SEETHAT: LEARNING TO OPTIMIZE CODE THROUGH RECURSIVE POLICY–DATA CO-EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Learning to optimize code requires feedback on both correctness and runtime, which program execution can provide. Beyond supplying rewards, execution also identifies valid program variants that can serve as inputs for further optimization. Recursively reusing these variants allows a policy to recursively expand its own training data, but also creates more candidate inputs than a finite training budget can afford. The resulting challenge is to identify which candidates offer useful learning opportunities before spending the budget to generate and execute their refinements. We introduce SEETHAT, which trains the code-optimization policy itself to predict whether an input will yield a rollout group with variation in validity or performance outcomes. These predictions are supervised by execution feedback already collected during training, with region-specific supervision for prediction and code generation. The policy then uses its predictions to rank newly admitted variants for future training, without generating or executing additional refinements or requiring a separate value network. In this way, policy learning and recursive training-data construction co-evolve. Across systems-code and competitive-programming optimization with both dense and mixture-of-experts models, SEETHAT achieves the lowest P90 and P95 runtimes among the compared training methods in all four model–benchmark settings, and reaches the lowest P75 runtimes in three settings.

1 INTRODUCTION

Code optimization improves software performance by transforming a program to reduce its execution cost while preserving its behavior. While modern compilers automate many low-level transformations, high-level optimization (including choices of algorithms, data structures, libraries, and program structure) can still yield non-trivial performance gains. Recent work has therefore explored large language models for code optimization and refinement, using execution feedback to learn performance-improving transformations (Shypula et al., 2025; HUANG et al., 2024; Gao et al., 2025; Du et al., 2025). Yet effective optimization remains challenging: useful transformations must preserve correctness while improving efficiency across a large space of possible refinements.

At the same time, this correctness requirement gives code optimization a useful structure for learning. In common reinforcement-learning formulations for code optimization (Du et al., 2025; Wei et al., 2026; Chambon et al., 2026), the policy generates multiple rollouts for each input program, and their execution outcomes provide rewards for policy updates. However, this standard training loop leaves a useful property of code optimization underexploited: *closure under verified refinement*. Each rollout is itself a program variant that, once verified to preserve the behavior, can serve as a new input to the same optimization task in a subsequent attempt. A rollout therefore provides not only a one-time training signal: an accepted output can also become future training data. Recursively applying this refinement process produces an expanding set of verified-equivalent program variants connected by refinement steps.

This closure property enables what we call *recursive policy–data co-evolution*. At training step t, the current policy generates candidate refinements, and verified variants are admitted to an evolving equivalence-class pool $\mathcal{C}_t$. These variants may be selected as inputs for later refinement. Training on selected candidates from the pool updates the policy, which in turn changes the variants generated in subsequent iterations. The training data therefore evolve with the policy, while the evolving

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.

policy recursively expands and reshapes its own future training distribution. We view this as a *restricted, data-mediated form of recursive self-improvement* (Good, 1965; Zelikman et al., 2024): learning algorithm remains fixed, while policy and its training data co-evolve, as in self-generated data training (Zelikman et al., 2022; Huang et al., 2026; Zhao et al., 2025).

Recursive reuse, however, creates a rapidly growing pool: if each input admits up to b accepted refinements, its size can grow as $O(b^d)$ after d refinement steps, far beyond what a fixed generation and execution budget can cover. We must therefore choose which candidates to expand with additional rollout groups. These choices also shape how the pool evolves: only selected inputs produce descendants, so each choice determines not only the current learning signal but also which parts of the pool are expanded next.

Which candidates should we expand? In group-relative RL, homogeneous rollout groups provide little learning signal, and a group with zero reward variance yields zero advantage and wastes its generation and execution budget. DAPO's dynamic sampling (Yu et al., 2025) identifies such groups only after generating and executing their rollouts, making it unsuitable for deciding which candidates in a recursively expanding pool to spend that budget in the first place. History-based filters (Zheng et al., 2025) instead rely on previously observed rollout rewards, which are unavailable for newly generated variants with no rollout history. We therefore predict, *before expansion*, whether a candidate is likely to yield an informative rollout group. Specifically, we predict two within-group contrasts: a *validity contrast* between valid non-trivial refinements and other attempts, and a *performance contrast* between near-baseline and substantially faster valid refinements.

We introduce SEETHAT, a framework that couples code-optimization policy learning with recursive training-data construction. Verified program variants enter an evolving candidate pool, and the policy itself predicts the two outcome contrasts to select variants for subsequent training. Execution outcomes already collected from training rollouts supervise the predictions, while separate objectives train prediction and code generation within the same policy.

Our contributions are:

- We show that closure under verified refinement enables *recursive policy–data co-evolution* in code-optimization RL, and identify the resulting bottleneck: selecting inputs from an exponentially growing equivalence-class pool, where executing every candidate is unaffordable.
- We propose SEETHAT, in which the optimization policy itself learns, from execution feedback it already receives, to predict which candidates will yield informative rollout groups *before allocating a rollout group to them*, without requiring a separate value network.
- On PerfCodeBench and PIE, with dense and mixture-of-experts models, SEETHAT achieves the lowest runtime at all four evaluated percentiles in three of the four settings and the lowest P90 and P95 in all four. It also tends to reach lower P75 runtimes in fewer training steps, while prediction increases training cost by only 3.65% for the same number of training steps.
- Ablations support recursive data evolution and training-value-guided selection: limiting reuse to depth-1 variants roughly quadruples P75 runtime, while randomizing selection with prediction training retained raises P95 runtime by $2.8\times$.

2 RELATED WORK

Execution-guided code improvement. Self-refinement and self-correction use feedback to revise earlier outputs (Madaan et al., 2023; Welleck et al., 2023; Shinn et al., 2023; Chen et al., 2024; Jin et al., 2026), with gains sensitive to feedback quality (Huang et al., 2024; Olausson et al., 2024). For efficiency optimization, PIE learns performance-improving edits from human traces (Shypula et al., 2025), while Afterburner and SuperCoder use reinforcement learning for source-code and assembly optimization, respectively (Du et al., 2025; Wei et al., 2026). Recent work examines the timing environment, reward design, and policy updates needed to learn from noisy execution times (Chambon et al., 2026). These approaches inform how optimization models are trained; our focus is selecting their training inputs.

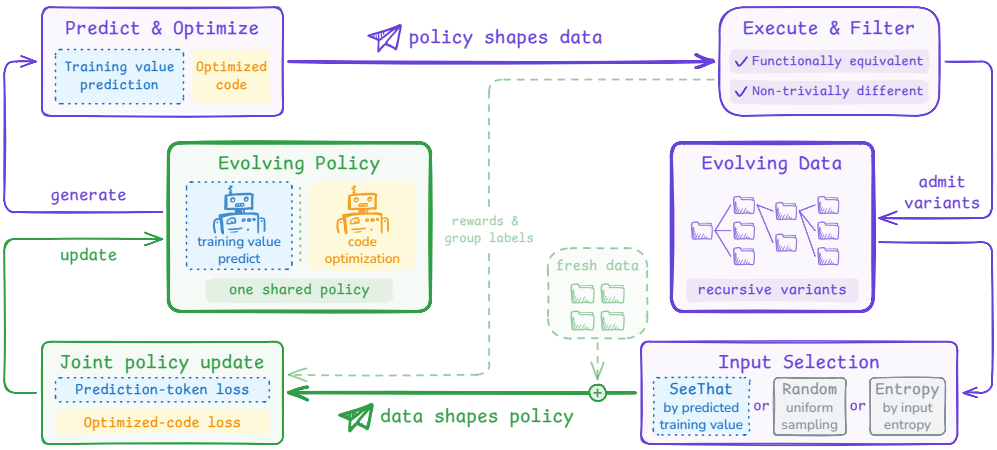


Figure 1: **Policy–data co-evolution in SEETHAT.** One shared policy predicts training value (blue) and optimizes code (orange). Policy shapes data: verified variants enter the evolving data pool as future inputs, and the policy's own predictions select which pool variants are blended with fresh inputs. Data shapes policy: group labels supervise prediction through binary cross-entropy, while execution rewards train optimized-code tokens through GRPO in a joint update. The input-selection stage supports interchangeable strategies: SEETHAT selects by predicted training value, while *Random* and *Select by entropy* are two baseline alternatives.

