GRADED EXECUTION FEEDBACK FOR CO-EVOLVING CODE AND TEST GENERATORS

Anonymous authors
Paper under double-blind review

ABSTRACT

Execution feedback for code generation is often represented as a binary pass/fail signal. This is natural for unit-test style correctness checking, but it discards useful information in tasks where multiple outputs are feasible and differ in solution quality. We study code generation for graded programming tasks, including combinatorial optimization and sequential decision-making, where programs are evaluated by normalized objective or rollout scores. We introduce Continuous Co-Evolution (CCE), a framework that trains a shared language model under both solution-program and test-generation prompts. Candidate programs are rewarded by their continuous performance on reference tests, while generated tests are rewarded when the program rankings they induce agree with rankings from reference tests. This ranking-based objective enables generated tests to provide informative feedback while preserving graded differences among candidate programs. Experiments on Traveling Salesman Problem (TSP), Set Cover, Parallel Machine Scheduling (PMS), and Game 2048 show that continuous test-feedback co-evolution improves over coder-only training and binary-feedback co-evolution baselines.

1 INTRODUCTION

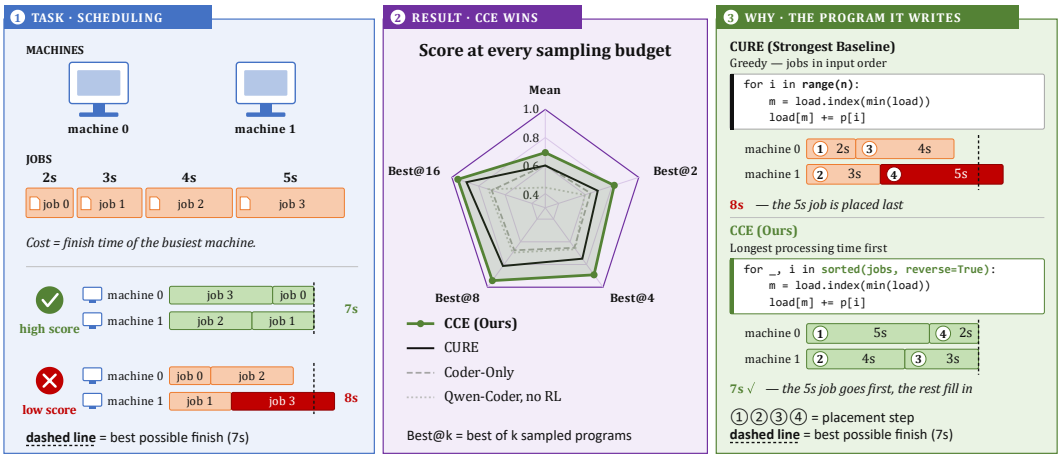


Figure 1: Illustration of the PMS task and learned program behavior. **Left:** Two feasible schedules have different makespans and therefore different quality scores. **Center:** Evaluation across program-sampling budgets. **Right:** On the illustrated instance, the CURE (Wang et al., 2025) checkpoint schedules jobs in input order, whereas CCE schedules longer jobs first and achieves a shorter makespan.

Modern code-generation training increasingly uses execution-based feedback from compilers, interpreters, unit tests, and task-specific evaluators (Le et al., 2022; Liu et al., 2023; Dou et al., 2024; Gehring et al., 2025). On conventional programming benchmarks, this feedback is commonly reduced to whether a generated program passes a set of tests. However, many code-generation tasks naturally provide *quantitative* feedback: for example, programs may differ in latency, optimization objective values, or scores achieved in computer games (Romera-Paredes et al., 2024; Liu et al.,

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.

