SEER: META-REINFORCEMENT LEARNING FOR EXPLORATORY SEARCH IN CODE LLM POST-TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning with verifiable rewards (RLVR) has become a powerful paradigm for improving the reasoning performance of large language models. Yet its improvement is constrained by a simple bottleneck: RL can directly reinforce only reasoning trajectories that the policy has already discovered. Increasing rollout budgets can expose rare successes, but through increasingly expensive, undirected sampling; meanwhile, objective-level interventions reshape the policy distribution without specifying where or how the model should explore. Consequently, useful solution strategies may remain effectively inaccessible under practical rollout budgets. We address this limitation with SEER, a meta-reinforcement learning framework for Code LLM post-training targeting useful solutions rarely exposed under practical rollout budgets. SEER organizes attempts into meta-trajectories, where prior code, verifier feedback, and reflections guide search. Credit extends across attempts: an attempt is valued for immediate correctness and for enabling later success, so failed but productive attempts receive downstream credit. A meta-GRPO objective combines direct and cross-attempt signals. Yet search-conditioned discovery alone is insufficient: behaviors found under search context may remain inaccessible from the original prompt. SEER therefore transfers discoveries to the original-prompt policy via a distillation-augmented objective, enabling context-free inference. SEER improves mean Pass@1 by 3.58–5.40 percentage points over the baseline average; it reaches 33.8% coverage on previously unseen cases and gains +5.88 points in meta-search over StepCoder, supporting structured exploration and reintegration as an path for extending RLVR.

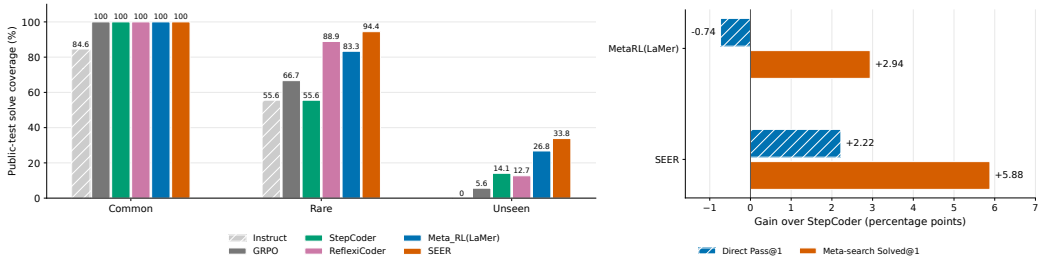


Figure 1: SEER improves discovery and retention of hard-to-reach solutions. (a) Cross-attempt credit expands rare/unseen coverage. (b) Reintegration transfers search gains to the direct policy.

1 INTRODUCTION

Reinforcement learning with verifiable rewards (RLVR) has emerged as a scalable and effective paradigm for post-training large language models on reasoning-intensive tasks such as mathematics and code generation Jiang et al. (2026c). By optimizing against automatically verifiable outcomes, methods such as Group Relative Policy Optimization (GRPO) can substantially improve the probability of generating correct solutions without relying on dense supervision or learned reward models Shao et al. (2024). Yet a higher success rate does not necessarily imply that RLVR has expanded

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.

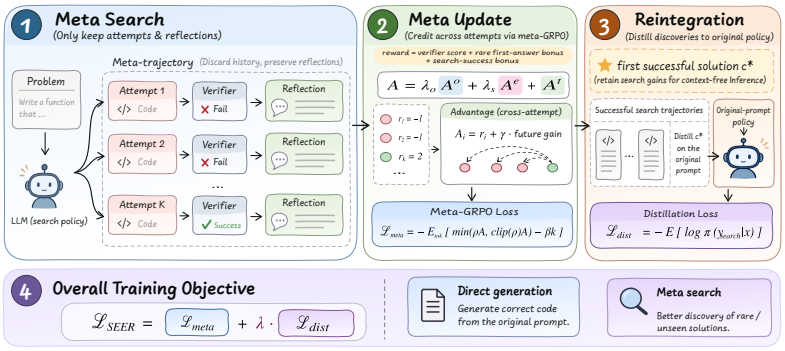


Figure 2: Overview of SEER. Feedback-conditioned meta-trajectory exploration uses cross-attempt credit, then reintegrates verified discoveries into the original-prompt policy for direct generation.

what the model is capable of solving. Recent evidence suggests that RL-trained models often outperform their base models under small sampling budgets, while sufficiently large-scale sampling from the base model can recover comparable or even higher pass@k performance; moreover, many reasoning trajectories favored after RL training already lie within the sampling distribution of the base model Chen et al. (2026b). These observations suggest that an important effect of RLVR is to redistribute probability mass toward already-discovered rewarding trajectories, improving sampling efficiency without necessarily expanding the set of reachable reasoning behaviors Chen et al. (2026b). This raises a more fundamental question for RLVR-based post-training: if reinforcement learning receives its direct learning signal only from behaviors exposed during rollout, how can it learn to discover useful solutions that are rarely, or never, sampled under limited rollout budgets?

This induces a finite-sampling exploration bottleneck in policy improvement: potentially useful paths may exist in low-probability regions of the policy distribution, yet remain effectively invisible if rarely sampled. Although parameter updates can indirectly affect unsampled trajectories, RLVR receives direct learning signal only from behaviors exposed during rollout. Prior analyses show that unexplored correct trajectories can dominate the remaining performance gap and that long-tail solutions are difficult to expose through ordinary sampling Chen et al. (2026b). The limitation is therefore not strict distributional support, but effective reachability under practical rollout budgets. From this perspective, RLVR's optimization frontier is shaped by exploration: RL can improve pass@1 by concentrating probability mass on successful trajectories, while broader coverage need not increase and can even shrink during training Chen et al. (2026b).Further policy improvement depends on exposing useful solution regions during rollout.

A natural response to this support bottleneck is to increase the rollout budget. Drawing more samples indeed raises the probability of encountering rare successful trajectories and can therefore improve coverage Hu et al. (2025). However, these gains still arise from repeatedly sampling from the same underlying policy distribution. For a low-probability trajectory y, the cost of discovering it grows inversely with its probability, making long-tail solutions increasingly expensive to uncover through brute-force sampling alone Huang et al. (2025). Rollout scaling therefore expands the volume of search without improving the structure of search: it changes how long the model searches, but not how it searches. A more promising direction is to make successive attempts dependent, allowing the model to use previous solutions, verifier feedback, and reflections to guide what it tries next. Yet conditioning later attempts on earlier ones is not sufficient to learn an exploration strategy. Under standard per-rollout RLVR, an exploratory attempt is primarily evaluated by its own immediate reward; a failed attempt that reveals useful information and ultimately enables a later success receives no explicit credit for that discovery Qu et al. (2025). What is missing, therefore, is a mechanism that propagates downstream success back to the exploratory decisions that made it possible. The key question is not merely how to search across attempts, but how to learn which search behaviors are valuable from their consequences on future attempts.

Motivated by this perspective, we propose SEER (Fig. 2), a meta-reinforcement learning framework for discovering useful code solutions that are rarely exposed under practical rollout budgets. Instead of treating rollouts as independent candidates, SEER organizes successive executable attempts into meta-trajectories, where prior code, verifier feedback, and reflections guide later search. Cru-

cially, credit propagates across attempts: an attempt is rewarded not only for immediate correctness but also for enabling later success, allowing failed yet productive attempts to receive downstream credit. A meta-GRPO objective combines direct solution quality with cross-attempt credit to learn this in-context search policy. Search-conditioned discovery alone, however, is insufficient: solutions reached with search history may remain unlikely from the original prompt. SEER therefore couples discovery with reintegration. The meta-policy searches beyond the effective reach of low-budget direct sampling, while a distillation-augmented objective transfers verified discoveries back to the original-prompt policy. This turns temporary search successes into solutions that can be generated directly at inference time, without search history. Empirically, SEER improves both discovery and retention (Fig. 1), covering 94.4% of rare and 33.8% of Unseen@4 cases, defined by four-sample success frequency under the Instruct reference policy (Appendix L). Relative to the MetaRL baseline, SEER gains +5.88 pp in meta-search and +2.22 pp in direct Pass@1, while the baseline gains +2.94 pp in search but drops 0.74 pp in direct generation.

Together, SEER turns RLVR exploration into a meta-learned search process whose discoveries are transferred back to the original-prompt policy. Our contributions are threefold:

- We identify an exploration-support bottleneck in RLVR: solutions may remain unreachable under limited rollouts, and formulate cross-attempt exploration as a meta-RL problem.
- We introduce SEER, a meta-RL framework that organizes successive attempts into meta-trajectories and uses a meta-level advantage objective to assign credit across attempts.
- We introduce search-to-policy transfer, which distills behaviors discovered under search into the original-prompt policy for direct inference without search history.

2 Related Work

Exploration in RLVR. RLVR has become a strong recipe for mathematical and code reasoning, from GRPO in DeepSeekMath Shao et al. (2024) to pure-RL reasoning in DeepSeek-R1 Guo et al. (2025). Recent work has questioned whether these gains reflect new behaviors or mainly reweight behaviors already present in the base policy. Yue et al. Chen et al. (2026b) report that large-budget base sampling recovers many trajectories favored after RL, whereas Wen et al. Wen et al. (2026) and ProRL Liu et al. (2026) provide evidence that sustained RL can extend reasoning boundaries under alternative evaluations. This debate has motivated explicit exploration methods. Tuyls et al. Tuyls et al. (2026) use representation bonuses, Chen et al. Chen et al. (2026a) suppress sampled responses, and Jiang et al. Jiang et al. (2025) use selective expert guidance; related work preserves low-probability reasoning tokens Huang et al. (2026) or reshapes advantages around uncertainty Xie et al. (2026). These methods broaden trajectories exposed to RL, but primarily improve *which trajectories are sampled*. SEER instead targets the finite-budget bottleneck by learning how successive attempts expose useful solutions rarely reached independently.

Iterative search and meta-reinforcement learning. A parallel line uses additional computation as structured search rather than independent resampling. Tree of Thoughts Yao et al. (2023), RAP Hao et al. (2023), and LATS Zhou et al. (2023) branch, evaluate, and revisit candidate reasoning paths, while Reflexion Shinn et al. (2023) conditions later attempts on feedback from earlier failures. Satori Shen et al. (2025) trains self-reflection and strategy exploration within extended traces, Yuan and Xie Yuan & Xie (2025) formulate iterative verify-and-improve reasoning as RL, and Qu et al. Qu et al. (2025) cast test-time reasoning as meta-RL using information gain. e3 Setlur et al. (2026) trains in-context exploration through generation, verification, and refinement; LAMER Jiang et al. (2026b) studies cross-episode learning, while RAGEN Wang et al. (2025) analyzes multi-turn agent RL. SEER shares the view that earlier outcomes should shape later behavior, but uses cross-attempt search to discover verified code solutions beyond low-budget direct sampling, with downstream credit assigned to attempts enabling those discoveries.

Transferring search discoveries into the policy. Search matters only if exposed behaviors improve the deployed policy. STaR Zelikman et al. (2022) bootstraps from successful self-generated rationales, while ReST-MCTS* Zhang et al. (2024) and rStar-Math Guan et al. (2025) use verifier-guided tree search to produce stronger training trajectories. Snell et al. Snell et al. (2024) show that verifier-guided search can use test-time compute more effectively than naive sampling. Kim et al. Kim et al. (2025) connect search, policy-gradient RL, and distillation, while Setlur et al. Setlur et al.