Adaptive sampling. Adaptive curricula (Graves et al., 2017) and prioritized replay (Schaul et al., 2016) use past learning signals to guide sampling. AceGRPO repurposes execution traces into an evolving buffer of training tasks and samples states by learnability potential (Cai et al., 2026), while GRESO skips prompts using historical reward statistics (Zheng et al., 2025). Their history-based selection signals require prior rollout observations from the same state or input and therefore do not provide a variant-specific estimate of prospective training value before a newly generated variant's first expansion. AceGRPO assigns new states a default initial potential until such observations are available. In RL with verifiable rewards, DAPO identifies zero-variance groups only after generating their rollouts (Yu et al., 2025). PCL trains a separate value model to select intermediate-difficulty prompts (Gao et al., 2026), and GPS uses a lightweight, history-conditioned predictor with difficulty and diversity criteria (Qu et al., 2026). SEETHAT instead trains the code-optimization policy itself to predict the training value of unseen program variants before allocating generation and execution budget.

3 POLICY–DATA CO-EVOLUTION

SEETHAT couples policy learning with training-data construction (see Figure 1). A policy both predicts the training value of an input and optimizes its code. Its verified outputs become future training inputs, and training on those inputs changes the policy that produces subsequent variants.

3.1 OVERVIEW OF THE CO-EVOLUTION LOOP

At training step t, the pipeline maintains policy parameters θ_t and an equivalence-class pool $\mathcal{C}_t$ of accepted candidate inputs, grouped by their original problem and program. Let x be a programming problem, $z^{(0)}$ its initial program, and $g = (x, z^{(d)})$ a code-optimization input at depth d, where d counts successive refinement steps from $z^{(0)}$. Let p_θ denote the autoregressive response distribution with parameters θ. Given g, we independently sample K full responses $s_i \sim p_{\theta_t}(\cdot \mid g)$, indexed by $i = 1, \ldots, K$; K is the rollout group size, set to 8 in our experiments. Each s_i contains the two predictions and a generated program $z_i^{(d+1)}$. An accepted variant preserves the problem specification and passes the verification, so $(x, z_i^{(d+1)})$ is itself a valid input to the same code-optimization task. Each training step follows the loop in Figure 1:

1. **Predict & Optimize.** For each input g, the policy θ_t generates K responses. Each response first predicts the prospective training value of g and then proposes an optimized program.
2. **Execute & Filter.** The K generated variants are executed to evaluate correctness and performance; their collective outcomes determine the training value for input g, while each generated variant receives its own code-optimization reward. Verified variants are added to $\mathcal{C}_t$ as candidate inputs for future training.
3. **Input Selection.** For candidate inputs in $\mathcal{C}_t$, the same policy produces only the training-value predictions, without generating new code. The resulting scores rank recursive variants to use in the next training batch.
4. **Joint policy update.** One policy update combines separate token-region objectives: binary cross-entropy on training-value prediction answers and GRPO on optimized-code tokens. The resulting policy θ_{t+1} then generates responses for the next batch.

3.2 Joint Prediction and Code Optimization

We estimate training value through two outcomes of a prospective rollout expansion: a *validity contrast*, indicating whether the resulting rollouts contain both valid non-trivial refinements and other attempts, and a *performance contrast*, indicating whether valid non-trivial refinements include both near-baseline and substantially faster programs. These outcomes provide a tractable proxy for learning opportunities offered by an input under the current policy.

We implement training-value prediction and code optimization within the same autoregressive response format. For input g, the prediction fields estimate properties of the K-rollout expansion of g. During training, each sampled response contains two prediction fields followed by optimized code:

```
<predict_validity_contrast>Y/N</predict_validity_contrast>
<predict_performance_contrast>Y/N</predict_performance_contrast>
<optimized_code>Optimized code</optimized_code>
```

During pool scoring, the policy evaluates the prediction probabilities defined in Section 3.3, without generating optimized code.

Execution-derived labels. After the K responses for an input g are executed, we derive the targets for the two prediction fields from their collective outcomes. For a fixed input g, write $z_i = z_i^{(d+1)}$, suppressing the refinement depth. We similarly omit the input index g from per-response quantities unless needed to distinguish inputs in a batch. Let V_x be the binary verifier for problem x, and let $e_i = V_x(z_i) \in \{0, 1\}$ indicate whether the generated program passes its equivalence checks. Here equivalence is assessed by the available tests, not formally proved. Define $v_i \in \{0, 1\}$ to be 1 exactly when $e_i = 1$ and the normalized edit distance from the input program exceeds 0.01, and 0 otherwise. Thus $v_i = 1$ denotes a valid non-trivial refinement, while $v_i = 0$ includes both programs that fail verification and verified but trivial edits. This criterion is distinct from the stricter family-level pool-admission checks in Section 3.4. For verified programs with valid timing measurements, define the speedup $u_i = T(z^{(0)})/T(z_i)$, where $T(\cdot)$ is execution time. The reference is the original program, so larger u_i means faster code. We treat speedups within $\epsilon = 0.1$ of 1 as near-baseline. Writing $\mathbb{1}[A]$ for the indicator of condition A, the two binary targets are

$$y_{\text{val}}(g) = \mathbb{1}\left[0 < \sum_{i=1}^{K} v_i < K\right], \tag{1}$$

$$y_{\text{perf}}(g) = \mathbb{1}[\exists i : v_i = 1,\ 1 - \epsilon \le u_i \le 1 + \epsilon] \cdot \mathbb{1}[\exists i : v_i = 1,\ u_i > 1 + \epsilon]\,. \tag{2}$$

Although each of the K responses independently emits the two prediction tokens, their targets are shared: $y_{\text{val}}(g)$ and $y_{\text{perf}}(g)$ are computed once from the execution outcomes of all K responses.

3.3 Data-to-Policy: Joint Prediction and Code-Optimization Loss

Prediction and code generation share an autoregressive policy but use different objectives: binary cross-entropy supervises the prediction answers, while GRPO trains the code region from execution rewards. For an update at step t, $\theta_{\text{old}} = \theta_t$ denotes the fixed parameters used to sample the responses, while θ denotes the parameters being optimized.

