CoVER: Reinforcement Learning with Code-Verified Process Rewards in Stateful Code Execution Environments

Anonymous authors
Paper under double-blind review

Abstract

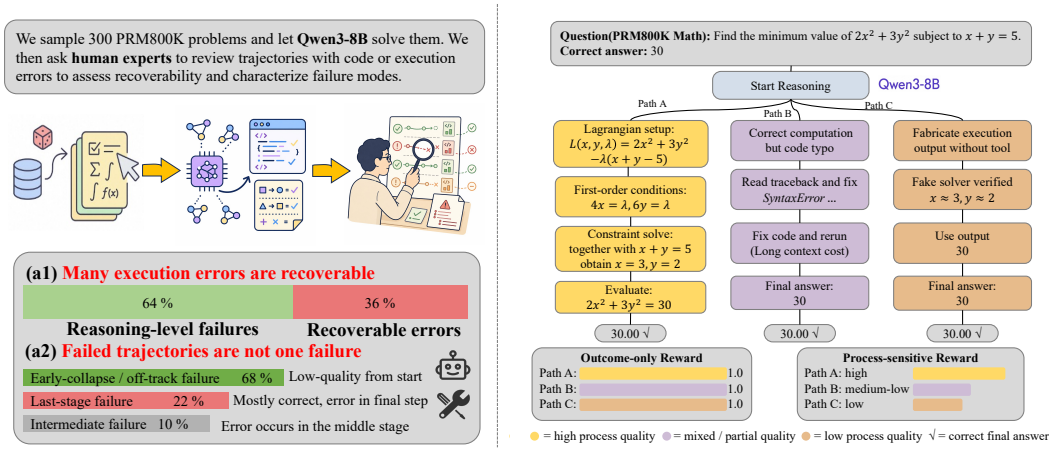
Large language models increasingly rely on external code execution for mathematical and code-intensive reasoning. However, most RL post-training assigns credit only through final-answer correctness, while prior process-supervision methods are largely designed for static traces and often require costly step labels or learned evaluators. We introduce **CoVER** (**Co**de-**Ver**ified Process **R**ewards), a reinforcement learning framework that turns stateful code execution into online process supervision. CoVER trains LLMs through iterative execution and self-correction in a persistent sandbox, using lightweight process rewards derived from execution-grounded sandbox signals. To optimize mixed agent–environment trajectories, we propose PR-GRPO, a process-reward variant of GRPO that applies masked log-probabilities and masked KL regularization only to policy-generated tokens and assigns credit with depth-wise step-level advantages. We construct CoVERBench by standardizing public reasoning sources and solver-verified tasks into a unified benchmark spanning mathematical reasoning, logical deduction, and constraint satisfaction. On CoVERBench, CoVER improves Qwen3-8B to 93.7% held-out validation accuracy, versus 82.1% for outcome-only GRPO under a matched protocol, with non-overlapping bootstrap confidence intervals. CoVER also transfers to unseen math and code benchmarks.

1 Introduction

Large language models (LLMs) are shifting from static text generation toward interactive agentic systems that use external computational environments for complex problem solving (Mialon et al., 2023; Qin et al., 2024; Wang et al., 2024a; Yao et al., 2023). Yet tasks requiring rigorous mathematical deduction, logical planning, and multi-step conditional reasoning remain challenging (Valmeekam et al., 2023; Bubeck et al., 2023): autoregressive decoding relies on implicit parametric knowledge and is brittle where symbolic precision, long-horizon state tracking, and exact numerical computation are required (Dziri et al., 2023; Saparov & He, 2022; Gou et al., 2023).

To enhance mathematical and logical reasoning, prior work has explored inference-time prompting strategies such as Chain-of-Thought (CoT) (Wei et al., 2022), Plan-and-Solve (Wang et al., 2023b), Chain of Code (Li et al., 2023), and Parsel (Zelikman et al., 2023), as well as post-training methods such as PPO (Schulman et al., 2017), DPO (Rafailov et al., 2023), GRPO (Shao et al., 2024), CodeRL (Le et al., 2022), and ReTool (Feng et al., 2025a). However, most RL pipelines rely on final-answer correctness or outcome-level rewards (Ouyang et al., 2022; Christiano et al., 2017; Xie et al., 2025b; Chua & Evans, 2025; Arcuschin et al., 2025), which provide sparse supervision for long-horizon reasoning. Process reward models (PRMs) instead assign feedback to intermediate reasoning steps and have improved mathematical reasoning (Uesato et al., 2022; Lightman et al., 2024; Wang et al., 2024b; Xu et al., 2025; Yao et al., 2026). Yet existing PRM approaches are largely designed for static text traces and often depend on human annotations, learned reward models, or probabilistic step evaluators. They therefore do not directly address stateful tool-integrated reasoning (TIR) (Xue et al., 2025; Ma et al., 2024), where low-cost supervision must be grounded in execution feedback, recovery behavior, and interaction efficiency.

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.



(a) Recoverable failures

(b) Same answer, different process

Figure 1: **Stateful TIR requires execution-grounded process supervision.** On 300 PRM800K (Lightman et al., 2024) problems solved by Qwen3-8B (Yang et al., 2025), 36% of execution-error trajectories are locally recoverable, and incorrect trajectories fail at heterogeneous stages. Same-answer trajectories also differ in process quality that final-answer rewards and static trace evaluation miss.

This gap becomes especially severe because such trajectories interleave model reasoning, executable actions, environment feedback, and subsequent revisions (Schick et al., 2023; Wang et al., 2023c; Zhou et al., 2023; Nakano et al., 2021; Zeng et al., 2024; Yang et al., 2023). In Figure 1a, 36% of execution-error trajectories are locally recoverable and incorrect trajectories fail at heterogeneous stages, so a single final reward obscures partial progress and local mistakes. Final correctness also masks large differences in process quality (Shinn et al., 2023; Liu et al., 2023b), as Figure 1b shows for three trajectories with the same correct answer. These findings motivate process-level, execution-grounded supervision generated online from the sandbox itself rather than dense human step annotations (analysis in Appendix F.3).

To bridge this gap, we propose **CoVER** (**Co**de-**Ver**ified Process **R**ewards), a reinforcement learning framework for stateful TIR that trains LLMs through iterative code generation, execution, and self-correction in a persistent sandbox. Unlike conventional PRMs requiring step labels or learned evaluators, CoVER derives lightweight process rewards from execution-grounded sandbox signals: execution success, format compliance, claim grounding against execution evidence, and interaction efficiency. This makes the supervision dynamic, grounded, and inexpensive to obtain once a task verifier and sandbox interface are available. To optimize such mixed agent–environment trajectories, we introduce **PR-GRPO**, a process-reward variant of GRPO with masked log-probabilities and masked KL regularization over policy-generated tokens only, together with depth-wise step-level advantages for multi-turn credit assignment.

We construct CoVERBench by standardizing established public reasoning sources and solver-verified logic tasks into a unified stateful code-execution protocol, spanning mathematical reasoning, logical deduction, and constraint satisfaction. Starting from Qwen3-8B, CoVER achieves **93.7%** accuracy on CoVERBench, compared to 82.1% for standard GRPO, and approaches the performance of strong proprietary models such as GPT-5 (Singh et al., 2025) on the same benchmark. We further evaluate CoVER on unseen benchmarks, including representative mathematical reasoning tasks such as AIME 2022-2025 AIME (2025) and GSM8K (Cobbe et al., 2021), as well as code-generation tasks such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), among others. CoVER performs well across these tasks, suggesting that code-verified process supervision improves not only tool interaction on CoVERBench, but also broader mathematical reasoning and executable problem-solving behavior.

Our contributions can be summarized as follows:

- We propose **CoVER**, an RL framework that turns stateful code execution into online process supervision, enabling LLMs to learn from verifiable execution feedback without dense human step annotations.

- We introduce **PR-GRPO**, a process-reward training algorithm for mixed agent–environment trajectories that jointly integrates verified transition rewards, policy-token masking, and depth-wise credit assignment.
- We construct **CoVERBench**, a standardized multi-domain benchmark that aggregates public reasoning sources and solver-verified tasks into a unified stateful code-execution protocol.
- We achieve strong results across mathematical reasoning, logical deduction, and code generation, while consistently outperforming standard GRPO and improving transfer to unseen benchmarks.

2 Background and Preliminaries

Process Reward Models and Tool-Integrated Reasoning. While outcome-based RL provides sparse credit signals in complex logical tasks (Stiennon et al., 2020; Liu et al., 2025; Wang et al., 2022; Madaan et al., 2023), PRMs address this sparsity through step-level supervision over intermediate reasoning processes (Zhang et al., 2025b). Recent methods reduce the annotation cost of early PRMs with automated process feedback, including self-rewarding label synthesis (Yuan et al., 2024; Wang et al., 2024b; Hosseini et al., 2024), confidence regularization (Zhao et al., 2025; Prabhudesai et al., 2025), and deterministic step-wise judgments from symbolic engines or theorem provers (Jha et al., 2024; Leang et al., 2025). In parallel, TIR offloads strict mathematical deduction and state tracking from LLMs to external environments (Parisi et al., 2022; Lu et al., 2023). PAL (Gao et al., 2023) and PoT (Chen et al., 2022) show that executing generated code provides deterministic feedback, and agent frameworks integrate such execution into open-ended task solving (Yin et al., 2023; Cai et al., 2023; Wang et al., 2023a).

Despite these advances, the combination of process supervision with stateful, multi-turn TIR remains underexplored. In execution environments, agents produce interleaved traces of textual reasoning, tool invocations, and environment observations (Huang et al., 2022; Hao et al., 2023). Policy optimization must therefore assign credit over dynamic execution trajectories rather than static text traces. CoVER addresses this gap by using code-verified execution states as online process rewards for optimizing multi-turn interactions. Prior and concurrent work also studies finer-grained credit assignment for multi-turn tool use, including turn-level reward design (Wei et al., 2025), GiGPO (Feng et al., 2025b), ToolRL (Qian et al., 2025), tool-call reward models (Ma et al., 2026), MatchTIR (Qu et al., 2026), GTPO (Ding et al., 2025), and THOR (Chang et al., 2025), as well as execution-feedback RL for code (Gehring et al., 2024; Li et al., 2025). CoVER differs in deriving deterministic per-turn rewards from verified transitions of a persistent sandbox rather than propagating terminal outcomes or matching reference tool calls, in masking environment-generated tokens out of the objective, and in normalizing credit among rollouts at the same interaction depth. Appendix H compares the designs.

Group Relative Policy Optimization. PPO (Schulman et al., 2017; Zheng et al., 2023) requires a value network whose memory and compute are costly at LLM scale (Touvron et al., 2023; Grattafiori et al., 2024; Team et al., 2024); GRPO (Shao et al., 2024) removes the critic by normalizing rewards within a sampled group. For a prompt $x \sim \mathcal{D}$, G completions $\{y_i\}_{i=1}^{G}$ are sampled from $\pi_{\theta_{\text{old}}}$, and each receives a scalar reward $r(x, y_i)$ from a reward model or a rule-based verifier (Lee et al., 2023; Bai et al., 2022; Xie et al., 2025b). The group-relative advantage $\hat{A}_i = (r(x, y_i) - \mu_r)/\sigma_r$, with group reward mean μ_r and standard deviation σ_r, is assigned to every token of y_i. The policy then maximizes a PPO-style clipped surrogate over token-level ratios $\rho_{i,t}(\theta)$ with a KL penalty toward π_{ref} (Appendix C.5). This suffices for completion-level supervision but not for stateful multi-turn TIR, where credit must be assigned across intermediate actions and execution states.

3 Process-Reward Policy Optimization Problem

Problem Formulation. We model stateful multi-turn code execution as a finite-horizon POMDP (Zhang et al., 2025a) $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \Omega, \mathcal{O}, \gamma \rangle$, where the latent state $s_t \in \mathcal{S}$ includes the input problem and the internal state of the persistent sandbox, and $\gamma \in (0, 1]$ is the discount factor. The policy does not observe s_t directly but instead conditions on the serialized interaction history $h_t = (o_1, a_1, o_2, a_2, \ldots, o_t)$, where observations $o_t \sim \mathcal{O}(\cdot \mid s_t)$ contain text-visible sandbox feedback such as `stdout`, `stderr`, or tracebacks. At each turn, the policy samples an assistant action $a_t = (y_{t,1}, \ldots, y_{t,L_t}) \sim \pi_\theta(\cdot \mid h_t)$, which contains natural-language reasoning, executable

code, or a final boxed answer. Code actions trigger sandbox execution, while a final boxed answer terminates the episode. The environment then evolves as

$$s_{t+1} \sim \mathcal{P}(\cdot \mid s_t, a_t), \qquad o_{t+1} \sim \mathcal{O}(\cdot \mid s_{t+1}). \tag{1}$$

Each turn receives a process reward $r_t^{\text{proc}} = \mathcal{R}_{\text{proc}}(s_t, a_t, s_{t+1}, o_{t+1})$ computed from execution feedback. At termination, an outcome reward $r_{\text{out}} = \mathcal{R}_{\text{out}}(\tau)$ scores final-answer correctness. A trajectory is therefore

$$\tau = (o_1, a_1, r_1, o_2, a_2, r_2, \ldots, o_{T_\tau}, a_{T_\tau}, r_{T_\tau}), \qquad T_\tau \le K, \tag{2}$$

where K is the maximum interaction horizon and the terminal reward includes both $r_{T_\tau}^{\text{proc}}$ and r_{out}.

Masked Policy Optimization. CoVER trajectories interleave policy-generated action tokens with environment-generated observation tokens. Since observations are produced by the environment kernel $\mathcal{O}$ rather than π_θ, sequence-level likelihoods over all tokens are not valid policy likelihoods. We flatten each trajectory into tokens $O = (y_1, \ldots, y_{|O|})$ and set a mask $M_k \in \{0, 1\}$, with $M_k = 1$ only for policy-generated tokens. All policy-dependent quantities are computed under this mask. We use the masked sampled KL penalty:

$$D_{\text{KL}}^{M}(\pi_\theta \| \pi_{\text{ref}}) = \frac{1}{\sum_{k=1}^{|O|} M_k} \sum_{k=1}^{|O|} M_k \left(\log \pi_\theta(y_k \mid y_{<k}) - \log \pi_{\text{ref}}(y_k \mid y_{<k}) \right). \tag{3}$$

Masking restricts both likelihoods and KL regularization to policy tokens (Appendices C.2–C.3).

Step-Level Credit Assignment and the PR-GRPO Objective. A single sequence-level advantage for the whole trajectory, as in standard GRPO, mismatches credit in multi-turn settings: an effective early code execution can be penalized because a later reasoning step fails. We therefore compute a step-level advantage from the per-turn rewards r_t. The return-to-go at turn t of trajectory i is $R_{i,t} = \sum_{j=t}^{T_i} \gamma^{j-t} r_{i,j}$. The step-level advantage uses depth-wise group normalization:

$$\hat{A}_{i,t} = \frac{R_{i,t} - \mu_t}{\sigma_t + \delta}, \tag{4}$$

where μ_t and σ_t are the mean and standard deviation of the returns across the active group $\{i : T_i \ge t\}$ at interaction depth t, and $\delta > 0$ is a small constant. Eq. (4) centers advantages to zero within each depth group (Appendix C.4), so each turn is scored against rollouts at the same depth. Combining the masked likelihoods with step-level credit yields the Process-Reward GRPO (PR-GRPO) objective. For each policy-generated token, let $d_{\text{KL},k}(\theta) = \log \pi_\theta(y_k \mid y_{<k}) - \log \pi_{\text{ref}}(y_k \mid y_{<k})$; training uses its low-variance form (`low_var_kl`, Appendix E.2). We optimize

$$J(\theta) = \frac{1}{G} \sum_{i=1}^{G} \sum_{t=1}^{T_i} \sum_{k \in \text{Turn}_t^{(i)}} M_k \left[\min\left(\rho_k \hat{A}_{i,t}, \text{clip}(\rho_k, 1-\epsilon, 1+\epsilon) \hat{A}_{i,t} \right) - \beta d_{\text{KL},k}(\theta) \right], \tag{5}$$

where $\text{Turn}_t^{(i)}$ is the token set of turn t in rollout i and $\rho_k = \pi_\theta(y_k \mid y_{<k}) / \pi_{\theta_{\text{old}}}(y_k \mid y_{<k})$.

4 Proposed Framework: CoVER

As illustrated in Figure 2, CoVER combines a stateful code-execution interface with two-stage post-training: supervised warm-up on code-interaction trajectories, followed by process-reward RL with PR-GRPO in the same execution environment.