(2025) show that cloning successful search traces alone can become suboptimal as compute scales. SEER focuses on the gap between search-conditioned discovery and direct generation: solutions found after prior attempts, verifier feedback, and reflection may remain unlikely under the original prompt. It therefore reintegrates these discoveries into the original-prompt policy, so meta-search discoveries can be generated directly without search history.

3 SEER: Structured Exploration and Experience Reintegration

SEER links code-generation attempts through feedback and reflection, assigning credit to immediate quality and downstream discoveries. It then transfers verified discoveries to the original-prompt policy, connecting exploration with direct-generation ability.

3.1 Code exploration as a meta-RL problem

For a programming problem $x \sim \mathcal{D}$ and code model π_θ, successive attempts interact with a meta-MDP:$\mathcal{M}_x = (\mathcal{S}_x, \mathcal{A}, \mathcal{P}_x, \mathcal{R}_x, \rho_x, \gamma), s_u = (x, h_u, t_u, \phi_u)$. The state contains history h_u, attempt index t_u, and phase $\phi_u \in \{\text{code}, \text{reflect}\}$. The initial distribution ρ_x starts in the code phase with empty history. Actions are complete programs or reflections; $\mathcal{P}_x$ executes programs or records reflections, and $\mathcal{R}_x$ supplies verifier rewards. Each code generation is one episode, with execution determining its outcome. A trial (meta-trajectory) connects episodes through reflections and ends at verified success or after H code attempts.

The policy uses a textual state encoding. With retained reflection memory m_t and $m_0 = \varnothing$, the phases are

$$c_t \sim \pi_\theta(\cdot \mid x, m_t), \qquad o_t = \mathcal{V}_x(c_t), \tag{1}$$

$$z_t \sim \pi_\theta(\cdot \mid x, m_t, c_t, o_t), \qquad m_{t+1} = \mathcal{U}(m_t, z_t). \tag{2}$$

After failure, z_t summarizes the attempted approach and proposes guidance from verifier feedback o_t; $\mathcal{U}$ retains it for the next program. Each episode generates a complete solution, permitting algorithm changes or local corrections. Execution resets program state but preserves memory. Success or budget exhaustion omits further reflection.

We collect M independent trials per problem, each with its own memory. Interleaving code and reflection actions gives

$$\tau_i = (s_{i,0}, a_{i,0}, r_{i,0}, \ldots, s_{i,K_i}), \qquad K_i \le 2H - 1. \tag{3}$$

Within a trial, fixed parameters adapt the action distribution through memory. Across tasks, outer updates learn this adaptation through trajectory-wide credit: failed attempts can receive credit for successful continuations.

3.2 Rewards for useful discoveries

SEER separates execution scores from incentives for fully correct solutions. In our development experiments, an additional full-success incentive improved convergence speed and final performance. We further distinguish rare direct solutions from search discoveries.

Let $p_x(c)$ be the fraction of tests passed, $q_x(c)$ indicate full verification, and $\eta_x(c) > 0$ encode failure severity. The verifier reward is

$$v_x(c) = p_x(c) - \eta_x(c)\mathbf{1}_{p_x(c)=0} + \kappa q_x(c). \tag{4}$$

The first two terms score execution; the fixed bonus κ separates full correctness from partial progress. Group-dependent bonuses additionally distinguish how success is found.

For M trials, let $n_0 = \sum_{i=1}^{M} q_x(c_{i,0})$ count successful initial episodes. Define

$$b_0(n) = \alpha_{0,h}\mathbf{1}_{0<n\le\ell_0} + \alpha_{0,l}\mathbf{1}_{\ell_0<n\le u_0}, \tag{5}$$

$$b_s(n) = \alpha_{s,0}\mathbf{1}_{n=0} + \alpha_{s,h}\mathbf{1}_{0<n\le\ell_s} + \alpha_{s,l}\mathbf{1}_{\ell_s<n\le u_s}. \tag{6}$$

Thresholds $\ell_0 < u_0$ and $\ell_s < u_s$ define scarcity levels, with $\alpha_{0,h} > \alpha_{0,l} > 0$ and $\alpha_{s,0} > \alpha_{s,h} > \alpha_{s,l} > 0$. Thus b_0 strengthens rare correct first answers, while b_s peaks when all first answers fail

and decreases as direct success becomes common. Both vanish above their upper thresholds; values are given in the experiments.

The shaped reward for code episode t is

$$r^c_{i,t} = v_x(c_{i,t}) + q_x(c_{i,t}) \left[\mathbf{1}_{t=0} b_0(n_0) + \mathbf{1}_{t>0} b_s(n_0) - d(t)\right], \qquad d(t) = \frac{\kappa t}{H+1}. \tag{7}$$

The count n_0 is fixed before retries. Early stopping makes b_s reward the first successful continuation. Correct first answers receive b_0 without search cost; among retries with equal bonuses, $d(t)$ favors earlier success.

Reflections receive zero immediate reward. For interleaved code and reflection actions, define

$$S_i = \sum_{u=0}^{K_i-1} r_{i,u}, \qquad G_{i,u} = \sum_{v=u}^{K_i-1} \gamma^{v-u} r_{i,v}. \tag{8}$$

S_i scores the complete trial; $G_{i,u}$ assigns remaining return, discounting across code and reflection actions. Both can receive credit for later discoveries.

3.3 META-GRPO

SEER combines first-answer quality, future return, and complete-search outcome. Statistics are computed per problem, using $\sigma_\epsilon(B) = \max(\mathrm{std}(B), \epsilon)$ with a positive numerical floor.

First-attempt advantage. First episodes share the original prompt and empty memory. Let μ_0 and σ_0 be the mean and floored standard deviation of their shaped rewards $\{r^c_{i,0}\}_{i=1}^M$, define A^0:

$$A^0_{i,u} = \mathbf{1}_{u=0} \frac{r^c_{i,0} - \mu_0}{\sigma_0}. \tag{9}$$

This term reinforces above-average first answers, preserving direct solution quality alongside search.

Episode-level advantage. We normalize the returns in Eq. equation 8 across all code and reflection actions sampled for the same problem, define A^e:

$$A^e_{i,u} = \frac{G_{i,u} - \mu_G}{\sigma_G}, \tag{10}$$

where μ_G and σ_G are the mean and ϵ-floored standard deviation of these returns. Actions within a trial share its outcome but differ in their remaining return. For a failed program, a zero-reward reflection, and a successful retry, the returns are $r_f + \gamma^2 r_s$, γr_s, and r_s, respectively. An early failure can therefore receive positive credit for a sufficiently valuable continuation, while reflection learns through downstream success.

Trajectory-level advantage. Let μ_S and $\sigma_S = \sigma_\epsilon(\{S_i\}_{i=1}^M)$ denote the mean and stabilized standard deviation of trial scores. With $L_i = \sum_u |a_{i,u}|$ and $\bar{L} = M^{-1} \sum_i L_i$, define A^t:

$$A^t_{i,u} = \frac{\bar{L}}{L_i} \frac{S_i - \mu_S}{\sigma_S}. \tag{11}$$

L_i counts generated code and reflection tokens, excluding prompts and padding. The trajectory score is broadcast across actions. Because success ends search early, lengths vary; without $\bar{L}/L_i$, longer trajectories repeat the score over more tokens. Normalization fixes the total broadcast coefficient at

$$\sum_{u=0}^{K_i-1} |a_{i,u}| A^t_{i,u} = \bar{L} \frac{S_i - \mu_S}{\sigma_S}. \tag{12}$$

Trajectory credit evaluates the whole search, while episode credit preserves temporal differences in future return.

We then combine the advantages and update the policy. The three components form

$$\widetilde{A}_{i,u} = \lambda_0 A^0_{i,u} + \lambda_s A^e_{i,u} + A^t_{i,u}. \tag{13}$$

Weights λ_0 and λ_s control first-answer and episode-return credit. For fully correct first answers, $A_{i,0} = \max(\widetilde{A}_{i,0}, 0)$ preserves direct successes when search bonuses alter the comparison; otherwise $A_{i,u} = \widetilde{A}_{i,u}$. A batch passing no tests receives zero policy-gradient advantage.

Let $h_{i,u,j}$ contain the prompt and tokens preceding token j. Relative to behavior policy π_{old}, the likelihood ratio is

$$\rho_{i,u,j} = \frac{\pi_\theta(a_{i,u,j} \mid h_{i,u,j})}{\pi_{\text{old}}(a_{i,u,j} \mid h_{i,u,j})}. \tag{14}$$

Each generated token receives its action's advantage. We minimize

$$\mathcal{L}_{\text{meta}} = -\frac{1}{N} \sum_{i,u,j} \left[\min(\rho_{i,u,j} A_{i,u}, \text{clip}(\rho_{i,u,j}, 1-\varepsilon, 1+\varepsilon) A_{i,u}) - \beta k_{i,u,j} \right]. \tag{15}$$

The sum covers N generated training tokens. For $q = \pi_{\text{ref}}(a \mid h)/\pi_\theta(a \mid h)$, the reference penalty is $k = q - \log q - 1$. The clipping width ε controls policy change and β controls regularization. Rewards and advantages remain fixed during updates; code and reflection tokens jointly train generation and adaptation.

3.4 Transferring search discoveries to direct generation

A solution likely under $\pi_\theta(\cdot \mid x, m_t)$ may remain unlikely under $\pi_\theta(\cdot \mid x)$: the search–policy gap. SEER transfers verified discoveries to the original-prompt policy to retain search improvements for direct generation.

We select each initially failed trial's first success $c^* = c_t$: $t > 0$, $q_x(c_t) = 1$, and $q_x(c_j) = 0$ for all $j < t$. Pairing it with x adds a target to replay buffer $\mathcal{B}$. Search history remains in the RL trajectory; distillation conditions only on the problem and starter code. Teacher forcing gives

$$\mathcal{L}_{\text{dist}}(\theta) = -\frac{1}{|\mathcal{B}|} \sum_{(x,c^*) \in \mathcal{B}} \frac{1}{|c^*|} \sum_{j=1}^{|c^*|} \log \pi_\theta(c^*_j \mid x, c^*_{<j}). \tag{16}$$

Only completion tokens receive supervision, normalized by completion length. Discoveries have equal positive weight and a bounded replay lifetime, allowing repeated use while new searches replenish the buffer. Targets remain fixed; an empty buffer contributes zero loss.

Both objectives update the same parameters:

$$\mathcal{L}_{\text{SEER}}(\theta) = \mathcal{L}_{\text{meta}}(\theta) + \lambda_d \mathcal{L}_{\text{dist}}(\theta). \tag{17}$$

Each outer update accumulates gradients from current meta-trajectories and retained discoveries before a shared optimizer step. RL trains code and reflection in search contexts through cross-episode credit; distillation increases solution likelihood under the original prompt. Search supplies successful targets, and their retention improves the starting distribution for future searches. The coefficient λ_d controls transfer strength. The trained model supports direct generation from x and feedback-conditioned search from (x, m_t).

4 Experiments

We evaluate whether SEER improves the original-prompt policy, finds correct programs under finite search budgets, and retains the discoveries made during search. The main experiments separate these questions through direct generation, cost-indexed search, component ablations, and controlled changes to search history. Additional model families, training dynamics, and robustness analyses appear in Appendix C–M.