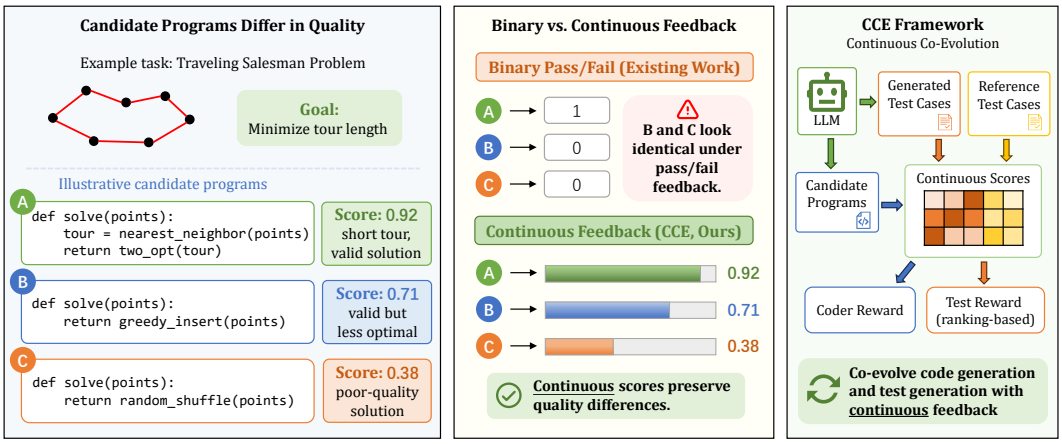


Figure 2: CCE preserves graded differences among candidate programs with *continuous* test feedback, while binary pass/fail signals collapse solution-quality differences, losing valuable information. A shared LLM co-evolves code generation and test generation using reference-based coder rewards and ranking-based test rewards.

2024; Ye et al., 2024). In such settings, reducing execution outcomes to pass/fail signals can discard meaningful differences in program quality.

Generated tests provide a scalable way to obtain execution feedback: they can expose failure modes, verify or rerank candidate programs, and provide training signals (Chen et al., 2023; Zeng et al., 2025; Ma et al., 2025; Wang et al., 2025). Yet existing generated-test methods typically treat each program-test interaction as pass or fail, even when execution produces a quantitative measure of solution quality. This motivates a natural question: how can program and test generators be co-trained to exploit graded, rather than binary, execution feedback?

This question arises naturally in (i) combinatorial optimization, where valid solutions can have different objective values; (ii) heuristic or approximation settings, where optimal solutions may be unavailable or computationally expensive to verify; and (iii) sequential decision-making, where program quality is reflected by multi-turn rollout performance. In all these cases, binary feedback collapses meaningful quality differences among candidate programs. We study this problem through representative graded programming tasks, including the Traveling Salesman Problem (TSP), Set Cover, Parallel Machine Scheduling (PMS), and Game 2048. As a concrete example, Figure 1 illustrates how feasible PMS schedules can differ in makespan (i.e., the time cost of the busiest machine) and how CCE's continuous feedback can distinguish the scheduling strategies that produce them.

To address this challenge, we propose Continuous Co-Evolution (CCE), a framework for jointly training program and test generation under *continuous* execution feedback. As illustrated in Figure 2, each test induces a normalized quality score for every candidate program. Reference tests reward programs according to solution quality, while generated tests are rewarded for preserving the reference-induced reliability ordering among programs. This formulation makes generated tests learned surrogate evaluators for graded tasks, while recovering binary coder-tester co-evolution when interaction scores are restricted to pass/fail outcomes. Beyond providing finer-grained rewards, continuous score vectors also retain distributional information that may support uncertainty-aware program selection, an exploratory direction we investigate in Appendix J and Appendix J.4.

Our main contributions are as follows:

- We formulate graded program-test co-evolution, in which candidate programs and generated tests interact through normalized quality scores rather than only binary correctness.
- We derive an efficiently computable, reliability-based pairwise reward that trains generated tests to preserve the program ordering induced by reference tests.
- We connect continuous and binary co-evolution formally: when interaction scores are binary, our pairwise objective naturally reduces to a binary unit-tester reward.