Joint objective. Let $\mathcal{B}$ be the current batch of inputs, combining fresh inputs with selected pool variants, and let $|\mathcal{B}|$ denote its size. For each input g, we independently sample K full responses from $p_{\theta_{\text{old}}}(\cdot \mid g)$ and minimize

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{B}|K} \sum_{g \in \mathcal{B}} \sum_{i=1}^{K} \left[(1-r)\mathcal{L}_{g,i}^{\text{GRPO}}(\theta) + r\mathcal{L}_{g,i}^{\text{BCE}}(\theta) \right]. \tag{3}$$

Here $\mathcal{L}_{g,i}^{\text{BCE}}$ is the mean loss over the two prediction targets, and $\mathcal{L}_{g,i}^{\text{GRPO}}$ is the mean code-token loss for response i; both are defined below. $r \in (0,1)$, and we use $r = 0.05$ in practice. Detailed weighting formulation is in Appendix A.3. Thus r controls the prediction share of token weights, not loss magnitudes or gradient norms. Only prediction answer positions receive the BCE objective, and only code positions receive the GRPO objective; prompt and prediction-delimiter positions are excluded as loss targets. Both losses update the same policy.

Prediction loss. Let $\mathcal{J} = \{\text{val}, \text{perf}\}$ and let $\hat{a}_j \in \{\texttt{Y}, \texttt{N}\}$ denote the model's predicted answer for target j, distinct from the execution-derived ground-truth binary label $y_j(g)$ defined in Eqs. 1–2. Let c denote the model input at a prediction answer position, including g, the prediction-field formatting, and any specified preceding answer. Conditioning on g below implicitly includes this fixed formatting. At an answer position, binary probabilities are obtained by normalizing the policy's probabilities over the two answer tokens:

$$p_\theta(\hat{a}_j = \texttt{Y} \mid c) = \frac{p_\theta(\texttt{Y} \mid c)}{p_\theta(\texttt{Y} \mid c) + p_\theta(\texttt{N} \mid c)}.$$

The validity prediction conditions on input g. For performance, we evaluate both possible validity answers and marginalize before computing the loss:

$$p_\theta(\hat{a}_{\text{perf}} = \texttt{Y} \mid g) = p_\theta(\hat{a}_{\text{perf}} = \texttt{Y} \mid g, \hat{a}_{\text{val}} = \texttt{N})\, p_\theta(\hat{a}_{\text{val}} = \texttt{N} \mid g) + p_\theta(\hat{a}_{\text{perf}} = \texttt{Y} \mid g, \hat{a}_{\text{val}} = \texttt{Y})\, p_\theta(\hat{a}_{\text{val}} = \texttt{Y} \mid g).$$

For either target, $p_\theta(\hat{a}_j = \texttt{N} \mid g) = 1 - p_\theta(\hat{a}_j = \texttt{Y} \mid g)$. Marginalization removes dependence on a single sampled validity answer without assuming independence between the predictions. The prediction loss averages binary cross-entropy over the two marginal probabilities:

$$\mathcal{L}_{g,i}^{\text{BCE}}(\theta) = -\frac{1}{|\mathcal{J}|} \sum_{j \in \mathcal{J}} \Big[y_j(g) \log p_\theta(\hat{a}_j = \texttt{Y} \mid g) + (1 - y_j(g)) \log p_\theta(\hat{a}_j = \texttt{N} \mid g) \Big]. \tag{4}$$

Both the marginal probabilities and the group-derived targets are shared across the K responses for input g, so this BCE is the same for every i; we retain the index to match the per-rollout notation in Eq. 3. Unlike group-relative rewards, the BCE loss is not centered across responses, so identical incorrect predictions still receive a corrective gradient.

Code-optimization loss. For generated program z_i, we define the reward as

$$R_i^{\text{code}} = -\mathbb{1}[e_i = 0] + \mathbb{1}[e_i = 1]\, u_i. \tag{5}$$

Failed equivalence checks receive -1; verified programs receive their speedup $u_i = T(z^{(0)})/T(z_i)$, including values below 1 for slowdowns. We set $u_i = 0$ for failed programs, which require no timing measurement. The non-trivial-edit criterion used in v_i does not gate this reward. For each sampled response s_i, we normalize rewards within its group of K responses to obtain the code advantage A_i^{code}. We minimize the standard clipped GRPO loss over code tokens:

$$\mathcal{L}_{g,i}^{\text{GRPO}}(\theta) = -\frac{1}{\max(1, n_i^{\text{code}})} \sum_{k \in \mathcal{T}_i^{\text{code}}} \min\left\{ \rho_{i,k}(\theta) A_i^{\text{code}}, \bar{\rho}_{i,k}(\theta) A_i^{\text{code}} \right\}. \tag{6}$$

Here k indexes token positions in s_i, $\mathcal{T}_i^{\text{code}}$ is the set of code-token positions, and $n_i^{\text{code}} = |\mathcal{T}_i^{\text{code}}|$ is its size. The quantity $\rho_{i,k}$ is the current-to-sampling-policy probability ratio, and $\bar{\rho}_{i,k}$ is that ratio clipped to $[1-\epsilon_{\text{clip}}, 1+\epsilon_{\text{clip}}]$. The parameter $\epsilon_{\text{clip}} > 0$ is the policy-ratio clipping threshold, distinct from the runtime tolerance ϵ. Appendix A.2 gives the full definitions.

3.4 Policy-to-Data: Constructing and Selecting Future Inputs

The policy-to-data path turns accepted outputs into new instances of the code-optimization task: execution determines admission, and the shared policy's predictions rank the resulting pool.

Admit distinct, valid variants. After execution, correct programs with valid positive timing measurements are considered for the pool. A candidate must differ non-trivially in code from every previously accepted member of its family, where a family contains the original program and its accepted descendants for the same problem. Its runtime must also differ from each family member by a ratio of at least 1.1 between the larger and smaller runtimes. Candidates are processed in descending speedup order, and an accepted program becomes a future input $g' = (x, z_i^{(d+1)})$. Its speedup reference remains the original program, not its immediate parent. These family-level admission checks do not remove responses from the current rollout group's reward or prediction-label computation.

Score new inputs by prediction log-odds. For each newly admitted input g, we reuse the marginal prediction probabilities defined in Section 3.3, without generating or executing code. The input's score is the sum of their log-odds:

$$S_\theta(g) = \sum_{j \in \{\text{val},\text{perf}\}} \log \frac{p_\theta(\hat{a}_j = \texttt{Y} \mid g)}{p_\theta(\hat{a}_j = \texttt{N} \mid g)}, \tag{7}$$

S_θ is a ranking heuristic, not a joint log-probability or an estimate of the performance gain from training on an input.

Following prior work that reuses historical signals for adaptive sampling (Zheng et al., 2025; Cai et al., 2026), we avoid repeatedly reevaluating past candidates: each candidate is scored once upon admission, before the current policy update, and its score is cached for subsequent selection.

Selecting inputs for the next round. Top-ranked pool entries fill up to half of each batch, and fresh inputs fill the remaining slots, including any left when the pool is small. We call this fraction of batch slots the *pool-sampling fraction*; it is fixed at 0.5 across all pool-based methods to isolate the input-construction mechanism rather than to optimize the mixture. Fresh inputs preserve exposure to the source distribution, while generated inputs support further optimization adaptive to training.

4 Experiments

We first compare SeeThat with GRPO and alternative pool-selection methods in final runtime, correctness, and training efficiency (Section 4.2). Ablations then test which parts of the method account for the gains (Section 4.3), and we measure the cost of prediction (Section 4.4).

