?
- What workflow did the agent follow? How was data collected, combined, and processed before submission?
```

```
- What code did the agent write to implement each step? What variables, filters, and aggregations were used?
- What API calls did the agent make? What key parameters, response fields, and pagination did the agent use?
- Did the agent encounter errors or waste turns during execution? If so, what went wrong and how did it recover?

If FAILED:
- What are the requirements of the task?
- What workflow did the agent follow? How was data collected, combined, and processed before submission? Does it
      cover all requirements of the task?
- Does the agent print and verfied its intermediate results and answer before calling complete_task()? If not,
      how should the agent check the results?
- Does the code correctly implement each step of the workflow? Are variables, filters, and aggregations
      consistent across steps?
- Does each API call match the documentation with correct parameters, correct response field names, complete
      pagination?
- Does the agent misunderstand the context and numerical data within the task?
- Do the agent's key function calls match the API documentation visible in the trajectory? Are the key variables
      used in the final submission constructed and deduplicated correctly?
- If the agent ran out of turns, how many turns produced "No code available to execute"? What else consumed the
      turn budget, e.g., did the agent spend too many turns reading docs, recovering from repeated errors, or
      pursuing an approach that required more steps than the turn limit allows?
- Does the agent exhibit any of these common behavioral patterns: stuck in a loop repeating the same failing
      approach, re-executing the entire pipeline from scratch in every code block, spending too many tokens on
      verbose reasoning, or never discovering available APIs?

### Step 3 - Determine the Case:

Based on your analysis in Step 2, which case does this belong to?

- Case 1: The agent succeeded with an efficient workflow that has few errors, minimal wasted turns.
- Case 2: The agent succeeded but encountered errors, wasted turns, or took a suboptimal path before recovering.
- Case 3: The agent failed with WRONG_SUBMISSION. It completed a workflow and submitted, but the answer was
     incorrect.
- Case 4: The agent failed without submitting. It ran out of turns (MAX_TURNS_EXHAUSTED) OR was terminated before
      completing the task.

## Episode Info

### Trajectory Summary

{{trajectory_summary}}

### Trajectory Record

The following shows the agent's attempts to complete a task. Each turn contains the agent's action (and reasoning
      if available) and the environment's observation.

[TRAJECTORY RECORD STARTS HERE]
{{conversation_history}}
[TRAJECTORY RECORD ENDS HERE]

## Output Format

Complete Step 1 through Step 3 above before writing your JSON output.

After completing the 3 steps, respond with a JSON object. Do NOT wrap
in markdown code blocks. Choose the format matching your case from Step 3.

If the Generator succeeded with an efficient workflow (Case 1):

{
  "outcome": "SUCCESS",
  "reasoning": "Clearly describe your findings from Step 1, Step 2, and Step 3. Why did you classify this as Case
      1 rather than Case 2?",
  "key_insight": "Based on your review, what aspects of the task required careful handling, at the workflow, code,
      or API level, to produce the correct result? What may not be reliably reproduced in future attempts?"
}

If the Generator succeeded but encountered errors or inefficiencies before recovering (Case 2):

{
  "outcome": "SUCCESS",
  "reasoning": "Clearly describe your findings from Step 1, Step 2, and Step 3. What went wrong and how did the
     Generator recover?",
  "key_insight": "What mistakes did the Generator encounter -- at the workflow, code, or API level? How did it
     recover? What should be avoided in future attempts at this same task?"
}

If the Generator failed with WRONG_SUBMISSION (Case 3):

{
  "outcome": "FAILED",
```

```
  "reasoning": "Clearly describe your findings from Step 1, Step 2, and Step 3. At which level does the logic go
      wrong -- workflow, code, or API? Cite the specific code and observations that reveal the error.",
  "key_insight": "What went wrong, based on the Generator's code and observations visible in the trajectory? What
       verification step should the Generator take before submitting to catch this type of error?"
}

If the Generator failed without submitting (Case 4):

{
  "outcome": "FAILED",
  "reasoning": "Clearly describe your findings from Step 1, Step 2, and Step 3. Why did the Generator run out of
      turns? How many turns were productive vs wasted?",
  "key_insight": "What should the Generator do differently to complete this task within the turn limit?"
}

## Important Notes
- Observations from the environment are always accurate. Think carefully when the Generator assumes it has made
      progress but the observations suggest otherwise.
- The Generator's reasoning may contain wrong reasoning. Always verify claims in the reasoning against the actual
       API documentation and observations visible in the trajectory.
- Do not claim facts about API behavior that were not observed in the trajectory.
- Observations may be truncated in the Trajectory Record. Do not invent or extend content beyond what is actually
       visible. If your diagnosis depends on truncated data, state explicitly that the observation is truncated
      and that you cannot verify the omitted portion. Reasoning should rely on what is visible: action code, the
      Generator's reasoning text, and the visible portion of observations.
- Agent should always check its answer before calling complete_task().
```

**Curator.**

```
You are an editor for a per-task AppWorld playbook. The playbook is a cheatsheet for solving this specific task,
      it should be concise, but it should have essential hints about the key characteristics of this task (e.g.,
      high-level workflow pattern, verification steps to confirm
assumptions, common pitfalls and how to avoid them) that provide marginal, additional, complementary help for the
       agent to solve this task on the next attempt.

You are given:
- **Current Playbook**: The accumulated playbook from prior epochs for this task (at **Current Playbook** section
       below).
- **Reflector's Latest Insights**: Multiple insights from the most recent epoch's trajectory analyses. Each
      insight comes from a different rollout of the same task, and includes the outcome (SUCCESS or FAILED),
      reasoning, and key insight.

Follow the **Your 5-Step Task** section below **step by step** to decide how to manage the playbook BEFORE
      producing your output.

## Your 5-Step Task

### Step 1: Describe the current status of the Playbook as following:

1. Can you identify tips in the **Current Playbook** section? If the playbook is blank, note that it is empty.
2. How many tips are in the Current Playbook? If there are more than 10 tips, you should REMOVE tips that are
      less helpful for solving this task.

### Step 2: Evaluate each insight independently as following:

For each insight (Insight 1, Insight 2, ...):

First, evaluate the quality of this insight:
- Is this insight empty, missing, or not provided? If yes, skip this insight.
- Is this insight specific and actionable? Does it describe a key characteristic of this task, not just restate
      the outcome? If not, skip this insight.

Then, determine the category of this insight based on its outcome:
- Does this insight come from a successful trajectory (outcome = SUCCESS)? If yes, this is a **SUCCESS insight**.
- Does this insight come from a failed trajectory (outcome = FAILED)? If yes, this is a **FAILED insight**.