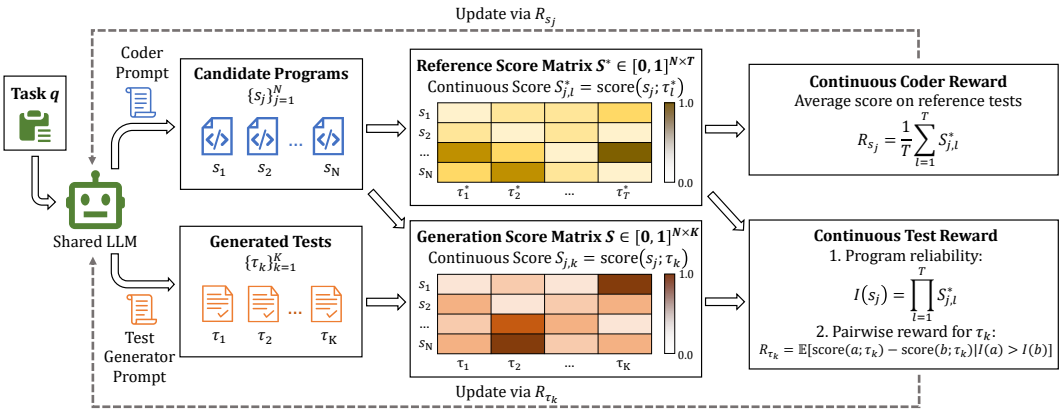


Figure 3: Overview of CCE framework. A shared LLM generates candidate programs and tests under different role prompts. Executing programs on reference tests produces the reference score matrix S^*, which is used to compute coder rewards and define program reliability. Executing programs on generated tests produces the generation score matrix S. The comparison of S and S^* helps evaluate the generated tests' quality (details in Sec. 3.3).

- Experiments on graded optimization and decision-making tasks show improvements over binary co-evolution, coder-only training, and direct inference, with the largest benefits when valid programs exhibit meaningful quality differences.

2 Related Work

Reinforcement Learning for Code Generation. Reinforcement learning (RL) has become an effective approach for improving LLM code generation, leveraging execution-based signals from compilers, interpreters, unit tests, and task-specific evaluators (Liu et al., 2023; Dou et al., 2024; Jin et al., 2025; Jiang et al., 2026). Many existing methods ultimately rely on discrete program-test outcomes, such as whether a program passes a test or compiles successfully (Le et al., 2022; Liu et al., 2023; Dou et al., 2024; Gehring et al., 2025). While these outcomes can be aggregated or transformed into richer rewards, the underlying program-test interaction remains correctness-based rather than graded by task quality.

Recent methods further densify coder-side supervision through execution-semantics alignment, step-level process supervision, or weighted partial unit-test success (Jiang et al., 2026; Li et al., 2025b; Wang et al., 2026). However, they derive supervision from fixed tests, execution traces, or process verifiers. In contrast, our focus is on jointly training program and test generators, where program-test interactions provide continuous, graded feedback for co-evolution.

LLM-Based Algorithm and Heuristic Discovery. LLMs have also been coupled with automatic evaluators to search directly for executable algorithms and heuristics. FunSearch combines a frozen language model with evolutionary program search and a task-specific evaluator, producing new mathematical constructions and online bin-packing heuristics (Romera-Paredes et al., 2024). EoH evolves both natural-language heuristic ideas and their code implementations (Liu et al., 2024), while ReEvo uses reflective evolution to search across heuristic spaces for combinatorial optimization (Ye et al., 2024). These works demonstrate that quantitative task objectives can provide effective feedback for program discovery. However, they use a fixed task evaluator and do not jointly train a test generator from interactions with candidate programs.

Unit-Test-Based Code Verification and Co-Evolution. Generated unit tests are widely used for execution-based verification, candidate selection, and training in code generation (Chen et al., 2023; Ma et al., 2025; Zeng et al., 2025). They have also been incorporated into coding pipelines for consistency-based selection, debugging, and iterative refinement (Huang et al., 2024; Ridnik et al., 2024; Li et al., 2025a). CURE extends generated testing to training-time co-evolution by jointly training code and unit-test generation, using program-test execution outcomes to couple and adapt both roles (Wang et al., 2025). However, its program-test interactions are binary pass/fail signals,

which do not capture solution-quality differences when execution yields graded objectives, heuristic scores, or rollout returns.

In contrast, our work models interactions through continuous task-defined scores and trains generated tests to preserve the program ordering induced by reference executions. These score vectors also retain distributional information that may support uncertainty-aware candidate selection, as explored in Appendix J and Appendix J.4.

3 METHOD

In this section, we formalize the problem setup, describe our proposed CCE framework with continuous rewards for the coder and reliability-based rewards for generated tests, and show how our CCE recovers CURE in the binary setting. Figure 3 shows an overview of our CCE.