Our main comparison uses Qwen2.5-Coder-Instruct Hui et al. (2024) on APPS+ Dou et al. (2024), MBPP+ Liu et al. (2023), TACO Li et al. (2023), and the code-generation task of LiveCodeBench (LCB) Jain et al. (2025), while search diagnostics use APPS*, is a held-out 102-problem subset of APPS+ used for computationally feasible 128-sample evaluation. HumanEval Chen et al. (2021) is reserved for additional analyses in the appendix. The main table compares SEER with adapted implementations of the core mechanisms of GRPO Shao et al. (2024), LaMer Jiang et al. (2026b),

Table 1: Main Results: Direct-Generation Performance Across Model Scales and Code Benchmarks

Scale	Benchmark	GRPO			LaMer			StepCoder			SEER			Δ_1
		Pass@1	Pass@2	Pass@3	Pass@1	Pass@2	Pass@3	Pass@1	Pass@2	Pass@3	Pass@1	Pass@2	Pass@3	(pp)
3B	APPS+	17.00	23.00	26.33	21.67	25.67	28.00	23.67	27.33	29.67	26.33	30.00	32.00	+5.55
	MBPP+	46.44	53.33	56.67	50.00	58.89	63.33	52.22	58.89	62.22	54.44	61.11	64.44	+4.89
	TACO	6.89	11.31	14.03	8.67	13.18	15.90	9.61	14.03	16.75	10.12	14.54	17.26	+1.73
	LCB	6.49	11.04	14.29	9.09	13.64	16.23	9.09	13.64	16.88	10.39	14.94	17.53	+2.17
	Avg.	19.21	24.67	27.83	22.36	27.85	30.86	23.65	28.47	31.38	25.32	30.15	32.81	+3.58
7B	APPS+	25.67	34.00	36.67	31.00	36.00	39.00	32.33	37.33	40.00	34.00	38.67	41.33	+4.33
	MBPP+	48.67	54.44	57.78	53.33	62.22	65.56	57.78	64.44	68.89	61.11	67.78	71.11	+7.85
	TACO	10.71	17.18	20.75	13.78	20.15	23.64	13.52	19.90	23.38	15.48	21.77	25.17	+2.81
	LCB	11.69	17.53	21.43	13.64	20.13	23.38	16.88	22.73	26.62	18.83	24.68	27.92	+4.76
	Avg.	24.19	30.79	34.16	27.94	34.62	37.89	30.13	36.10	39.72	32.36	38.22	41.38	+4.94
14B	APPS+	30.67	39.67	42.33	37.67	42.33	44.67	39.33	43.67	46.33	42.67	47.00	49.33	+6.78
	MBPP+	53.11	60.00	63.33	58.89	65.56	68.89	62.22	68.89	72.22	65.56	72.22	74.44	+7.49
	TACO	12.50	18.88	22.36	17.69	23.72	27.04	16.58	22.70	26.11	18.37	24.40	27.64	+2.78
	LCB	15.58	22.08	25.32	19.48	25.32	28.57	21.43	27.27	30.52	23.38	28.57	31.82	+4.55
	Avg.	27.97	35.16	38.33	33.43	39.23	42.29	34.89	40.63	43.80	37.50	43.05	45.81	+5.40

and StepCoder Dou et al. (2024) under a unified training and evaluation framework. Additional analyses include SEER and a ReflexiCoder-inspired baseline Jiang et al. (2026a). Instruct is the unmodified starting policy. Direct evaluation conditions only on the original problem with zero search budget. A SEER search unit is one meta-trajectory with up to three code episodes and two reflections; thus, Meta-search Solved@k counts k meta-searches, while direct baselines use k independent code samples. Token-budget comparisons also include reflection tokens. Appendix A provides implementation metrics, and training recipes.

Direct-generation gains across scales and tasks. Table 1 compares SEER, GRPO, LaMer, and StepCoder on four code benchmarks using Qwen2.5-Coder at 3B, 7B, and 14B. Values are percentages, and Avg. is the unweighted mean across benchmarks. Δ_1 is SEER's Pass@1 gain over the mean of the three baselines in each setting; its average is taken over these setting-wise gains. Across the 12 model–benchmark settings, SEER consistently outperforms the baseline average, with Pass@1 gains increasing from 3.58 points at 3B to 5.40 points at 14B. The improvement also extends to TACO and LCB, where the margins are generally larger at 7B and 14B than at 3B, though the magnitude remains task dependent. This pattern is consistent with SEER propagating credit from informative failures to later successes, whereas GRPO's rollout-local credit assignment cannot exploit such cross-episode signals as effectively under a fixed training budget. SEER also retains gains as the sampling budget increases, with mean Pass@3 improvements of 2.79–4.34 points across model scales. The benefit therefore persists beyond single-answer generation, indicating that SEER improves the original-prompt policy in a way that remains visible across multiple samples.

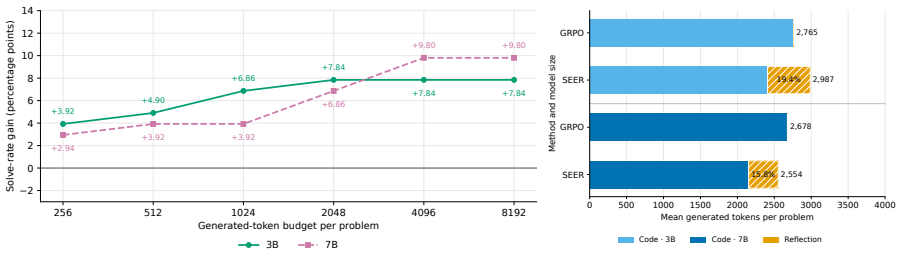


Figure 3: Search efficiency: (a) SEER solve-rate gains across token budgets; (b) code and reflection token costs.

Does SEER improve finite-budget exploration? We compare independent sampling with normalized SEER's feedback-conditioned search on APPS*. For the GRPO baseline, search cost is measured as the total generated-token length of the three independent code samples used for Pass@3,

whereas SEER's cost includes both code and reflection tokens. Figure 3b reports the mean cost per problem. At 3B, SEER uses 2,987 tokens compared with 2,765 for GRPO; at 7B, SEER uses 2,554 tokens compared with 2,678 for GRPO. Reflection contributes 19.4% and 15.8% of SEER's total token cost at 3B and 7B, respectively. Figure 3a compares solve rates across token budgets for both model scales. At a budget of 4,096 tokens, SEER improves solve rate by 7.84 points at 3B and 9.80 points at 7B. Full token- and verifier-budget curves are provided in Appendix F.

Figure 4 further examines how direct-generation performance scales with the sampling budget on APPS*. All three policies independently sample answers from the original prompt. In panel (a), SEER achieves the highest Pass@k across all evaluated budgets, with the largest gains appearing at smaller k. Panel (b) shows that SEER's advantage over StepCoder is 2.45 points at Pass@1, increases to 3.08 points at Pass@16, and then narrows to 0.33 points at Pass@128. The gaps of SEER and StepCoder over GRPO also decrease as k grows. SEER's margin over GRPO falls from 5.52 points at Pass@1 to 2.63 at Pass@64 and 0.67 at Pass@128, while StepCoder's corresponding margin declines from 3.07 to 1.47 and 0.33 points. Thus, larger direct-sampling budgets allow GRPO to recover much of the gap to the stronger policies. At Pass@128, SEER, StepCoder, and GRPO reach 54.00%, 53.67%, and 53.33%, respectively. Overall, SEER remains strongest across the evaluated budgets, with its clearest advantage under tighter sampling constraints.

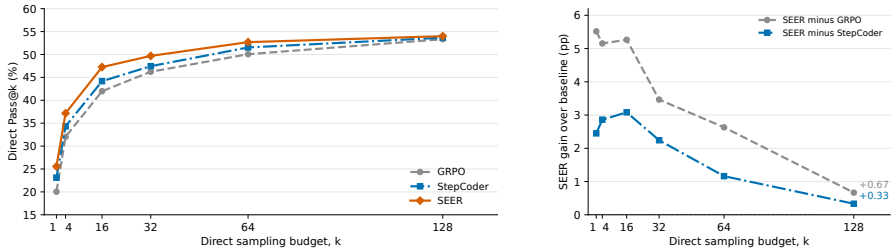


Figure 4: Direct-sampling reachability across budgets. (a) Pass@k. (b) SEER gains over baselines.

Figure 5 examines which problems are solved on APPS under a budget of 128 units per problem. Relative to Instruct, SEER solves 15 additional problems and misses one, yielding a net gain of 14. Relative to GRPO, it solves 12 additional problems and misses one, yielding a net gain of 11. In both comparisons, SEER solves 87 of the 102 problems; the net gains over Instruct and GRPO are therefore 14 and 11 problems, respectively. These set differences show that SEER's coverage gain comes mainly from problems missed by the comparison policies.

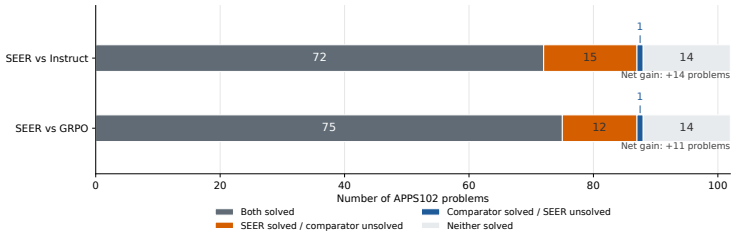


Figure 5: Solved-problem overlap between SEER and direct-sampling baselines at 128 search units

What do cross-episode credit and reintegration contribute? We remove distillation, replace downstream credit with immediate credit, replace the full method with successful-sample replay, or disable reflection. Figure 6 reports each change relative to full SEER, separately for original-prompt generation and search. This separation tests the two responsibilities of the method: producing useful discoveries and making their solutions accessible without the discovery history.

The ablation results reveal distinct roles across components. Distillation mainly preserves search-derived knowledge for direct generation, as its removal hurts direct performance more than meta-search. In contrast, immediate credit, success replay, and reflection primarily affect search, suggesting that they shape feedback-conditioned search trajectories. Immediate credit is especially important, highlighting the value of credit assignment through successful continuations, while success

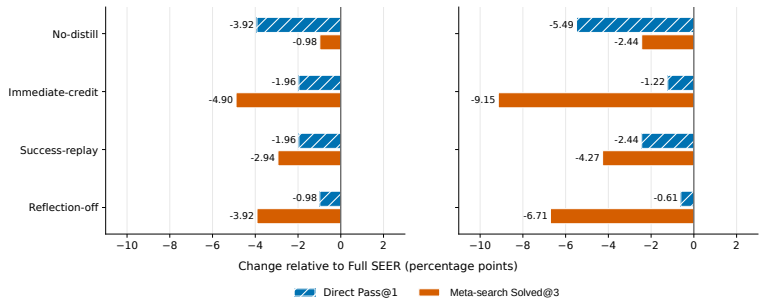


Figure 6: Component ablations: (a) APPS* and (b) HumanEval.

replay reinforces effective trajectories and reflection refines them. Overall, search-side mechanisms improve exploration and learning, whereas distillation transfers these gains into direct generation.

How does prior experience affect recovery? To examine how prior experience shapes recovery, we branch three additional code episodes from a shared failed first answer under four conditions: full history, reset history, no verifier feedback, and shuffled feedback. The HumanEval29 evaluation uses 12 matched problem–seed instances across eight problems, with recovery measured on hidden tests. Figure 7 shows recovery by the first successful episode. With full history, SEER recovers 75.0% of initial failures versus 66.7% for StepCoder, with four additional solutions found after the first retry. Resetting history or removing verifier feedback lowers recovery to 66.7%. Shuffled feedback also reaches 75.0%, but all successes occur at episode 2. These interventions therefore affect both recovery rate and recovery timing.