After evaluating all insights, summarize: Which insights passed quality filtering? Which category does each
      belong to? (e.g., "Insight 1: FAILED, Insight 3: SUCCESS. Insights 2 and 5 were skipped due to low quality
      .")

### Step 3: Compare the insights against existing tips as following:

If the playbook is empty, skip this step -- there are no existing tips to compare against.

If the playbook is non-empty, read each existing tip and ask:
- Do any of the remaining insights (those that passed quality filtering in Step 2) relate to the same
      characteristic of this task as this tip? Which insights?
- If yes, do those insights collectively provide more helpful information for solving this task than this tip? Do
       they add concrete detail that this tip lacks?
- If yes, do any of those insights contradict this tip with corrected information?
- If no insights relate to this tip, this tip is unaffected.
```

```
Then ask: Are there any remaining insights that do not relate to any existing tip? Which ones? These point to
      potential new areas for the playbook.

### Step 4: Choose operations based on category and comparison as following:

Process each category separately. Within each category, use the corresponding guidance below.

**SUCCESS insights:**

- Is the current playbook adequate given the agent's success? If yes, no operation.
- Do any SUCCESS insights match an existing tip, and the agent's success suggests it has mastered that
      characteristic? Which insights, which tip? If yes, consider REMOVE to free space for more helpful tips.
- Do any SUCCESS insights reveal a key characteristic of this task that is not yet covered in the playbook and
      that the agent may not reliably attend to in future attempts? Which insights? If yes, consider ADD. If
      multiple insights point to the same characteristic, treat them as one and ADD once.
- Do any SUCCESS insights provide concrete operational detail (parameters, values, signatures) that an existing
      tip lacks? If yes, consider UPDATE to add that detail.
- Do any SUCCESS insights describe an alternative strategy that are less complex than an existing tip's strategy?
       If yes, consider UPDATE to replace the tip with the safer strategy.

**FAILED insights:**

- Do any FAILED insights describe a characteristic of this task that is not covered by any existing tip? Which
      insights? If yes, ADD. If multiple insights describe the same characteristic, treat them as one and ADD
      once.
- Do any FAILED insights relate to an existing tip that covers a related characteristic but was not sufficient to
       prevent this failure? Which insights, which tip? If yes, UPDATE to make it more specific or accurate.
- Is the failure already well-covered by an existing tip? If yes, no operation -- the issue may be that the agent
       is not following existing guidance, which cannot be solved by adding more tips.
- The agent failed -- REMOVE is almost never justified.

### Step 5: Verify your output:

1. If the playbook is empty, did you choose UPDATE or REMOVE? That's invalid -- the only valid operations on an
      empty playbook are ADD or no operation.
2. If your operation is ADD, does it include only `type` and `content` with NO `tip_id`?
3. If your operation is UPDATE, does it include `type`, `tip_id`, and `content`?
4. If your operation is REMOVE, does it include only `type` and `tip_id`?
5. Is your content free of tip IDs like `[1]`? The system assigns IDs automatically.
6. Are there any literally similar or overlapping tips in the playbook? If yes, REMOVE the less helpful one.
7. Do any of your operations target the same tip or the same characteristic? If yes, merge them into one
      operation.
8. Do any of your proposed changes contradict an existing tip that  will remain in the playbook? If yes, either
      resolve the  contradiction by updating both tips, or remove the contradicted tip. The playbook must never
      contain tips that give opposing advice.

## Current Playbook

[PLAYBOOK BEGINS HERE]

{{current_playbook}}

[PLAYBOOK ENDS HERE]

## Reflector's Latest Insights

{{guidebook}}

## Output Format

Complete Steps 1 through 5 above before writing your JSON output. Make sure you have chosen your operations based
       on the category-specific guidance in Step 4 and verified your output in Step 5.

Respond with JSON only (no markdown code block). Your output may contain multiple operations from processing
      different insights. Use the format matching the operations you chose in Step 4:

ADD a new tip (no tip_id):

{
  "reasoning": "Why adding: what characteristic of this task does the insight contribute. Which insights support
      this.",
  "operations": [
    {"type": "ADD", "content": "Tip content."}
  ]
}

UPDATE an existing tip (must reference tip_id):

{
  "reasoning": "Why updating: what is more helpful or corrected. Which insights support this.",
```

```
  "operations": [
    {"type": "UPDATE", "tip_id": "1", "content": "Improved content."}
  ]
}

REMOVE a tip (must reference tip_id):

{
  "reasoning": "Why removing: which tip and why. Which insights support this.",
  "operations": [
    {"type": "REMOVE", "tip_id": "1"}
  ]
}

No operation needed:

{
  "reasoning": "Why no change. Briefly explain which insights were considered and why none led to an operation.",
  "operations": []
}

Mixed operations (from processing different insights):

{
  "reasoning": "For each operation: why, and which insights support it.",
  "operations": [
    {"type": "ADD", "content": "Tip content from FAILED insights."},
    {"type": "UPDATE", "tip_id": "2", "content": "Improved content from FAILED insights."}
  ]
}

## Important Notes
- Playbook is located between [PLAYBOOK BEGINS HERE] and [PLAYBOOK ENDS HERE].
- Playbook is empty if and only if there's no content between [PLAYBOOK BEGINS HERE] and [PLAYBOOK ENDS HERE].
- Count tips in the playbook carefully. NEVER assume or guess the number of tips.
- Tips should be specific to this task's key characteristics.
- You are given multiple insights from different rollouts of the same task. Always reference specific insights (
      Insight 1, Insight 2, ...) when explaining your reasoning. Do not treat all insights as a single
      undifferentiated block.
```

### C.4 MINISWEBENCH

**Actor.**

```
Use this task-specific playbook to guide repository exploration, repair, and verification.
Treat tips as hypotheses supported by earlier attempts; check them against the current files.

<playbook>
{{playbook}}
</playbook>
```

**Reflector.**

```
Analyze this repository repair attempt. A partial reward does not mean the issue is resolved.
Use only the issue and observed tool outputs. Identify a concrete cause of failure or a useful
successful technique; do not invent tests, source code, or hidden evaluator behavior.

Issue {{instance_id}}:
{{problem_statement}}

Outcome: {{outcome}}
{{trajectory_summary}}

Current playbook:
{{playbook}}

Attempt:
{{conversation_history}}

Return JSON only: {"reasoning": "Evidence linking the actions to the outcome", "key_insight": "One concise,
      actionable lesson for the next attempt"}.
```

**Curator.**

```
Edit the task-specific repository repair playbook using the supplied reflections.
Keep at most ten concise, evidence-based tips. Merge duplicates; remove contradicted tips.
Do not invent hidden test results. Prefer useful file locations, reproduction steps, root
causes, and verification commands over generic advice. An empty playbook permits ADD only.