3.1 PROBLEM SETUP AND METHOD OVERVIEW

Problem Setup. We consider a programming task $q \sim \mathcal{D}$, where the goal is to generate a program that produces high-quality solutions under a task-specific objective. We use a single shared language-model policy π_θ for both code generation and test-case generation, where θ denotes the trainable parameters of the shared model. The two roles are distinguished by role-specific prompts. Given a task q, the shared policy samples N candidate programs s_j under the coder prompt p_{code}:

$$\{s_j\}_{j=1}^{N} \sim \pi_\theta(\cdot \mid q, p_{\text{code}}),$$

and samples K task-conditioned test cases τ_k under the test-generator prompt p_{test}:

$$\{\tau_k\}_{k=1}^{K} \sim \pi_\theta(\cdot \mid q, p_{\text{test}}).$$

Each generated test τ_k specifies a problem instance or input case (e.g., a specific map for the Traveling Salesman Problem) on which candidate programs can be executed. Our generated tests induce continuous feedback: executing a candidate program s_j on τ_k produces a task-specific solution, whose cost/utility (e.g., distance traveled in the Traveling Salesman Problem) is normalized into a continuous score

$$\text{score}(s_j; \tau_k) \in [0, 1],$$

where larger values indicate better task performance on the generated test. Executing all candidate programs on all *generated* tests produces a continuous *generation score matrix*

$$S \in [0, 1]^{N \times K}, \qquad S_{j,k} = \text{score}(s_j; \tau_k).$$

For each task q, we also use a set of T reference (ground-truth) tests, denoted by $\{\tau_l^*\}_{l=1}^{T}$. Executing candidate programs on these *reference* tests produces a continuous *reference score matrix*

$$S^* \in [0, 1]^{N \times T}, \qquad S^*_{j,l} = \text{score}(s_j; \tau_l^*).$$

This reference score matrix S^* is used to compute coder rewards and to estimate the reliability ordering of candidate programs. The comparison of S and S^* helps evaluate the generated tests' quality (details in Sec. 3.3).

Method Overview. Figure 3 shows an overview of our CCE framework. A shared LLM generates candidate programs and test cases under different role prompts; the resulting *continuous* score matrices are used to construct rewards for the coder and the test-case generator.

3.2 CONTINUOUS REWARDS FOR THE CODER

Given the reference score matrix S^*, we define the coder reward by aggregating the scores obtained by each candidate program s_j on reference tests $\{\tau_l^*\}_{l=1}^{T}$. For a generated program s_j, its reward is

$$R_{s_j} = \frac{1}{T} \sum\nolimits_{l=1}^{T} S^*_{j,l} = \frac{1}{T} \sum\nolimits_{l=1}^{T} \text{score}(s_j; \tau_l^*). \tag{1}$$

This continuous reward directly optimizes the coder for performance on reference task instances while preserving differences among valid but suboptimal solutions.

The key requirement here is that task-specific outcomes are mapped to a normalized score in $[0, 1]$, with larger values indicating better solution quality. For maximization tasks with an exact optimal value $V^*(\tau)$, a candidate output with value $V(s_j; \tau)$ can be scored as $\text{score}(s_j; \tau) = \frac{V(s_j;\tau)}{V^*(\tau)}$. For minimization tasks with an exact optimal cost $C^*(\tau)$, a candidate output with cost $C(s_j; \tau)$ can be scored as $\text{score}(s_j; \tau) = \frac{C^*(\tau)}{C(s_j;\tau)}$. Invalid or infeasible outputs are assigned score 0, and scores are clipped to $[0, 1]$. For tasks without an instance-level optimum, such as Game 2048, we use a fixed task-specific reference procedure to construct bounded training rewards, while reporting raw task metrics for final evaluation.

3.3 Reliability Reward for Generated Tests

We next define the reward for each generated test. The purpose of a generated test is to provide scores that rank candidate programs consistently with their performance on reference tests. To formalize this, we define a reliability score for each candidate program s_j using the reference score matrix S^* from the T reference tests. Specifically,

$$I(s_j) = \prod_{l=1}^{T} S^*_{j,l}.$$

This multiplicative form assigns high reliability only to programs that perform well *across all* reference tests. If any reference test assigns a low score to s_j, then $I(s_j)$ is correspondingly reduced. In the implementation, we compute this quantity in the log domain for numerical stability (more details in Appendix D).