4.1 Experimental setup

4.1.1 Training and evaluation setup

We use Qwen3-8B and Qwen3.6-35B-A3B as backbone models. Training uses Tinker (Thinking Machines Lab, 2025) fine-tuning API, with batch size 32 code-optimization inputs, group size $K = 8$, LoRA rank 32, learning rate 4×10^{-5}, and random seeds 41, 42, and 43. Each run trains for 200 steps and final test results use the last checkpoint. Evaluation code is executed sequentially on an AWS `c7i.2xlarge` instance (8 vCPUs, 16 GB RAM) under Ubuntu.

4.1.2 Datasets and contamination control

We evaluate on two code-optimization benchmarks, PerfCodeBench and PIE. **PerfCodeBench represents systems-code optimization**, which contains performance-critical systems tasks with executable correctness and runtime checks (Jing et al., 2026). **PIE represents competitive-programming code optimization**, which contains human-written slow–fast program pairs from CodeNet submissions (Shypula et al., 2025); we use the slower program as input for the optimization task and evaluate the generated program for correctness and runtime.

In both settings, the model optimizes an existing program rather than generating from scratch. For each benchmark, we sample 100 programs for validation and 300 programs for testing from official partitions, respectively. To reduce train–test contamination, we exclude any program whose word 5-gram Jaccard similarity with a training program is at least 0.9 (Appendix A.1).

4.1.3 BASELINES

We compare four training-data regimes across both benchmarks, all using GRPO for policy optimization. *GRPO* is the original-only baseline: it trains only on the original programs and does not reuse generated rollouts as training inputs. The other three methods recursively construct equivalence-class pools with no explicit depth bound. *Random selection* samples pooled inputs uniformly, *Entropy selection* prioritizes inputs on which the current policy has the highest entropy, and SEETHAT ranks inputs by predicted training value (S_θ). As reference points, we also evaluate GPT-5.4 and GPT-5.6 Sol, each sampled three times with temperature 1 and top-$p = 0.95$.

4.1.4 EVALUATION METRICS

Evaluation uses single-step code optimization: given $(x, z^{(0)})$, the model generates one candidate $z^{(1)}$, with no output fed back for another inference step. *Correctness* is the fraction of inputs for which the generated program passes all tests in the provided suite from the benchmark; passing the tests suite does not establish functional equivalence beyond those tests.

For runtime measurement, we execute each program 10 times after one warmup run and use the mean as the runtime of a program. We report P50, P75, P90, and P95 runtime statistics across evaluation inputs. To prevent incorrect outputs from affecting the performance gain results, all reported runtimes are failure-padded: a generated program that fails the test suite is assigned the runtime of its original program rather than excluded. We evaluate learning curves every 10 training steps on the validation set and report final results on the held-out test subsets.

4.2 MAIN RESULTS

Table 1 reports final test results across models and benchmarks. SEETHAT achieves the lowest P90 and P95 runtime in all four model–benchmark settings, and the lowest runtime at all four reported percentiles in three of the four model–benchmark settings. The exception is PerfCodeBench with Qwen3.6-35B-A3B, where Random selection achieves lower P50 and P75 runtime, while SEETHAT remains substantially lower at P90 and P95. These runtime improvements do not always achieve higher correctness. Appendix A.4 further compares against GPT-5.4 and GPT-5.6 sol on the test subsets. On PIE, SEETHAT with either backbone achieves lower runtimes than both GPT models at all four percentiles. On PerfCodeBench, SEETHAT outperforms GPT-5.4 at all percentiles with either backbone, and with Qwen3.6-35B-A3B has lower P90 and P95 runtimes than GPT-5.6 sol.

Recursive reuse is already beneficial without learned selection. On PerfCodeBench with Qwen3-8B, all three pool-based methods reduce P75 runtime to less than one third of that under GRPO. SEETHAT provides the largest gains at the higher runtime percentiles: for example, its P95 runtime is 199.52 compared with 335.21 for Random selection and 1433.56 for GRPO. Figure 2 shows the corresponding evaluation curves. SEETHAT reaches low P75 runtime earlier in training in several settings, most clearly on PerfCodeBench with Qwen3-8B. In three of the four settings, it also stays below the GPT-5.6 Sol P75 reference for most late-training checkpoints, a pattern the other trained methods do not consistently achieve.

4.3 ABLATIONS

Table 2 compares SEETHAT with five variants on PerfCodeBench with Qwen3-8B. SEETHAT has lower runtime at all four percentiles and higher mean correctness than every variant. Figure 3 in Appendix A.5 shows the corresponding learning curves.

Is recursion beyond the first refinement necessary? *Depth-1 only* restricts selected variants to depth $d = 1$ while keeping prediction-based ranking and the fresh–generated mixture. Its P75 runtime rises to 204.79 ms, about $3.9\times$ that of SEETHAT and close to GRPO, suggesting that much of the benefit comes from variants beyond the first refinement. We additionally study selected-candidate depth without imposing an explicit depth limit; the maximum observed depth saturates at 5 on both benchmarks (Appendix A.6).

Original-input replay ranks pooled variants as in the full method, but replaces each selected variant $(x, z^{(d)})$ with its original input $(x, z^{(0)})$, while keeping the same pool-sampling fraction and objectives. Its P75 remains close to the full method, but P95 increases from 199.52 ms to 479.16 ms, indicating that training directly on generated program variants, rather than only revisiting the corresponding original problems, is important for the tail-runtime improvements.

Table 1: Final test performance on the 300-program test subsets, reported as mean ± one standard deviation over three independent runs. Runtimes are in milliseconds (ms); correctness is in percent. Best means are bolded, including ties. Original denotes the original input program, and Base model the checkpoint before code-optimization training. All runtimes are padded.