CoVERBench Design. We construct CoVERBench by standardizing established public reasoning sources and solver-verified tasks into a unified stateful code-execution protocol. CoVERBench draws on PRM800K-style mathematical reasoning (Lightman et al., 2024), selected logic-oriented and structured reasoning tasks from BBH (Suzgun et al., 2023), and solver-verified logical deduction and constraint-satisfaction tasks motivated by logical memorization and rule-based RL (Xie et al., 2025a;b). SFT and RL share the same code-interaction interface but differ in supervision. SFT

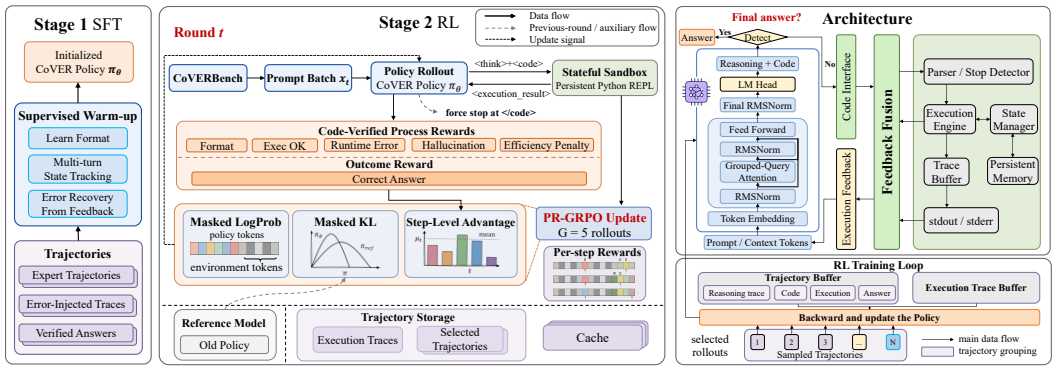


Figure 2: **Overview of CoVER.** (Left) Supervised warm-up on expert trajectories teaches the code-interaction format and recovery behavior. (Middle) Online process-reward RL runs in a persistent sandbox with masked step-level PR-GRPO updates. (Right) The runtime executes code actions, returns execution feedback, and stores mixed trajectories for policy optimization.

instances are complete expert trajectories (reasoning, executable code, sandbox observations, final answers, and, in a subset, error recovery), synthesized with GPT-5 and validated by sandbox replay and deterministic verifiers. RL instances contain only the problem statement, interaction instruction, and verifier-side metadata; the policy improves from online execution feedback. Open-weight teachers regenerate comparable warm-up data (Appendix I.3); construction and contamination controls are in Appendices D and I.2.

Code Actions in a Stateful Sandbox. CoVER treats executable code as the policy's externalized reasoning action. Given the serialized interaction history, the policy emits either a code action or a final boxed answer. Code actions are executed in a persistent Python sandbox, and feedback such as `stdout`, `stderr`, or tracebacks is appended as the next observation. The sandbox keeps variables and intermediate results across turns for reuse and recovery. A restricted runtime blocks unsafe imports, network access, subprocesses, and unauthorized file operations (Appendix E.1).

Stage 1: Supervised Warm-up. The first stage initializes the policy with expert trajectories from the SFT split of CoVERBench. These trajectories demonstrate the desired interaction format, multi-turn state tracking, execution-based reasoning, and recovery from injected code errors. For a serialized SFT trajectory $\tau = (y_1, \ldots, y_{|\tau|})$, we optimize a masked next-token objective:

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{\tau \sim \mathcal{D}_{\mathrm{SFT}}} \left[\frac{\sum_{k=1}^{|\tau|} M_k \log \pi_\theta(y_k \mid y_{<k})}{\sum_{k=1}^{|\tau|} M_k} \right], \tag{6}$$

where $M_k = 1$ only for assistant-generated policy tokens and $M_k = 0$ for user prompts and environment feedback. The resulting checkpoint serves as the initial RL policy and as the reference policy for KL regularization.

Stage 2: Process-Reward RL with PR-GRPO. After warm-up, CoVER performs online RL on the RL training split of CoVERBench following Algorithm 1. Here, T_i is the number of turns of rollout i, and $\oplus$ denotes appending an action or observation to the serialized interaction history. For each prompt x, we sample a group of G rollouts from the old policy $\pi_{\theta_{\mathrm{old}}}$ in the stateful sandbox. Before advantage estimation, we retain the top κG rollouts ranked by cumulative process reward. At turn t of rollout i, the model observes history $h_{i,t-1}$, emits action $a_{i,t}$, and, if the action is executable code, receives environment observation $o_{i,t}$. The reward combines a dense process component and a terminal outcome component, both deterministic functions of the sandbox transition and verifier metadata, so no learned evaluator or step label is needed:

$$r_{i,t} = R_{\mathrm{proc}}(h_{i,t-1}, a_{i,t}, o_{i,t}) + \mathbf{1}[t = T_i]\, R_{\mathrm{out}}(\hat{y}_i, y^\star), \tag{7}$$

where $\hat{y}_i$ is the extracted boxed answer and $y^\star$ is the verifier-side reference answer. For final-answer actions that do not invoke the sandbox, we set $o_{i,t} = \emptyset$. The process reward is computed from

Algorithm 1 CoVER training with stateful process-reward GRPO.

Require: SFT-initialized policy π_θ, RL prompts $\mathcal{D}_{\mathrm{RL}}$, sandbox $\mathcal{E}$, group size G, retention ratio κ, max turns K

Initialize reference policy $\pi_{\mathrm{ref}} \leftarrow \pi_\theta$
for each iteration **do**
  Set old policy $\pi_{\theta_{\mathrm{old}}} \leftarrow \pi_\theta$ and draw prompts $\mathcal{X} \sim \mathcal{D}_{\mathrm{RL}}$
  ▽ **Stateful Rollout & Process Feedback**
  for each prompt $x \in \mathcal{X}$ **do**
    for $i = 1, 2, \ldots, G$ **do**
      Reset $\mathcal{E}$ and initialize $h_{i,0} \leftarrow x$
      for $t = 1, 2, \ldots, K$ **do**
        Sample $a_{i,t} \sim \pi_{\theta_{\mathrm{old}}}(\cdot \mid h_{i,t-1})$
        if final answer is detected **then** `# Detect \boxed{} as the final answer`
          Set $o_{i,t} \leftarrow \emptyset$, extract $\hat{y}_i$, and compute $r_{i,t} \leftarrow R_{\mathrm{proc}}(h_{i,t-1}, a_{i,t}, o_{i,t}) + R_{\mathrm{out}}(\hat{y}_i, y^\star)$
          Set $h_{i,t} \leftarrow h_{i,t-1} \oplus a_{i,t}$, $T_i \leftarrow t$, and **break**
        else
          Execute $a_{i,t}$ in $\mathcal{E}$, observe $o_{i,t}$, and compute $r_{i,t} \leftarrow R_{\mathrm{proc}}(h_{i,t-1}, a_{i,t}, o_{i,t})$
          Set $h_{i,t} \leftarrow h_{i,t-1} \oplus a_{i,t} \oplus o_{i,t}$
        if $t = K$ **then**
          Extract $\hat{y}_i$ if available, add terminal outcome reward to $r_{i,t}$, and set $T_i \leftarrow t$
    Retain the top κG rollouts by cumulative process reward `# Trajectory retention`
  ▽ **Step-Level Advantage Estimation**
  for $t = 1, 2, \ldots, \max_i T_i$ **do**
    Compute $R_{i,t} = \sum_{j=t}^{T_i} \gamma^{j-t} r_{i,j}$ for active rollouts i
    Normalize across active rollouts to obtain $\hat{A}_{i,t}$ using Eq. (4)
  ▽ **Masked PR-GRPO Update**
  Compute D_{KL}^M and $J(\theta)$ using Eq. (3) and Eq. (5) over policy-generated tokens only
  $\theta \leftarrow \theta + \eta \nabla_\theta J(\theta)$

Table 1: Code-verified process reward terms and the terminal outcome reward.

Term	Value	Trigger
r^{fmt}	$+0.1$	Valid code / final-answer format
r^{exec}	$+0.2$ / -0.2	Successful execution / exception, timeout, or disallowed operation
r^{hall}	-0.5	Fabricated sandbox feedback or unsupported checkable claim
r^{eff}	$-0.1 \max(0, t-2)$	Turns beyond the first two
R_{out}	$+2.0$	Final boxed answer matches the verifier reference

execution-grounded interaction signals:

$$R_{\mathrm{proc}} = r^{\mathrm{fmt}} + r^{\mathrm{exec}} + r^{\mathrm{hall}} + r^{\mathrm{eff}}, \tag{8}$$

with the terms and triggers summarized in Table 1. The penalty r^{hall} targets fabricated sandbox feedback and mechanically checkable claims that the execution evidence contradicts or does not support; it ignores claims it cannot check (Appendix C.1).

From the per-turn rewards of the retained rollouts, CoVER computes return-to-go values and depth-wise group-normalized step advantages (Eq. (4)). It then maximizes the PR-GRPO objective (Eq. (5)) with the masked KL penalty (Eq. (3)). Credit is assigned per interaction step. Gradients reach only policy-generated tokens; sandbox observations serve as conditioning context.

5 Experiments

CoVERBench is split 9:1 into training and held-out validation sets, stratified by domain and source and disjoint by problem instance and template family. Exact-match, MinHash, embedding-similarity, and logical-signature deduplication removed all detectable cross-split overlap (Appendix I.2). We use rollout group size $G = 5$ with 80% trajectory retention, prompt batch size $|\mathcal{B}| = 32$, maximum assistant turns 9, and maximum response length 8,192 tokens. Training is implemented with FSDP and vLLM on 16 A100-80GB GPUs (details in Appendix E.2).

5.1 CoVERBench Results with GRPO and Strong LLMs

Figure 3 and Tables 2–3 summarize the CoVERBench validation results.

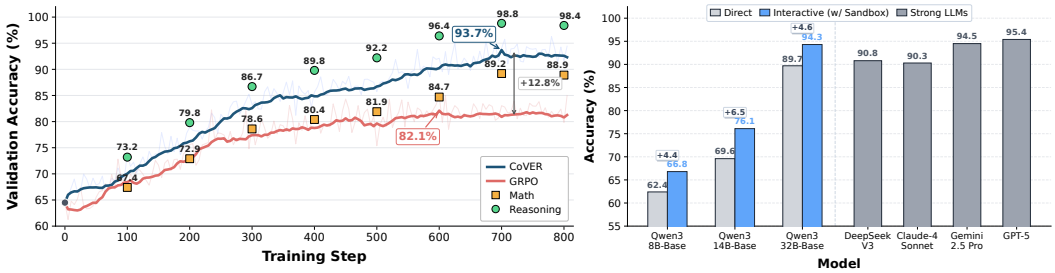


Figure 3: **Main results on CoVERBench.** CoVER and GRPO start from the same SFT checkpoint. Across Qwen3 scales, sandbox interaction outperforms direct answering.

(1) CoVER improves both math and logical reasoning abilities. Figure 3 shows steady gains during training on the Math and Reasoning subsets: Math accuracy rises from 67.4% (step 100) to 84.7% (step 600) and 89.2% (step 700), and Reasoning accuracy from 73.2% to 96.4% and 98.8%.

(2) Sandbox interaction consistently helps Qwen3 base models. Sandbox interaction outperforms direct answering at every scale, from 62.4% to 66.8% (8B), 69.6% to 76.1% (14B), and 89.7% to 94.3% (32B), so executable feedback helps even without RL.

(3) CoVER-8B reaches the range of strong LLM references. CoVER-8B scores 93.7%, 11.6 points above matched GRPO. This baseline shares CoVER's SFT checkpoint, prompts, sandbox, verifier, rollout budget, decoding, and policy-token masking, and uses an outcome-only reward with sequence-level advantages. The gap thus reflects process rewards and step-level credit, not sandbox access, which alone gives Qwen3-8B 66.8%. On the same instances, CoVER-8B also exceeds DeepSeek-V3 (90.8%) (Liu et al., 2024) and Claude-4-Sonnet (90.3%) (Anthropic, 2025), is close to Gemini-2.5-Pro (94.5%) (Comanici et al., 2025) and GPT-5 (95.4%), and approaches sandbox-enhanced Qwen3-32B (94.3%).

(4) Each PR-GRPO mechanism is necessary. With all else fixed (Table 3), outcome-only GRPO reaches 82.1%, process rewards with sequence-level credit 89.2%, and full PR-GRPO with depth-wise step-level credit 93.7%. The variant without policy-token masking keeps process rewards and step-level credit but optimizes environment tokens (stdout, stderr, tracebacks) in its log-probability and KL terms as if they were actions, so it falls to 68.4%, below masked GRPO at 82.1%. Removing the execution reward or the claim-grounding penalty costs 8.4 and 5.3 points (Appendix F.1).

(5) The gains are statistically robust. Table 2 separates evaluation, decoding, and training variability. Problem-level bootstrap 95% intervals (10,000 resamples) are disjoint. Across three decoding seeds, the 95% CI of the gain is $[10.8, 12.4]$ points. Across three matched reduced-scale training runs per method, it is $[4.0, 7.4]$ points (Cohen's $d = 8.1$). Every CoVER training run stays above every GRPO run at each evaluated checkpoint after step 100. CoVER also varies less across seeds than GRPO in both settings (SD 0.3 vs. 0.4 for decoding; 0.6 vs. 0.8 for training). Both seed tests remain significant after Bonferroni correction. The gain does not require trajectory selection: retaining all rollouts yields 93.1%, and 60–100% retention stays within 92.8–93.7%. Open-weight teachers (GLM-5.2,

Table 2: **Robustness of the CoVERBench gain over matched GRPO** (mean±SD; gain with 95% CI; two-sided Welch's t-tests). †Matched reduced-scale runs: 300 updates, identical data and budget.

Evidence	CoVER	GRPO	Gain [95% CI]	Test
Bootstrap CI	$[92.7, 94.7]$	$[80.5, 83.7]$	$+11.6$	disjoint
3 decoding seeds	93.7 ± 0.3	82.1 ± 0.4	$+11.6\ [10.8, 12.4]$	t=40.2, $p<10^{-5}$
3 training seeds†	84.6 ± 0.6	78.9 ± 0.8	$+5.7\ [4.0, 7.4]$	t=9.9, $p<0.001$
All rollouts kept	93.1	82.1	$+11.0$	–

Table 3: **PR-GRPO mechanism ablation** (%) with all else fixed: accuracy, error encounter rate, and recovery rate.

Variant	Acc.	Err.	Rec.
CoVER (PR-GRPO)	**93.7**	**3.2**	**74.7**
w/o step-level advantage	89.2	7.8	58.3
w/o policy-token masking	68.4	31.5	18.2
GRPO (outcome-only)	82.1	14.6	46.5

Table 4: **Zero-shot transfer performance across math and code benchmarks.** Math results are reported in acc@1 (AC) and maj@32 (M32) (%); code results in Pass@1 (%). Benchmarks include AIME (2022–2025), GSM8K (Cobbe et al., 2021), HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), EvalPlus variants (Plus) (Liu et al., 2023a), and BigCodeBench (Zhuo et al., 2024). Best results are **bolded** and second-best are underlined.

Model	Size	Mathematical reasoning (0-shot)		Code generation (0-shot, Pass@1)		
		AIME AC / M32 (%)	GSM8K AC / M32 (%)	HumanEval Base / Plus	MBPP Base / Plus	BigCodeBench Full / Hard
Qwen3-32B (thinking)	32B	**53.3 / 70.8**	93.4 / **96.1**	**92.7 / 87.2**	90.2 / **75.9**	50.5 / 26.8
Llama-3.1-70B-Inst	70B	19.2 / 30.8	92.3 / 94.7	81.2 / 74.4	86.1 / 70.1	**54.8 / 27.7**
Qwen3-8B (thinking)	8B	47.5 / 64.2	89.8 / 93.6	89.6 / 84.1	86.8 / 72.5	48.6 / 23.1
+ SFT	8B	45.8 / 62.5	89.3 / 93.8	89.0 / 82.9	85.7 / 72.2	48.2 / 22.0
+ GiGPO	8B	48.3 / 65.8	92.0 / 95.0	90.0 / 84.0	88.1 / 72.8	49.1 / 23.9
+ GTPO	8B	49.2 / 66.7	92.7 / 95.3	90.3 / 84.6	89.1 / 73.1	49.6 / 24.8
+ SFT + PR-GRPO (**CoVER**)	8B	50.8 / 68.3	**93.9** / 95.8	90.9 / 85.4	**90.7** / 73.5	50.2 / 25.8

Math-specialized baselines

Model	Size	AIME AC / M32 (%)	GSM8K AC / M32 (%)
InternLM2-Math	20B	15.8 / 32.5	82.6 / 87.2
Math-Shepherd-PRM	7B	22.5 / 34.2	89.1 / 92.5
NuminaMath-TIR	7B	24.2 / 39.2	84.6 / 88.7
DeepSeek-Math-Inst.	7B	25.8 / 40.8	88.2 / 91.6
LogicReward	7B	23.3 / 38.3	86.4 / 90.4

Code-specialized baselines

Model	Size	HumanEval Base / Plus	MBPP Base / Plus	BigCodeBench Full / Hard
Codestral	22B	81.1 / 73.8	78.2 / 68.8	36.4 / 16.1
DS-Coder-V2	16B	81.7 / 75.6	82.2 / 70.4	36.9 / 16.5
CodeLlama	13B	65.8 / 56.0	68.2 / 50.5	31.7 / 9.5
SemCoder	7B	87.8 / 83.5	84.1 / 70.6	41.1 / 19.2
CodeI/O	7B	85.4 / 81.1	81.7 / 69.3	40.2 / 18.2

DeepSeek-V4-Pro) regenerate the warm-up data with QC pass rates within 1.2 points of GPT-5 (Appendices I.1 and I.3).