A good generated test should preserve the ordering induced by this reliability measure. That is, if a candidate program a is more reliable than another candidate program b (i.e., $I(a) > I(b)$), the *generated* test τ_k should assign a higher score for a than for b. We therefore define the reward for the generated test τ_k as the expected score difference over reliability-ordered pairs:

$$R_{\tau_k} = \mathbb{E}\left[\text{score}(a; \tau_k) - \text{score}(b; \tau_k) | I(a) > I(b)\right].$$

This pairwise formulation rewards discriminative tests: a generated test receives higher reward when its induced scores agree with the reliability ordering of candidate programs, and lower reward when it reverses that ordering. Thus, the objective encourages ranking consistency with reference-test performance rather than uniformly high scores.

In practice, we estimate this expectation from the sampled candidate programs for each task. For clarity, first suppose that their reliabilities are distinct. Let $\sigma : [N] \to [N]$ sort the candidates by increasing reliability, so that

$$I(s_{\sigma(1)}) < I(s_{\sigma(2)}) < \cdots < I(s_{\sigma(N)}).$$

The empirical reward for generated test τ_k is then the average score difference over all ordered pairs:

$$\widehat{R}_{\tau_k} = \frac{2}{N(N-1)} \sum\nolimits_{1 \le i < j \le N} \left(S_{\sigma(j),k} - S_{\sigma(i),k}\right). \tag{2}$$

The tie-aware definition used in the implementation is given in Appendix D.

Proposition 3.1 (Efficient Computation of the Pairwise Reward). *After sorting candidates by reliability, the empirical generated-test reward can be computed in linear time in N, including when reliability ties are present. When all candidate reliabilities are distinct, the reward in Eqn. 2 admits the closed form*

$$\widehat{R}_{\tau_k} = \frac{2}{N(N-1)} \sum\nolimits_{r=1}^{N} (2r - N - 1) S_{\sigma(r),k}.$$

The complete proof is available in Appendix A. Prop. 3.1 above shows that the reward can be computed by a single pass over the sorted scores, avoiding explicit enumeration of all $O(N^2)$ pairs.

3.4 CURE as a Special Case of CCE

Below we show that our CCE can recover CURE as a binary special case.

Theorem 3.1 (**CURE as a Special Case of CCE**). *Suppose each generated test τ induces a binary pass/fail score,*

$$\mathrm{score}(s;\tau) = \begin{cases} 1, & \text{if } s \text{ passes } \tau, \\ 0, & \text{otherwise}. \end{cases}$$

Let $c_s \in \{0,1\}$ denote whether a program s is correct, and let $c_\tau \in \{0,1\}$ denote whether a generated test τ is correct. Under the binary execution model of CURE (Wang et al., 2025),

$$\Pr(\mathrm{score}(s;\tau)=1) = \begin{cases} 1, & c_s=1,\ c_\tau=1, \\ 0, & c_s=1,\ c_\tau=0, \\ p_{01}, & c_s=0,\ c_\tau=1, \\ p_{00}, & c_s=0,\ c_\tau=0, \end{cases}$$

where p_{01} is the probability that an incorrect program passes a correct test, and p_{00} is the probability that an incorrect program passes an incorrect test. Let

$$p_\tau = \Pr(c_\tau = 1)$$

be the probability that a generated test τ is correct. Then under the notation $p_\tau = p_u$, our CCE's pairwise generated-test reward recovers CURE's unit-tester reward

$$R_\tau = p_\tau(1-p_{01}) - (1-p_\tau)p_{00}.$$

The complete proof is available in Appendix A. Theorem 3.1 shows that our CCE generalizes CURE from binary pass/fail outcomes to general continuous-valued outcomes, with the latter as a special case of the former. This drastically broadens the range of tasks that our CCE can solve.

3.5 Training

We optimize a single instruction-tuned LLM as both the coder and the test-case generator. The two roles share the same parameters and differ only in their role-specific prompts.