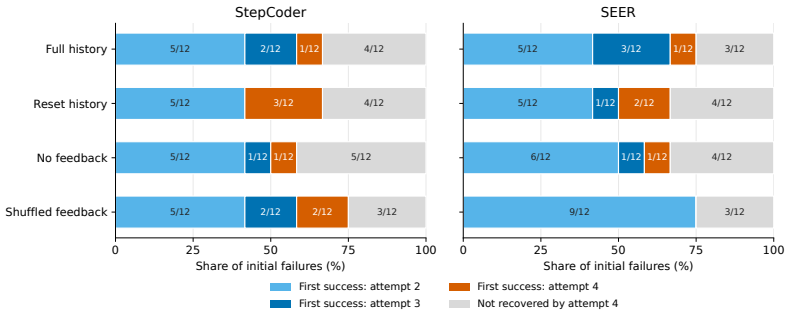


Figure 7: Recovery after initial failure

We stratify APPS* by success frequency under a four-sample reference budget into Common, Rare, and Unseen@4. At Meta-search Solved@4, SEER reaches 100% on Common, 94.4% on Rare, and 33.8% on Unseen@4, exceeding Meta-RL (LaMer) by 7.0 points on problems never solved by the reference sampler. Full results and Solved@1–4 trends are reported in Appendix L, Figure 19.

Overall, SEER benefits most from exploiting later successful continuations and retaining their solutions for direct generation. Additional analyses examine whether these gains persist across backbones, datasets, normalization choices, and training and search budgets.

5 Conclusion

We introduced SEER, a meta-RL framework for the finite-budget exploration bottleneck in RLVR. SEER links code attempts through feedback, credits failed attempts that enable later success, and distills verified discoveries back into the original-prompt policy. Across benchmarks and model scales, SEER improves both direct generation and finite-budget search, with the largest gains on rare and previously unreached problems. Ablations show that cross-attempt credit drives discovery, while reintegration preserves discoveries for direct use. These results suggest that structured exploration and consolidation can extend RLVR more effectively than scaling independent sampling alone.

AI USE STATEMENT

In this work, we used generative AI tools solely for language editing and polishing, including improving grammar, clarity, and readability of the manuscript. We did not use generative AI tools for generating research ideas, conducting experiments, analyzing results, producing figures, or writing code, and other disclosure categories are not applicable to this work. We have reviewed all AI-assisted edits and verified that they did not alter the technical meaning, claims, or conclusions of the paper. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Liang Chen, Xueting Han, Qizhou Wang, Bo Han, Jing Bai, Hinrich Schuetze, and Kam-Fai Wong. Eepo: Exploration-enhanced policy optimization via sample-then-forget. In *International Conference on Learning Representations*, volume 2026, pp. 150418–150434, 2026a.

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.

Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *Advances in Neural Information Processing Systems*, 38:57654–57689, 2026b.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, et al. Stepcoder: improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4571–4585, 2024.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. volume 645, pp. 633–638. Nature Publishing Group UK London, 2025.

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.

Jian Hu, Mingjie Liu, Ximing Lu, Fang Wu, Zaid Harchaoui, Shizhe Diao, Yejin Choi, Pavlo Molchanov, Jun Yang, Jan Kautz, et al. Brorl: Scaling reinforcement learning via broadened exploration. *arXiv preprint arXiv:2510.01180*, 2025.

Audrey Huang, Adam Block, Qinghua Liu, Nan Jiang, Akshay Krishnamurthy, and Dylan J Foster. Is best-of-n the best of them? coverage, scaling, and optimality in inference-time alignment. *arXiv preprint arXiv:2503.21878*, 2025.

Guanhua Huang, Tingqiang Xu, Mingze Wang, Qi Yi, Xue Gong, Siheng Li, Ruibin Xiong, Kejiao Li, Yuhao Jiang, and Bo Zhou. Low-probability tokens sustain exploration in reinforcement learning with verifiable reward. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 24158–24188, 2026.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Juyong Jiang, Jiasi Shen, Sunghun Kim, Kang Min Yoo, Jeonghoon Kim, and Sungju Kim. Reflexicoder: Teaching large language models to self-reflect on generated code and self-correct it via reinforcement learning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 37543–37562, 2026a.

Yulun Jiang, Liangze Jiang, Damien Teney, Michael Moor, and Maria Brbic. Meta-rl induces exploration in language agents. In *International Conference on Learning Representations*, volume 2026, pp. 3279–3303, 2026b.

Yuxin Jiang, Yufei Wang, Qiyuan Zhang, Xingshan Zeng, Liangyou Li, Jierun Chen, Chaofan Tao, Haoli Bai, and Lifeng Shang. From verifiable dot to reward chain: Harnessing verifiable reference-based rewards for reinforcement learning of open-ended generation. *arXiv preprint arXiv:2601.18533*, 2026c.

Zishang Jiang, Jinyi Han, Tingyun Li, Xinyi Wang, Sihang Jiang, Jiaqing Liang, Zhaoqian Dai, Shuguang Ma, Fei Yu, and Yanghua Xiao. Selective expert guidance for effective and diverse exploration in reinforcement learning of llms. *arXiv preprint arXiv:2510.04140*, 2025.

Juno Kim, Denny Wu, Jason Lee, and Taiji Suzuki. Metastable dynamics of chain-of-thought reasoning: Provable benefits of search, rl and distillation. *arXiv preprint arXiv:2502.01694*, 2025.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.

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, 2023.

Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. Prorl: Prolonged reinforcement learning expands reasoning boundaries in large language models. *Advances in Neural Information Processing Systems*, 38:17998–18031, 2026.

Yuxiao Qu, Matthew YR Yang, Amrith Setlur, Lewis Tunstall, Edward Emanuel Beeching, Ruslan Salakhutdinov, and Aviral Kumar. Optimizing test-time compute via meta reinforcement fine-tuning. *arXiv preprint arXiv:2503.07572*, 2025.

Amrith Setlur, Nived Rajaraman, Sergey Levine, and Aviral Kumar. Scaling test-time compute without verification or rl is suboptimal. *arXiv preprint arXiv:2502.12118*, 2025.

Amrith Setlur, Matthew Yang, Charlie Snell, Jeremiah Greer, Ian Wu, Virginia Smith, Max Simchowitz, and Aviral Kumar. e3: Learning to explore enables extrapolation of test-time compute for llms. In *International Conference on Learning Representations*, volume 2026, pp. 127323–127361, 2026.

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.

Maohao Shen, Guangtao Zeng, Zhenting Qi, Zhang-Wei Hong, Zhenfang Chen, Wei Lu, Gregory Wornell, Subhro Das, David Cox, and Chuang Gan. Satori: Reinforcement learning with chain-of-action-thought enhances llm reasoning via autoregressive search. *arXiv preprint arXiv:2502.02508*, 2025.

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.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Jens Tuyls, Dylan Foster, Akshay Krishnamurthy, and Jordan Ash. Representation-based exploration for language models: From test-time to post-training. In *International Conference on Learning Representations*, volume 2026, pp. 120621–120645, 2026.

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

Xumeng Wen, Zihan Liu, Shun Zheng, Shengyu Ye, Zhirong Wu, Yang Wang, Zhijian Xu, Xiao Liang, Junjie Li, Ziming Miao, et al. Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base llms. In *International Conference on Learning Representations*, volume 2026, pp. 49450–49483, 2026.

Can Xie, Ruotong Pan, Xiangyu Wu, Zhang Yunfei, Jiayi Fu, Tingting Gao, and Guorui Zhou. Unlocking exploration in rlvr: Uncertainty-aware advantage shaping for deeper reasoning. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 19057–19076, 2026.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Yurun Yuan and Tengyang Xie. Reinforce llm reasoning through multi-agent reflection. *arXiv preprint arXiv:2506.08379*, 2025.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. Rest-mcts*: Llm self-training via process reward guided tree search. *Advances in Neural Information Processing Systems*, 37:64735–64772, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

A EVALUATION PROTOCOLS AND REFERENCE TRAINING CONFIGURATION

Direct generation and search measure different units. Direct generation uses a search budget of zero: it conditions only on the problem, and one code generation is one episode. In meta-search, reflection conditions on the previous program, execution feedback, and retained memory. The resulting summary and guidance update the memory used by the next code episode. A complete independent search is one outer unit. Unless a depth experiment specifies otherwise, that unit permits three code episodes and two reflections. Meta-search Solved@k therefore uses k independent searches, not k total code generations. Reflection is a generated action connecting episodes and is charged to the token budget.

For direct sampling, let n_x be the number of saved completions and c_x the number passing all evaluation tests for question x. We compute

$$\widehat{\text{Pass@}k} = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \left[1 - \frac{\binom{n_x - c_x}{k}}{\binom{n_x}{k}} \right]. \tag{18}$$

The ratio is zero when $n_x - c_x < k$. For a greedy single-answer evaluation, direct accuracy is instead the fraction of questions whose one generated answer passes all tests. We distinguish that score from sampled Pass@1. Independent search units can be aggregated with the same formula after fixing the success rule for a unit.

Problem populations and test access. For APPS+, we sample a fixed validation set of 300 problems in the original difficulty proportions: 117 introductory, 160 interview, and 23 competition problems. APPS* provides public tests for search feedback. The HumanEval adaptation split contains 131 training and 33 validation problems, with 29 validation problems used for feedback interventions. MBPP+, TACO, and LCB use their respective prepared evaluation sets, with LCB restricted to code generation.

Sampling and adaptation. The 128-sample direct protocol uses temperature 0.8, top-p 0.91, at most 512 generated tokens, and a 1,536-token problem input. It retains duplicate and unsuccessful completions. The problem-input bound excludes additional reflection/search history, whose allowance is configured separately.

Table 2 gives the reference training configuration.

Table 2: SEER training configuration

Component	Reference value
LoRA rank / scaling	$r = 16$, $\alpha = 32$; dropout 0
Adapted projections	Query, key, value, and output attention projections
Search group	Five trials per problem, at most three code episodes per trial
Full-success bonus	$\kappa = 2$
Zero-pass penalty	$\eta_x(c) = 0.8 + 0.4 f_x(c)$
First-answer scarcity	$(\ell_0, u_0) = (1, 2)$; $(\alpha_{0,h}, \alpha_{0,l}) = (2, 1)$
Search-discovery scarcity	$(\ell_s, u_s) = (2, 4)$; $(\alpha_{s,0}, \alpha_{s,h}, \alpha_{s,l}) = (2, 1, 0.5)$
Early-success incentive	$d(t) = 2t/(H + 1)$
Return discount	$\gamma = 0.46$ across code and reflection actions
Advantage weights	$\lambda_0 = 0.5$, $\lambda_s = 1$
Distillation	Coefficient 0.1; first success after failure; four replay uses
Length handling	Trajectory-credit normalization
Learning rate	Initial 2×10^{-6}, peak 5×10^{-6}, final 2×10^{-6}

Here $f_x(c)$ is the fraction of tests producing compilation or runtime errors when no test passes. The zero-pass reward ranges from -0.8 for wrong answers to -1.2 when every test produces such an error. Scarcity thresholds count successful first answers among the five trials; the corresponding bonuses vanish above their upper thresholds.

B DISCOVERY AND REINTEGRATION DURING TRAINING

We track full SEER and SEER without distillation at 50-update intervals to examine how search discoveries become available to the original-prompt policy. Figure 8 reports direct Pass@1 on APPS+, Meta-search Solved@4 on APPS*, and the cumulative number of distinct training problems with a successful discovery. A search unit permits up to three code episodes with feedback and reflection.