5.2 Generalization to Unseen Benchmarks

Table 4 evaluates zero-shot transfer to unseen math and code benchmarks. SFT alone lowers nine of the ten reported metrics relative to Qwen3-8B thinking. After RL, CoVER improves on all ten metrics and exceeds both GiGPO and GTPO across the reported benchmarks. Compared with the stronger of these two baselines, GTPO, CoVER increases AIME acc@1 from 49.2% to 50.8%, GSM8K acc@1 from 92.7% to 93.9%, and BigCodeBench-Hard Pass@1 from 24.8% to 25.8%.

Among the general models in Table 4, CoVER ranks second on AIME and first on GSM8K acc@1, and it approaches Qwen3-32B on BigCodeBench-Hard with a quarter of its parameters. It also outperforms every math-specialized baseline on AIME and GSM8K and every code-specialized baseline on every code benchmark.

5.3 Token and Compute Efficiency Comparison

We compare the inference cost of CoVER-8B with Qwen3-8B and Qwen3-32B thinking models on 120 AIME problems (2022–2025). The thinking baselines use a 16,384-token single-pass cap; CoVER interacts with the sandbox over multiple turns and stops at its final answer.

Table 5: **Inference efficiency on AIME.** Tok./correct: total tokens generated across all problems divided by the number of correctly solved problems; cap hit: fraction of responses reaching the 16,384-token cap; rel. FLOPs/correct estimated from model size and tokens/correct and normalized to CoVER; matched subsets: problems solved by all three models (39) or by none (30).

Model	All 120 AIME problems				Matched subsets: tokens/problem	
	Tok./prob.	Tok./correct	Cap hit	Rel. FLOPs/correct	All correct (39)	All wrong (30)
Qwen3-8B	12,899	27,156	47.5%	4.1×	9,183	15,951
Qwen3-32B	12,047	22,588	41.7%	13.7×	7,822	15,975
CoVER-8B	**3,349**	**6,588**	–	**1.0**×	**1,987**	**3,651**

As shown in Table 5, CoVER-8B uses 3,349 tokens per problem and approximately 6,588 tokens per correct solution, versus 12,047–12,899 and 22,588–27,156 for the thinking baselines. The advantage

persists on the all-correct matched subset, and on all-wrong cases CoVER averages 3,651 tokens versus 15,951–15,975 for the baselines. Estimated from model size and token usage, Qwen3-8B and Qwen3-32B require approximately $4.1\times$ and $13.7\times$ the FLOPs per correct solution of CoVER. End to end, at batch size 1 on one A100, CoVER needs 63.0 s per problem versus 234.5 s for Qwen3-8B thinking; sandbox execution and kernel management add 2.1 s (3.3%; Appendix J).

5.4 PR-GRPO Improves Traceback-Based Error Recovery

When generated code raises an exception, the sandbox returns a traceback that the model must use to revise its code. We compare PR-GRPO with GRPO on three metrics: error encounter rate, conditional recovery rate after an error, and error-to-correct rate. PR-GRPO reduces the error encounter rate from 14.6% to 3.2% (Figure 4), indicating that dense execution-level rewards discourage brittle code generation. After an error, PR-GRPO raises the recovery rate from 46.5% to 74.7%: the model learns to use tracebacks as diagnostic signals.

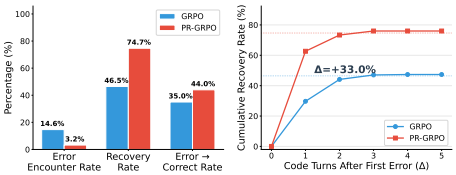


Figure 4: **Traceback-based error recovery.** PR-GRPO reduces execution errors and recovers more rapidly after the first traceback.

PR-GRPO also corrects errors faster. Within one code turn after the first traceback, it recovers 62.7% of errored trajectories versus 29.7% for GRPO. After two turns, its cumulative recovery approaches 75%, while GRPO plateaus near 47%. It also raises the end-to-end error-to-correct rate from 35.0% to 44.0%. Of CoVER's 146 residual validation errors, only 1.4% are runtime, syntax, or tool-usage failures; 61.6% are reasoning errors and 19.2% verifier false negatives (Appendix F.2).

5.5 Process Reward Analysis

We analyze whether CoVER's process rewards provide supervision beyond terminal correctness by summing turn-level rewards for each rollout, excluding the terminal outcome reward. As shown in Figure 5, correct trajectories consistently receive higher process rewards than incorrect ones, especially in early-to-mid training, showing that the reward aligns with final success rather than merely format compliance. The gap peaks during bootstrap ($\Delta = 0.44$ at step 25) and then shrinks as both groups enter the high-reward regime, suggesting that malformed code, execution errors, and ineffective recovery are progressively reduced.

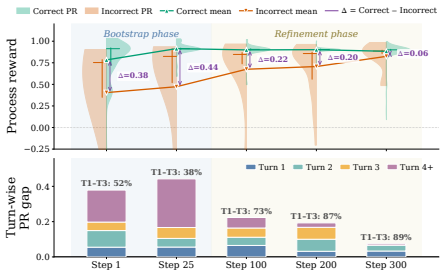


Figure 5: **Process reward analysis.** Correct trajectories receive higher process rewards, and the gap shrinks as tool use stabilizes.

The lower panel decomposes this gap by turn group. Early in training, Turn 4+ contributes most of the signal, accounting for 62% of the gap at step 25, indicating that failed rollouts often reveal themselves through prolonged debugging or repeated execution attempts. This contribution drops to 13% by step 200, and by step 300 the remaining small gap is dominated by Turns 1–3. This shift indicates that training first improves the mechanics of multi-turn tool use, leaving later-stage performance differences to be driven more by the quality of the initial reasoning steps.

6 Conclusion

We introduced **CoVER**, which turns stateful code execution in a persistent sandbox into online process supervision for LLM reasoning, and **PR-GRPO**, which optimizes the resulting mixed agent–environment trajectories with policy-token masking and depth-wise step-level advantages. Starting from Qwen3-8B, CoVER reaches 93.7% on CoVERBench, 11.6 points above matched outcome-only GRPO. Bootstrap intervals are disjoint, and the margins are significant across three decoding seeds and three matched reduced-scale training runs. Across unseen math/code benchmarks, CoVER improves accuracy, recovery behavior, and inference efficiency, demonstrating the effectiveness of code-verified process rewards for training reliable multi-turn reasoning agents.

AI USE STATEMENT

In this work, we used generative AI tools for two purposes. (i) To aid and polish writing: editing the manuscript for readability and formatting LaTeX. (ii) To generate synthetic datasets: the $1{,}500$ supervised warm-up trajectories of CoVERBench were synthesized with GPT-5 (and, for the teacher-independence analysis, with GLM-5.2 and DeepSeek-V4-Pro), then replayed in the sandbox and filtered by deterministic verifiers and human quality control (Section 4; Appendices D.2 and I.3). We did not use generative AI tools for retrieval and discovery, research ideation or execution, drafting sections of the paper, or proving mathematical claims; the LLMs evaluated as baselines are objects of study, not authoring tools. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects or personal data. CoVERBench is constructed from public reasoning datasets and solver-generated tasks, and the sandbox executes only model-generated Python code under a restricted runtime that blocks unsafe imports, network access, subprocesses, and unauthorized file operations. Stronger code-executing agents nevertheless raise safety concerns, including misuse for automated code generation, unsafe tool invocation, and over-trust in apparently verified outputs; deployments should enforce sandbox isolation, resource limits, auditing, and domain-specific verification before use in high-stakes settings. A broader discussion is given in Appendix B.

REPRODUCIBILITY STATEMENT

The problem formulation, PR-GRPO objective, and training procedure are fully specified in Sections 3 and 4 and Algorithm 1. The complete reward formulation, including all constants, is given in Appendix C.1, and proofs of the masked-likelihood, masked-KL, and step-level advantage results are provided in Appendix C. The construction, composition, and quality control of CoVERBench are described in Appendix D; the stateful runtime, rollout implementation, and all training hyperparameters are listed in Appendix E. An anonymized version of CoVERBench is provided as supplementary material, and the full training code and reproduction scripts will be released upon publication.

REFERENCES

AIME. AIME problems and solutions. `https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions`, 2025.

Anthropic. Claude Sonnet 4. `https://www.anthropic.com/news/claude-4`, 2025.

Iván Arcuschin, Jett Janiak, Robert Krzyzanowski, Senthooran Rajamanoharan, Neel Nanda, and Arthur Conmy. Chain-of-thought reasoning in the wild is not always faithful. *arXiv preprint arXiv:2503.08679*, 2025.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv preprint arXiv:2303.12712*, 2023.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.

Qikai Chang, Zhenrong Zhang, Pengfei Hu, Jun Du, Jiefeng Ma, Yicheng Pan, Jianshu Zhang, Quan Liu, and Jianqing Gao. THOR: Tool-integrated hierarchical optimization via RL for mathematical reasoning. *arXiv preprint arXiv:2509.13761*, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

James Chua and Owain Evans. Are DeepSeek R1 and other reasoning models more faithful? *arXiv preprint arXiv:2501.08156*, 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Wei Zhu, Yuan Ni, Guotong Xie, Zhiyuan Liu, and Maosong Sun. UltraFeedback: Boosting language models with high-quality feedback. *arXiv preprint arXiv:2310.01377*, 2023.

Yifeng Ding, Hung Le, Songyang Han, Kangrui Ruan, Zhenghui Jin, Varun Kumar, Zijian Wang, and Anoop Deoras. Empowering multi-turn tool-integrated agentic reasoning with group turn policy optimization. *arXiv preprint arXiv:2511.14846*, 2025.

Hanze Dong, Wei Xiong, Deepanshu Goyal, Yihan Zhang, Winnie Chow, Rui Pan, Shizhe Diao, Jipeng Zhang, Kashun Shum, and Tong Zhang. RAFT: Reward ranked finetuning for generative foundation model alignment. *arXiv preprint arXiv:2304.06767*, 2023.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. Faith and fate: Limits of transformers on compositionality. *Advances in neural information processing systems*, 36:70293–70332, 2023.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. ReTool: Reinforcement learning for strategic tool use in LLMs. *arXiv preprint arXiv:2504.11536*, 2025a.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for LLM agent training. In *Advances in Neural Information Processing Systems*, 2025b.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *International conference on machine learning*, pp. 10764–10799. PMLR, 2023.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*, 2023.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173, 2023.

Arian Hosseini, Xingdi Yuan, Nikolay Malkin, Aaron Courville, Alessandro Sordoni, and Rishabh Agarwal. V-star: Training verifiers for self-taught reasoners. *arXiv preprint arXiv:2402.06457*, 2024.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

Piyush Jha, Prithwish Jana, Pranavkrishna Suresh, Arnav Arora, and Vijay Ganesh. RLSF: Fine-tuning LLMs via symbolic feedback. *arXiv preprint arXiv:2405.16661*, 2024.

Wouter Kool, Herke van Hoof, and Max Welling. Buy 4 reinforce samples, get a baseline for free! *ICLR 2019 Workshop on Deep Reinforcement Learning Meets Structured Prediction*, 2019.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Joshua Ong Jun Leang, Giwon Hong, Wenda Li, and Shay B Cohen. Theorem prover as a judge for synthetic data generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 29941–29977, 2025.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. RLAIF: Scaling reinforcement learning from human feedback with AI feedback. *arXiv preprint arXiv:2309.00267*, 2023.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*, 2023.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. ToRL: Scaling tool-integrated RL. *arXiv preprint arXiv:2503.23383*, 2025.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The twelfth international conference on learning representations*, 2024.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Hanmeng Liu, Zhizhang Fu, Mengru Ding, Ruoxi Ning, Chaoli Zhang, Xiaozhang Liu, and Yue Zhang. Logical reasoning in large language models: A survey. *arXiv preprint arXiv:2502.09100*, 2025.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023a.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. AgentBench: Evaluating LLMs as agents. *arXiv preprint arXiv:2308.03688*, 2023b.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36:43447–43478, 2023.

Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jianguang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, and Dongmei Zhang. WizardMath: Empowering mathematical reasoning for large language models via reinforced evol-instruct. *arXiv preprint arXiv:2308.09583*, 2023.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. *Advances in neural information processing systems*, 37:74325–74362, 2024.

Da Ma, Ziyue Yang, Hongshen Xu, Haotian Fang, Kai Yu, and Lu Chen. Empowering LLM tool invocation with tool-call reward model. In *International Conference on Learning Representations*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

Rémi Munos, Michal Valko, Daniele Calandriello, Mohammad Gheshlaghi Azar, Mark Rowland, Zhaohan Daniel Guo, Yunhao Tang, Matthieu Geist, Thomas Mesnard, Côme Fiegel, et al. Nash learning from human feedback. In *Forty-first International Conference on Machine Learning*, 2024.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. WebGPT: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Aaron Parisi, Yao Zhao, and Noah Fiedel. TALM: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

Mihir Prabhudesai, Lili Chen, Alex Ippoliti, Katerina Fragkiadaki, Hao Liu, and Deepak Pathak. Maximizing confidence alone improves reasoning. *arXiv preprint arXiv:2505.22660*, 2025.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. ToolRL: Reward is all tool learning needs. In *Advances in Neural Information Processing Systems*, 2025.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–40, 2024.

Changle Qu, Sunhao Dai, Hengyi Cai, Jun Xu, Shuaiqiang Wang, and Dawei Yin. MatchTIR: Fine-grained supervision for tool-integrated reasoning via bipartite matching. *arXiv preprint arXiv:2601.10712*, 2026.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Abulhair Saparov and He He. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. *arXiv preprint arXiv:2210.01240*, 2022.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. OpenAI GPT-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in neural information processing systems*, 33:3008–3021, 2020.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 13003–13051, 2023.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the planning abilities of large language models-a critical investigation. *Advances in neural information processing systems*, 36:75993–76005, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023a.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 2609–2634, 2023b.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024a.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, 2024b.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. MINT: Evaluating LLMs in multi-turn interaction with tools and language feedback. *arXiv preprint arXiv:2309.10691*, 2023c.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Quan Wei, Siliang Zeng, Chenliang Li, Zhongruo Wang, William Brown, Oana Frunza, Wei Deng, Anderson Schneider, Yuriy Nevmyvaka, Yang Katie Zhao, Alfredo Garcia, and Mingyi Hong. Reinforcing multi-turn reasoning in LLM agents via fine-grained reward structure and credit assignment. *arXiv preprint arXiv:2505.11821*, 2025.

Chulin Xie, Yangsibo Huang, Chiyuan Zhang, Da Yu, Xinyun Chen, Bill Yuchen Lin, Bo Li, Badih Ghazi, and Ravi Kumar. On memorization of large language models in logical reasoning. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pp. 2742–2785, 2025a.

Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. Logic-RL: Unleashing LLM reasoning with rule-based reinforcement learning. *arXiv preprint arXiv:2502.14768*, 2025b.

Jundong Xu, Hao Fei, Huichi Zhou, Xin Quan, Qijun Huang, Shengqiong Wu, William Yang Wang, Mong-Li Lee, and Wynne Hsu. LogicReward: Incentivizing LLM reasoning via step-wise logical supervision. *arXiv preprint arXiv:2512.18196*, 2025.

Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun Ma, and Bo An. SimpleTIR: End-to-end reinforcement learning for multi-turn tool-integrated reasoning. *arXiv preprint arXiv:2509.02479*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. InterCode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854, 2023.

Jiarui Yao, Ruida Wang, and Tong Zhang. PRL: Process reward learning improves LLMs' reasoning ability and broadens the reasoning boundary. *arXiv preprint arXiv:2601.10201*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2023.

Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. Lumos: Learning agents with unified data, modular design, and open-source LLMs. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2023.

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

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions. *Advances in Neural Information Processing Systems*, 36:31466–31523, 2023.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. AgentTuning: Enabling generalized agent abilities for LLMs. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3053–3077, 2024.

Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, et al. The landscape of agentic reinforcement learning for LLMs: A survey. *arXiv preprint arXiv:2509.02547*, 2025a.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. The lessons of developing process reward models in mathematical reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 10495–10516, 2025b.

Xuandong Zhao, Zhewei Kang, Aosong Feng, Sergey Levine, and Dawn Song. Learning to reason without external rewards. *arXiv preprint arXiv:2505.19590*, 2025.

Rui Zheng, Shihan Dou, Songyang Gao, Yuan Hua, Wei Shen, Binghai Wang, Yan Liu, Senjie Jin, Qin Liu, Yuhao Zhou, et al. Secrets of RLHF in large language models part i: PPO. *arXiv preprint arXiv:2307.04964*, 2023.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. WebArena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.