In each training iteration, for each task q, the current checkpoint samples candidate programs $\{s_j\}_{j=1}^{N}$ and generated tests $\{\tau_k\}_{k=1}^{K}$. When fresh samples do not provide enough valid programs or discriminative tests, we supplement the batch with previously useful programs or tests from a replay buffer. All selected programs are executed on both reference tests and generated tests to obtain the *reference score matrix* S^* and the *generation score matrix* S. The coder reward is the average reference score defined in Eqn. 1. The test reward is the empirical reliability-based reward defined in Eqn. 2. Invalid or non-discriminative generated tests are filtered before reward construction, while invalid programs receive zero scores on failed executions.

We optimize the shared policy with a GRPO-style objective (Shao et al., 2024). Let $g(o) \in \{\mathrm{code}, \mathrm{test}\}$ denote the role of a sampled response o, let $\mathcal{G}_{g(o)}$ be its role-specific rollout group, and let R_o be its raw coder or test reward. With $\overline{R}_{g(o)}$ and $\widehat{\sigma}_{g(o)}$ as the mean and population standard deviation of the rewards in that group, respectively, we define the group-relative advantage:

$$A_o = \frac{R_o - \overline{R}_{g(o)}}{\widehat{\sigma}_{g(o)} + \epsilon_{\mathrm{norm}}}, \tag{3}$$

where $\epsilon_{\mathrm{norm}} > 0$ is a numerical stabilizer. Code-generation and test-generation rewards are normalized separately, leading to the resulting advantages for program and test responses, A_{s_j} and A_{τ_k}, respectively. Exact numerical constants and implementation details are given in Appendix D.

For a sampled response o with advantage A_o, let

$$r_\theta(o) = \exp\left(\log \pi_\theta(o \mid q) - \log \pi_{\theta_{\mathrm{old}}}(o \mid q)\right),$$

where $\pi_{\theta_{\mathrm{old}}}$ is the phase-start policy used to compute old log-probabilities. We minimize

$$\mathcal{L}(\theta) = -\mathbb{E}\Big[\min\left(r_\theta(o)A_o, \mathrm{clip}(r_\theta(o), 1-\epsilon, 1+\epsilon)A_o\right)\Big] + \beta \widehat{D}_{\mathrm{KL}}(\pi_\theta, \pi_{\theta_{\mathrm{old}}}),$$

where ϵ is the policy-ratio clipping radius, β is the KL-regularization coefficient, and $\widehat{D}_{\mathrm{KL}}$ is an approximate KL penalty to the phase-start policy.

Training proceeds in two consecutive role-specific update phases on the same model. We first update the shared policy on code-generation responses using coder rewards. We then recompute old log-probabilities for the test-generation responses and update the same policy using generated-test rewards. The resulting checkpoint is used for the next rollout iteration.

Table 1: TSP performance on hidden tests. Scores are normalized to $[0, 1]$; higher is better. Best and second-best values are bolded and underlined, respectively.

Type	Method	Mean	Median	P75	P90	P95	Best@2	Best@4	Best@8	Best@16
RL-Trained	Coder-Only	<u>0.621</u>	**0.940**	<u>0.944</u>	**0.997**	**0.998**	0.838	**0.958**	<u>0.986</u>	<u>0.995</u>
	CURE	0.613	**0.940**	0.940	0.966	0.991	<u>0.841</u>	0.945	0.972	0.985
	CCE (Ours)	**0.629**	**0.940**	**0.950**	<u>0.993</u>	<u>0.998</u>	**0.850**	<u>0.957</u>	**0.987**	**0.995**
Direct	Qwen2.5-7B-Instruct	0.518	<u>0.706</u>	0.940	0.964	0.995	0.761	0.917	0.971	0.987
	Qwen2.5-Coder-7B-Instruct	0.579	**0.940**	0.940	0.940	0.940	0.807	0.922	0.942	0.943

Table 2: Set Cover performance on hidden tests. Scores are normalized to $[0, 1]$; higher is better. Best and second-best values are bolded and underlined, respectively.