Current playbook:
```

```
{{current_playbook}}

Reflections:
{{guidebook}}

Return JSON only with "reasoning" and "operations". Each operation must be one of:
{"type": "ADD", "content": "A new actionable tip"}
{"type": "UPDATE", "bullet_id": "1", "content": "Replacement text for that existing tip"}
{"type": "REMOVE", "bullet_id": "1"}
Use the actual existing tip ID for UPDATE/REMOVE, never put an ID inside content.
Use {"reasoning": "No supported improvement", "operations": []} when no edit is needed.
```

# D ALGORITHMIC DETAILS

## D.1 FROM THE HANDOFF OBJECTIVE TO SPARSE COMPONENT UPDATES

**From the future-only objective to delayed credit.** We first derive the policy gradient of Eq. (5), then identify the approximations used by ACE-RL. Write $s = (q, P)$, $X = R_q(\tau)$, and $Y = R_q(\tilde{\tau})$ for the initial and subsequent actor rewards. The input distribution $s \sim \mu$ is held fixed for this local derivation. The objective concerns the value of one sampled candidate; selecting the best of several candidates defines a different outer execution law. Assume differentiable policies, parameter-independent environment dynamics and rewards, and conditions allowing differentiation under the expectation. All scores below are evaluated at the same policy, with role prompts and the deterministic formatter suppressed:

$$\begin{aligned} S_0 &= \nabla_\theta \log \pi_\theta(\tau \mid q, P), & S_{\mathrm{r}} &= \nabla_\theta \log \pi_\theta(z \mid q, P, \tau), \\ S_{\mathrm{c}} &= \nabla_\theta \log \pi_\theta(u \mid P, z), & S_1 &= \nabla_\theta \log \pi_\theta(\tilde{\tau} \mid q, P'). \end{aligned} \tag{9}$$

An actor score sums over its generated action tokens; environment observations contribute no score. Factoring the execution probability into these four policy calls and applying the likelihood-ratio identity (Williams, 1992; Sutton & Barto, 2018) yields

$$\nabla_\theta J_{\mathrm{ACE}} = \mathbb{E}[Y(S_0 + S_{\mathrm{r}} + S_{\mathrm{c}} + S_1)]\,. \tag{10}$$

Parameter sharing does not merge these occurrences: the initial actor affects the evidence available to the handoff, whereas the subsequent actor must use the resulting playbook.

Define $Q_0 = \mathbb{E}[Y \mid s, \tau]$, $Q_{\mathrm{r}} = \mathbb{E}[Y \mid s, \tau, z]$, and $Q_{\mathrm{c}} = \mathbb{E}[Y \mid s, \tau, z, u]$. For an admissible edit, $Q_{\mathrm{c}} = V_\theta(q, P')$, the expected reward of the actor using its candidate. By conditional expectation,

$$\nabla_\theta J_{\mathrm{ACE}} = \mathbb{E}[Q_0 S_0 + Q_{\mathrm{r}} S_{\mathrm{r}} + Q_{\mathrm{c}} S_{\mathrm{c}} + Y S_1]\,. \tag{11}$$

Thus descendants provide delayed credit to each ancestor. For these identities, an inadmissible branch may be assigned terminal payoff zero and no evaluator score. We distinguish that convention from the implementation's admissible-only reflector average below.

**A full comparison tree.** Sample $N$ initial actor trajectories, $M$ reflections per trajectory, $C$ curator outputs per reflection, and $L$ evaluations per candidate. Let $Y_{ijk\ell}$ be a leaf reward. The nested estimates

$$\widehat{Q}^{\mathrm{c}}_{ijk} = \frac{1}{L}\sum_{\ell=1}^{L} Y_{ijk\ell}, \qquad \widehat{Q}^{\mathrm{r}}_{ij} = \frac{1}{C}\sum_{k=1}^{C} \widehat{Q}^{\mathrm{c}}_{ijk}, \qquad \widehat{Q}^{0}_{i} = \frac{1}{M}\sum_{j=1}^{M} \widehat{Q}^{\mathrm{r}}_{ij} \tag{12}$$

are unbiased for their respective conditional values. Consequently,

$$\begin{aligned} \widehat{g}_{\mathrm{full}} = \frac{1}{N}\sum_{i} \widehat{Q}^{0}_{i} S_{0,i} + \frac{1}{NM}\sum_{ij} \widehat{Q}^{\mathrm{r}}_{ij} S_{\mathrm{r},ij} \\ + \frac{1}{NMC}\sum_{ijk} \widehat{Q}^{\mathrm{c}}_{ijk} S_{\mathrm{c},ijk} + \frac{1}{NMCL}\sum_{ijk\ell} Y_{ijk\ell} S_{1,ijk\ell}, \end{aligned} \tag{13}$$

with $\mathbb{E}[\widehat{g}_{\mathrm{full}}] = \nabla_\theta J_{\mathrm{ACE}}$. The result follows by applying the tower property to each term; leaves sharing ancestors need not be independent unconditionally. Siblings at every component share an input, providing the comparison groups for delayed, joint training. Subtracting an action-independent baseline preserves each expected score term. Choosing empirical group standardization and a clipped loss then gives a delayed joint GRPO *surrogate* (Shao et al., 2024; Schulman et al., 2017); this choice is distinct from the exact identity in Eq. (13). Appendix B shows that a chain-wise sampling pattern, one continuation per initial trajectory, is the special case $M = C = L = 1$ of this estimator.

**Conditional handoff learning and the current actor.** The implemented actor uses $X$ as its reward and does not train on candidate-evaluation trajectories. To isolate this credit choice, assume all outputs are admissible, omit filtering and normalization, and replace the implementation's token weighting by equal role-wise means. The resulting idealized direction is

$$\begin{aligned} g_{\mathrm{local}} &= \mathbb{E}[X S_0 + Y S_{\mathrm{r}} + Y S_{\mathrm{c}}], \\ \nabla J_{\mathrm{ACE}} - g_{\mathrm{local}} &= \mathbb{E}[(Y - X) S_0 + Y S_1]. \end{aligned} \tag{14}$$

Current actor credit therefore requires an additional interpretation. Treat the initial trajectory as already collected evidence from a fixed checkpoint $\bar{\theta}$ and define

$$J_{\mathrm{cond}}(\theta;\bar{\theta}) = \mathbb{E}_{\substack{s\sim\mu,\ \tau\sim\pi_{\bar{\theta}} \\ z,u,\tilde{\tau}\sim\pi_\theta}}[Y],$$
$$\nabla_\theta J_{\mathrm{cond}}|_{\theta=\bar{\theta}} = \mathbb{E}[Y(S_{\mathrm{r}} + S_{\mathrm{c}} + S_1)]. \tag{15}$$

The seed distribution is fixed only in this conditional objective; using a fixed checkpoint during rollout collection alone does not remove $S_0$ from Eq. (10).