Both methods improve their search performance throughout training, reaching 19.61% and 18.70% Solved@4 at update 300. Their discovery coverage is similarly close: full SEER and the no-distillation variant discover successful solutions to 160 and 163 distinct problems, respectively. Search therefore continues to supply useful solutions in both runs. The direct-generation curves separate more clearly in the second half of training. At update 150, both are near 23.76%; by update 300, full SEER reaches 28.00%, compared with 26.00% without distillation. Full SEER gains 6.33 points over its initial direct performance, while the no-distillation variant gains 3.33 points. The larger direct improvement alongside similar search and discovery coverage supports reintegration as the mechanism that makes discovered solutions reusable without search history.

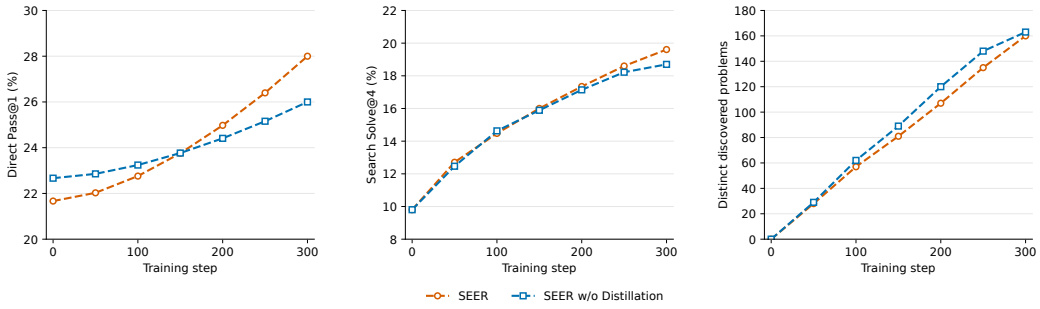


Figure 8: Discovery–reintegration training dynamics: (a) direct generation on APPS+; (b) meta-search on APPS*; (c) cumulative discovery.

Figure 9 shows how the two runs use their successful discoveries. Full SEER adds 38, 41, 43, 43, 44, and 38 records across the six intervals, adding 247 training targets cumulatively. The no-distillation run produces more records overall (42, 63, 45, 50, 47, and 22; 269 in total), but none enter a buffer. Despite this larger discovery count, its direct Pass@1 ends at 26.00%, below the 28.00% reached by full SEER. This pattern is consistent with reuse: full SEER converts each eligible discovery into an original-prompt target throughout training, so later updates can improve direct generation even when search coverage is similar.

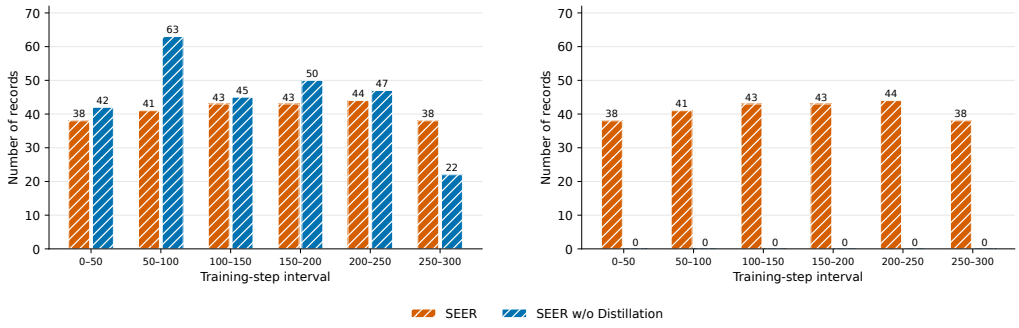


Figure 9: Discovery reuse: (a) new eligible discoveries; (b) new distillation-buffer records.

C BACKBONE GENERALIZATION

The family comparison replaces Qwen with DeepSeek-Coder-6.7B-Instruct or CodeGemma-7B-IT and retains the existing reward and optimization logic. Tokenization, chat formatting, special tokens,

model loading, and generation termination follow each backbone's own interface. This keeps the comparison focused on whether the training method remains useful across pretrained code policies.

Table 3 compares direct generation with Qwen2.5-Coder-7B-Instruct, DeepSeek-Coder-6.7B-Instruct, and CodeGemma-7B-IT on APPS+, HumanEval, and TACO. We report Pass@1, Pass@2, and Pass@3 for GRPO and SEER. Values are percentages. Avg. is the unweighted mean over the three benchmarks. Δ_1 is SEER minus GRPO at Pass@1, in percentage points; its Avg. entry averages the setting-wise differences.

Table 3: Direct-generation performance across model families

Backbone	Benchmark	GRPO			SEER			Δ_1
		Pass@1	Pass@2	Pass@3	Pass@1	Pass@2	Pass@3	(pp)
Qwen2.5-Coder 7B	APPS+	25.67	34.00	36.67	34.00	38.67	41.33	+8.33
	HumanEval	83.33	91.67	94.23	87.18	93.59	95.51	+3.85
	TACO	10.71	17.18	20.75	15.48	21.77	25.17	+4.77
	Avg.	39.90	47.62	50.55	45.55	51.34	54.00	+5.65
DeepSeek-Coder 6.7B	APPS+	26.00	31.33	34.00	32.00	37.00	39.67	+6.00
	HumanEval	83.49	90.38	93.59	85.26	92.31	94.23	+1.77
	TACO	10.03	16.41	19.90	14.12	20.41	23.89	+4.09
	Avg.	39.84	46.04	49.16	43.79	49.91	52.60	+3.95
CodeGemma 7B-IT	APPS+	24.67	30.33	33.33	30.67	36.00	39.00	+6.00
	HumanEval	82.92	89.74	91.67	85.90	91.03	94.87	+2.98
	TACO	8.76	15.48	19.05	12.67	19.30	22.87	+3.91
	Avg.	38.78	45.18	48.02	43.08	48.78	52.25	+4.30

SEER exceeds GRPO at all three sampling budgets in all nine backbone–task combinations. Its mean Pass@1 gains are 5.65, 3.95, and 4.30 percentage points for Qwen, DeepSeek, and CodeGemma, respectively. On APPS+, the gains are 8.33, 6.00, and 6.00 points; on TACO, they are 4.77, 4.09, and 3.91 points. The benefit therefore extends beyond Qwen and a single code benchmark.

The advantage also persists with more independent samples: mean Pass@3 gains are 3.45, 3.43, and 4.23 points across the three families. On HumanEval, SEER remains ahead at every sampling budget despite smaller and non-monotonic margins for some backbones. Overall, these results show that SEER's direct-generation gains hold across model families and r

D Trajectory-length normalization and optimization contribution

The length normalization acts on the trajectory-level component A^t. Broadcasting an unnormalized trajectory score over all generated tokens multiplies its total coefficient by trajectory length. Multiplication by $\bar{L}/L_i$ removes that length factor from the trajectory component. The first-answer and episode-level components retain their own token-level contributions.

Figure 10 examines this mechanism in 3B APPS trajectories. Each point pairs the generated completion length with an absolute token-weighted advantage proxy. The analysis uses the 0–1,024 generated-token range, comprising 2,517 normalized and 2,678 unnormalized trajectories. Fits use the trajectories in each panel; confidence bands resample whole generation batches to preserve within-batch dependence.

In Figure 10, Qwen 3B, APPS, training seed 69. The proxy sums absolute advantage contributions weighted by completion length. Lines are ordinary least-squares fits; bands are 95% generation-batch bootstrap intervals.

The fitted slope falls from 1.546 without normalization to 1.452 with normalization, a reduction of about 6.1%. The Pearson correlation falls from 0.677 to 0.625. Both remain positive: normalization

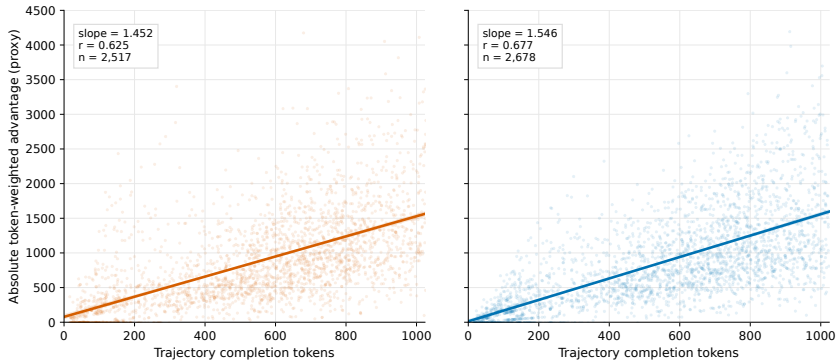


Figure 10: Length and advantage contribution: (a) normalized SEER; (b) unnormalized SEER.

rescales the broadcast trajectory score, while the aggregate proxy also includes other advantage components and differences between trajectories.

E TRANSFER FROM APPS TO UNSEEN TRAINING DOMAINS

We keep the APPS-trained checkpoint fixed and evaluate it on MBPP+ and TACO without target-specific training. This tests whether the acquired behavior can be used outside the training domain. Direct generation and feedback-conditioned search are evaluated separately.

Figure 11 separates direct generation from feedback-conditioned meta-search on MBPP+ and TACO. The gray, blue, and orange bars denote GRPO direct generation, SEER direct generation, and SEER meta-search, respectively.

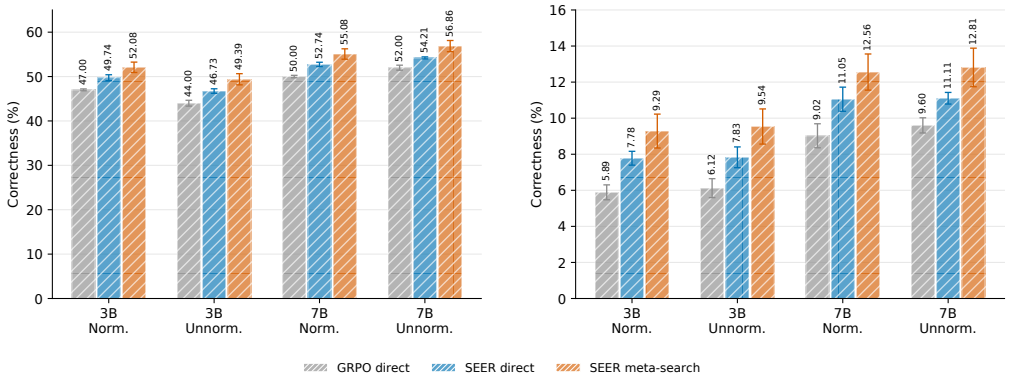


Figure 11: Cross-dataset transfer on MBPP+ (left) and TACO (right): GRPO direct generation versus SEER direct generation and meta-search.

SEER improves direct generation on both transfer tasks. Relative to GRPO, the gains are 2.21–2.74 percentage points on MBPP+ and 1.51–2.03 points on TACO. Meta-search adds a further 2.34–2.66 and 1.51–1.71 points, respectively, over SEER direct generation. This pattern holds across 3B and 7B and both normalization settings, showing that training gains extend to the target tasks and that feedback-conditioned search retains additional problem-solving capability.

The improvements in both direct generation and meta-search are consistent with SEER's aim of reintegrating search experience into the policy. The further increase in solve rate with meta-search suggests that execution feedback and subsequent attempts can still help resolve problems left unsolved by the initial response.

F SEARCH-COST ACCOUNTING AND VERIFIER BUDGETS

Generated-token cost includes code and reflection completions. Prompt tokens are excluded from this axis, and verifier requests are counted separately. One verifier request evaluates one candidate program; it may execute several test cases. Hidden-test scoring is performed offline and is not counted as feedback made available to the policy.

For a recorded sequence of candidates, we replay events in outer-search order and then episode order. A budget admits only completed candidates whose cumulative cost is within the limit. Any preceding reflection is included in the cost of reaching its subsequent candidate. This gives an empirical coverage curve over the stored records. Coverage plateaus once all stored candidates for a problem have been evaluated.