Method	Runtime P50 ↓	Runtime P75 ↓	Runtime P90 ↓	Runtime P95 ↓	Correctness ↑
PerfCodeBench, Qwen3-8B					
Original	81.07 ± 0.32	475.08 ± 6.68	1567.67 ± 18.57	2368.35 ± 8.95	-
Base model	73.78 ± 0.16	407.03 ± 14.85	1394.25 ± 23.39	2119.31 ± 15.01	81.22 ± 0.19
GRPO	49.35 ± 0.31	191.47 ± 1.46	707.29 ± 26.87	1433.56 ± 99.96	93.44 ± 0.19
Entropy selection	9.96 ± 0.93	57.83 ± 1.22	164.33 ± 8.91	359.94 ± 14.09	93.78 ± 0.19
Random selection	8.31 ± 0.08	53.90 ± 0.12	138.50 ± 4.08	335.21 ± 7.83	**95.44 ± 0.38**
SeeThat	**6.89 ± 0.44**	**52.22 ± 0.33**	**116.78 ± 4.33**	**199.52 ± 10.62**	94.67 ± 0.88
PerfCodeBench, Qwen3.6-35B-A3B					
Original	81.07 ± 0.32	475.08 ± 6.68	1567.67 ± 18.57	2368.35 ± 8.95	-
Base model	47.29 ± 2.17	234.58 ± 2.89	1075.83 ± 42.38	2988.22 ± 5.38	66.33 ± 1.00
GRPO	4.93 ± 0.16	39.97 ± 1.06	158.99 ± 12.74	499.01 ± 1.39	84.44 ± 0.51
Entropy selection	5.31 ± 0.03	41.40 ± 1.16	146.90 ± 0.98	986.64 ± 51.45	79.44 ± 0.19
Random selection	**3.26 ± 0.05**	**24.19 ± 0.82**	100.36 ± 4.01	212.44 ± 49.48	90.22 ± 0.19
SeeThat	4.56 ± 0.05	25.71 ± 0.42	**76.68 ± 0.77**	**135.37 ± 4.18**	**90.56 ± 0.38**
PIE, Qwen3-8B					
Original	4.56 ± 0.06	5.38 ± 0.56	6.45 ± 0.63	7.56 ± 0.41	-
Base model	4.51 ± 0.02	5.36 ± 0.50	6.71 ± 0.70	8.31 ± 0.32	72.00 ± 0.88
GRPO	1.48 ± 0.00	1.54 ± 0.01	1.80 ± 0.01	1.94 ± 0.01	77.33 ± 0.88
Entropy selection	1.48 ± 0.01	1.54 ± 0.01	1.78 ± 0.03	1.85 ± 0.01	78.56 ± 0.77
Random selection	1.50 ± 0.00	1.65 ± 0.00	1.82 ± 0.01	2.03 ± 0.11	**81.00 ± 0.67**
SeeThat	**1.19 ± 0.00**	**1.49 ± 0.00**	**1.56 ± 0.04**	**1.75 ± 0.08**	79.33 ± 0.58
PIE, Qwen3.6-35B-A3B					
Original	4.56 ± 0.06	5.38 ± 0.56	6.45 ± 0.63	7.56 ± 0.41	-
Base model	4.27 ± 0.06	4.72 ± 0.09	5.44 ± 0.19	6.23 ± 0.22	71.78 ± 2.69
GRPO	**2.76 ± 0.02**	2.88 ± 0.02	4.45 ± 0.19	5.14 ± 0.09	**88.56 ± 0.69**
Entropy selection	2.81 ± 0.04	2.90 ± 0.02	4.33 ± 0.09	5.03 ± 0.07	86.00 ± 0.67
Random selection	2.80 ± 0.02	2.88 ± 0.02	4.17 ± 0.02	4.67 ± 0.05	**88.56 ± 0.19**
SeeThat	**2.76 ± 0.02**	**2.87 ± 0.03**	**4.09 ± 0.01**	**4.62 ± 0.03**	81.33 ± 1.86

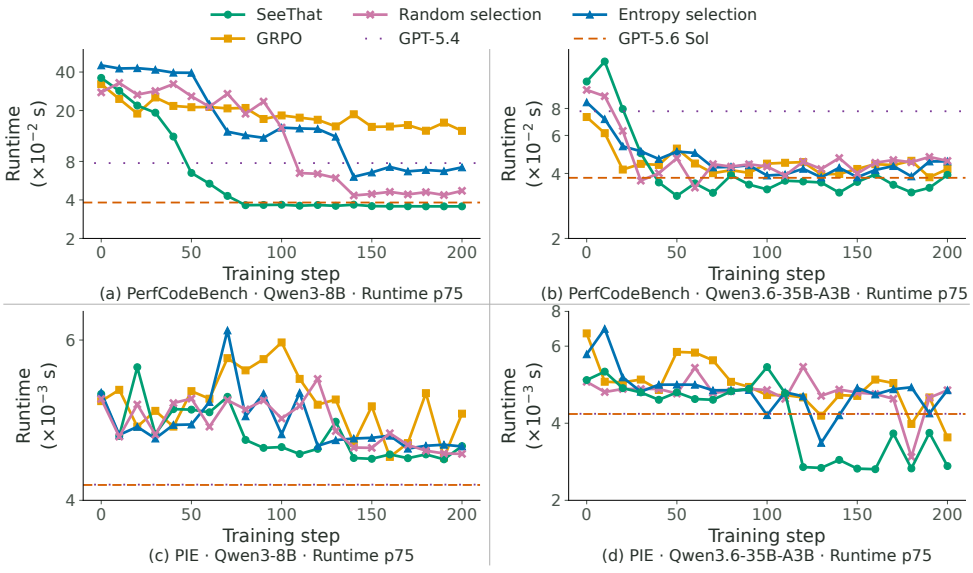


Figure 2: **P75 runtime over training.** Failure-padded P75 runtime on the 100 programs validation subset for Qwen3-8B and Qwen3.6-35B-A3B on PerfCodeBench (a–b) and PIE (c–d). Numbers averaged over three runs. Horizontal lines show GPT-5.4 and GPT-5.6 Sol for reference; they overlap in (c–d). The y-axis is logarithmic and lower is better. Final test results are reported in Table 1.

Does using prediction scores improve selection? Appendix A.7 reports prediction AUROC over training on PerfCodeBench with Qwen3-8B for (a) validity contrast and (b) performance contrast. Both predictions generally achieve above-chance discrimination, supporting their use as signals for input selection.

Table 2: Ablation results on PerfCodeBench with Qwen3-8B (mean ± one standard deviation) on the test subset. Runtimes are in milliseconds (ms); correctness is in percent. SEETHAT (full) repeats the main-table result. Best means are bolded.

Method	Runtime P50 ↓	Runtime P75 ↓	Runtime P90 ↓	Runtime P95 ↓	Correctness ↑
Random selection (prediction retained)	9.19 ± 0.32	57.09 ± 0.41	182.25 ± 3.92	551.02 ± 12.71	93.78 ± 0.19
Shared credit	8.90 ± 0.12	62.60 ± 0.44	169.39 ± 7.77	335.22 ± 11.55	93.00 ± 0.00
Original-input replay	7.54 ± 0.04	56.74 ± 0.45	165.58 ± 5.74	479.16 ± 16.11	93.44 ± 0.38
Depth-1 only	39.07 ± 0.91	204.79 ± 10.87	805.16 ± 12.25	1681.23 ± 100.77	93.44 ± 0.38
DAPO (sampling only)	48.81 ± 0.46	192.85 ± 2.61	663.66 ± 8.98	1173.75 ± 2.10	91.00 ± 0.00
SEETHAT (full)	$\mathbf{6.89 \pm 0.44}$	$\mathbf{52.22 \pm 0.33}$	$\mathbf{116.78 \pm 4.33}$	$\mathbf{199.52 \pm 10.62}$	$\mathbf{94.67 \pm 0.88}$

Table 3: Matched-step prediction-cost comparison for Qwen3-8B on PerfCodeBench ($K = 8$, mean ± standard deviation across three seeds). Token counts are in millions (M); costs are in USD using Tinker pricing as of September 17, 2026. Overhead is computed from the mean values.

Configuration	Input tokens (M)	Output tokens (M)	Cost (USD)
Without prediction	16.65 ± 2.92	13.71 ± 2.89	11.48 ± 2.27
With prediction	18.48 ± 2.96	13.82 ± 2.89	11.90 ± 2.27
Overhead (%)	10.98	0.76	3.65

Random selection (prediction retained) keeps the prediction task and candidate scoring unchanged, but does not actually use the scores when selecting pool entries; it instead samples uniformly from $\mathcal{C}_t$. Its P95 runtime increases to 551.02 ms, $2.8\times$ that of SEETHAT. Moreover, it is worse than Random selection without the prediction objective (335.21 ms), indicating that joint prediction interferes with code optimization. So SEETHAT's improvement does not come from prediction alone: using predictions for input selection compensates for this interference and improves substantially over both random-selection variants. *DAPO (sampling only)* adopts only DAPO's dynamic-sampling mechanism (Yu et al., 2025): fresh inputs whose rollout groups have zero advantage variance are replaced. This filter-after-execution strategy remains close to GRPO (P75 192.85 ms), substantially behind methods that select recursive candidates before allocating their rollout budget.