Playbook persistence explains how learning to use the handoff can be distributed across encounters. For a continuing sequence of the same task, let $P_i$ be the actual incumbent, $\tau_i$ its actor trajectory, and $R_i = R_q(\tau_i)$. The next input $P_{i+1}$ includes whatever selection and reset transition the harness performs. Pathwise,

$$\frac{1}{T}\sum_{i=0}^{T-1} R_{i+1} = \frac{1}{T}\sum_{i=0}^{T-1} R_i + \frac{R_T - R_0}{T}. \tag{16}$$

Since rewards lie in $[0,1]$, the difference is at most $1/T$. When the long-run averages exist, they coincide without requiring identical marginal distributions of $P_i$ and $P_{i+1}$. At one fixed policy, write $S_{a,i} = \nabla_\theta \log \pi_\theta(\tau_i \mid q, P_i)$. The direct reader terms also satisfy

$$\frac{1}{T}\sum_{i=0}^{T-1} R_{i+1}S_{a,i+1} = \frac{1}{T}\sum_{i=0}^{T-1} R_i S_{a,i} + \frac{R_T S_{a,T} - R_0 S_{a,0}}{T}. \tag{17}$$

The expected norm of the boundary vanishes if $\sup_i \mathbb{E}\|R_i S_{a,i}\| < \infty$. Hence a current encounter can supply the direct reader contribution of the preceding *retained* handoff, with matching weighting. This is an identity for that gradient component, not a derivation of the full sequential gradient from convergence of reward averages.

In particular, persistence does not remove the actor's effect on future evidence. For the continuing finite-horizon objective $J_T = \mathbb{E}[\sum_{t=0}^{T} R_t]$, causality gives the contribution from incumbent actor trajectories

$$g_{\mathrm{act},T} = \mathbb{E}\left[\sum_{i=0}^{T} S_{a,i}\left(R_i + \sum_{t=i+1}^{T} R_t\right)\right]. \tag{18}$$

Current-reward training omits the future-return term. A sufficient condition for its expectation to vanish is $\mathbb{E}[\sum_{t=i+1}^{T} R_t \mid x_i, \tau_i] = b_i(x_i)$, where $x_i$ includes the full relevant pre-action state; the weaker condition is orthogonality of future return to the actor score in expectation. Without such an assumption, treating the seed as fixed evidence is an explicit local approximation to the all-policy objective. Similarly, one-handoff rewards do not capture effects of reflection and curation beyond the next attempt.

There are three further limits to applying the temporal identity to training. First, the next encounter visits a retained winner or incumbent, whereas handoff credit uses all evaluated proposals; reindexing does not equate those distributions. Second, resets are valid transitions in Eq. (16), but a pre-reset proposal need not become the next input. Treating each reset interval as a separate proposal chain produces repeated boundary terms. Third, consecutive training encounters use changing policy checkpoints; identifying their score contributions requires an approximation that the policy changes slowly, or an explicit correction. Thus persistence motivates current actor training under the conditional interpretation, while it does not establish equality of the implemented update and the complete delayed gradient.

**What sparse sampling preserves.** For a fixed candidate with independent evaluation episodes,

$$\widehat{V}_L = \frac{1}{L}\sum_{\ell=1}^{L} Y_\ell, \qquad \mathbb{E}[\widehat{V}_L \mid q, P'] = V_\theta(q, P'), \qquad \mathrm{Var}(\widehat{V}_L \mid q, P') = \frac{\mathrm{Var}(Y \mid q, P')}{L}. \tag{19}$$

Setting $L = 1$ preserves the conditional mean and weakly increases its variance. Replacing multiple curator descendants by one has the same property when inadmissible outcomes are counted

consistently. Let $\mathcal{T}$ be a realized full tree, with reflector and curator group contributions $G_i^{\mathrm{r}}$ and $G_{ij}^{\mathrm{c}}$. Uniformly selecting $I \in [N]$ and $H \in [M]$ gives

$$\mathbb{E}_I[G_I^{\mathrm{r}} \mid \mathcal{T}] = \frac{1}{N}\sum_i G_i^{\mathrm{r}}, \qquad \mathbb{E}_{I,H}[G_{IH}^{\mathrm{c}} \mid \mathcal{T}] = \frac{1}{NM}\sum_{ij} G_{ij}^{\mathrm{c}}. \tag{20}$$

These identities preserve raw score means with action-independent baselines, fixed scales, and matching role weights; sums require inverse inclusion-probability weights. Sampling an intact group also preserves the average of its normalized, clipped loss if that group's distribution is unchanged. Reducing the descendant samples used to compute its rewards does not satisfy that qualification.

**Scope of equivalence for the implemented surrogate.** Empirical GRPO normalization is nonlinear. Even mean centering with a sample's own group mean introduces a finite-group factor: for $G$ independent siblings with unbiased rewards and scores $S_i$,

$$\mathbb{E}\left[\frac{1}{G}\sum_i (\widehat{Q}_i - \overline{\widehat{Q}})S_i\right] = \frac{G-1}{G}\,\mathbb{E}[QS]. \tag{21}$$

A leave-one-out mean removes this factor. Let $\sigma_Q$ be the sample standard deviation of the conditional values $Q_{1:G}$. Dividing by the sampled reward standard deviation further implies, in general,

$$\mathbb{E}\left[\frac{\widehat{Q}_i - \overline{\widehat{Q}}}{\widehat{\sigma}+\epsilon}\,\middle|\, Q_{1:G}\right] \neq \frac{Q_i - \bar{Q}}{\sigma_Q + \epsilon}. \tag{22}$$

Thus one evaluation retains the same surrogate form and an unbiased value estimate, but can change the expected normalized update as well as its variance. Clipping adds a further surrogate approximation; the raw score identities apply at the collection policy with clipping inactive and the stipulated credit and weighting.