APPENDIX CONTENTS

A Scope and Extensions 18

B Broader Impact 18

C Theoretical Formulations and Omitted Proofs 18

C.1 Detailed Reward Formulation 18

C.2 Masked Trajectory Log-Likelihood 20

C.3 Masked KL Regularization 20

C.4 Group-Centered Property of Step-Level Advantage 21

C.5 The Standard GRPO Objective 21

D Dataset Construction: CoVERBench 22

D.1 Overview and Dataset Composition 22

D.2 SFT Trajectory Serialization and Synthesis 22

D.3 RL Prompt Construction and Domain-Specific Sources 23

D.4 Preprocessing, Verification, and Quality Control 24

E System Runtime and Implementation Details 25

E.1 Stateful Runtime and Rollout Implementation 25

E.2 Training and Implementation Details 26

F Empirical Analysis and Ablation Studies 26

F.1 Ablation Study 26

F.2 CoVERBench Validation Error Analysis 27

F.3 Analysis of Execution-Grounded Process Supervision 29

G Representative CoVER Solution Trajectories 33

H Relation to Turn-Level RL for Tool Use 43

I Statistical Robustness, Sensitivity, and Contamination Controls 44

I.1 Uncertainty Estimates and Seed Robustness 44

I.2 Contamination Controls 45

I.3 Teacher Independence of the Warm-up Data 45

J End-to-End Inference Cost 46

A Scope and Extensions

Where CoVER applies. CoVER is designed for tasks with safe, resettable, and mechanically verifiable execution or solver feedback, such as numerical and symbolic computation, structured constraint solving, deterministic tests, and stateful code interaction. In these settings every reward term is a deterministic function of the sandbox transition and the verifier metadata (Table 1), so process supervision is obtained online, at every executed turn, without step labels, learned evaluators, or reference tool-call trajectories. The claim-grounding penalty r^{hall} checks mechanically extractable claims against the execution evidence, including numeric and symbolic values, structured assignments, constraint-satisfaction statements, and test-passing claims (Appendix C.1). Within this scope, the gains are supported by matched comparisons, ablations, and seed and contamination analyses (Section 5.1; Appendix I): the 93.7% CoVERBench result is obtained on a held-out validation split that is disjoint by instance and template family, the advantage over GRPO is significant across three decoding seeds and three matched reduced-scale training seeds (Table 13), the selection-free setting reaches 93.1%, and the gains are accompanied by transfer to unseen external benchmarks (Table 4).

Natural extensions. The framework extends along three directions. (i) *Scale:* applying PR-GRPO to larger backbones with longer contexts and interaction horizons, and scaling the training data and compute budget. (ii) *Richer verifiers:* the deterministic checker $\mathrm{Check}(c; \mathcal{E}_{i,t})$ is an interface into which formal proof checkers, property-based tests, or domain-specific validators can be plugged to supervise intermediate semantic correctness. (iii) *Broader environments:* checkpoint-and-rollback sandboxes make environments with side effects resettable, which would bring software engineering with hidden dependencies and real-world tool use into the same execution-grounded training loop.

B Broader Impact

CoVER can improve the reliability and efficiency of LLM reasoning systems by encouraging models to ground intermediate steps in verifiable execution feedback rather than relying solely on final-answer rewards or unverifiable chains of thought. Because its process rewards require no step labels, it avoids the cost of human process annotation and can benefit applications in mathematics, education, programming assistance, and scientific problem solving. However, stronger code-executing agents also introduce safety concerns, including potential misuse for automated code generation, unsafe tool invocation, or over-trust in apparently verified outputs. Deployments should therefore enforce sandbox isolation, resource limits, auditing, and domain-specific verification before use in high-stakes settings.

C Theoretical Formulations and Omitted Proofs

This appendix provides additional formal details for the CoVER training objective and the PR-GRPO optimization procedure introduced in Section 3. We first specify the process-reward components used to convert sandbox execution signals into dense online supervision. We then give omitted derivations for masked trajectory log-likelihood, masked KL regularization, and the group-centered property of step-level advantages. These results clarify why, in mixed agent-environment trajectories, policy-gradient and regularization terms should be computed only over policy-generated tokens, while environment-generated observations are treated as external feedback.

C.1 Detailed Reward Formulation

In the stateful rollout setting described in Section 4, CoVER provides dense process rewards at each interaction turn and a terminal outcome reward at the end of the episode. For rollout i at turn t, the process reward is

$$R_{\mathrm{proc}}(h_{i,t-1}, a_{i,t}, o_{i,t}) = r^{\mathrm{fmt}}_{i,t} + r^{\mathrm{exec}}_{i,t} + r^{\mathrm{hall}}_{i,t} + r^{\mathrm{eff}}_{i,t}. \tag{9}$$

Table 1 in the main text summarizes the reward terms; the components are defined as follows.

Formatting reward. The formatting reward $r_{i,t}^{\text{fmt}}$ encourages the model to follow the required interaction protocol:

$$r_{i,t}^{\text{fmt}} = 0.1 \cdot \mathbf{1}[\text{the action follows the required protocol}]. \tag{10}$$

For code actions, this requires a well-formed `<code>...</code>` block. For final-answer actions, this requires a valid boxed answer.

Execution reward. For code actions, the execution reward $r_{i,t}^{\text{exec}}$ is

$$r_{i,t}^{\text{exec}} = \begin{cases} +0.2, & \text{if the code executes successfully without exception or timeout,} \\ -0.2, & \text{if the code raises an exception, times out, or triggers a disallowed operation.} \end{cases} \tag{11}$$

For final-answer actions that do not invoke the sandbox, this term is set to 0. This term rewards reliable execution; final-answer correctness is supervised by the terminal outcome reward, and mechanically checkable claims by the claim-grounding penalty below.

Hallucination and claim-grounding penalty. The hallucination penalty $r_{i,t}^{\text{hall}}$ penalizes two types of unsupported behavior: fabricated sandbox feedback and unsupported mechanically checkable claims. We define

$$r_{i,t}^{\text{hall}} = -\lambda_{\text{hall}} \cdot \mathbf{1}\left[H_{i,t}^{\text{fb}} \vee H_{i,t}^{\text{claim}}\right], \qquad \lambda_{\text{hall}} = 0.5. \tag{12}$$

Here $H_{i,t}^{\text{fb}} = 1$ if the agent emits sandbox-only feedback fields, such as `<execution_result>`, instead of waiting for the environment-generated observation; otherwise $H_{i,t}^{\text{fb}} = 0$.

The second indicator, $H_{i,t}^{\text{claim}}$, captures unsupported execution-visible claims. Let

$$\mathcal{C}_{i,t} = \mathcal{C}(h_{i,t-1}, a_{i,t}) \tag{13}$$

denote the set of mechanically extractable claims made by the agent in turn t, such as candidate numeric values, structured assignments, constraint-satisfaction statements, test-passing claims, or claims that a printed output implies a particular answer. Let

$$\mathcal{E}_{i,t} = (o_{i,t}, \text{State}_{i,t}, \text{Meta}_i) \tag{14}$$

denote the verifier-side evidence available after the action, including the sandbox observation $o_{i,t}$, the persistent runtime state $\text{State}_{i,t}$, and task metadata Meta_i, such as the task type and deterministic verifier.

We use a lightweight deterministic checker

$$\text{Check}(c; \mathcal{E}_{i,t}) \in \{1, 0, \perp\}, \tag{15}$$

where 1 means that claim c is supported by the evidence, 0 means that it is contradicted or lacks the required executable support, and $\perp$ means that the claim is not mechanically checkable by our rules and is therefore ignored. We set

$$H_{i,t}^{\text{claim}} = \mathbf{1}\left[\exists c \in \mathcal{C}_{i,t} \text{ s.t. } \text{Check}(c; \mathcal{E}_{i,t}) = 0\right]. \tag{16}$$

The checker targets claims that can be grounded in execution-visible evidence, which keeps it deterministic. For math tasks, this includes numeric or symbolic consistency checks, candidate-value verification, and constraint-satisfaction checks. For logic and constraint-satisfaction tasks, this includes structured assignment checks against the deterministic solver or partial consistency verifier. For code-generation tasks, this includes claims about unit-test success, assertion results, and printed test summaries. Thus $r_{i,t}^{\text{hall}}$ grounds the claims exposed through the sandbox interface in the execution evidence, at no annotation cost.

Efficiency penalty. The efficiency penalty discourages unnecessarily long interactions and repeated stalling behavior:

$$r_{i,t}^{\text{eff}} = -0.1 \cdot \max(0, t - 2). \tag{17}$$

At the terminal step, CoVER additionally applies an outcome reward:

$$R_{\text{out}}(\hat{y}_i, y^\star) = 2.0 \cdot \mathbf{1}[\hat{y}_i = y^\star], \tag{18}$$

where $\hat{y}_i$ is the extracted boxed answer and $y^\star$ is the verifier-side reference answer. If no valid boxed answer is produced, the outcome reward is 0.

C.2 MASKED TRAJECTORY LOG-LIKELIHOOD

Proposition 3.1 (Masked Trajectory Log-Likelihood). *Let τ be a realized interaction trajectory as defined in Eq.* (2)*, and let $O = (y_1, \ldots, y_{|O|})$ denote the flattened token sequence containing both policy-generated action tokens and environment-generated observation tokens. Let $M_k \in \{0, 1\}$ indicate whether token y_k is generated by the policy. Then the policy-dependent score of the trajectory is*

$$\nabla_\theta \log P_\theta(\tau) = \sum_{k=1}^{|O|} M_k \nabla_\theta \log \pi_\theta(y_k \mid y_{<k}). \tag{19}$$

Proof. Consider the augmented version of the trajectory in Eq. (2) that also includes the latent sandbox states. By the POMDP dynamics in Eq. (1), its probability factorizes into policy terms and environment terms:

$$P_\theta(\tau) = C(\tau) \prod_{t=1}^{T_\tau} \pi_\theta(a_t \mid h_t), \tag{20}$$

where $h_t = (o_1, a_1, o_2, a_2, \ldots, o_t)$ is the observable interaction history, and $C(\tau)$ collects all terms from the initial-state distribution, sandbox transition kernel, observation kernel, and reward-generation mechanism. These environment terms are independent of the learnable policy parameters θ. Taking logarithms and differentiating gives

$$\nabla_\theta \log P_\theta(\tau) = \sum_{t=1}^{T_\tau} \nabla_\theta \log \pi_\theta(a_t \mid h_t). \tag{21}$$

Each assistant action $a_t = (y_{t,1}, \ldots, y_{t,L_t})$ is generated autoregressively:

$$\log \pi_\theta(a_t \mid h_t) = \sum_{m=1}^{L_t} \log \pi_\theta(y_{t,m} \mid h_t, y_{t,<m}). \tag{22}$$

After flattening the mixed trajectory into O, the policy-generated action tokens are exactly those with $M_k = 1$, while environment feedback tokens, such as `stdout`, `stderr`, and tracebacks, have $M_k = 0$. Substituting the autoregressive factorization into Eq. (21) yields Eq. (19).

C.3 MASKED KL REGULARIZATION

Proposition 3.2 (Masked KL Regularization). *In mixed agent-environment trajectories, the KL regularizer used for policy optimization should be applied only to policy-generated tokens, i.e., indices with $M_k = 1$. Including environment-generated tokens introduces terms that do not measure deviations between the current and reference policies over the model's action distribution.*

Proof. KL regularization in PPO-style language-model fine-tuning constrains the updated policy π_θ from deviating too far from a reference policy π_{ref} (Schulman et al., 2017; Ziegler et al., 2019; Ouyang et al., 2022). For mixed trajectories, CoVER uses the masked token-level KL defined in Eq. (3), which averages only over policy-generated tokens.

If the mask is omitted, the regularizer additionally contains environment-token terms of the form

$$\sum_{k:M_k=0} \left[\log \pi_\theta(y_k \mid y_{<k}) - \log \pi_{\text{ref}}(y_k \mid y_{<k})\right]. \tag{23}$$

For these indices, y_k is not sampled from π_θ, but from the external environment through sandbox execution. Such tokens include standard output, error messages, and tracebacks. Therefore, Eq. (23) is not a divergence between π_θ and π_{ref} over the policy's action distribution. Its gradient includes

$$\sum_{k:M_k=0} \nabla_\theta \log \pi_\theta(y_k \mid y_{<k}), \tag{24}$$

which pushes the policy to assign likelihood to environment-generated strings that it did not choose as actions. This creates spurious optimization pressure and biases the trust-region penalty away from its intended role of controlling policy deviation on model-generated tokens.

Masking removes these non-policy terms and yields a KL regularizer defined only on tokens sampled from the policy. Thus, Eq. (3) is the appropriate KL penalty for mixed agent-environment trajectories.

C.4 Group-Centered Property of Step-Level Advantage

Proposition 3.3 (Group-Centered Step-Level Advantage). *For any turn depth t with a non-empty active group of trajectories $\mathcal{G}_t$, the unclipped step-level advantages in Eq.* (4) *are zero-centered:*

$$\sum_{i \in \mathcal{G}_t} \hat{A}_{i,t} = 0. \tag{25}$$

Proof. Following the group-relative normalization used in GRPO (Shao et al., 2024) and related multi-sample baseline estimators (Kool et al., 2019), define the active group at turn depth t as

$$\mathcal{G}_t = \{i : T_i \geq t\}. \tag{26}$$

For trajectory i, the return-to-go at turn t is

$$R_{i,t} = \sum_{j=t}^{T_i} \gamma^{j-t} r_{i,j}. \tag{27}$$

Let

$$\mu_t = \frac{1}{|\mathcal{G}_t|} \sum_{i \in \mathcal{G}_t} R_{i,t} \tag{28}$$

be the depth-specific group mean. The centered returns sum to zero:

$$\sum_{i \in \mathcal{G}_t} (R_{i,t} - \mu_t) = \sum_{i \in \mathcal{G}_t} R_{i,t} - |\mathcal{G}_t| \mu_t = 0. \tag{29}$$

Using the step-level advantage definition in Eq. (4), where $\sigma_t + \delta$ is shared by all trajectories in $\mathcal{G}_t$, we obtain

$$\sum_{i \in \mathcal{G}_t} \hat{A}_{i,t} = \frac{1}{\sigma_t + \delta} \sum_{i \in \mathcal{G}_t} (R_{i,t} - \mu_t) = 0. \tag{30}$$

Thus, at each interaction depth, the unclipped advantages are centered within the active rollout group. This makes the update depend on relative performance among rollout branches at the same turn depth, providing a simple baseline for multi-turn credit assignment.

C.5 The Standard GRPO Objective

For reference, we state the standard GRPO objective (Shao et al., 2024) that PR-GRPO (Eq. (5)) generalizes. Inspired by leave-one-out reward baselines (Kool et al., 2019; Dong et al., 2023), GRPO removes the critic by normalizing rewards within a sampled group. Specifically, for a prompt $x \sim \mathcal{D}$, a group of G completions $Y = \{y_1, \ldots, y_G\}$ is sampled from $\pi_{\theta_{\text{old}}}(\cdot \mid x)$. Each completion y_i receives a scalar reward $r(x, y_i)$ from a reward model (Lee et al., 2023; Yuan et al., 2024; Bai et al., 2022; Munos et al., 2024; Cui et al., 2023) or a rule-based verifier (Xie et al., 2025b; Luo et al., 2023; Singh et al., 2023). The group-relative advantage is computed as

$$\hat{A}_i = \frac{r(x, y_i) - \mu_r}{\sigma_r}, \tag{31}$$

where μ_r and σ_r denote the mean and standard deviation of the reward set $\{r(x, y_j)\}_{j=1}^{G}$. GRPO then updates the policy π_θ by maximizing a clipped surrogate objective with KL regularization toward π_{ref}:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{x \sim \mathcal{D},\, Y \sim \pi_{\theta_{\text{old}}}} \left[\frac{1}{G} \sum_{i=1}^{G} \left(\mathcal{L}_{\text{clip}}(x, y_i; \theta) - \beta \mathbb{D}_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}) \right) \right]. \tag{32}$$

The clipped surrogate loss is defined as $\mathcal{L}_{\text{clip}} = \sum_{t=1}^{|y_i|} \min\left(\rho_{i,t}(\theta)\hat{A}_i, \text{clip}(\rho_{i,t}(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_i\right)$, where $\rho_{i,t}(\theta) = \pi_\theta(a_{i,t} \mid s_{i,t}) / \pi_{\theta_{\text{old}}}(a_{i,t} \mid s_{i,t})$ is the token-level probability ratio, and β is the KL regularization coefficient.

D DATASET CONSTRUCTION: COVERBENCH

This section details the construction of **CoVERBench**, a standardized stateful code-execution benchmark that aggregates established public reasoning sources and solver-verified logic tasks under a unified interaction protocol. CoVERBench is a method-agnostic testbed: it reformulates diverse reasoning problems into a common code-interaction format so that models can be trained and evaluated under the same execution interface. The benchmark provides both supervised interaction trajectories for warm-up and prompt-level instances for online RL, containing 1,500 quality-controlled supervised trajectories and 23,229 RL prompt-level instances, which are split into training and validation sets with a 9:1 ratio.

D.1 OVERVIEW AND DATASET COMPOSITION

CoVERBench is constructed around the principle of *interaction alignment*: the SFT and RL stages share the same visible code-interaction paradigm, but use different forms of supervision. In the SFT stage, each example is a complete expert-quality interaction trajectory containing reasoning, executable code, environment observations, possible error recovery, and a final answer. In the RL stage, each instance contains only the problem statement, the unified interaction instruction, and verifier-side metadata; no reference trajectory is exposed to the policy. This design makes supervised warm-up an initialization stage for interaction behavior, while RL performs policy improvement under real execution feedback.