In Table 4, costs cover 102 problems and 408 outer units. The table reports code, reflection, and verifier costs for the two search configurations at each model scale. Verifier calls count candidate evaluations.

Table 4: Search costs

Size	Method	Code tokens	Reflection tokens	Total tokens	Calls
3B	SEER	245,690	59,063	304,753	1,052
3B	LaMer	231,147	58,009	289,156	1,056
7B	SEER	219,476	41,069	260,545	995
7B	LaMer	228,649	43,427	272,076	979

The token-coverage curve uses the recorded generated-token budget; the left panel is 3B and the right panel is 7B. The verifier-coverage curve applies the corresponding candidate-call budget. Both curves replay fixed search records in search-ID and attempt order, and additionally compare normalized and unnormalized SEER under the same budget accounting. Generated tokens include code and reflection. Hidden-test oracle coverage counts a problem as solved once any completed candidate passes its hidden tests; hidden outcomes are used only for offline scoring.

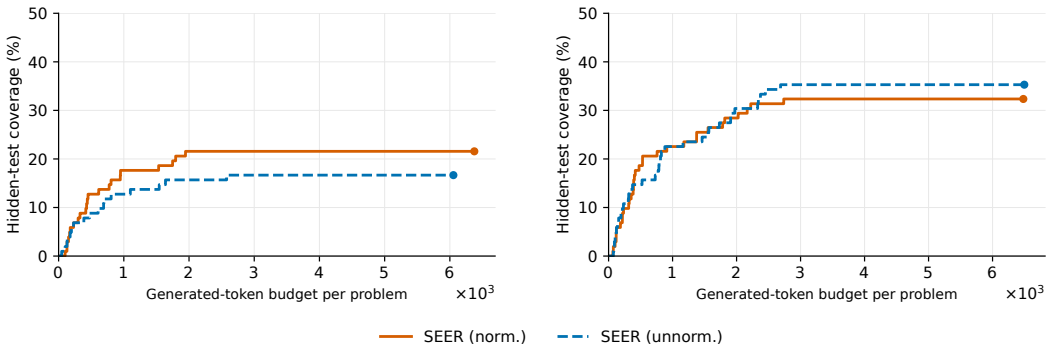


Figure 12: Token-budget coverage: (a) 3B; (b) 7B.

Figure 12 shows a rapid increase in coverage at lower token budgets, followed by a plateau. At 3B, normalized SEER has higher endpoint coverage than the unnormalized variant; at 7B, the ordering is reversed, and both variants exceed their corresponding 3B results. Figure 13 shows the gains from additional verifier calls: the 3B curves flatten at approximately eight calls, whereas the 7B curves continue to discover solutions, with the unnormalized variant reaching the highest endpoint. The two views measure search cost in generated tokens and verifier calls, respectively, while also illustrating the effect of trajectory-length normalization across search budgets.

G TRAINING SEARCH DEPTH AND DIMINISHING RETURNS

Figure 14 compares the 3B models trained with at most $H = 3, 4, 5$ code episodes per meta-search, using a fixed evaluation budget on APPS and TACO. On APPS, SEER remains at 26.33%, 26.00%,

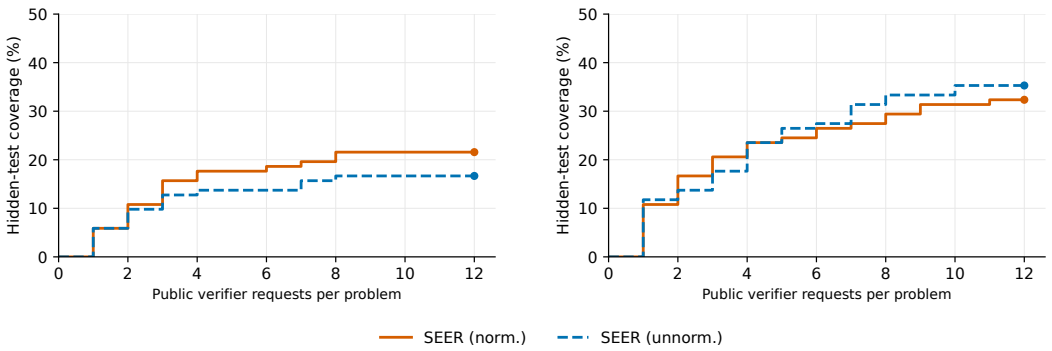


Figure 13: Verifier-budget coverage: (a) 3B; (b) 7B.

and 26.00%. LaMer stays at 25.67% across all three settings, while ReflexiCoder rises from 24.67% to 25.00% and 25.67%. SEER therefore leads at every budget; the largest baseline improvement is still insufficient to close the gap at $H = 5$.

TACO benefits more from deeper training searches. SEER rises from 18.62% to 21.60% and 22.36%, while LaMer rises from 14.97% to 16.67% and 17.09%. ReflexiCoder, StepCoder, and GRPO reach 13.95%, 12.84%, and 13.86% at $H = 5$. The SEER–LaMer gap grows from 3.65 to 5.27 points, and the gap over the strongest of the three direct baselines grows from 6.38 to 8.41 points. Increasing training depth thus preserves the APPS lead and expands the TACO lead.

The marginal return declines at the largest budget. SEER gains 2.98 points from $H = 3$ to $H = 4$ but only 0.76 points from $H = 4$ to $H = 5$; LaMer gains 1.70 and 0.42 points. The final increments for ReflexiCoder, StepCoder, and GRPO are 0.26, 0.08, and 0.17 points. The smaller final increment indicates diminishing returns from increasing training depth. SEER retains the largest final increment on TACO, suggesting that longer training trajectories continue to improve the learned policy under the fixed evaluation budget.

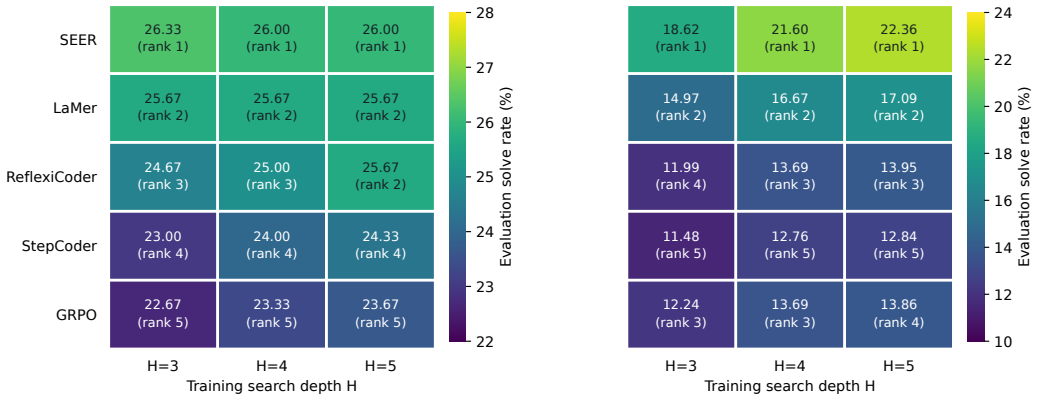


Figure 14: Training search depth: (a) APPS+; (b) TACO. Models are trained with $H = 3, 4, 5$ code episodes per meta-search and evaluated at a fixed budget.

H Training rollout budget and compute

Figure 15 compares SEER, LaMer, GRPO, and Instruct across three rollout settings, labeled $h = 2, 3, 4$. The horizontal axis measures generated code tokens, and the panels report overall performance and coverage on Unseen@4 problems. The settings allocate 9, 12, or 15 candidates per problem to the direct-sampling controls and at most 12, 15, or 18 code candidates to the search methods. Successful searches terminate early, so their realized token cost depends on when a solution is found.

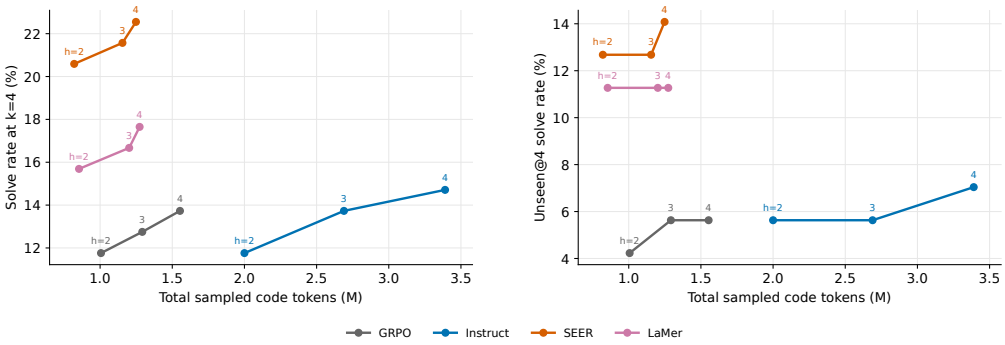


Figure 15: Training budget and search performance: (a) left, solve rate on all 102 APPS* problems; (b) right, solve rate on the 71-problem Unseen@4 subset.

Increasing the rollout allocation improves all four methods, but SEER maintains the highest performance throughout. From $h = 2$ to $h = 4$, its score rises from 20.59% to 22.55%, compared with 15.69% to 17.65% for LaMer, 11.76% to 13.73% for GRPO, and 11.76% to 14.71% for Instruct. In the configuration summarized in Table 5, SEER reaches 21.57% with 1.155 million code tokens, exceeding GRPO by 8.82 points while using 10.6% fewer code tokens. It also exceeds LaMer by 4.90 points at a slightly lower code-token cost (1.155 versus 1.201 million). The SEER–LaMer gap remains at 4.90 points across the three settings, showing a sustained performance advantage as the rollout allocation increases.

Table 5 reports the configuration with 12 candidates for GRPO and Instruct and at most 15 for SEER and LaMer. Token counts are in millions, and scores are percentages.

Table 5: Rollout configurations

Method	Candidates	Code tokens	Optimizer tokens	Reported score at $k = 4$
GRPO	12	1.292	1.292	12.75
Instruct	12	2.689	—	13.73
SEER	≤ 15	1.155	2.006	21.57
LaMer	≤ 15	1.201	2.088	16.67

Figure 15(b) examines coverage on Unseen@4 problems missed by the initial reference samples. As the rollout allocation increases, SEER's coverage rises from 12.68% to 14.08%, while LaMer remains at 11.27%. At $h = 3$, SEER reaches 12.68%, compared with 11.27% for LaMer and 5.63% for both GRPO and Instruct. SEER's coverage advantage thus extends to initially unreached problems.

Table 5 separates generated code tokens from tokens processed by the optimizer. SEER uses 1.155 million code tokens and 2.006 million optimizer tokens, compared with 1.292 million of each for GRPO. SEER therefore generates fewer code tokens while processing more tokens during optimization. Relative to LaMer, SEER uses fewer tokens in both categories (1.155 versus 1.201 million code tokens and 2.006 versus 2.088 million optimizer tokens) and achieves a 4.90-point higher score.

I LoRA CAPACITY AND DIFFICULTY GROUPS

The capacity comparison varies LoRA rank over $16, 32, 64, 128$ with scaling $\alpha = 2r$. It reports the final 3B checkpoint on APPS+ and HumanEval. Figure 16(a) breaks down APPS+ accuracy by difficulty, while Figure 16(b) compares the full set with the subset excluding the 23 competition questions. We compare final checkpoints; the rank-16 reference uses a different rollout microbatch.