Does region-specific supervision matter? *Shared credit* replaces the separate token-region objectives with a shared learning signal applied to prediction and code tokens. Its P75/P95 runtimes increase from 52.22 ms/199.52 ms to 62.60 ms/335.22 ms, respectively, while correctness drops from 94.67% to 93.00%. This supports region-specific supervision for prediction and code generation.

4.4 COST OF PREDICTION

SEETHAT adds overhead from scoring newly admitted candidates and from the prediction fields generated in each training rollout. To isolate this overhead, Table 3 compares Qwen3-8B runs on PerfCodeBench with and without prediction at matched training steps ($K = 8$, three seeds), so the comparison reflects prediction overhead alone rather than any benefit from faster learning (Figure 2 (a)). Prediction increases the mean cost per run by \$0.42 (3.65%), from \$11.48 to \$11.90. Most of the additional usage comes from input tokens (+10.98%) for candidate-scoring, while output tokens grow by only 0.76% because each rollout adds only a short prediction. The small increase in output tokens keeps the overall monetary overhead modest despite the higher price.

5 CONCLUSION

We study code optimization as recursive policy–data co-evolution, where verified outputs become future training inputs and input selection shapes the policy's subsequent learning opportunities. Across PerfCodeBench and PIE with dense and mixture-of-experts models, it achieves the lowest runtime at all four reported percentiles in three of four settings, and the lowest P90/P95 in all four among the compared training methods. It also generally reaches lower P75 runtimes earlier, with prediction adding 3.65% to training cost at matched step counts. Ablations support both recursive reuse and prediction-based selection. These findings support learned input selection for allocating a finite budget as training data expands.

AI USE STATEMENT

Generative AI tools were used to assist with code generation for the research implementation and to polish the manuscript's language, clarity, and organization. Responsibility for reviewing and testing the code and verifying the text, technical claims, and references rests with the authors, who take full responsibility for the correctness, originality, and integrity of the final work, including all AI-assisted content.

ETHICS STATEMENT

This work studies automated code optimization. Generated program variants may contain errors or security vulnerabilities, and passing a finite test suite does not guarantee functional equivalence or safe deployment. Such programs should undergo independent validation and appropriate security review before use. The use and redistribution of code and datasets should respect their applicable licenses and usage restrictions.

REPRODUCIBILITY STATEMENT

All source datasets are publicly available. The dataset construction and contamination-control protocols are described in Section 4.1.2 and Appendix A.1. An anonymous code repository is available at `https://anonymous.4open.science/r/SeeThat-B0F4/`. Sections 3 and 4 describe the method and experimental protocol, with additional implementation details in Appendix A.

REFERENCES

Yuzhu Cai, Zexi Liu, Xinyu Zhu, Cheng Wang, Yanfeng Wang, and Siheng Chen. Acegrpo: Adaptive curriculum enhanced group relative policy optimization for autonomous machine learning engineering, 2026. URL https://arxiv.org/abs/2602.07906.

Pierre Chambon, Kunhao Zheng, Juliette Decugis, Benoit Sagot, and Gabriel Synnaeve. Reinforcement learning for code optimization, 2026. URL https://arxiv.org/abs/2607.25970.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=KuPixIqPiq.

Mingzhe Du, Anh Tuan Luu, Yue Liu, Yuhao QING, Dong HUANG, Xinyi He, Qian Liu, Zejun MA, and See-Kiong Ng. Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=At8OUlyTOu.

Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael R. Lyu. Search-based llms for code optimization. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pp. 578–590. IEEE Press, 2025. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00021. URL https://doi.org/10.1109/ICSE55347.2025.00021.

Zhaolin Gao, Joongwon Kim, Wen Sun, Thorsten Joachims, Sid Wang, Richard Yuanzhe Pang, and Liang Tan. Prompt curriculum learning for efficient LLM post-training. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=zqOCacBD3P.

Irving John Good. Speculations concerning the first ultraintelligent machine. volume 6, pp. 31-88. 1965. doi: https://doi.org/10.1016/S0065-2458(08)60418-0. URL https://www.sciencedirect.com/science/article/pii/S0065245808604180.

Alex Graves, Marc G. Bellemare, Jacob Menick, Rémi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *International Conference on Machine Learning (ICML)*, pp. 1311-1320, 2017. URL http://proceedings.mlr.press/v70/graves17a.html.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning LLM from zero data. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=96apU6YzSO.

Dong HUANG, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao QING, Heming Cui, Zhijiang Guo, and Jie Zhang. Effilearner: Enhancing efficiency of generated code via self-optimization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=KhwOuB0fs9.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IkmD3fKBPQ.

Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. Reveal: Self-evolving code agents via reliable self-verification. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=q56ZI1Co43.

Huihao Jing, Wenbin Hu, Haochen Shi, Hanyu Yang, Sirui Zhang, Shaojin Chen, Haoran Li, and Yangqiu Song. Perfcodebench: Benchmarking llms for system-level high-performance code optimization. *CoRR*, abs/2605.15222, May 2026. URL https://doi.org/10.48550/arXiv.2605.15222.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=S37hOerQLB.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=y0GJXRungR.

Yun Qu, Qi Wang, Yixiu Mao, Heming Zou, Yuhang Jiang, Weijie Liu, Clive Bai, Kai Yang, Yangkun Chen, Saiyong Yang, and Xiangyang Ji. Small generalizable prompt predictive models can steer efficient RL post-training of large reasoning models. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=vCHOBek0dv.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05952.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=vAElhFcKW6.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Automated high-level code optimization for warehouse performance. *IEEE Micro*, 45(4):60-71, 2025. doi: 10.1109/MM.2025.3590033.

Thinking Machines Lab. Tinker. https://thinkingmachines.ai/tinker/, 2025.

Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. Supercoder: Assembly program superoptimization with large language models, 2026. URL https://openreview.net/forum?id=30iarHLvCS.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=hH36JeQZDaO.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=2a36EMSSTp.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STar: Bootstrapping reasoning with reasoning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_3ELRdg2sgI.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=46Zgqo4QIU.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=neZSGqhxDa.

Haizhong Zheng, Yang Zhou, Brian R. Bartoldson, Bhavya Kailkhura, Fan Lai, Jiawei Zhao, and Beidi Chen. Act only when it pays: Efficient reinforcement learning for LLM reasoning via selective rollouts. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=x5lITYXmW2.

A Additional Experimental Details

A.1 Dataset preparation

Splits and sampling. For both PerfCodeBench and PIE, we follow the benchmark's official train/evaluation/test partitions. Training inputs come from the training partition. From the evaluation and test partitions, we sample 100 evaluation programs and 300 test programs uniformly at random with a fixed seed (42).

Near-duplicate filtering. To limit leakage between training and held-out data, we remove any evaluation or test program whose word 5-gram Jaccard similarity with some training program is at least 0.9, and sample the held-out subsets from the remaining programs. This dataset-level filter does not rule out prior exposure through base-model pretraining.