CoVERBench draws on prior work in process supervision and tool-augmented reasoning. PRM800K (Lightman et al., 2024) motivates intermediate reasoning supervision, BBH (Suzgun et al., 2023) provides diverse reasoning tasks with stable answer formats, and recent studies of logical memorization and rule-based RL (Xie et al., 2025a;b) motivate solver-verified logic tasks. CoVERBench standardizes these sources into a stateful code-execution setting where trajectories include not only natural-language reasoning, but also executable actions and environment observations. The full dataset composition is summarized in Table 6.

Table 6: Composition of CoVERBench. SFT instances are complete multi-turn code-interaction trajectories, while RL instances are prompt-level problems with verifier metadata only.

Split	Domain / category	# Instances	Supervision or verification
SFT	Math reasoning	500	Teacher-synthesized trajectories with execution and answer checks
SFT	BBH reasoning	500	Teacher-synthesized trajectories from selected BBH-style tasks
SFT	Logic / constraint satisfaction	500	Teacher-synthesized trajectories for solver-verified logic puzzles
SFT	Subtotal	1,500	Human quality control after automatic validation
RL	Math reasoning	12,500	Numeric, symbolic, or exact-match verification
RL	BBH: Boolean Expressions	600	Rule-based or exact-match verification
RL	BBH: Date Understanding	650	Normalized exact-match verification
RL	BBH: Dyck Languages	500	Structured string verification
RL	BBH: Logical Deduction variants	900	Multiple-choice or rule-based verification
RL	BBH: Multi-Step Arithmetic Two	669	Numeric verification
RL	BBH: Tracking Shuffled Objects variants	750	Multiple-choice or state-tracking verification
RL	BBH subtotal	4,069	Verification-friendly reasoning tasks
RL	Logic / constraint satisfaction	6,660	Solver-verified structured matching
RL	Subtotal	23,229	Training and validation for online RL and evaluation
Total	–	24,729	1,500 SFT trajectories and 23,229 RL instances

For the BBH portion of the RL split, we select sub-tasks that are naturally compatible with code-assisted reasoning, deterministic verification, or structured logical evaluation. These tasks allow the execution environment to provide useful intermediate support, such as evaluating Boolean formulas, simulating symbolic state updates, checking bracket structures, and performing arithmetic computation.

D.2 SFT TRAJECTORY SERIALIZATION AND SYNTHESIS

All CoVERBench examples follow a unified multi-turn message schema, so that supervised warm-up and RL share the same visible code-interaction pattern. Here we describe the data-level serialization

and synthesis of completed SFT trajectories; the online execution protocol, stop detector, persistent sandbox, and rollout buffer are described in Appendix E.1.

Each SFT instance is represented as an alternating sequence of user, assistant, and environment messages. The user message contains the task instruction and problem statement. Assistant messages contain policy-generated reasoning, executable code actions, or the final boxed answer. Environment messages contain recorded execution feedback, such as printed outputs or tracebacks, produced by running the corresponding code. This format preserves the action-observation structure of stateful code reasoning while clearly separating policy-generated tokens from environment-generated tokens. During tokenization, the SFT loss is applied only to assistant tokens; user prompts, system instructions, and environment feedback are masked out, mirroring the masked likelihood and masked KL treatment used during RL.

The SFT split contains 1,500 trajectories, with 500 examples each from math reasoning, BBH reasoning, and Knights-and-Knaves logic. Each trajectory is synthesized by a strong teacher model and then manually quality-controlled. Given a problem q, the teacher is prompted to produce a concise multi-turn solution with local reasoning, executable Python code, and a final boxed answer. We typically constrain demonstrations to 2–4 reasoning/code steps and instruct the teacher not to hard-code the final answer; numerical, symbolic, or logical conclusions should be derived through computation or explicit checking.

Generated code blocks are executed to construct the corresponding environment observations. For each trajectory, the code is replayed in a sandbox, and the resulting `stdout`, `stderr`, or traceback is wrapped as an `execution_result` message. Only trajectories that pass format validation, execution validation, and final-answer verification are retained. The final SFT instance is then assembled as a standardized multi-turn conversation consisting of the initial problem, assistant code actions, environment observations, and the final answer.

To provide a supervised warm-up signal for recovery behavior, a subset of SFT trajectories includes recoverable execution errors. During synthesis, the teacher is occasionally instructed to introduce a local bug in an intermediate code block and then repair it after observing the real execution feedback. The injected errors cover common failures in code-assisted reasoning, such as variable-name typos, type mismatches, index errors, missing imports, malformed loops, and off-by-one logic errors. This yields demonstrations of the form

$$\text{buggy code} \;\rightarrow\; \text{traceback observation} \;\rightarrow\; \text{diagnosis and corrected code}.$$

Such trajectories reduce the distribution shift between supervised warm-up and online RL: instead of seeing only flawless code during warm-up, the model also learns to interpret runtime feedback and continue solving after local failures.

D.3 RL Prompt Construction and Domain-Specific Sources

The RL split differs from the SFT split in that it provides no reference reasoning trajectory, code solution, or expert demonstration. Each RL instance contains only the task prompt and verifier-side metadata. During training, rollout workers create a stateful sandbox session, sample actions from the current policy, execute generated code, and compute process and outcome rewards from the resulting interaction.

Each RL instance contains the following fields:

- **Problem text**: the natural-language task statement shown to the model.
- **Interaction instruction**: the same code-interaction protocol used during supervised warm-up.
- **Reference answer**: hidden from the policy and used only by the verifier.
- **Match type**: the verification rule, such as exact match, numeric equivalence, symbolic equivalence, Boolean match, multiple-choice match, or structured dictionary match.
- **Environment metadata**: maximum number of turns, timeout, and task category.

The RL split covers three types of reasoning problems. The math subset contains verifiable quantitative reasoning tasks requiring numerical computation, algebraic manipulation, or multi-step symbolic

reasoning. The BBH subset consists of verification-friendly reasoning tasks, including Boolean Expressions, Date Understanding, Dyck Languages, Logical Deduction variants, Multi-Step Arithmetic Two, and Tracking Shuffled Objects variants. These tasks are naturally compatible with code-assisted reasoning because the sandbox can support intermediate operations such as evaluating Boolean formulas, checking bracket structures, simulating symbolic state updates, or performing arithmetic computation.

The logic and constraint-satisfaction subset consists of programmatically generated Knights-and-Knaves puzzles with structured answer verification. Each puzzle contains a set of characters, each of whom is either a knight, who always tells the truth, or a knave, who always lies. The model must infer the identity of every character from their statements. We generate these puzzles with randomized names, variable numbers of characters, shuffled statements, paraphrased templates, and diverse logical forms such as self-identity claims, statements about other characters, conjunctions, disjunctions, implications, and simple cardinality constraints. Candidate puzzles are solved by exhaustive enumeration over all possible knight/knave assignments, and we retain only instances satisfying the desired solver criterion, typically uniqueness.

The construction of this subset follows two motivations from recent logical-reasoning studies: reducing memorization from repeated templates (Xie et al., 2025a) and enabling deterministic rule-based verification for RL (Xie et al., 2025b). To further reduce template memorization, we deduplicate puzzles by canonical logical signatures rather than surface text alone. Final answers are represented as structured mappings, e.g.,

$$\{\texttt{"Alice"} : \texttt{"Knight"},\ \texttt{"Bob"} : \texttt{"Knave"},\ \texttt{"Carol"} : \texttt{"Knight"}\}.$$

D.4 Preprocessing, Verification, and Quality Control

All CoVERBench data are processed through a unified validation pipeline before training. The pipeline includes format validation, execution validation, answer verification, human inspection for SFT trajectories, and duplication/leakage control.

Format validation. For SFT trajectories, we check that code tags are well-formed, execution-result fields follow the corresponding code actions, role alternation matches the expected conversation schema, and the final assistant response contains a boxed answer. For RL prompts, we verify that the system instruction, problem text, reference answer, match type, and environment metadata are present.

Execution validation. For SFT data, recorded code blocks are replayed in the sandbox, and the replayed outputs must match the stored `execution_result` up to whitespace normalization and non-deterministic traceback paths. Trajectories with inconsistent execution records are discarded or regenerated. This ensures that supervised warm-up uses real environment feedback rather than fabricated observations.

Answer validation. Each final answer is checked by the corresponding verifier. Math problems use exact string match, numeric comparison, or symbolic equivalence depending on the answer type. BBH tasks use task-specific exact or normalized matching, while Knights-and-Knaves tasks use structured dictionary comparison after canonicalizing character names and labels. The same verifier family is used during online RL to compute the terminal outcome reward.

Human quality control. The 1,500 SFT trajectories are synthesized by a teacher model and manually inspected. Human quality control checks whether the trajectory follows the interaction format, whether code execution is meaningful rather than decorative, whether intermediate reasoning is consistent with printed outputs, and whether the final boxed answer is supported by the preceding computation or logical analysis. Trajectories with hallucinated execution results, unsupported answers, or excessive irrelevant code are removed or regenerated.

Leakage and duplication control. For RL, no solution trajectory is stored in the policy-visible prompt; reference answers are retained only in verifier metadata. We remove duplicate or near-duplicate instances by normalized problem text and, for Knights-and-Knaves tasks, by canonical

logical form. This is important for logic tasks, where superficially different statements can encode the same underlying constraint system. These controls remove duplicate, near-duplicate, and logically equivalent instances from the RL data.

E System Runtime and Implementation Details

This appendix provides the system-level details needed to reproduce CoVER's stateful rollout and training pipeline. We first describe the online runtime that parses assistant actions, executes code in a persistent sandbox, returns environment observations, and records mixed agent-environment trajectories with token-level masks. We then report the training configuration, including rollout parameters, optimization hyperparameters, sequence limits, and compute resources used in the main experiments.

E.1 Stateful Runtime and Rollout Implementation

This section provides implementation details for the stateful code-action runtime used by CoVER during online rollout. Appendix D.2 describes how completed SFT trajectories are serialized as data; here we describe how new trajectories are generated through interaction with the execution environment.

Runtime action protocol. During online rollout, each assistant turn is parsed as either an executable code action or a final-answer action. A code action contains a short reasoning plan followed by a Python block:

<think> ⋯ </think> <code> ⋯ </code>.

A final-answer action is detected when the model emits an answer in the form \boxed{·}. The same action protocol is used to serialize SFT trajectories and to collect RL rollouts, while online RL additionally requires live sandbox execution.

Stop detection and execution dispatch. During rollout, generation is monitored by a stop detector. If a closing </code> tag is detected, decoding is interrupted, the enclosed Python code is extracted, and the code is dispatched to the sandbox execution engine. If a final boxed answer is detected instead, the rollout terminates without executing the answer text as code. This design prevents the model from fabricating execution results inside the same assistant turn and ensures that all execution feedback comes from the actual environment.

Persistent sandbox state. The execution environment is implemented as a persistent Python runtime, similar to a Jupyter kernel. Within a single episode, allowed imports, defined variables, intermediate objects, and previous computations are preserved across turns. Thus, later code actions can refer to results produced by earlier executions. The state manager resets the runtime at the beginning of each new rollout, ensuring isolation across prompts and sampled trajectories.

Execution feedback. After executing a code block, the sandbox returns the actual execution feedback. The feedback may contain normal output, runtime errors, timeout messages, or traceback messages, and is serialized as:

<execution_result> ⋯ </execution_result>.

The runtime inserts this message into the context as an environment observation and marks its tokens with $M_k = 0$ in the trajectory buffer. Policy-generated assistant tokens are marked with $M_k = 1$. This masking convention is used consistently in supervised warm-up and in RL, so that losses, KL regularization, and policy-gradient updates are applied only to policy-generated tokens.

Safety and timeout constraints. For safety and stability, sandbox execution is restricted. The runtime blocks dangerous imports and system-level operations, including network access, subprocess creation, process control, and unauthorized file-system operations. File access, when enabled, is limited to the isolated execution workspace for the current rollout. Each execution is also subject to a step-level timeout. If a code block times out, raises an exception, or attempts a disallowed operation, the sandbox returns the corresponding timeout message, exception, or traceback as the next

environment observation. These failures are also exposed to the process-reward function, allowing the policy to learn recovery behavior from real execution feedback.

Trajectory buffer. For each sampled rollout, CoVER stores the serialized trajectory, policy-token mask, old log-probabilities, execution outputs, per-turn process rewards, terminal verification result, and final outcome reward. The buffer is used to compute return-to-go values, depth-wise group-normalized advantages, the masked KL term, and the PR-GRPO objective. This separation between rollout collection and policy optimization ensures that environment feedback is used as verifiable observation and reward signal, while gradients are applied only to policy-generated tokens.

E.2 Training and Implementation Details

This section reports the training and implementation details used for the main CoVER experiments. Unless otherwise specified, the main results use the full curriculum containing BBH, Knights-and-Knaves, and math reasoning tasks. Table 7 summarizes the main hyperparameters and system configuration. The SFT stage is used primarily to initialize the interaction protocol and recovery behavior. Trajectory retention denotes the fraction of valid sampled rollouts retained for the PR-GRPO update.

Table 7: Hyperparameters and implementation details for CoVER training.

Group	Hyperparameter	Value
Model	Backbone	Qwen3-8B
	Initialization	SFT warm-up checkpoint
	SFT data	1,500 trajectories, 500 each from BBH, Knights, and Math
	SFT epochs / learning rate / warm-up ratio	3 / 2×10^{-5} / 0.1
RL algorithm	Optimizer	PR-GRPO
	Advantage estimator	Step-level group-normalized return-to-go
	Rollout group size G	5
	Trajectory retention	80% of valid rollouts
	Prompt batch size $\lvert\mathcal{B}\rvert$	32
	Update mini-batch size	32
	Micro-batch size per GPU	1
	Actor optimizer / learning rate	AdamW / 1×10^{-6}
	Discount factor γ	0.95
	KL loss type / coefficient	`low_var_kl` / 1×10^{-3}
	KL in reward	False; KL is applied only in the loss
	Entropy coefficient	0
Rollout	Rollout engine	vLLM
	Tensor parallel size	2
	Temperature / top-p / top-k	0.6 / 0.95 / 20
	Decoding stop	`</code>` or final-answer detector
Multi-turn interaction	Maximum assistant / environment turns	9 / 8
	Maximum tool-response length	1,024 tokens
	Agent loop	Stateful tool-use rollout
Sequence length	Maximum prompt length	2,048 tokens
	Maximum generated response length	8,192 tokens
	Approximate total context length	10,240 tokens
Training system	Hardware	16 A100-80GB GPUs, 2 nodes
	Gradient checkpointing	True
	Reference parameter offload	True
	Total RL epochs	2

F Empirical Analysis and Ablation Studies

F.1 Ablation Study

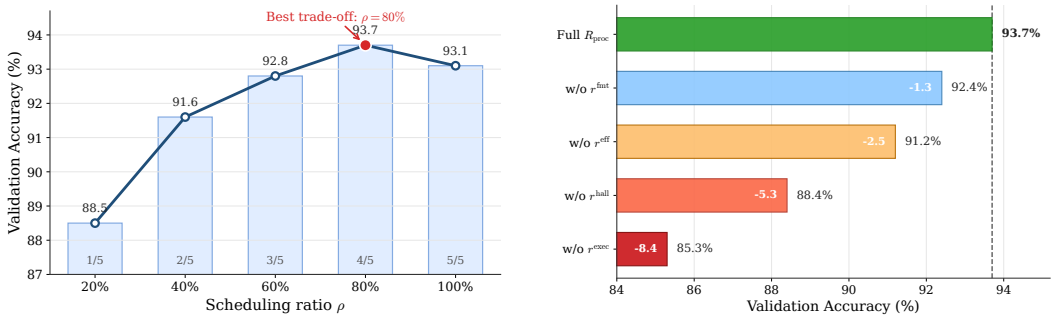
We conduct ablation studies to isolate the key design choices of CoVER. Unless otherwise specified, all variants use the same Qwen3-8B backbone, supervised warm-up checkpoint, CoVERBench RL split, rollout group size $G = 5$, optimizer, decoding configuration, maximum interaction horizon, and verifier. The full model denotes CoVER trained with PR-GRPO, masked optimization, step-level advantage estimation, the complete code-verified process reward, and trajectory scheduling with the best ratio.

For trajectory scheduling, each prompt samples $G = 5$ trajectories in the stateful sandbox. Before the PR-GRPO update, trajectories are ranked by their cumulative process reward,

$$s_i = \sum_{t=1}^{T_i} R_{\text{proc}}(h_{i,t-1}, a_{i,t}, o_{i,t}), \tag{33}$$

where the terminal outcome reward is excluded. We retain the top κ fraction of trajectories for policy optimization. Since $G = 5$, $\kappa \in \{20\%, 40\%, 60\%, 80\%, 100\%\}$ exactly corresponds to retaining $1, 2, 3, 4, 5$ trajectories. As shown in Figure 6a, retaining 80% of trajectories achieves the best validation accuracy. This setting removes the worst rollout in each group, such as early-collapsed, invalid-code, or hallucinated-feedback trajectories, while preserving enough positive and negative contrast for group-relative learning. Retaining only 20% of trajectories lowers accuracy to 88.5%, indicating that recoverable failures provide useful contrast for learning error diagnosis and correction, while all settings from 60% to 100% stay within 92.8–93.7% (Table 14).