In Figure 16(a), introductory accuracy ranges from 45.3% to 47.0%, and interview accuracy from 14.4% to 16.9%. Competition accuracy peaks at 8.7% with rank 64 and falls to zero at ranks 32

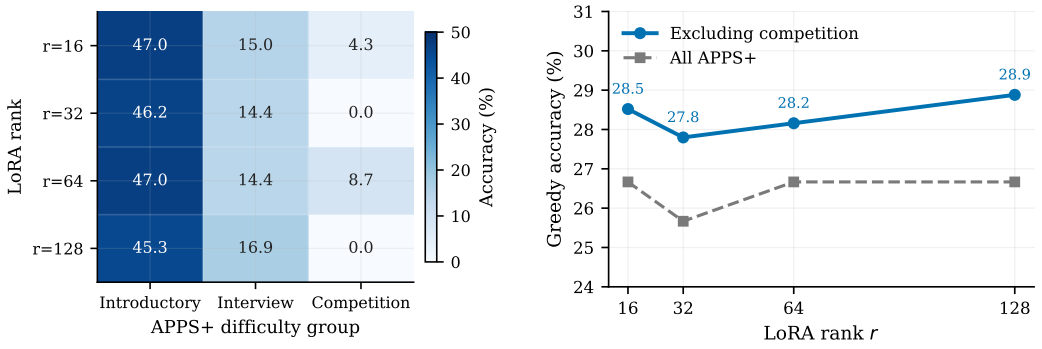


Figure 16: LoRA rank and accuracy on APPS+: (a) accuracy by difficulty; (b) accuracy on the full set and the subset excluding competition questions.

and 128. The difficulty breakdown further shows that increasing rank does not consistently improve accuracy.

Figure 16(b) compares accuracy on the 300-question APPS+ evaluation set and its 277-question introductory/interview subset, using the same predictions.

Across LoRA ranks 16 to 128, accuracy on the full APPS+ evaluation set ranges from 25.67% to 26.67%, with no monotonic improvement. Excluding competition questions gives accuracies of 27.80%–28.88%; rank 128 performs best on this subset, while ranks 16, 64, and 128 tie on the full set. The difficulty breakdown shows how these totals arise: rank 64 solves two competition problems, whereas rank 128 solves more interview problems. Increasing adaptation capacity changes which difficulty groups benefit, with limited change in overall accuracy in this setting.

J SENSITIVITY TO HISTORY AND FEEDBACK HANDLING

History determines what a later episode can learn from earlier attempts. We compare full SEER with removal of feedback, the no-reset-history setting, and shuffled feedback. Figure 17 expresses each score relative to full SEER, with full SEER set to one.

In Figure 17, parenthesized values are the absolute percentages; bars divide them by full SEER's 71.33%. The evaluation cohort differs from the matched first-failure cohort.

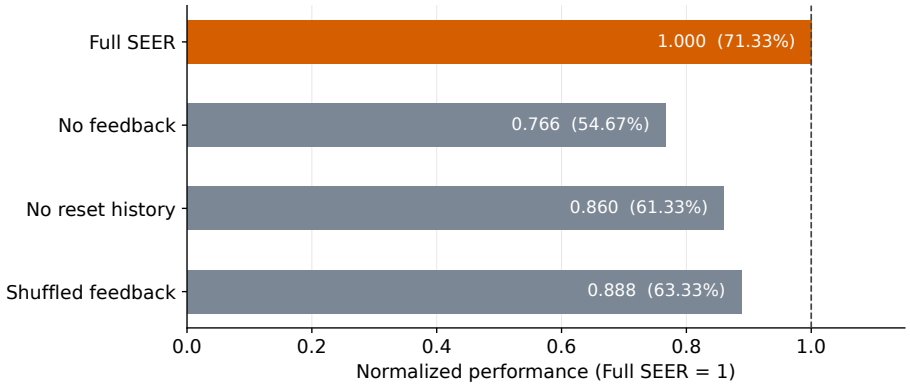


Figure 17: History and feedback sensitivity

The full score is 71.33%, compared with 54.67% without feedback, 61.33% without history reset, and 63.33% under shuffled feedback. These correspond to relative scores of 0.766, 0.860, and 0.888. Full history gives the highest performance, while removing feedback causes the largest drop, showing that execution outcomes provide useful information for later episodes. The no-reset and

shuffled-feedback conditions also change the final score, indicating that search depends on how prior information is organized.

K VARIATION ACROSS REPEATED EVALUATIONS

Table 6: Repeated-evaluation variability

Method	HE D	HE S	APPS+ D	APPS+ S
SEER	84.77±1.79	96.55±0.00	28.44±0.51	38.00±1.00
GRPO	70.69±1.72	91.95±1.99	20.11±1.35	28.00±2.52
StepCoder	72.99±1.00	93.10±0.00	23.67±0.58	32.00±0.58
LaMer	78.16±1.99	93.10±0.00	25.00±0.58	33.00±1.15

We compare SEER, GRPO, StepCoder, and LaMer across three repeated evaluations on the HumanEval adaptation subset and APPS+ (Figure 18). We report the mean and sample standard deviation separately for direct generation and search to examine whether the method ordering persists across evaluation randomness. In Table 6, values are means and sample standard deviations over three repetitions, in percent. D denotes Direct Pass@1 and S denotes Meta-search Solved@4 in the direct and search panels in Figure 18. Figure 18 shows the results for evaluation seeds 114, 514, and 810, with one point per seed.

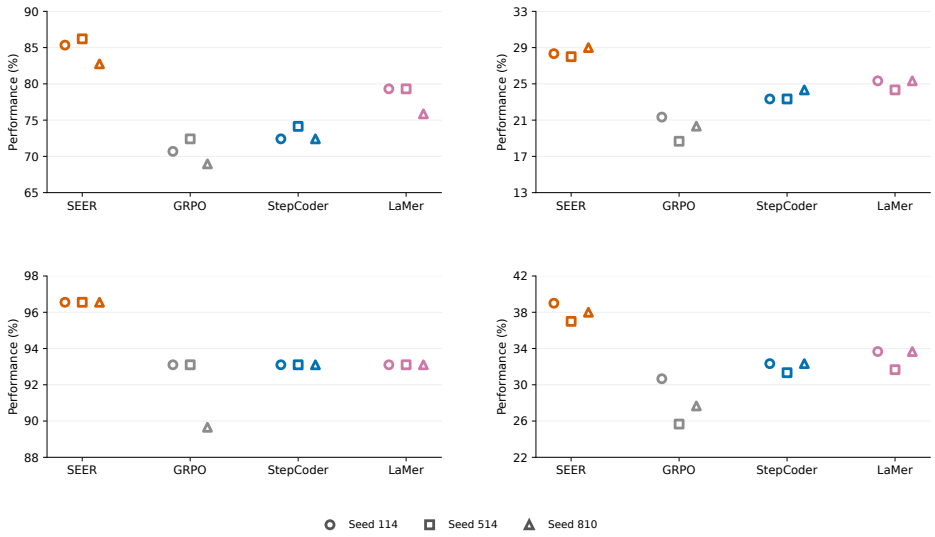


Figure 18: Variability across evaluation seeds: (a) HumanEval direct; (b) APPS+ direct; (c) HumanEval meta-search; (d) APPS+ meta-search

SEER has the highest mean in all four panels. On APPS+, its direct score is 28.44%, compared with 25.00% for LaMer, 23.67% for StepCoder, and 20.11% for GRPO. Its search score reaches 38.00%, exceeding LaMer, StepCoder, and GRPO by 5.00, 6.00, and 10.00 points, respectively. SEER's search standard deviation is 1.00 points, compared with 1.15 for LaMer, 0.58 for StepCoder, and 2.52 for GRPO. Thus, its higher average coverage is accompanied by less variation than GRPO, while StepCoder has the smallest variation at a lower mean.

On HumanEval, SEER reaches 84.77% in direct generation, ahead of LaMer at 78.16%, StepCoder at 72.99%, and GRPO at 70.69%. Its search score is 96.55% in all three evaluations; LaMer and StepCoder reach 93.10%, and GRPO averages 91.95%. SEER maintains its lead across all three evaluation seeds in both direct generation and search on the two datasets.

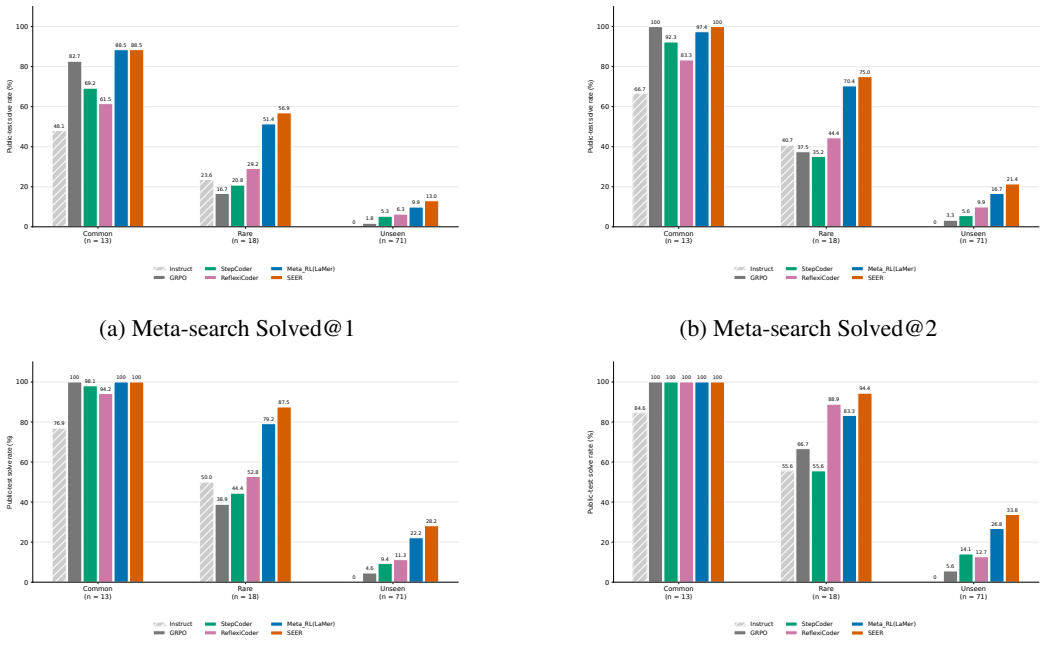
Table 7: Low-budget coverage by reachability group

Group	Budget	Instruct	GRPO	StepCoder	ReflexiCoder	LaMer	SEER
Common	$k = 1$	48.1	82.7	69.2	61.5	88.5	88.5
	$k = 2$	66.7	100.0	92.3	83.3	97.4	100.0
	$k = 3$	76.9	100.0	98.1	94.2	100.0	100.0
	$k = 4$	84.6	100.0	100.0	100.0	100.0	100.0
Rare	$k = 1$	23.6	16.7	20.8	29.2	51.4	56.9
	$k = 2$	40.7	37.5	35.2	44.4	70.4	75.0
	$k = 3$	50.0	38.9	44.4	52.8	79.2	87.5
	$k = 4$	55.6	66.7	55.6	88.9	83.3	94.4
Unseen@4	$k = 1$	0.0	1.8	5.3	6.3	9.9	13.0
	$k = 2$	0.0	3.3	5.6	9.9	16.7	21.4
	$k = 3$	0.0	4.6	9.4	11.3	22.2	28.2
	$k = 4$	0.0	5.6	14.1	12.7	26.8	33.8

L REACHABILITY STRATA AND LARGER SAMPLING BUDGETS

We analyze APPS* using three fixed reference-sampling strata: Common (13 problems, three or four successful reference samples), Rare (18, one or two), and Unseen@4 (71, none).