Type	Method	Mean	Median	P75	P90	P95	Best@2	Best@4	Best@8	Best@16
RL-Trained	Coder-Only	0.327	0.027	0.794	**0.959**	**0.959**	0.513	0.758	0.896	0.951
	CURE	<u>0.504</u>	<u>0.645</u>	<u>0.841</u>	**0.959**	**0.959**	<u>0.735</u>	<u>0.878</u>	<u>0.930</u>	<u>0.953</u>
	CCE (Ours)	**0.550**	**0.839**	**0.863**	**0.959**	**0.959**	**0.757**	**0.888**	**0.933**	**0.956**
Direct	Qwen2.5-7B-Instruct	0.134	0.000	0.004	<u>0.712</u>	<u>0.841</u>	0.255	0.426	0.629	0.822
	Qwen2.5-Coder-7B-Instruct	0.086	0.000	0.045	0.292	0.490	0.168	0.259	0.426	0.609

4 EXPERIMENTS

In this section, we compare CCE with existing methods on four graded programming tasks.

4.1 TASKS AND EVALUATION PROTOCOL

We evaluate CCE on **TSP**, **Set Cover**, **Parallel Machine Scheduling (PMS)**, and **Game 2048**. These tasks cover both one-shot combinatorial optimization, where a program outputs a complete solution for a given instance, and sequential decision making, where a program implements a policy that is rolled out from an initial state. For all tasks, the model is prompted to generate executable Python code, which is run on held-out instances and scored by a task-specific evaluator. Concrete interface examples for TSP, Set Cover, PMS, and Game 2048 are provided in Appendix B.

Combinatorial Optimization Tasks. TSP asks a program to find a short tour over two-dimensional points, Set Cover asks it to select a minimum-cost collection of sets covering the universe, and PMS asks it to assign jobs to parallel machines while minimizing makespan. TSP and Set Cover scores use the exact instance optimum, whereas PMS uses a reference makespan. Invalid, infeasible, or unparsable outputs receive score 0. Program validity statistics are reported in Appendix G.

Game 2048. For Game 2048, a generated program implements a next-move policy,

```
next_move(board: list[int]) -> str,
```

where the board is a flattened 4×4 state and the action is one of `UP`, `DOWN`, `LEFT`, or `RIGHT`. The evaluator rolls out the policy using deterministic transition and tile-spawning rules, with a fixed maximum number of steps. Since Game 2048 does not have a tractable instance-level optimum, we use a random-gated normalization based on random-policy rollouts to normalize training rewards on each initial board. This provides a bounded training signal, but is less suitable as the main evaluation metric. For final evaluation, we therefore report raw rollout metrics, including the number of valid steps and merge score, which directly reflect policy quality in the game. Detailed task-specific scoring rules, including the Game 2048 training normalization, are provided in Appendix C.

Evaluation Protocol. For each task, final evaluation is performed on hidden test instances that are never used during training and are separate from the reference tests used to compute training rewards. Direct-inference baselines and trained checkpoints are evaluated on the same hidden tests with the same parsers and scoring functions. Game 2048 rollouts are capped at 500 moves.

For final evaluation, each method samples $C = 256$ candidate programs and evaluates each program on $H = 256$ fixed hidden test instances, producing a $C \times H$ metric table M. For TSP, Set Cover, and PMS, $M_{c,h}$ is the normalized score of program c on hidden test h. We first average over hidden tests for each program: $\bar{m}_c = \frac{1}{H} \sum_{h=1}^{H} M_{c,h}$. Reported distribution statistics, such as median, 90th percentile (P90), and 95th percentile (P95), are computed over the C per-program averages $\{\bar{m}_c\}_{c=1}^{C}$.

Table 3: PMS performance on hidden tests. Scores are normalized to $[0, 1]$ using the reference makespan; higher is better. Best and second-best values are bolded and underlined, respectively.

Type	Method	Mean	Median	P75	P90	P95	Best@2	Best@4	Best@8	Best@16
RL-Trained	Coder-Only	0.604	**0.626**	0.626	0.677	0.677	0.638	0.653	0.675	0.701
	CURE	0.600	**0.626**	0.677	0.677	**0.956**	0.693	0.749	0.813	0.890
	CCE (Ours)	**0.693**	**0.626**	**0.956**	**0.956**	**0.956**	**0.817**	**0.891**	**0.942**	**0.955**
Direct	Qwen2.5-7B-Instruct	0.404	**0.626**	0.626	0.626	0.626	0.546	0.620	0.639	0.650
	Qwen2.5-Coder-7B-Instruct	0.444	**0.626**	0.638	0.677	0.677	0.594	0.666	0.695	0.726

Table 4: Game 2048 performance on hidden tests, measured by merge score. Higher is better; best and second-best values are bolded and underlined, respectively.