Table 3 in the main text ablates the two core mechanisms of PR-GRPO ("Err." denotes the error encounter rate and "Rec." the recovery rate). Removing step-level advantage estimation reduces validation accuracy from 93.7% to 89.2%, and lowers the recovery rate from 74.7% to 58.3%. This confirms that sequence-level credit assignment is too coarse for stateful multi-turn execution: a useful early code action may be incorrectly penalized if a later reasoning step fails. Removing masked optimization is more damaging, dropping validation accuracy to 68.4%, increasing the error encounter rate to 31.5%, and reducing recovery to 18.2%. This is because stdout, stderr, and traceback tokens are generated by the environment rather than the policy. Including them in log-probability or KL terms introduces spurious gradients and forces the policy to fit sandbox outputs instead of optimizing its own actions. Standard GRPO obtains 82.1% validation accuracy, with a 14.6% error encounter rate and 46.5% recovery rate, showing that outcome-only reward provides insufficient supervision for stateful execution.



(a) Trajectory scheduling ratio.

(b) Process reward components.

Figure 6: Ablation studies on trajectory scheduling and code-verified process rewards.

Finally, we study the contribution of each component in the code-verified process reward. As shown in Figure 6b, removing r_{exec} causes the largest degradation, reducing validation accuracy to 85.3%. This shows that direct execution feedback is essential for learning reliable runnable code. Removing the hallucination / claim-grounding penalty r_{hall} also substantially hurts performance, dropping accuracy to 88.4%. This ablation disables fabricated-feedback detection and grounding checks for unsupported checkable claims, making the model more likely to fabricate sandbox feedback, ignore execution evidence, or draw conclusions not supported by computed outputs. The efficiency term r_{eff} prevents stalling and redundant code executions, while the formatting reward r_{fmt} stabilizes the code-interaction protocol. Overall, these ablations show that CoVER's gains come from the combination of trajectory scheduling, masked optimization, step-level credit assignment, and execution-grounded process rewards.

F.2 CoVERBench Validation Error Analysis

This section analyzes the remaining failure modes on the CoVERBench validation set. The validation set contains 2,322 examples and is constructed to evaluate mathematical reasoning and logic-oriented

reasoning under the stateful code-execution interface. At the reported validation accuracy of 93.7%, the evaluated CoVER model produces 146 incorrect predictions. We audit these incorrect predictions and categorize each failure according to its dominant cause. The analysis characterizes the residual errors after CoVER has learned stable code interaction.

Table 8 summarizes the distribution of audited error types. The dominant category is *reasoning error*, accounting for 61.6% of the remaining errors. In these cases, the generated code is syntactically valid and usually executes successfully, but the model implements an incorrect mathematical strategy, misses a key condition, or draws an invalid conclusion from intermediate results.

The second-largest category is *false negative*, accounting for 19.2% of errors. These are cases where the model's answer is semantically correct but rejected by the automatic verifier due to formatting or normalization mismatch, such as decimal-versus-fraction forms or equivalent symbolic expressions. The automatic verifier can therefore underestimate CoVER's true accuracy.

Hallucination accounts for 13.0% of the remaining errors. These failures occur when the model fabricates intermediate quantities, ignores execution outputs, reports unsupported test or constraint results, or produces a final answer unsupported by the computed evidence. This category is the target of the claim-grounding part of r_{hall}, which penalizes mechanically checkable unsupported claim during training. Diagram understanding errors account for 4.8% and mainly appear in geometry or spatial reasoning problems. Low-level execution and tool-use failures are rare: runtime errors and syntax or inappropriate tool-usage errors each account for only 0.7% of the audited errors. This supports the conclusion that CoVER successfully teaches the model to produce well-formed and executable code, leaving high-level reasoning as the primary bottleneck.

We note that this analysis differs from the error-encounter statistics in Section 5.4. There, an error is counted whenever a traceback appears during an intermediate code-execution turn. Here, we count only the dominant cause of final incorrect validation predictions. Thus, many intermediate execution errors that are successfully recovered do not appear as final validation failures.

Table 8: Error-type distribution on the CoVERBench validation set. Percentages are computed over 146 audited incorrect predictions.

Error type	Count	Share	Avg. confidence
Reasoning error	90	61.6%	0.975
False negative	28	19.2%	0.985
Hallucination	19	13.0%	0.983
Diagram understanding error	7	4.8%	0.960
Runtime error	1	0.7%	0.900
Syntax error / inappropriate tool usage	1	0.7%	0.900

We further break down the audited errors by problem difficulty in Table 9. The distribution shows that the nature of residual errors changes with difficulty. For easy problems, false negatives and diagram-related failures take a relatively larger share, indicating that some apparent failures are caused by verifier mismatch or surface interpretation issues. For harder problems, reasoning errors dominate: their share increases from 42.9% on easy problems to 64.6% on hard problems. This pattern shows that code generation is no longer CoVER's bottleneck: code-level failures account for only 1.4% of the residual errors, which concentrate in high-level reasoning on hard problems.

Table 9: Error-type distribution by difficulty. Percentages are normalized within each difficulty group.

Difficulty	Reasoning error	False negative	Hallucination	Diagram understanding	Runtime error	Syntax / tool usage
Easy	42.9%	28.6%	7.1%	14.3%	0.0%	7.1%
Medium	60.6%	18.2%	15.2%	6.1%	0.0%	0.0%
Hard	64.6%	18.2%	13.1%	3.0%	1.0%	0.0%

Takeaways. The error analysis yields two main observations. First, CoVER largely resolves low-level execution and tool-use failures: syntax errors, inappropriate tool usage, and unrecovered runtime errors together account for only 1.4% of the remaining validation errors. Second, the remaining failures are dominated by reasoning-level mistakes, especially on harder problems. The natural levers

for further gains are therefore stronger symbolic planning and richer answer normalization in the verifier, the latter targeting the 19.2% false-negative share.

F.3 Analysis of Execution-Grounded Process Supervision

This section provides the details underlying Figure 1. We analyze 300 PRM800K (Lightman et al., 2024) problems solved by Qwen3-8B (Yang et al., 2025) under a stateful code-execution interface, where trajectories interleave model reasoning, generated code, sandbox feedback, revisions, and final answers. Unlike static reasoning traces, stateful TIR exposes dynamic supervision signals such as execution errors, traceback types, recovery behavior, hallucinated feedback, and interaction efficiency.

Table 10: Compact statistics underlying Figure 1. Percentages are rounded to match the visualization.

Panel	Category	Value
Overall	Total trajectories	300
	Execution-error trajectories	111
	Incorrect-final-answer trajectories	126
	Correct-final-answer trajectories	174
Recoverability	Reasoning-level failures	64%
	Recoverable errors	36%
Failure stage	Early-collapse / off-track failure	68%
	Last-stage failure	22%
	Intermediate failure	10%

Table 10 summarizes the statistics visualized in Figure 1. Among 300 trajectories, 111 produce execution errors and 126 yield incorrect final answers. For execution-error trajectories, 36% are locally recoverable, while 64% correspond to reasoning-level failures. For incorrect trajectories, failures occur at heterogeneous stages: 68% collapse early, 22% are near-correct but fail at the last stage, and 10% fail in an intermediate stage. These results show that neither final correctness nor execution success alone is sufficient to characterize process quality.

Many failures are recoverable and highly heterogeneous. The first phenomenon is that execution errors are not homogeneous. Some errors indicate a fundamentally wrong solution strategy, but others are local implementation mistakes that can be repaired once the model observes sandbox feedback. Example 1-2 illustrates this distinction.

Example 1: Locally Repairable Error — Correct reasoning with a surface-level syntax bug

Problem. Find the number of solutions in the interval $[0, 2\pi]$ to

$$\tan x + \sec x = 2\cos x.$$

Ground truth. 2

Model output excerpt.

```
# Solve tan(x) + sec(x) = 2*cos(x)
from sympy import *

x = symbols('x')
solutions = solve(tan(x) + sec(x) - 2*cos(x), x)

# Filter solutions in [0, 2*pi]
valid = [s for s in solutions if 0 <= s <= 2\pi]
print(len(valid))
```

Execution result.

```
  File "<stdin>", line 8
    valid = [s for s in solutions if 0 <= s <= 2\pi]
                                                  ^
SyntaxError: unexpected character after line continuation character
```

Annotation. Locally repairable with confidence 0.95. The model identified a plausible symbolic-solving strategy, but the generated code contains a surface-level syntax issue caused by mixing LaTeX notation with Python syntax. Replacing the raw `\pi` with `pi` would repair the program without changing the mathematical approach.

Example 2: Reasoning-Level Failure — Execution error caused by an unrelated strategy

Problem. Find the integer n, $0 \le n \le 5$, such that

$$n \equiv -3736 \pmod 6.$$

Ground truth. 2

Model output excerpt.

```python
# Try to compute a recursive sequence value related to 3736.
def fib(n):
    return fib(n - 1) + fib(n - 2)

result = fib(3736)
print(result % 6)
```

Execution result.

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
    result = fib(3736)
  File "<stdin>", line 3, in fib
    return fib(n - 1) + fib(n - 2)
  File "<stdin>", line 3, in fib
    return fib(n - 1) + fib(n - 2)
  File "<stdin>", line 3, in fib
    return fib(n - 1) + fib(n - 2)
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Annotation. Reasoning-level failure with confidence 1.0. The generated program attempts to compute a Fibonacci-style recursion, which is unrelated to the modular arithmetic problem. The correct solution only requires reducing -3736 modulo 6:

$$-3736 \equiv 2 \pmod 6.$$

Even if the recursion error were repaired by adding base cases or memoization, the underlying strategy would still be incorrect. This example should therefore receive a qualitatively different process signal from Example 1: the failure is not a local implementation bug, but a fundamental reasoning error.

Same answer, different process quality. The second phenomenon is that final correctness can hide large differences in process quality. Two trajectories may reach the same answer, but one may be grounded and efficient while the other relies on brittle or hallucinated corrections. This is the setting illustrated in Figure 1b.

Example 3: Same Correct Answer, Different Process Quality

Problem. Count the number of ordered triples (a, b, c) of nonnegative integers such that

$$a + b + c = 20, \qquad a \le 8, \quad b \le 10, \quad c \le 12.$$

Ground truth. 63

Trajectory A: grounded and efficient.

Turn 1.

```python
# Use stars and bars with inclusion-exclusion.
# Without upper bounds, the number of nonnegative solutions to
# a + b + c = 20 is C(22, 2).
#
# Subtract single violations:
# a >= 9  -> a' + b + c = 11, contributing C(13, 2)
# b >= 11 -> a + b' + c = 9,  contributing C(11, 2)
# c >= 13 -> a + b + c' = 7,  contributing C(9, 2)
#
# Add back feasible double violations.
# Only a >= 9 and b >= 11 is possible:
# a' + b' + c = 0, contributing 1.
# The other double violations have negative remaining sum.

from math import comb

total = comb(22, 2)
bad_a = comb(13, 2)
bad_b = comb(11, 2)
bad_c = comb(9, 2)
add_back = 1

ans = total - bad_a - bad_b - bad_c + add_back
print(ans)
```

Execution output.

```
63
```

Final answer.

$$\boxed{63}$$

Trajectory B: hallucinated correction but eventually correct.

Turn 1.

```python
# Attempt inclusion-exclusion, but use an unjustified template:
# subtract all single violations and then add back "two boundary overlaps".
# This is not derived from the constraints. In fact, only one double
# violation is feasible.

from math import comb

total = comb(22, 2)
bad_a = comb(13, 2)
bad_b = comb(11, 2)
bad_c = comb(9, 2)

# Hallucinated correction: assume two pairwise boundary overlaps exist.
ans = total - bad_a - bad_b - bad_c + 2
print(ans)
```

Execution output.

```
64
```

Turn 2.

```python
# The previous result is close. Assume one endpoint case was
# double-counted and subtract 1.
# This is a post-hoc correction, not a derived inclusion-exclusion term.

ans = 64 - 1
print(ans)
```

Execution output.

```
63
```

Final answer.

$$\boxed{63}$$

Analysis. Both trajectories reach the same correct answer and would receive identical outcome rewards. However, their process quality is fundamentally different. Trajectory A derives the correct inclusion-exclusion formula and uses code as a faithful executable verifier. Trajectory B first applies an invalid inclusion-exclusion

correction, obtains the wrong intermediate result 64, and then performs an unsupported post-hoc adjustment to arrive at 63. The final answer is correct, but the path is not faithful: the correction is hallucinated rather than grounded in the feasible constraint intersections. This illustrates why final correctness alone is insufficient.

The statistics and examples show why stateful TIR benefits from execution-grounded process supervision. Sandbox feedback makes local implementation errors, recovery behavior, failure stage, and inefficient tool use observable. CoVER converts these online signals into process rewards, enabling RL to prefer trajectories that are not only correct, but also faithful, recoverable, and efficient.

G REPRESENTATIVE COVER SOLUTION TRAJECTORIES

This section presents representative solution trajectories generated by CoVER. The examples cover mathematical competition problems, mathematical reasoning, logical reasoning, elementary arithmetic, algorithmic state tracking, and code generation. Each trajectory illustrates the same code-verified interaction pattern: the model first formulates an executable subproblem, runs code in the sandbox, observes real execution feedback, and then revises or finalizes its answer. For readability, we show compact excerpts rather than the full raw transcripts. All execution outputs shown below are sandbox-generated observations.

AIME 2024 I-3 Game-Theory Counting — Dynamic programming with executable verification

Task. Alice and Bob play the following game. A stack of n tokens lies before them. The players take turns, with Alice going first. On each turn, the player removes either 1 token or 4 tokens from the stack. Whoever removes the last token wins. Find the number of positive integers $n \leq 2024$ for which Bob has a strategy that guarantees Bob will win regardless of Alice's play.
Ground truth. 809

Trajectory summary. CoVER formulates the game as a finite dynamic program. Let `dp[i]` denote whether the player to move with i tokens can force a win. A state is losing if all legal moves lead to winning states for the opponent. Since Alice moves first, Bob can force a win exactly when the initial state is losing for the player to move.

Turn 1.
<think> Set up the losing positions using dynamic programming. A position is winning if there exists a legal move, removing either 1 or 4 tokens, that leaves the opponent in a losing position. Compute all losing positions up to 2024.

```python
max_n = 2024

# dp[i] = True means the player to move at i tokens can force a win.
# dp[i] = False means the player to move at i tokens loses with optimal play.
dp = [False] * (max_n + 1)

# Base case: with 0 tokens, the player to move cannot move and loses.
dp[0] = False

for i in range(1, max_n + 1):
    can_take_1 = (i - 1 >= 0 and dp[i - 1] == False)
    can_take_4 = (i - 4 >= 0 and dp[i - 4] == False)

    if can_take_1 or can_take_4:
        dp[i] = True
    else:
        dp[i] = False

losing_positions = [i for i in range(1, max_n + 1) if dp[i] == False]

print("First losing positions:", losing_positions[:20])
print("Count:", len(losing_positions))
```

Execution output.

```
First losing positions: [2, 5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40, 42,
      45, 47, 50]
Count: 809
```

Turn 2.
<think> Verify the computed losing positions by checking small cases and identifying the periodic structure. The first few losing positions suggest that the losing states are exactly those congruent to 0 or 2 modulo 5.

```python
# Verify the residue pattern of the losing positions.
residues = sorted(set(i % 5 for i in losing_positions))
print("Residues modulo 5:", residues)

# Check that every number n <= 2024 with residue 0 or 2 modulo 5 is losing,
# and that no other number is losing.
pattern_positions = [i for i in range(1, max_n + 1) if i % 5 in {0, 2}]

print("Pattern count:", len(pattern_positions))
print("Matches DP:", pattern_positions == losing_positions)
```

Execution output.

```
Residues modulo 5: [0, 2]
Pattern count: 809
Matches DP: True
```

Turn 3.
<think> Compute the final answer using the verified modular pattern. The valid starting sizes are exactly $n \equiv 0 \pmod 5$ or $n \equiv 2 \pmod 5$.

```
# Count n <= 2024 such that n is 0 or 2 modulo 5.

count_mod_0 = 2024 // 5

# Numbers congruent to 2 modulo 5 are:
# 2, 7, 12, ..., 2022.
count_mod_2 = ((2024 - 2) // 5) + 1

answer = count_mod_0 + count_mod_2

print("Count with residue 0 modulo 5:", count_mod_0)
print("Count with residue 2 modulo 5:", count_mod_2)
print("Answer:", answer)
```

Execution output.

```
Count with residue 0 modulo 5: 404
Count with residue 2 modulo 5: 405
Answer: 809
```

Turn 4.
<think> The dynamic program and the modular-pattern verification agree. Therefore, the number of positive integers $n \leq 2024$ for which Bob can force a win is 809.
Final answer.

$$\boxed{809}$$

Analysis. This trajectory illustrates execution-grounded mathematical reasoning. CoVER first constructs the standard winning-state recurrence for a subtraction game and uses code execution to enumerate all losing states up to 2024. It then inspects the computed states, discovers the period-5 structure, and independently verifies that the losing states are precisely those with residues 0 and 2 modulo 5. Finally, it converts this verified pattern into a closed-form count, $404 + 405 = 809$. Thus the code execution serves both as a computational tool and as a consistency check for the derived mathematical structure.