The implementation also averages reflector reward only over admissible edits. If independent edits are admissible with probability $p$ and their mean conditional value is $v$, $n$ descendants have expected valid-only average $[1-(1-p)^n]v$, counting an empty valid set as zero. Its mean depends on $n$; counting invalid edits as zeros in the original denominator would instead give $pv$ for every $n$. Finally, a loss divided by the total retained token count changes role weights when branches are removed. Selecting among usable reflections and filtering groups can change inclusion probabilities as well. Exact sampling identities therefore do not imply equivalence of this random ratio to the full-tree estimator. The implemented method is a sparse component-wise GRPO surrogate with conditional handoff credit and current actor learning; its sampling and credit approximations are explicit rather than an asserted equality of expected gradients.

**From the full tree to the sparse tree.** The full tree above samples $M$ reflections of every actor rollout, $C$ edits of every reflection, and $L$ evaluations of every candidate. Three reductions yield the implemented scheme.

- *One reflector group.* All $N$ rollouts are kept for actor training, but only one uniformly chosen seed is reflected on. This preserves a same-input reflector group and removes the factor $N$ from all downstream sampling.
- *One curator group.* One uniformly chosen usable reflection receives $C$ edits and every other usable reflection receives one. This preserves a same-input curator group and still supplies a downstream value for every reflection.
- *One evaluation per candidate.* Replacing an average of $L$ evaluations by one preserves the candidate's conditional expected verifier reward, at higher conditional variance. Empirical standardization and the implementation choices discussed above qualify equivalence of the resulting updates.

Per task and training step, the full tree with $(N, M, C, L) = (8, 4, 4, 1)$ generates 8 actor rollouts, 32 reflections, 128 curator edits, and 128 candidate evaluations, 296 outputs in total. The three reductions bring this to $8 + 4 + 7 + 7 = 26$ outputs for ACE-RL, and the chain-wise design of Appendix B uses $8 + 8 + 8 + 8 = 32$. Figure 8 reports the measured token cost of the implemented methods.

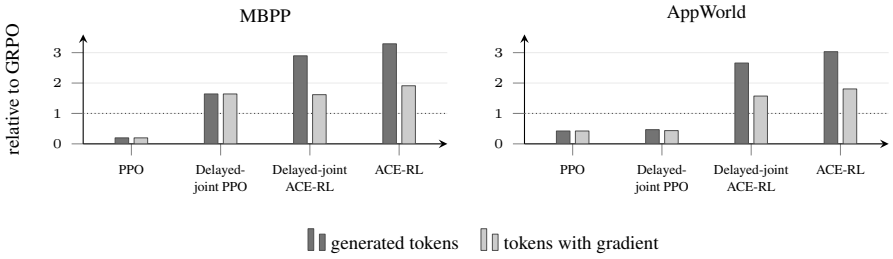



Figure 8: **Tokens per training step relative to GRPO** (dotted line), mean over training steps, for the benchmarks where every method logs token counts. Dark: generated tokens; light: tokens that receive a gradient. PPO logs half as many trajectories per step as GRPO in our runs and generates 0.2–0.4× its tokens; delayed-joint PPO generates 1.6× on MBPP and 0.47× on AppWorld. The two ACE-RL designs generate 2.6–3.3× as many tokens as GRPO, of which 55–60% receive a gradient; the rest are candidate-evaluation rollouts and filtered groups, which are generated but not trained on.

**Algorithm 1** ACE-RL: one training step.

**Require:** task batch $\mathcal{Q}$; policy $\theta$; per-task playbooks $\{P_q\}$ and encounter counts $\{n_q\}$; initial playbook $P_0$; group sizes $(N, M, C)$; reset interval $F$

$\mathcal{B} \leftarrow \varnothing$; $\theta_{\text{old}} \leftarrow \theta$
**for** each task $q \in \mathcal{Q}$ **do**
  $n_q \leftarrow n_q + 1$; **if** $n_q \bmod F = 0$ **then** $P_q \leftarrow P_0$ ▷ periodic reset
  Sample $N$ actor rollouts $\tau_1{:}\tau_N \sim \pi_{\theta_{\text{old}}}(\cdot \mid \iota_{\text{act}}(q, P_q))$; rewards $R_q(\tau_j)$
  Pick seed $j^\star \sim \text{Unif}[N]$; sample $M$ reflections $z_1{:}z_M$ of $\tau_{j^\star}$
  If any reflections are usable, choose $z_h$ uniformly; sample $C$ edits of $z_h$ and one of each other usable reflection
  Apply admissible edits to copies of $P_q$; score each candidate with one fresh rollout (Eq. 6)
  Assign component rewards (Eq. 7); update $P_q$ (Eq. 23)
  Add actor, reflector, and curator outputs with rewards and group ids to $\mathcal{B}$
**end for**
Filter groups (Appendix D); normalize advantages within each role's group (Section 3.2)
Update $\theta$ on $\mathcal{B}$ with the clipped token-level surrogate (Appendix D)

### D.2 Playbook Selection, Reset, and the Training Step

Training credit and memory selection are different decisions. The rewards above rank sampled outputs for learning; separately, the harness must choose which playbook task $q$ carries into its next encounter. Let $s = R_q(\tau_{j^\star})$ be the seed's reward. Among admissible candidates whose value is at least $s$, we retain the one with the highest value (ties broken uniformly), and otherwise keep $P$:

$$P^{\text{next}} = \begin{cases} \arg\max_{P'} \ \widehat{V}(P') & \text{if } \exists P' : \ \widehat{V}(P') \geq s, \\ P & \text{otherwise.} \end{cases} \tag{23}$$

Rejecting an update does not erase its evaluated rewards; the corresponding outputs remain eligible for training subject to group filtering. Because edits accepted under early, weak parameters can persist and mislead later actors, we also reset each task's playbook to $P_0$ every $F$ training encounters of that task ($F = 3$ in all experiments). The reset bounds the age of retained experience and repeatedly exposes training to the empty-playbook condition that evaluation starts from. Algorithm 1 summarizes one training step.

### D.3 Implementation Details

**Sampling and provenance.** The main configuration samples $N = 8$ actor rollouts per task with the current playbook, selects one seed uniformly, samples $M = 4$ reflections of the seed, selects one usable reflection uniformly to receive $C = 4$ curator samples, and gives each remaining usable reflection one curator sample. With all four reflections usable this yields 8 actor, 4 reflector, and 7

curator outputs and up to 7 candidate playbooks, each scored by one fresh actor episode on a newly initialized environment for the same task. Candidate-evaluation trajectories are not added to the actor's training group; the actor trains on its 8 seed rollouts. Every candidate records its source task, seed trajectory, and reflection, so downstream values can be attributed to the outputs that produced them. Reflector and curator generations use the same sampling temperature (0.7) as the actor.