Meta-search Solved@1–4 by reachability group. Figure 19(a)–(d) reports public-test Meta-search Solved@1, Meta-search Solved@2, Meta-search Solved@3, and Meta-search Solved@4, respectively, for the same three groups. Meta-search Solved@4 is the four-unit solve coverage also shown in the teaser. Each meta-search allows up to three code episodes, including retries conditioned on feedback.



(a) Meta-search Solved@1

(b) Meta-search Solved@2

(c) Meta-search Solved@3

(d) Meta-search Solved@4

Figure 19: Reachability-group analysis

We partition APPS* according to the number of successful solutions among four direct samples from the Instruct reference policy: Common (3–4 successes), Rare (1–2), and Unseen@4 (0). Thus, by construction, Instruct has 0% coverage on the Unseen@4 subset under this four-sample reference

budget. Table 7 and Figure 19 jointly summarize coverage over these three groups for $k = 1, 2, 3, 4$, reported in percentages. The table provides the corresponding numerical values, while the figure visualizes how coverage evolves with increasing budget. Direct methods are evaluated using Pass@k, whereas SEER and LaMer use Meta-search Solved@k. The results show distinct saturation patterns across the three groups. On Common problems, SEER increases from 88.5% at Meta-search Solved@1 to 100% at Meta-search Solved@2 and remains saturated for larger k, indicating that additional search beyond two units provides no further coverage gain. Rare problems improve more gradually, from 56.9% to 75.0%, 87.5%, and 94.4% as k increases from 1 to 4. The successive gains of 18.1, 12.5, and 6.9 percentage points show diminishing returns as coverage approaches saturation. In contrast, Unseen@4 continues to benefit throughout the full budget range, increasing from 13.0% at $k = 1$ to 21.4%, 28.2%, and 33.8% at $k = 2, 3, 4$. Thus, Common saturates quickly, Rare approaches saturation more gradually, and Unseen@4 retains substantial headroom at $k = 4$.

Compared with Meta-RL (LaMer), SEER's advantage on Unseen@4 increases from 3.1 percentage points at Meta-search Solved@1 to 4.7, 6.0, and 7.0 points at the subsequent budgets. On Rare problems, the corresponding gaps are 5.5, 4.6, 8.3, and 11.1 points. At Meta-search Solved@4, SEER also exceeds the direct baselines on Rare, achieving 94.4% coverage compared with 88.9% for ReflexiCoder, 66.7% for GRPO, and 55.6% for both StepCoder and Instruct. On Unseen@4, Instruct remains at 0% across all four budgets, while ReflexiCoder reaches 6.3%, 9.9%, 11.3%, and 12.7%. SEER reaches 33.8% at $k = 4$, approximately 2.7× the coverage of ReflexiCoder.

M Representative successful search trajectories

Three APPS* searches from 3B SEER illustrate input handling, task-logic repair, and algorithm restructuring. We show all seven code-generation responses and four reflections in chronological order, with execution feedback summarized alongside each response. Success denotes passing the public tests used by the search verifier. Complete prompts and JSON feedback are provided in the supplementary trajectory files, indexed by problem and search ID.

Table 8: Three routes from failure to a verified solution

Problem / search	Main change	Tests passed by episode
2428 / 2	Integer-to-string conversion	$0/8 \rightarrow 8/8$
5691 / 2	Preserve the specified traversal order	$0/1 \rightarrow 1/1$
7005 / 0	Sliding window and executable input/output	$1/2 \rightarrow 0/2 \rightarrow 2/2$

Shared prompt structure. Code generation receives the problem and starter code, followed by retained reflection for retries. Reflection receives the problem, attempted program, and execution feedback. The shared reflection instruction is:

Reflection prompt

```
Output exactly 2 SHORT and CONCISE bullets:

- #Attempt Summary: Briefly summarize the current approach and
    observed failure signal.
- #Guidance: Suggest the most likely fix or one better next approach,
    using only provided feedback.

Keep SHORT and CONCISE. Do not invent unseen tests. Output no
    reasoning, steps, pseudocode, code, or extra text.
```

M.1 Input format and type handling (problem 2428, search 2)

The task classifies a card number by prefix and length: VISA uses prefix 4 and length 13 or 16; AMEX uses 34 or 37 and length 15; Discover uses 6011 and length 16; Mastercard uses 51–55 and length 16. Unmatched inputs return Unknown.

The card-issuer task receives an integer card number. Episode 1 applies string operations directly to that integer and raises an AttributeError. The reflection identifies the type mismatch; episode 2 converts the number to a string before testing prefixes and lengths, and passes all eight public tests. The repair preserves the classification structure while correcting its input representation.

Episode 1: model response

```
```python
def get_issuer(number):
    if number.startswith('4') and (len(number) == 13 or len(number) ==
     16):
        return "VISA"
    elif number.startswith('34') or number.startswith('37') and len(
    number) == 15:
        return "AMEX"
    elif number.startswith('6011') and len(number) == 16:
        return "Discover"
    elif number.startswith('51') or number.startswith('52') or number.
    startswith('53') or number.startswith('54') or number.startswith
    ('55') and len(number) == 16:
        return "Mastercard"
    else:
        return "Unknown"
```
```

Tests passed	Verifier feedback
0/8	Runtime error: integer input has no startswith method.

Reflection before episode 2

```
- #Attempt Summary: The function attempts to match a card number's
    prefix and length against known card types. It uses `startswith`
    but fails because the input is an integer instead of a string.
- #Guidance: Convert the integer `number` to a string before using `
    startswith`.
```

Episode 2: model response

```
```python
def get_issuer(number):
    number_str = str(number)
    if number_str.startswith('4') and len(number_str) in [13, 16]:
        return 'VISA'
    elif number_str.startswith('3') and (number_str.startswith('34')
    or number_str.startswith('37')) and len(number_str) == 15:
        return 'AMEX'
    elif number_str.startswith('6011') and len(number_str) == 16:
        return 'Discover'
    elif number_str.startswith('5') and number_str[1] in ['1', '2',
    '3', '4', '5'] and len(number_str) == 16:
        return 'Mastercard'
    else:
        return 'Unknown'
```
```

Tests passed	Verifier feedback
8/8	All public tests passed.

M.2 RESTORING THE TASK LOGIC (PROBLEM 5691, SEARCH 2)

The input gives the number of fragments and their sector positions in traversal order. The required output is the sum of absolute differences between consecutive positions.

The disk-fragment task asks for travel distance in the supplied fragment order. Episode 1 sorts the sectors and introduces a starting position of one, changing the requested traversal. After feedback, the reflection proposes summing distances between consecutive fragments. Episode 2 preserves the input order and implements that sum, passing the public test.

Episode 1: model response

```
```python
n = int(input())
fragments = list(map(int, input().split()))

# Sort the fragments to minimize the total distance traveled
fragments.sort()

total_distance = 0
current_sector = 1

for fragment in fragments:
    # Calculate the distance from the current sector to the next
    fragment's sector
    distance = abs(current_sector - fragment)
    total_distance += distance

    # Move to the next fragment's sector
    current_sector = fragment

print(total_distance)
```
```

Tests passed	Verifier feedback
0/1	Wrong answer; reported output: 2.

Reflection before episode 2

```
- #Attempt Summary: The incorrect code sorts the fragments to minimize
     the total distance traveled but fails to account for the total
    distance the head needs to travel for all fragments.
- #Guidance: To fix the issue, calculate the total distance the head
    needs to travel for each fragment by summing up the absolute
    differences between consecutive fragments' sectors, then print
    this total distance.
```

Episode 2: model response

```
```python
n = int(input())
fragments = list(map(int, input().split()))

# Calculate the total distance traveled by the head
total_distance = 0
for i in range(1, n):
    total_distance += abs(fragments[i] - fragments[i - 1])

print(total_distance)
```
```

Tests passed	Verifier feedback
1/1	All public tests passed.

M.3 RESTRUCTURING THE ALGORITHM (PROBLEM 7005, SEARCH 0)

The input gives student skill values. The output is the largest team whose maximum and minimum skill differ by at most five.

The balanced-team task requires the largest group whose maximum and minimum skill differ by at most five. Episode 1 uses frequency counts and an endpoint check, passing one of two public tests. The first reflection proposes a sliding window. Episode 2 introduces two pointers but omits executable input/output and advances the left boundary only once per iteration; it passes neither test. Episode 3 restores input/output and uses a while loop to enforce the window constraint, passing both public tests. The complete trajectory shows the move from frequency-based counting to a sorted sliding window, followed by repairs to its implementation.

Episode 1: model response

```
```python
import sys
from collections import Counter

def max_balanced_team(n, skills):
    skills.sort()
    freq = Counter(skills)
    max_count = 0

    for skill, count in freq.items():
        max_count = max(max_count, count)

        # Check for consecutive skills within 5 range
        if skill + 5 in freq:
            max_count = max(max_count, min(count, freq[skill + 5]))

    return max_count

if __name__ == "__main__":
    n = int(input())
    skills = list(map(int, input().split()))
    print(max_balanced_team(n, skills))
```
```

Tests passed	Verifier feedback
1/2	Wrong answer; reported output: 1.

Reflection before episode 2

```
- #Attempt Summary: The code attempts to sort the skills and count
    occurrences. It checks for consecutive skills within 5 range but
    fails to handle large datasets efficiently.

- #Guidance: Consider using a sliding window approach to find the
    longest subarray where the difference between any two elements is
    at most 5. This will help in finding the maximum balanced team
    more efficiently.
```

Episode 2: model response

````
```python
from typing import *
from collections import *

def max_balanced_team(n: int, skills: List[int]) -> int:
    skills.sort()

    left = 0
    right = 0
    max_team_size = 0

    while right < n:
        if right > left and skills[right] - skills[left] > 5:
            left += 1

        max_team_size = max(max_team_size, right - left + 1)
        right += 1

    return max_team_size

# Example usage:
# n = int(input())
# skills = list(map(int, input().split()))
# print(max_balanced_team(n, skills))
```

This solution uses a sliding window approach to find the longest
    subarray where the difference between any two elements is at most
    5. This is efficient for handling large datasets.
````

Tests passed	Verifier feedback
0/2	Wrong answer; empty output on both public tests.

Reflection before episode 3

```
- #Attempt Summary: The code sorts the skills and uses two pointers to
     find the maximum team size with a difference of no more than 5
    between any two skills. It fails on large datasets.
- #Guidance: Consider using a sliding window approach to efficiently
    find the maximum team size without sorting, which can handle
    larger datasets more effectively.
```

Episode 3: model response

```
```python
from typing import List

def max_balanced_team(students: List[int]) -> int:
    students.sort()
    n = len(students)
    max_team_size = 0

    left = 0
    for right in range(n):
        while students[right] - students[left] > 5:
            left += 1
        max_team_size = max(max_team_size, right - left + 1)

    return max_team_size

if __name__ == "__main__":
    import sys
    input = sys.stdin.read
    data = list(map(int, input().split()))
    n = data[0]
    students = data[1:]
    print(max_balanced_team(students))
```
```

Tests passed	Verifier feedback
2/2	All public tests passed.

How the search develops. The first two cases repair the input representation and traversal logic within one retry. The third changes the algorithm, then repairs its implementation after another failed episode. Its public-test score falls from $1/2$ to $0/2$ before reaching $2/2$: the intermediate failure lies on a successful continuation. The reflections propose directions that subsequent code revises; the final program retains sorting despite the second reflection suggesting otherwise. Together, the trajectories illustrate how feedback-conditioned attempts can move from local correction to algorithm restructuring.