AIME 2025 II-6 Coordinate Geometry — Symbolic derivation with executable verification

Task. Circle ω_1 with radius 6 centered at point A is internally tangent at point B to circle ω_2 with radius 15. Points C and D lie on ω_2 such that $\overline{BC}$ is a diameter of ω_2 and $\overline{BC} \perp \overline{AD}$. Rectangle $EFGH$ is inscribed in ω_1 such that $\overline{EF} \perp \overline{BC}$. Triangles $\triangle DGF$ and $\triangle CHG$ have equal areas. The area of rectangle $EFGH$ is m/n, where m and n are relatively prime positive integers. Find $m + n$.
Ground truth. 293

Trajectory summary. CoVER places the larger circle at the origin, derives the key points A, B, C, D, parameterizes the inscribed rectangle by its half-width and half-height, and uses the equal-area constraint to solve for the rectangle. The final area is then verified symbolically.

Turn 1.
<think> Set up coordinates. Put the center of ω_2 at $(0, 0)$ with radius 15. Since ω_1 has radius 6 and is internally tangent to ω_2 at $B = (15, 0)$, its center is $A = (9, 0)$. Then $C = (-15, 0)$. Because $\overline{AD}$ is perpendicular to $\overline{BC}$, point D lies on the vertical line through A, so $D = (9, 12)$.

```
import sympy as sp

# Coordinates:
# Larger circle omega_2: center O = (0, 0), radius 15.
# Smaller circle omega_1: center A = (9, 0), radius 6.
A = sp.Point(9, 0)
B = sp.Point(15, 0)
C = sp.Point(-15, 0)
D = sp.Point(9, 12)

# Verify D lies on omega_2.
```

```
print("Distance OD squared:", D.x**2 + D.y**2)
print("Radius squared:", 15**2)

# Verify D is vertically above A, hence AD is perpendicular to BC.
print("A:", A)
print("B:", B)
print("C:", C)
print("D:", D)
```

Execution output.

```
Distance OD squared: 225
Radius squared: 225
A: Point2D(9, 0)
B: Point2D(15, 0)
C: Point2D(-15, 0)
D: Point2D(9, 12)
```

Turn 2.
<think> Parameterize the rectangle. Since EF is perpendicular to BC, the rectangle has vertical and horizontal sides. Because it is inscribed in the circle centered at $A = (9, 0)$, its center is A. Let the half-width be a and the half-height be b. Then

$$E = (9 + a, -b), \quad F = (9 + a, b), \quad G = (9 - a, b), \quad H = (9 - a, -b),$$

and the circle condition is $a^2 + b^2 = 36$.
The area of $\triangle DGF$ is $a(12 - b)$, since $GF = 2a$ and the height from D to GF is $12 - b$. The area of $\triangle CHG$ is $b(24 - a)$, since $HG = 2b$ and the horizontal distance from $C = (-15, 0)$ to the line $x = 9 - a$ is $24 - a$. Equal areas give

$$a(12 - b) = b(24 - a).$$

```
a, b = sp.symbols("a b", positive=True)

# Circle condition for a rectangle centered at A and inscribed in omega_1.
circle_eq = sp.Eq(a**2 + b**2, 36)

# Equal-area condition:
# area(DGF) = a * (12 - b)
# area(CHG) = b * (24 - a)
area_eq = sp.Eq(a * (12 - b), b * (24 - a))

print("Circle equation:", circle_eq)
print("Area equation:", area_eq)

# Simplify the area equation.
simplified_area_eq = sp.simplify(a * (12 - b) - b * (24 - a))
print("Simplified area difference:", simplified_area_eq)
```

Execution output.

```
Circle equation: Eq(a**2 + b**2, 36)
Area equation: Eq(a*(12 - b), b*(24 - a))
Simplified area difference: 12*a - 24*b
```

Turn 3.
<think> The equal-area equation simplifies to $12a - 24b = 0$, so $a = 2b$. Substitute this into $a^2 + b^2 = 36$ to determine a and b, then compute the rectangle area $4ab$.

```
# From 12*a - 24*b = 0, we have a = 2*b.
solution_b = sp.solve(sp.Eq((2*b)**2 + b**2, 36), b)
print("Solutions for b:", solution_b)

# Use the positive solution.
b_val = [s for s in solution_b if s > 0][0]
a_val = 2 * b_val

area_rect = 4 * a_val * b_val
area_rect = sp.simplify(area_rect)

print("a:", a_val)
print("b:", b_val)
print("Rectangle area:", area_rect)
```

Execution output.

```
Solutions for b: [6*sqrt(5)/5]
a: 12*sqrt(5)/5
b: 6*sqrt(5)/5
Rectangle area: 288/5
```

Turn 4.
<think> The rectangle area is $288/5$. Thus $m = 288$ and $n = 5$, which are relatively prime, so $m + n = 293$.

```
from math import gcd

m = 288
n = 5

print("gcd:", gcd(m, n))
print("m+n:", m + n)
```

Execution output.

```
gcd: 1
m+n: 293
```

Final answer.

$$\boxed{293}$$

Analysis. This example illustrates how CoVER uses code execution to support a coordinate geometry derivation. The model first fixes a coordinate system consistent with the internal tangency, obtaining $A = (9, 0)$, $B = (15, 0)$, $C = (-15, 0)$, and $D = (9, 12)$. It then parameterizes the inscribed rectangle by half-width a and half-height b. The equal-area constraint simplifies to $a = 2b$, while the circle constraint gives $a^2 + b^2 = 36$. Solving these equations yields $a = 12/\sqrt{5}$ and $b = 6/\sqrt{5}$, so the rectangle area is $4ab = 288/5$. Therefore $m + n = 293$.

CoVERBench Logical Reasoning — Exhaustive verification for a Knights-and-Knaves puzzle

Task. A special island is inhabited only by knights and knaves. Knights always tell the truth, and knaves always lie. We meet six inhabitants: Lily, Isabella, Ella, James, Elizabeth, and Sebastian. Their statements are:

- Lily: "James is not a knight."
- Isabella: "Ella is a knight or Ella is a knave."
- Ella: "Lily is a knave and Lily is a knight."
- James: "Elizabeth is a knave or Ella is a knight."
- Elizabeth: "If Ella is a knave then Isabella is a knave."
- Sebastian: "Lily is a knight and Elizabeth is a knight."

Determine who is a knight and who is a knave, and output the result as a JSON dictionary.
Ground truth.

```
{"Lily": false, "Isabella": true, "Ella": false, "James": true, "Elizabeth": false, "
    Sebastian": false}
```

Trajectory summary. CoVER solves the logical puzzle by exhaustive enumeration over all 2^6 possible truth assignments. For each assignment, it checks whether each person's statement has the same truth value as that person's type: true for a knight and false for a knave. The trajectory also illustrates a small interaction-level repair: the model initially constructs the correct solution list but forgets to print it, observes empty sandbox output, and then explicitly prints the result.

Turn 1.
<think> Enumerating all truth assignments with code.

```
from itertools import product

# Persons: Lily, Isabella, Ella, James, Elizabeth, Sebastian
names = ["Lily", "Isabella", "Ella", "James", "Elizabeth", "Sebastian"]

```

```
solutions = []

for vals in product([True, False], repeat=6):
    Lily, Isabella, Ella, James, Elizabeth, Sebastian = vals

    def says(speaker, statement):
        return statement if speaker else not statement

    ok = True

    # Lily: "James is not a knight."
    ok &= says(Lily, (not James))

    # Isabella: "Ella is a knight or Ella is a knave."
    # This is a tautology.
    ok &= says(Isabella, (Ella or (not Ella)))

    # Ella: "Lily is a knave and Lily is a knight."
    # This is a contradiction.
    ok &= says(Ella, ((not Lily) and Lily))

    # James: "Elizabeth is a knave or Ella is a knight."
    ok &= says(James, ((not Elizabeth) or Ella))

    # Elizabeth: "If Ella is a knave then Isabella is a knave."
    # In Python booleans, p <= q matches implication p -> q.
    ok &= says(Elizabeth, ((not Ella) <= (not Isabella)))

    # Sebastian: "Lily is a knight and Elizabeth is a knight."
    ok &= says(Sebastian, (Lily and Elizabeth))

    if ok:
        solutions.append(dict(zip(names, vals)))

solutions
```

Execution output.

```
(no output)
```

Turn 2.
<think> We need to print the solutions list to see the result.

```
solutions
```

Execution output.

```
(no output)
```

Turn 3.
<think> We must explicitly print the variable; simply writing its name does not display it in this environment.

```
print(solutions)
```

Execution output.

```
[{'Lily': False, 'Isabella': True, 'Ella': False, 'James': True, 'Elizabeth': False, '
    Sebastian': False}]
```

Turn 4.
<think> Based on the computed assignment, the final answer is the corresponding JSON dictionary.
Final answer.

```
{"Lily": false, "Isabella": true, "Ella": false, "James": true, "Elizabeth": false, "
    Sebastian": false}
```

Analysis. This trajectory demonstrates execution-grounded logical reasoning through finite model checking. Instead of relying on informal case analysis, CoVER represents each inhabitant's type as a Boolean variable and enumerates all possible assignments. The helper function `says` enforces the Knights-and-Knaves semantics: a knight's statement must be true, while a knave's statement must be false.

The resulting unique satisfying assignment is:

$$\text{Lily} = \text{false}, \quad \text{Isabella} = \text{true}, \quad \text{Ella} = \text{false},$$
$$\text{James} = \text{true}, \quad \text{Elizabeth} = \text{false}, \quad \text{Sebastian} = \text{false}.$$

The process also shows a lightweight but important form of tool-use correction. The first two turns compute or reference the solution list but do not produce visible output in the sandbox. After observing the empty execution result, CoVER corrects the interaction by calling `print(solutions)`, obtaining the verified assignment before producing the final JSON answer.

CoVERBench Mathematical Reasoning — Rectilinear geometry with area verification

Task. The area of polygon $ABCDEF$ is 52, with $AB = 8$, $BC = 9$, and $FA = 5$. The polygon is rectilinear as shown in the diagram: AB is horizontal, BC is vertical, CD is horizontal, DE is vertical, EF is horizontal, and FA is vertical. Find $DE + EF$.
Ground truth. 9

Trajectory summary. CoVER reconstructs the right-angled polygon from the side-length constraints. The polygon can be viewed as an 8×9 rectangle with a smaller bottom-left rectangle removed. Since $BC = 9$ and $FA = 5$, the removed rectangle has height $9 - 5 = 4$, which is also DE. The total area is 52, so the removed area is $72 - 52 = 20$, giving $EF = 20/4 = 5$. Therefore $DE + EF = 4 + 5 = 9$.

Turn 1.
<think> Set up coordinates for the rectilinear polygon. Use the known lengths $AB = 8$, $BC = 9$, and $FA = 5$. The full bounding rectangle has width 8 and height 9. Since $FA = 5$, the lower horizontal segment EF lies 5 units below the top, so its height above the bottom is $9 - 5 = 4$. Thus $DE = 4$. Let $EF = x$, then solve for x using the given area.

```python
import sympy as sp

# Given lengths
AB = 8
BC = 9
FA = 5
area_polygon = 52

# The bounding rectangle has dimensions AB by BC.
area_rectangle = AB * BC

# Since FA is the vertical segment from F up to A,
# the lower notch has height BC - FA.
DE = BC - FA

# Let EF = x. The removed rectangle has area DE * EF.
x = sp.symbols("x", positive=True)

eq = sp.Eq(area_rectangle - DE * x, area_polygon)
solution = sp.solve(eq, x)

print("Bounding rectangle area:", area_rectangle)
print("DE:", DE)
print("Solution for EF:", solution)
```

Execution output.

```
Bounding rectangle area: 72
DE: 4
Solution for EF: [5]
```

Turn 2.
<think> Verify the result by constructing coordinates for the polygon and computing its area using the shoelace formula. With $DE = 4$ and $EF = 5$, a consistent coordinate placement is

$$A = (0, 9), \quad B = (8, 9), \quad C = (8, 0), \quad D = (5, 0), \quad E = (5, 4), \quad F = (0, 4).$$

This gives $AB = 8$, $BC = 9$, $FA = 5$, and area 52.

```python
# Verify by shoelace formula.
coords = [
    (0, 9),  # A
    (8, 9),  # B
    (8, 0),  # C
    (5, 0),  # D
```

```
    (5, 4),  # E
    (0, 4),  # F
]

def shoelace(poly):
    total = 0
    n = len(poly)
    for i in range(n):
        x1, y1 = poly[i]
        x2, y2 = poly[(i + 1) % n]
        total += x1 * y2 - x2 * y1
    return abs(total) / 2

computed_area = shoelace(coords)

# Compute the relevant side lengths.
AB_len = coords[1][0] - coords[0][0]
BC_len = coords[1][1] - coords[2][1]
DE_len = coords[4][1] - coords[3][1]
EF_len = coords[4][0] - coords[5][0]
FA_len = coords[0][1] - coords[5][1]

print("Area:", computed_area)
print("AB:", AB_len)
print("BC:", BC_len)
print("DE:", DE_len)
print("EF:", EF_len)
print("FA:", FA_len)
print("DE + EF:", DE_len + EF_len)
```

Execution output.

```
Area: 52.0
AB: 8
BC: 9
DE: 4
EF: 5
FA: 5
DE + EF: 9
```

Turn 3.
<think> The coordinate verification matches all given side lengths and the required area. Therefore $DE + EF = 9$.
Final answer.

$$\boxed{9}$$

Analysis. This example illustrates how CoVER grounds a short geometric argument through explicit executable checks. The key observation is that the right-angled hexagon is the outer 8×9 rectangle with a bottom-left rectangle removed. Because $BC = 9$ and $FA = 5$, the removed rectangle has height

$$DE = 9 - 5 = 4.$$

The outer rectangle has area $8 \cdot 9 = 72$, while the polygon has area 52. Hence the removed rectangle has area $72 - 52 = 20$. Since its height is 4, its width is

$$EF = \frac{20}{4} = 5.$$

Therefore,

$$DE + EF = 4 + 5 = 9.$$

The sandbox verification reconstructs coordinates with these side lengths and confirms the area by the shoelace formula before producing the final answer.

GSM8K Elementary Math — Two-turn arithmetic verification

Task. Peter has $70 and wishes to spend an equal amount each day for one week. From Sunday through Wednesday, he spends his money on wooden action figures, which cost $5 each. For the rest of the week, he buys plastic action figures, which cost $2 each. How many total action figures will he have by the end of the week?

Ground truth. 23

Trajectory summary. CoVER computes the daily spending as $\$70/7 = \$10$. Sunday through Wednesday contains four days, so Peter spends $\$40$ on wooden figures and buys 8 of them. The remaining three days account for $\$30$, which buys 15 plastic figures. The total number of action figures is $8 + 15 = 23$.

Turn 1.
<think> Peter spends an equal amount each day for one week. First compute the daily budget. Then count the number of days from Sunday through Wednesday and the number of remaining days. Use the two item prices to compute wooden and plastic figures separately.

```
# Peter has 70 dollars and spends the same amount each day for 7 days.
total_money = 70
days_in_week = 7
daily_spending = total_money / days_in_week

# Sunday through Wednesday is 4 days: Sunday, Monday, Tuesday, Wednesday.
wooden_days = 4

# The rest of the week is Thursday, Friday, Saturday.
plastic_days = days_in_week - wooden_days

# Costs of the two types of action figures.
wooden_cost = 5
plastic_cost = 2

# Spending and item counts.
wooden_spending = wooden_days * daily_spending
plastic_spending = plastic_days * daily_spending

wooden_figures = wooden_spending / wooden_cost
plastic_figures = plastic_spending / plastic_cost

total_figures = wooden_figures + plastic_figures

print("Daily spending:", daily_spending)
print("Wooden spending:", wooden_spending)
print("Plastic spending:", plastic_spending)
print("Wooden figures:", wooden_figures)
print("Plastic figures:", plastic_figures)
print("Total figures:", total_figures)
```

Execution output.

```
Daily spending: 10.0
Wooden spending: 40.0
Plastic spending: 30.0
Wooden figures: 8.0
Plastic figures: 15.0
Total figures: 23.0
```

Turn 2.
<think> The executed calculation gives 8 wooden figures and 15 plastic figures, for a total of 23 action figures. Print only the final answer.

```
daily_spending = 70 / 7

wooden_days = 4
plastic_days = 3

wooden_figures = (wooden_days * daily_spending) / 5
plastic_figures = (plastic_days * daily_spending) / 2

answer = int(wooden_figures + plastic_figures)
print(answer)
```

Execution output.

```
23
```

Final answer.

$$\boxed{23}$$

Analysis. This trajectory demonstrates a compact execution-grounded solution for an elementary arithmetic word problem. The main reasoning step is the temporal decomposition: Sunday through Wednesday is four

days, leaving three days for the rest of the week. Since Peter spends $10 per day, he spends

$$4 \cdot 10 = 40$$

dollars on wooden figures and

$$3 \cdot 10 = 30$$

dollars on plastic figures. Therefore he buys

$$40/5 = 8$$

wooden figures and

$$30/2 = 15$$

plastic figures, for a total of

$$8 + 15 = 23.$$

The second turn recomputes the result in a minimal form and extracts the final answer.