**Reflector input.** The benchmark adapter formats the seed trajectory for the reflector as a *trajectory summary* (outcome, tests passed or verifier result, number of turns, termination reason) followed by the full *trajectory record* (the actor's messages interleaved with environment observations), truncated to a history budget of 20k tokens. The reflector prompt asks for a three-step analysis (extract facts, explain the actor's behaviour, classify the case: first-turn success, later self-correction, knowledge gap, or ineffective revision) and a JSON object with `outcome`, `reasoning`, and `key_insight`. Prompts are listed in Appendix C.

**Playbook and curator edits.** A playbook is a list of lines `[id] tip`. The curator returns a JSON list of operations, each `ADD` (new tip), `UPDATE` (rewrite tip by id), or `REMOVE` (by id). Operations are applied in order, duplicates are removed, and ids are renumbered. A curator output is *admissible* when it parses, contains at least one operation that applies to the current playbook, and the resulting playbook is at most 2,048 tokens; otherwise it receives reward 0, is excluded from the set $\mathcal{C}(z)$ used to average its reflection's reward, and produces no candidate. The initial playbook is empty on every benchmark.

**Comparison groups and filtering.** Actor outputs share a group per task; the four reflections share a group keyed by the seed trajectory; the four curator outputs of the selected reflection share a group keyed by that reflection; the other curator outputs are singletons. Before normalization, any group whose rewards are all $\leq 0$ or all $\geq 1$ is dropped, and singletons are dropped with it under binary rewards. For MiniSWEBench, whose shaped rewards can be equal but strictly between 0 and 1, the filter instead requires a nonzero within-group range. With the threshold filter and fractional rewards, as in MBPP, a singleton can survive; the backend then uses mean 0 and scale 1, producing an outcome-weighted update rather than a group comparison. Constant multi-output groups that survive the threshold filter have zero centered advantage. Rollout batches are over-generated by a factor of two so that enough informative groups survive the filter (Yu et al., 2025b).

**Objective.** Each retained output $i$ receives the scalar advantage $A_i = (r_i - \mu_G)/(\sigma_G + \epsilon)$, where $\mu_G$ and $\sigma_G$ are the mean and standard deviation of rewards in its role's group $G$ (Section 3.2), broadcast over its generated tokens. Writing $\rho_{i,t}(\theta) = \pi_\theta(y_{i,t} \mid x_{i,t})/\pi_{\theta_{\text{old}}}(y_{i,t} \mid x_{i,t})$ and $M_{i,t}$ for the loss mask over generated tokens, the shared policy maximizes

$$\mathcal{L}_{\text{GRPO}}(\theta) = \frac{\sum_{i\in\mathcal{B}}\sum_t M_{i,t}\,\ell_{\text{clip}}\big(\rho_{i,t}(\theta), A_i\big)}{\sum_{i\in\mathcal{B}}\sum_t M_{i,t}}, \qquad \mathcal{B} = \mathcal{B}^{\text{act}} \cup \mathcal{B}^{\text{ref}} \cup \mathcal{B}^{\text{cur}}, \tag{24}$$

with the asymmetric clipped surrogate $\ell_0(\rho, A) = \min\{\rho A, \text{clip}(\rho, 1-\delta_-, 1+\delta_+)A\}$, $(\delta_-, \delta_+) = (0.2, 0.28)$ (Yu et al., 2025b), and a dual clip $\max\{\ell_0, cA\}$ with $c = 3$ for $A < 0$. Token-mean aggregation means roles are weighted by their token counts; we do not impose explicit role weights. There is no KL penalty, no entropy bonus, and no critic. Candidate values and advantages are constants with respect to $\theta$.

**Selection and reset.** Playbook selection follows Eq. (23) (Appendix D.2). Each task's playbook is reset to empty before its 3rd, 6th, ... training encounter; the counter is per task, not per optimizer step, and the evaluation store is never reset by the training schedule.

**Hyperparameters.** All methods use the same actor rollout count, token budgets, learning rate, and clip settings within a benchmark. The outputs of all trained roles are concatenated into one batch and updated jointly with Eq. (24); the PPO-based methods replace the group-relative advantage with GAE from a critic. Methods differ only in what defines them: the advantage estimator (group-relative or GAE), the group filter (off for critic-based methods), the number of evaluations per candidate (eight for delayed-joint PPO, one otherwise), and the actor training source (seed rollouts, except delayed-joint PPO, which trains on the accepted candidate's rollouts). Table 6 lists shared optimizer and rollout settings; Table 7 lists per-benchmark budgets.

Table 6: Shared training hyperparameters. Baseline-specific settings: PPO and delayed-joint PPO add a critic initialized from the policy (learning rate $10^{-5}$, GAE with $\gamma = 1$, $\lambda = 0.95$, 10 warm-up steps) and disable group filtering so the critic sees constant-reward batches.

| Setting | Value |
|---|---|
| Tasks per step / actor rollouts per task | 8 / 8 |
| Reflector, curator samples $(M, C)$ | 4, 4 |
| Candidate evaluations per candidate | 1 |
| Playbook reset interval $F$ | 3 encounters |
| Playbook length cap | 2,048 tokens |
| Sampling temperature (all roles, train and eval) | 0.7 |
| Actor learning rate | $10^{-6}$ |
| PPO epochs per batch / mini-batch | 1 / 8 tasks |
| Clip $(\delta_-, \delta_+)$, dual clip $c$ | $(0.2, 0.28)$, 3 |
| KL loss / entropy bonus | none / none |
| Loss aggregation | token mean |
| Group over-generation factor | 2 |
| Reported training horizon | 300 (Mastermind, MBPP), 400 (AppWorld), 60 (MiniSWEBench) |