PIE-specific processing. PIE pairs a slower and a faster C++ submission to the same CodeNet problem; we use the slower program as the optimization input. Because a slower program can appear in several pairs, we deduplicate programs by their problem and submission identifiers within each partition, and discard the data if the same identifier maps to conflicting code or test selections. Each program is evaluated on the test cases specified for it by PIE, or on all complete test cases of its problem when none are specified.

A.2 GRPO loss for code optimization

For the code-optimization rewards defined in Section 3.3, we compute a group-relative advantage from all K responses to input g:

$$\overline{R}_g = \frac{1}{K}\sum_{i=1}^{K} R_i^{\text{code}}, \qquad \sigma_g = \sqrt{\frac{1}{K}\sum_{i=1}^{K}(R_i^{\text{code}} - \overline{R}_g)^2}, \qquad A_i^{\text{code}} = \frac{R_i^{\text{code}} - \overline{R}_g}{\sigma_g + \delta},$$

Here $\overline{R}_g$ and σ_g are the mean and population standard deviation of the group's rewards, and $\delta > 0$ ensures numerical stability. A group of identical rewards has zero code advantage. Let s_i be the full sampled response, $\mathcal{T}_i^{\text{code}}$ its code-token positions, and $n_i^{\text{code}} = |\mathcal{T}_i^{\text{code}}|$. Write $s_{i,k}$ for token k of response i and $s_{i,<k}$ for all its preceding tokens. For the sampling policy $p_{\theta_{\text{old}}}$, define the token-level importance ratio

$$\rho_{i,k}(\theta) = \frac{p_\theta(s_{i,k} \mid g, s_{i,<k})}{p_{\theta_{\text{old}}}(s_{i,k} \mid g, s_{i,<k})}.$$

The per-response clipped GRPO loss used in Eq. 3 is

$$\mathcal{L}_{g,i}^{\text{GRPO}}(\theta) = -\frac{1}{\max(1, n_i^{\text{code}})} \sum_{k \in \mathcal{T}_i^{\text{code}}} \min\Big\{ \rho_{i,k}(\theta) A_i^{\text{code}}, \; \text{clip}(\rho_{i,k}(\theta), 1-\epsilon_{\text{clip}}, 1+\epsilon_{\text{clip}})\, A_i^{\text{code}} \Big\}. \tag{8}$$

Here $\epsilon_{\text{clip}} > 0$ controls policy-ratio clipping and is distinct from the runtime tolerance ϵ. The operator $\text{clip}(a, l, u) = \min(\max(a, l), u)$ restricts a to $[l, u]$. Rewards, advantages, and sampling-policy probabilities are held fixed during optimization. If no code tokens are present, the sum is empty and contributes zero.

A.3 Token-count scaling

For response i to input g, let $n_{g,i}^{\text{code}}$ and $n_{g,i}^{\text{pred}}$ count the code and prediction tokens included in the loss. Only the two single-token Y/N answer positions contribute to the prediction count; prompt and delimiter positions are excluded. The two conditional evaluations used to marginalize the performance prediction contribute to one BCE target, not two additional prediction targets. Thus $n_{g,i}^{\text{code}}$ is n_i^{code} with its input index made explicit, and $n_{g,i}^{\text{pred}} = |\mathcal{J}| = 2$ when both prediction targets

are present. For nonempty code and prediction regions, each code token has weight 1, while each prediction token has relative weight

$$\alpha_{g,i} = \frac{r}{1-r}\frac{n_{g,i}^{\text{code}}}{n_{g,i}^{\text{pred}}}.$$

This depends only on token counts, not reward or advantage magnitudes, and ensures

$$\frac{\alpha_{g,i} n_{g,i}^{\text{pred}}}{n_{g,i}^{\text{code}} + \alpha_{g,i} n_{g,i}^{\text{pred}}} = r.$$

Because the two per-response losses are token averages, normalizing the weighted token-loss sum by the total token weight yields

$$\frac{n_{g,i}^{\text{code}} \mathcal{L}_{g,i}^{\text{GRPO}}(\theta) + \alpha_{g,i} n_{g,i}^{\text{pred}} \mathcal{L}_{g,i}^{\text{BCE}}(\theta)}{n_{g,i}^{\text{code}} + \alpha_{g,i} n_{g,i}^{\text{pred}}} = (1-r)\mathcal{L}_{g,i}^{\text{GRPO}}(\theta) + r\mathcal{L}_{g,i}^{\text{BCE}}(\theta),$$

which gives the compact batch objective in Eq. 3. The fraction r controls token weights, not loss magnitudes or gradient norms.

A.4 Comparison with proprietary models

Table 4 compares SeeThat with GPT-5.4 and GPT-5.6 Sol on the test subsets. The two GPT models are each sampled three times with temperature 1 and top-p 0.95; their entries are mean $\pm$ one standard deviation over the three samples. We repeat each benchmark-level GPT result in both backbone settings to match the 2×2 organization of Table 1; the other rows are copied from that table.

A.5 Ablation learning curves

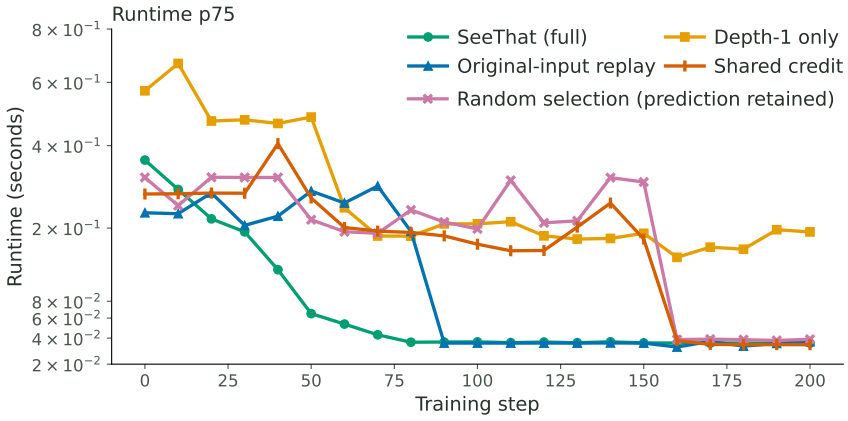


Figure 3: **P75 runtime over training for the ablation variants.** We compare SeeThat (full), Random selection (prediction retained), Shared credit, Original-input replay, and Depth-1 only. The horizontal axis shows training steps; the vertical axis shows runtime in seconds on a square-root scale. Lower runtime is better.

A.6 Selected-candidate depth

Figure 4 shows the maximum depth among candidates selected by SeeThat at each training step. Although no explicit depth limit is imposed, the maximum observed depth reaches 5 for both PerfCodeBench and PIE with Qwen3-8B.

Table 4: Final test performance of GPT-5.4, GPT-5.6 sol, and the Qwen models before training (Base model), after GRPO, and after SeeThat. Entries are mean ± one standard deviation; runtimes are padded and in milliseconds (ms), and correctness is in percent. Best means within each model–benchmark setting are bolded, including ties.