HumanEval Code Generation — Test-driven validation of a date parser

Task. Implement the function `valid_date(date)`. The function should return `True` if the input string is a valid date and `False` otherwise. A valid date must satisfy:

- the string is not empty;
- the format is `mm-dd-yyyy`;
- the month is between 1 and 12;
- months $1, 3, 5, 7, 8, 10, 12$ have at most 31 days;
- months $4, 6, 9, 11$ have at most 30 days;
- month 2 has at most 29 days.

Ground truth. The submitted function passes all unit tests.

Trajectory summary. For this code-generation task, CoVER writes an executable implementation of `valid_date`, then validates it against prompt examples and additional edge cases. The final answer is the submitted function rather than a numeric value.

Turn 1.
<think> Parse the date string into month, day, and year components. The string must use hyphens, have exactly three fields, and match the `mm-dd-yyyy` format. Then check whether the month and day values are within the allowed ranges.

```python
def valid_date(date):
    # The format must be mm-dd-yyyy, so there must be exactly two hyphens.
    if not date or date.count("-") != 2:
        return False

    parts = date.split("-")
    if len(parts) != 3:
        return False

    month_str, day_str, year_str = parts

    # The format requires two digits for month, two for day, and four for year.
    if len(month_str) != 2 or len(day_str) != 2 or len(year_str) != 4:
        return False

    # All fields must be numeric.
    if not (month_str.isdigit() and day_str.isdigit() and year_str.isdigit()):
        return False

    month = int(month_str)
    day = int(day_str)

    # Month must be in the range 1 to 12.
    if month < 1 or month > 12:
        return False

    # Determine the maximum number of days in the given month.
    if month in [1, 3, 5, 7, 8, 10, 12]:
        max_day = 31
    elif month in [4, 6, 9, 11]:
        max_day = 30
```

```
    else:
        max_day = 29

    # Day must be in the valid range for the month.
    if day < 1 or day > max_day:
        return False

    return True
```

Execution output.

```
(Code executed successfully with no stdout output. This is normal for function
     definitions.)
```

Turn 2.
<think> Run the implementation on the examples from the prompt and on additional edge cases, including invalid separators, invalid months, invalid days, and malformed field lengths.

```
# Prompt examples.
assert valid_date("03-11-2000") == True
assert valid_date("15-01-2012") == False
assert valid_date("04-0-2040") == False
assert valid_date("06-04-2020") == True
assert valid_date("06/04/2020") == False

# Additional format checks.
assert valid_date("") == False
assert valid_date("3-04-2020") == False
assert valid_date("03-4-2020") == False
assert valid_date("03-04-20") == False
assert valid_date("03-04-20200") == False
assert valid_date("03--2020") == False

# Month validity.
assert valid_date("00-10-2020") == False
assert valid_date("13-10-2020") == False

# Day validity for different month types.
assert valid_date("01-31-2020") == True
assert valid_date("04-30-2020") == True
assert valid_date("04-31-2020") == False
assert valid_date("02-29-2020") == True
assert valid_date("02-30-2020") == False
assert valid_date("12-31-2020") == True

print("All tests passed.")
```

Execution output.

```
All tests passed.
```

Turn 3.
<think> The implementation satisfies the prompt examples and the additional edge cases. Submit the final function.
Final submitted implementation.

```
def valid_date(date):
    if not date or date.count("-") != 2:
        return False

    parts = date.split("-")
    if len(parts) != 3:
        return False

    month_str, day_str, year_str = parts

    if len(month_str) != 2 or len(day_str) != 2 or len(year_str) != 4:
        return False

    if not (month_str.isdigit() and day_str.isdigit() and year_str.isdigit()):
        return False

    month = int(month_str)
    day = int(day_str)

    if month < 1 or month > 12:
```

```
        return False

    if month in [1, 3, 5, 7, 8, 10, 12]:
        max_day = 31
    elif month in [4, 6, 9, 11]:
        max_day = 30
    else:
        max_day = 29

    if day < 1 or day > max_day:
        return False

    return True
```

Final answer.

```
DONE
```

Analysis. This example illustrates how CoVER handles code-generation tasks differently from mathematical question answering. The objective is not to compute a scalar answer, but to produce a function that satisfies a behavioral specification. CoVER first implements a parser for the required `mm-dd-yyyy` format, then uses sandbox execution to validate the function against both the prompt examples and additional edge cases. The generated solution checks the problem at three levels:

1. **Syntactic validity:** the string must contain exactly two hyphens and have field lengths 2, 2, and 4.
2. **Numeric validity:** all three fields must consist only of digits.
3. **Calendar validity:** the month must be between 1 and 12, and the day must not exceed the maximum allowed value for that month.

The second turn provides execution-grounded evidence that the implementation handles invalid separators, malformed lengths, out-of-range months, and month-specific day limits. The final submitted artifact is the function body itself, with `DONE` indicating completion of the code-generation task.

H Relation to Turn-Level RL for Tool Use

Several prior and concurrent works study fine-grained credit assignment for multi-turn tool use. CoVER's contribution is the specific combination of (i) online process rewards derived from verified transitions of a persistent sandbox, (ii) masked policy optimization for interleaved policy/environment tokens, and (iii) depth-wise group-relative credit assignment. Table 11 summarizes the component-level design differences.

Outcome-supervised and reference-supervised TIR. ReTool (Feng et al., 2025a), ToRL (Li et al., 2025), and SimpleTIR (Xue et al., 2025) train tool-integrated reasoning with primarily outcome-level rewards; SimpleTIR additionally stabilizes training by filtering invalid turns, which is complementary to process supervision. Our outcome-only GRPO baseline instantiates this paradigm inside our environment, so the $82.1\% \rightarrow 93.7\%$ gap in Table 3 is a controlled comparison against this family. ToolRL (Qian et al., 2025) and MatchTIR (Qu et al., 2026) derive fine-grained rewards from agreement with ground-truth tool names, arguments, or reference call sequences; CoVER requires no reference trajectory at RL time, so it applies wherever a task verifier and a sandbox are available. THOR (Chang et al., 2025) also uses execution feedback, within a hierarchical PPO-style scheme with binary step-execution signals, whereas CoVER combines execution, grounding, formatting, and efficiency signals with critic-free, depth-wise group-relative optimization and policy/environment masking.

Which design choices matter. Three design choices distinguish CoVER within this family. (1) *Source of intermediate supervision.* CoVER derives process rewards at every executed turn from the actual sandbox transition (execution success or failure, fabricated-feedback and checkable-claim grounding, format compliance, and interaction efficiency); these deterministic signals remain informative even when all sampled rollouts of a prompt are terminally incorrect, where purely outcome-derived credit provides no contrast. (2) *Optimization of mixed policy/environment trajectories.* PR-GRPO restricts both likelihood ratios and KL regularization to policy-generated tokens (Appendices C.2 and C.3), so environment-generated stdout, stderr, and tracebacks never enter the gradient. (3) *Credit normalization and environment state.* CoVER normalizes active rollouts separately at each interaction depth, so that early and late turns with systematically different return horizons are

Table 11: Component-level positioning of PR-GRPO relative to prior and concurrent turn-level RL methods for tool use.

Method	Supervision and credit assignment	Design of CoVER
Turn-level reward design (Wei et al., 2025)	Verifiable intermediate rewards with turn-level PPO/GRPO	Persistent sandbox state, claim–evidence grounding, and explicit optimization over mixed policy/environment trajectories
GiGPO (Feng et al., 2025b)	Critic-free inter-/intra-group step-level credit assignment	Deterministic code-transition rewards, policy-token masking, and normalization over active branches at each interaction depth
ToolRL (Qian et al., 2025)	Fine-grained decomposition of tool-use rewards	Rewards from live sandbox state and verifier evidence without a prescribed reference action trajectory
Tool-call reward model (Ma et al., 2026)	Learned process rewards for tool calls	Deterministic execution and evidence checks rather than a learned reward model
MatchTIR (Qu et al., 2026)	Reference-call matching with dual-level advantages	No reference tool-call trajectory required during RL
RLEF (Gehring et al., 2024)	Iterative execution feedback for code models	Decomposed per-turn rewards with policy-token-only optimization and depth-wise group-relative credit
GTPO (Ding et al., 2025)	Discounted turn-level returns with similarity-based reward shaping	Per-turn verified rewards, claim grounding, environment-token masking, and depth-wise credit

never compared directly, and operates in a persistent sandbox that preserves state across up to nine turns. Each choice is measurable: replacing depth-wise credit with sequence-level process-reward credit reduces validation accuracy from 93.7% to 89.2% and recovery from 74.7% to 58.3%, and removing policy/environment masking reduces accuracy to 68.4% (Table 3); among the reward components, execution feedback (-8.4 points) and claim grounding (-5.3) matter most (Figure 6b).

I Statistical Robustness, Sensitivity, and Contamination Controls

I.1 Uncertainty Estimates and Seed Robustness

We quantify evaluation-set, decoding, and training variability separately; each source independently confirms CoVER's advantage over matched GRPO.

Evaluation-set variability. Table 12 reports 95% confidence intervals from $10{,}000$ non-parametric bootstrap resamples over evaluation problems. On CoVERBench, the intervals of CoVER ($[92.7, 94.7]$) and GRPO ($[80.5, 83.7]$) are disjoint, so the 11.6-point improvement holds under problem resampling. On AIME 2022–2025 (n=120), CoVER scores 50.8% versus 47.5% for its Qwen3-8B backbone.

Table 12: Problem-level bootstrap 95% confidence intervals (10,000 resamples).

Benchmark	Model	Score	95% CI
CoVERBench (validation)	GRPO	82.1%	$[80.5, 83.7]$
CoVERBench (validation)	CoVER	93.7%	$[92.7, 94.7]$
AIME 2022–2025 (n=120)	Qwen3-8B (thinking)	47.5%	$[38.6, 56.4]$
AIME 2022–2025 (n=120)	CoVER	50.8%	$[41.9, 59.7]$

Decoding and training seeds. Table 13 summarizes the seed analyses. All tests are two-sided Welch's t-tests on the per-seed accuracies (df ≈ 3.7 in both cases), which do not assume equal variances across methods, and the 95% confidence intervals of the gain use the same Welch standard

error and degrees of freedom. *Decoding seeds:* re-decoding the CoVERBench validation set with three decoding seeds gives $93.7 \pm 0.3\%$ for CoVER and $82.1 \pm 0.4\%$ for GRPO (mean$\pm$SD; $t = 40.2$, $p < 10^{-5}$). The 11.6-point gain has a 95% confidence interval of $[10.8, 12.4]$ and is about 29 times the larger of the two seed standard deviations. *Training seeds:* to measure optimization variability, we trained each method with three independent training seeds in a matched reduced-scale configuration (300 updates with identical data and budget). CoVER reaches $84.6 \pm 0.6\%$ and GRPO $78.9 \pm 0.8\%$ ($t = 9.9$, $p < 0.001$); the 5.7-point gain has a 95% confidence interval of $[4.0, 7.4]$, well away from zero. The seed ranges of the two methods do not overlap at any evaluated checkpoint after step 100, i.e., from that point on every CoVER run outperforms every GRPO run. The between-method difference amounts to about eight pooled seed standard deviations (Cohen's $d = 8.1$); with the small-sample correction for three seeds per method, Hedges' $g = 6.4$ remains eight times the conventional threshold of 0.8 for a large effect. Both seed tests also remain significant at the 0.01 level after Bonferroni correction for the two comparisons. Under matched reduced-scale training, the advantage of CoVER over GRPO is therefore reproducible across independent runs rather than the product of a favorable seed.

Table 13: Seed robustness on CoVERBench validation (mean$\pm$SD over three seeds per method; two-sided Welch's t-tests; the 95% confidence interval of the gain uses the Welch standard error). †Training seeds use the matched reduced-scale configuration (300 updates, identical data and budget).

Seeds (3 per method)	CoVER	GRPO	Gain [95% CI]	Welch's t	p	Cohen's d
Decoding seeds	93.7 ± 0.3	82.1 ± 0.4	$+11.6$ $[10.8, 12.4]$	40.2	$< 10^{-5}$	–
Training seeds†	84.6 ± 0.6	78.9 ± 0.8	$+5.7$ $[4.0, 7.4]$	9.9	< 0.001	8.1

Retention ratio. Table 14 lists the full retention sweep of Figure 6a. Performance stays within 92.8–93.7% for 60–100% retention, and the default κ=80% is the best setting of the sweep. The selection-free setting that retains all rollouts reaches 93.1%, 11.0 points above GRPO, so the gain does not rely on trajectory selection, and every setting of the sweep stays above GRPO (82.1%). Top-rollout retention acts as a stability filter that removes the lowest-ranked, often early-collapsed or invalid, rollout in each group.

Table 14: Sensitivity to the trajectory retention ratio (CoVERBench validation accuracy).

Retained rollouts	20%	40%	60%	80%	100%
Validation accuracy	88.5%	91.6%	92.8%	**93.7%**	93.1%

I.2 Contamination Controls

The 9:1 CoVERBench split is disjoint by instance and template family and stratified by domain and source; generated variants were grouped by base-problem/template ID before splitting, SFT solution trajectories are not included in RL prompts, and reference answers and verifier metadata are never policy-visible. Across SFT, RL-training, and validation prompts we applied (i) normalized exact matching; (ii) MinHash over normalized 8-gram shingles (Jaccard ≥ 0.8) together with `bge-large` cosine similarity ≥ 0.90; and (iii) canonical logical-signature matching for generated Knights-and-Knaves instances. These checks removed 14 exact and 41 near/canonical matches between SFT and RL data, and 6 exact and 23 near/canonical matches between RL-training and validation prompts. After filtering, no detectable cross-split matches remained, including SFT–validation prompt overlap or shared canonical logical signatures across RL partitions. These controls cover exact, paraphrased, template-level, and logically equivalent overlap, so CoVERBench validation accuracy is measured on held-out instances free of detectable overlap, and the unseen external benchmarks in Table 4 additionally measure transfer.

I.3 Teacher Independence of the Warm-up Data

GPT-5 is used only for offline candidate generation of the 1,500 warm-up trajectories, not for SFT/RL optimization, reward computation, environment feedback, or evaluation; all candidates are sandbox-replayed for trace consistency and checked by deterministic answer verifiers, and the accepted

trajectories are retained as static artifacts, so reproducing the reported run does not require future GPT-5 access. To test whether comparable data can be regenerated without a proprietary teacher, we repeated synthesis with two open-weight models, GLM-5.2 and DeepSeek-V4-Pro, using the same problems, domain mix, 2–4-turn synthesis and error-injection protocol, sandbox/verifiers, and QC rubric, retaining 500 math, 500 BBH-style, and 500 logic/constraint trajectories per teacher (Table 15). Both open-weight teachers yield the full $1{,}500$-trajectory set with closely comparable QC rates, matching or exceeding GPT-5 on replay consistency and answer verification, which indicates that accepted-data quality is enforced primarily by replay, deterministic verification, and QC rather than by a particular teacher's style.

Table 15: Quality-control pass rates of warm-up trajectory synthesis with different teachers. Replay-consistent means that re-execution reproduces all stored observations (modulo whitespace and nondeterministic traceback paths); human QC additionally checks meaningful code use, claim–output consistency, strategy validity, and unsupported answers.

SFT teacher	Open weights	Format pass	Replay-consistent	Answer pass	Human-QC pass
GPT-5 (paper)	No	97.6%	94.3%	92.2%	89.1%
GLM-5.2	Yes	96.9%	95.5%	92.4%	88.7%
DeepSeek-V4-Pro	Yes	97.8%	94.1%	93.1%	88.6%

J End-to-End Inference Cost

The FLOPs comparison in Table 5 counts only model computation. To account for sandbox execution, kernel management, and serialization, we measured end-to-end latency for the same-backbone pair on the 120 AIME problems of Table 5 at batch size 1 on one A100-80GB with vLLM (bf16, temperature 0.6, top-p 0.95; $16{,}384$-token cap for the baseline). Batch-1 inference isolates per-problem latency from serving-scheduler effects; sandbox execution is timed on the same node's CPU. Both models decode at approximately 55 tokens/s. As Table 16 shows, CoVER's sandbox-related overhead totals 2.1 s, about 3.3% of its end-to-end latency, while its latency per correct answer is $4.0\times$ lower than that of the thinking baseline.

Table 16: Per-problem end-to-end inference cost on AIME (batch size 1, one A100-80GB).

Per problem	Qwen3-8B (thinking)	CoVER-8B
Generated tokens	12,899	3,349
LM decode time	234.5 s	60.9 s
Sandbox execution	—	1.2 s
Kernel init/reset and serialization	—	0.9 s
End-to-end latency	234.5 s	**63.0 s**
Latency per correct answer	493.7 s	**124.0 s**