| Checkpoint and evaluation interval | every 10 steps (5 on MiniSWEBench) |
| Training runs per method | 1 |
| Hardware per run | 2 GPUs (H100 or H200; H200 for AppWorld and MiniSWEBench) |

Table 7: **Benchmark settings.** Turn limits are per actor episode; timeouts are per training trajectory.

| Benchmark | Model | Reward per episode | Turns | Timeout | Train / held-out | Playbook content |
|---|---|---|---|---|---|---|
| Mastermind | Qwen3-0.6B (non-thinking) | best guess score $(e + v/2)/L$; 1 when solved | 20 | 30 min | 200 / 100 | constraints implied by prior guesses |
| MBPP | Qwen3-0.6B (non-thinking) | best fraction of the 3 unit tests passed | 6 | 30 min | 374 / 100 | failing assertions, edge cases |
| AppWorld | Qwen3-4B-Instruct-2507 | 1 iff the task's state checks pass | 30 | 20 min | 90 / 57 (dev) | API dependencies, entity bookkeeping |
| MiniSWEBench | Qwen3-4B-Instruct-2507 | 0 / 0.05 / 0.2 / 1 (see text) | 30 | 2 h | 100 / 100 (Verified) | relevant files, failure causes |

## E Benchmarks and Reward Design

We use four benchmarks that span short feedback loops and long-horizon tool use (Table 7). All roles share one model per benchmark, the playbook is initially empty, and every reward lies in $[0, 1]$. The information a playbook can carry differs across benchmarks: implied constraints on the hidden code, failing assertions and edge cases, API dependencies and entity bookkeeping, and relevant files and failure causes. The 0.6B model is small enough that recursive self-improvement is weak before training (Figure 2); the 4B instruct model tests whether an already post-trained model shows the same train–inference mismatch.

**Mastermind.** Each game draws a secret code of length $L \in \{3, 4, 5\}$ over digits $\{0, \ldots, D-1\}$ with $D \in \{4, 5, 6\}$, from a fixed generator seed; the 200 training and 100 held-out games share no seeds. The agent submits $L$ space-separated digits per turn and observes the number of exact matches $e$ (right digit, right position) and value matches $v$ (right digit, wrong position, with multiset accounting). A guess scores $(e + v/2)/L$, so a solved game scores 1; a malformed guess scores 0 and returns an error observation. The episode reward is the best guess score, and a game ends on an exact match or after 20 turns.

**MBPP.** We use the full MBPP release (Austin et al., 2021), training on its 374-problem train split and evaluating on the first 100 problems of its test split. Each problem has three unit tests. The actor submits a program in a code block each turn; each assert runs in a subprocess with a 10-second timeout, and a turn's reward is the fraction of asserts that pass. After a failed turn the environment shows up to three failing asserts with their error messages, and the episode ends when all tests pass or after six turns. The episode reward is the best turn reward, so held-out rewards are multiples of $1/3$ per task. Because the displayed asserts are the ones that are scored, a harness can carry them forward; every method sees the same feedback.

**AppWorld.** The actor writes Python that calls app APIs through the AppWorld server and finishes by calling the task's completion API (Trivedi et al., 2024). The episode reward is 1 if AppWorld's state-based evaluator reports success for the task and 0 otherwise, including when the actor runs out of its 30 turns. We train on the 90 tasks of the train split and evaluate on the 57 tasks of the dev split.

**MiniSWEBench.** Tasks are GitHub issues solved in the mini-SWE-agent scaffold[3] (Yang et al., 2024): the actor inspects and edits a repository through shell commands in a container and submits within 30 steps. Training uses 100 instances of a SWE-Gym subset and evaluation the first 100 instances of SWE-bench Verified (Jimenez et al., 2023), both graded with the official evaluation harness when the episode ends. The shaped reward is 0 when the final diff is empty, 0.05 when a diff exists but grading fails to run, 0.2 when the diff is graded but the issue is not resolved (including diffs that do not apply), and 1 when all fail-to-pass and pass-to-pass tests pass.

**Rollout accounting at evaluation.** Vanilla evaluation draws 8 independent actor episodes per task. Harnessed evaluation runs 8 sequential visits per task, each with one actor episode, one reflector call, and one curator call; no additional rollout scores the curator's edit. RSI@$K$ and pass@$K$ therefore use the same $K$ actor episodes, and RSI@$K$ additionally uses $K$ reflector and $K$ curator calls.

**Turn-level curves.** For Figure 10 the reward trace of each task is padded to the turn limit with its last reward, and a task that has been solved is counted as solved in later visits; the traces are then averaged over tasks per turn. This is a plotting convention and adds no interaction.

# F Additional Results and Diagnostics

## F.1 Statistical Analysis

Every evaluation draws 8 actor episodes per held-out task: 8 independent samples in vanilla mode and 8 sequential visits in harnessed mode (Section 4.2). For each task $i \in \{1, \dots, n\}$ we first compute the task-level statistic $r_i$ from its 8 episodes: the mean reward for pass@1, the expected best of $K$ samples drawn without replacement for pass@$K$, and the best of the first $K$ visits for RSI@$K$. We then report

$$\bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i, \qquad \mathrm{SEM} = \sqrt{\frac{1}{n(n-1)}\sum_{i=1}^{n}(r_i - \bar{r})^2}, \qquad \mathrm{CI}_{95} = \bar{r} \pm 1.96\,\mathrm{SEM}. \tag{25}$$