Method	Runtime P50 ↓	Runtime P75 ↓	Runtime P90 ↓	Runtime P95 ↓	Correctness ↑
PerfCodeBench, Qwen3-8B					
Original	81.07 ± 0.32	475.08 ± 6.68	1567.67 ± 18.57	2368.35 ± 8.95	-
Base model	73.78 ± 0.16	407.03 ± 14.85	1394.25 ± 23.39	2119.31 ± 15.01	81.22 ± 0.19
GRPO	49.35 ± 0.31	191.47 ± 1.46	707.29 ± 26.87	1433.56 ± 99.96	93.44 ± 0.19
GPT-5.4	10.53 ± 0.87	70.85 ± 2.66	273.15 ± 13.59	605.91 ± 63.17	93.33 ± 1.33
GPT-5.6 sol	$\mathbf{3.70 \pm 0.10}$	$\mathbf{24.36 \pm 1.36}$	$\mathbf{87.48 \pm 2.51}$	$\mathbf{174.74 \pm 12.31}$	$\mathbf{96.44 \pm 0.51}$
SeeThat	6.89 ± 0.44	52.22 ± 0.33	116.78 ± 4.33	199.52 ± 10.62	94.67 ± 0.88
PerfCodeBench, Qwen3.6-35B-A3B					
Original	81.07 ± 0.32	475.08 ± 6.68	1567.67 ± 18.57	2368.35 ± 8.95	-
Base model	47.29 ± 2.17	234.58 ± 2.89	1075.83 ± 42.38	2988.22 ± 5.38	66.33 ± 1.00
GRPO	4.93 ± 0.16	39.97 ± 1.06	158.99 ± 12.74	499.01 ± 1.39	84.44 ± 0.51
GPT-5.4	10.53 ± 0.87	70.85 ± 2.66	273.15 ± 13.59	605.91 ± 63.17	93.33 ± 1.33
GPT-5.6 sol	$\mathbf{3.70 \pm 0.10}$	$\mathbf{24.36 \pm 1.36}$	87.48 ± 2.51	174.74 ± 12.31	$\mathbf{96.44 \pm 0.51}$
SeeThat	4.56 ± 0.05	25.71 ± 0.42	$\mathbf{76.68 \pm 0.77}$	$\mathbf{135.37 \pm 4.18}$	90.56 ± 0.38
PIE, Qwen3-8B					
Original	4.56 ± 0.06	5.38 ± 0.56	6.45 ± 0.63	7.56 ± 0.41	-
Base model	4.51 ± 0.02	5.36 ± 0.50	6.71 ± 0.70	8.31 ± 0.32	72.00 ± 0.88
GRPO	1.48 ± 0.00	1.54 ± 0.01	1.80 ± 0.01	1.94 ± 0.01	77.33 ± 0.88
GPT-5.4	4.10 ± 0.02	4.18 ± 0.05	4.57 ± 0.52	5.04 ± 0.61	$\mathbf{92.44 \pm 0.39}$
GPT-5.6 sol	4.11 ± 0.02	4.17 ± 0.04	4.60 ± 0.60	5.05 ± 0.59	87.56 ± 0.77
SeeThat	$\mathbf{1.19 \pm 0.00}$	$\mathbf{1.49 \pm 0.00}$	$\mathbf{1.56 \pm 0.04}$	$\mathbf{1.75 \pm 0.08}$	79.33 ± 0.58
PIE, Qwen3.6-35B-A3B					
Original	4.56 ± 0.06	5.38 ± 0.56	6.45 ± 0.63	7.56 ± 0.41	-
Base model	4.27 ± 0.06	4.72 ± 0.09	5.44 ± 0.19	6.23 ± 0.22	71.78 ± 2.69
GRPO	$\mathbf{2.76 \pm 0.02}$	2.88 ± 0.02	4.45 ± 0.19	5.14 ± 0.09	88.56 ± 0.69
GPT-5.4	4.10 ± 0.02	4.18 ± 0.05	4.57 ± 0.52	5.04 ± 0.61	$\mathbf{92.44 \pm 0.39}$
GPT-5.6 sol	4.11 ± 0.02	4.17 ± 0.04	4.60 ± 0.60	5.05 ± 0.59	87.56 ± 0.77
SeeThat	$\mathbf{2.76 \pm 0.02}$	$\mathbf{2.87 \pm 0.03}$	$\mathbf{4.09 \pm 0.01}$	$\mathbf{4.62 \pm 0.03}$	81.33 ± 1.86

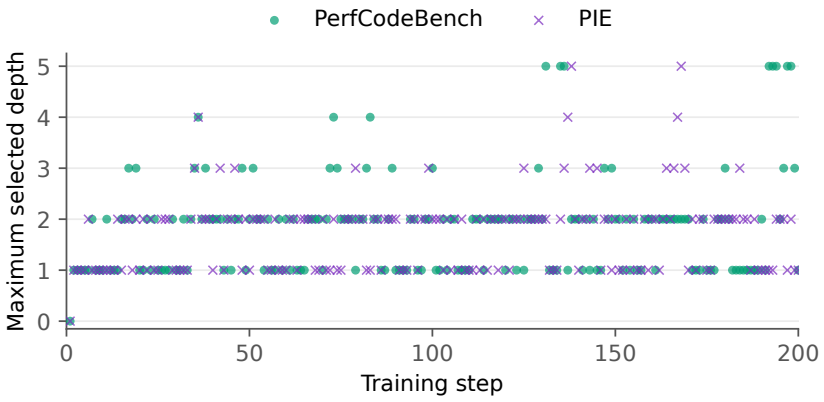
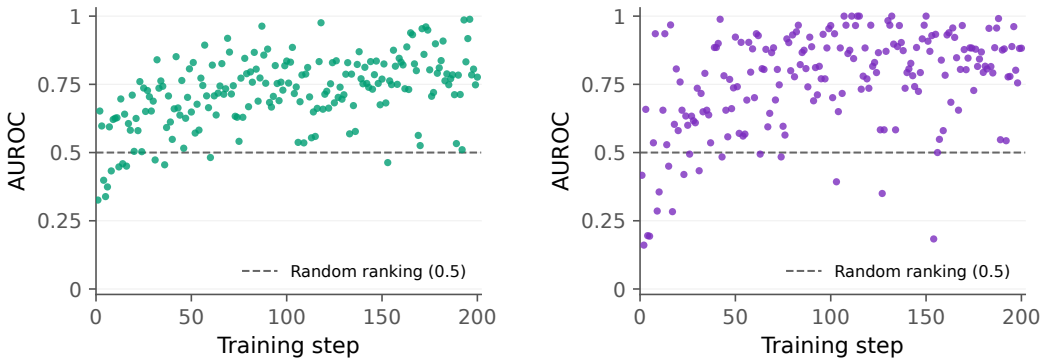


Figure 4: **Maximum selected depth over training.** Results are shown for SeeThat with Qwen3-8B on PerfCodeBench and PIE.

A.7 PREDICTION QUALITY

Figure 5 reports prediction AUROC of the two training-value predictions over training on PerfCodeBench with Qwen3-8B. We compute AUROCs for validity and performance contrast using the corresponding prediction log-odds for each input. At each training step, we omit AUROC when the evaluated inputs do not contain both positive and negative targets. Both predictors achieve AUROC above random ranking for most of training, with performance contrast generally exhibiting stronger ranking quality.



(a) Validity contrast

(b) Performance contrast

Figure 5: **Prediction AUROC over training.** AUROC for predicting whether a rollout group contains both valid non-trivial refinements and other attempts (left), or both near-baseline and substantially faster programs among valid non-trivial refinements (right). Higher is better; 0.5 corresponds to random ranking.