These intervals summarize variation in the task-level statistics, conditional on their sampled episodes, but not over training runs: each method was trained once. Paired comparisons between two methods on the same task set use the per-task differences $r_i^A - r_i^B$ in Eq. (25). Task counts $n$ per benchmark are listed in Table 7.

## F.2 Training Reward

## F.3 Turn-Level Curves

## F.4 Handoff Diagnostics over Training

## F.5 Critic Diagnostics

[3] https://github.com/SWE-agent/mini-swe-agent

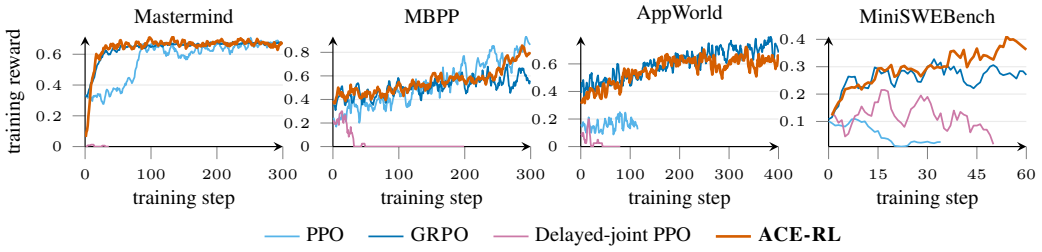



Figure 9: **Training reward** (centered five-step mean) of the methods in Figure 4. It is not comparable across methods: actor-only methods are scored with an empty playbook, harness methods under the inherited playbook.

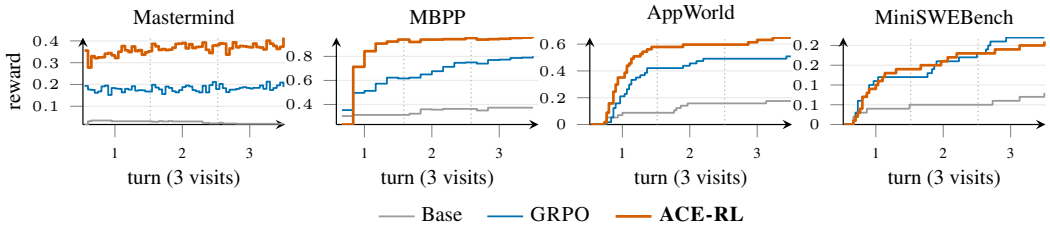



Figure 10: **Turn-level reward over the first three harnessed visits**, at the last checkpoint evaluated for both GRPO and ACE-RL (Mastermind and MBPP step 290, AppWorld 390, MiniSWEBench 55), and for the base model. Dotted lines mark visit boundaries. Within a visit the curve shows how quickly the actor gains reward; at a boundary the step shows what the handoff contributed. ACE-RL is above GRPO at nearly every turn on Mastermind, MBPP, and AppWorld; on MiniSWEBench the curves overlap.

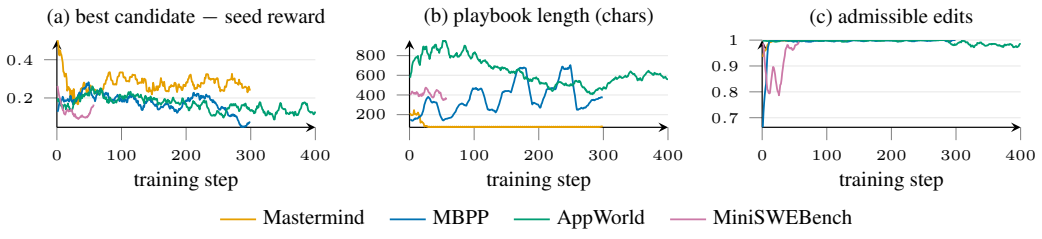



Figure 11: **Handoff diagnostics for ACE-RL** from the training logs (ten-step centered mean). **(a)** Value of the best candidate playbook minus the seed trajectory's reward. **(b)** Length of the retained playbooks. **(c)** Fraction of curator outputs that produce an admissible candidate. The best candidate improves on the seed throughout training on every benchmark; playbooks stay far below the 2,048-token cap; and nearly every curator output is admissible after the first tens of steps.

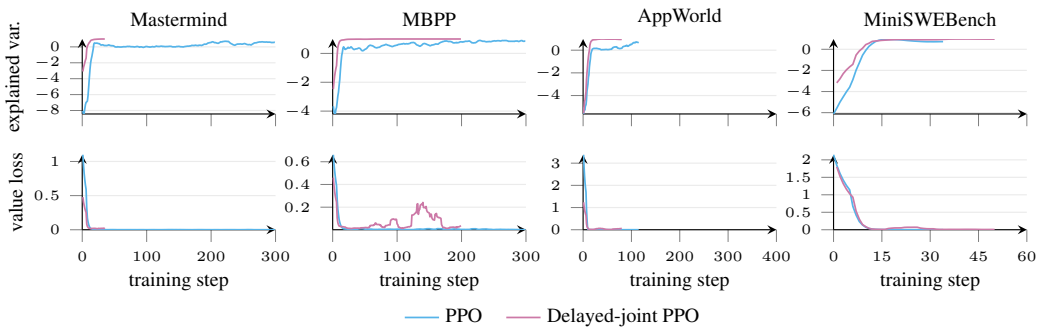



Figure 12: **Critic diagnostics of PPO and delayed-joint PPO** from the training logs (ten-step centered mean). Top: explained variance of the value function on the training batch. Bottom: value loss. Both critics start with negative explained variance and fit the returns within the first tens of steps, reaching 0.55–1.0. On MBPP the delayed-joint critic's loss rises again at steps 110–150, after the policy's reward has collapsed (Section 5.1). A well-fit critic therefore does not prevent the collapse.

# G Qualitative Material

## G.1 What the Trained Playbooks Contain

We inspected the playbooks that the final ACE-RL checkpoints write during the harnessed evaluation (after the third visit, all held-out tasks). On AppWorld and MiniSWEBench they are task-specific: the 57 AppWorld playbooks hold 133 tips, of which 132 are distinct, and the 100 MiniSWEBench playbooks hold 245 distinct tips. The tips name APIs, entities, files, and failure causes observed in earlier visits. On the two 0.6B benchmarks the handoff carries little task information. On Mastermind every tip written by the trained curator is the same template string, "Tip content from FAILED insights." (300 tips over 100 tasks), and the delayed-joint design behaves the same way; the base model and the GRPO checkpoint write generic but contentful tips about using feedback and avoiding repeated guesses. On MBPP most tasks are solved at the first visit, and the 15 tips written over 100 tasks restate the reflector's case label (for example, "Tip content from Case 3 insights: ..."). The Mastermind and MBPP gains of ACE-RL therefore come mainly from an actor trained to work inside the harness rather than from the content of the handoff (Section 5.3).

## G.2 Examples of a Helpful Handoff

Each example is a held-out task from the harnessed evaluation of the final ACE-RL checkpoint, where the first visit fails, the curator writes the tip shown, and the following visits succeed.

**AppWorld (task 383cbac_1; rewards over visits 1–4: 0, 1, 1, 1).** After the failed first visit, the curator writes:

> [1] Payments to colleagues (e.g., sending money to Spencer) represent personal contributions, not the manager's total payment. If the manager paid for everyone, then everyone else should have sent money to the manager, not to a colleague. The presence of payments to colleagues means the manager's total payment must be calculated as the sum of all payments from others to the manager, plus the personal contribution to colleagues.

The following visits, whose playbooks begin with this tip, complete the task.

**MiniSWEBench (task django__django-11179; rewards over visits 1–4: 0, 1, 1, 1).** After the failed first visit, the curator writes:

> [1] Always verify that the reproduction script runs successfully in a valid Django environment before attempting to verify a fix. This includes checking for required settings, database configuration, or model registration. Without a working environment, any verification step is invalid and the fix cannot be reliably assessed.

The following visits, whose playbooks begin with this tip, resolve the issue.

# H Limitations

ACE-RL requires a harness whose handoff artifacts can be resampled and evaluated in isolation; agents that entangle acting, summarizing, and remembering in one generation fall back to chain-level credit, which our delayed-joint variant approximates. Candidate evaluation makes each training step more expensive than a GRPO step (Figure 8). The playbook is task-local, and we do not study transfer of experience across tasks. Our evidence concerns models up to 4B parameters and four benchmarks; whether the same recipe is needed, or sufficient, at frontier scale is open. Finally, training models to pass experience to their future selves in autonomous software and interface agents raises safety questions about scoping, monitoring, and release that our task-local, verifier-bounded setting does not resolve.