Type	Method	Mean	Median	P75	P90	P95	Best@2	Best@4	Best@8	Best@16
RL-Trained	Coder-Only	27.0	0.0	**29.9**	52.5	108.2	48.7	82.7	133.5	209.5
	CURE	24.4	**17.9**	**29.9**	44.4	69.7	41.6	66.0	100.8	154.2
	CCE (Ours)	**46.1**	1.5	**29.9**	55.6	121.8	**84.6**	**149.1**	**254.1**	**418.8**
Direct	Qwen2.5-7B-Instruct	14.6	0.0	25.5	37.9	62.5	25.9	42.2	62.4	86.6
	Qwen2.5-Coder-7B-Instruct	33.7	0.0	**29.9**	**74.1**	**126.8**	61.4	105.9	173.5	272.9

We also report best-of-B performance. For each value of B, we repeatedly sample a subset $\mathcal{B} \subseteq \{1, \ldots, C\}$ with $|\mathcal{B}| = B$ and compute $\max_{j \in \mathcal{B}} \bar{m}_j$. We repeat this procedure 1,000 times and report the average as Best@B. For Game 2048, $M_{c,h}$ measures raw rollout metrics such as valid steps or merge score, and we use the same per-program aggregation and best-of-B protocol with the corresponding task-specific metric.

4.2 Baselines

We compare against four baselines, two training baselines that use the same base model and optimization pipeline, and two direct-inference baselines without RL fine-tuning.

Coder-Only Training. This baseline removes test-case generation from the training loop. The model samples only candidate programs, executes them on reference instances, and is updated using the same coder reward as CCE. It uses the same base model, execution pipeline, and RL optimizer, but skips the generated-test reward and test-generator update phase.

CURE (Binary-Feedback Co-evolution). This baseline keeps the same co-evolutionary training pipeline as CCE, but thresholds task-specific execution scores into binary pass/fail indicators, yielding a CURE-style interaction table. Thresholding details are provided in Appendix D. This baseline tests whether preserving graded objective values is more effective than discretizing them into pass/fail rewards.

Direct Inference. We also evaluate pretrained models without RL fine-tuning. These models are prompted to generate solver code and are evaluated with the same parsers, hidden test instances, and scoring functions as the trained models. We include direct inference with the base model, Qwen2.5-7B-Instruct (Qwen et al., 2024), and with the code-specialized Qwen2.5-Coder-7B-Instruct (Hui et al., 2024).

4.3 Implementation Details

All RL-trained models, including CCE and the training baselines, are initialized from Qwen2.5-7B-Instruct (Qwen et al., 2024) and trained for 120 iterations. In each iteration, the current policy is sampled under the coder and test-generator prompts using a shared rollout protocol with task-specific filtering. We use oversampling before filtering invalid or duplicate samples. For Game 2048, the coder generates a `next_move` policy and the test generator produces initial boards. More details are in Appendix D.

4.4 Main Results

We report normalized objective scores for TSP, Set Cover, and PMS in Table 1-3. Game 2048 has no tractable instance-level optimum; we therefore report raw merge score and valid rollout steps in Table 4-5.

Table 5: Game 2048 performance on hidden tests, measured by valid rollout steps. Higher is better; best and second-best values are bolded and underlined, respectively.

Type	Method	Mean	Median	P75	P90	P95	Best@2	Best@4	Best@8	Best@16
RL-Trained	Coder-Only	3.66	0.00	5.46	8.66	13.61	6.38	10.22	15.25	22.18
	CURE	3.72	**2.31**	5.46	7.55	10.88	6.10	9.06	12.67	17.79
	CCE (Ours)	**5.63**	0.36	**5.58**	9.54	14.77	**9.96**	**16.61**	**26.60**	**41.54**
Direct	Qwen2.5-7B-Instruct	2.33	0.00	5.15	6.34	9.34	4.06	6.38	8.90	11.54
	Qwen2.5-Coder-7B-Instruct	4.47	0.01	5.46	**10.22**	**17.80**	7.90	13.00	20.11	30.10

Overall Comparison. CCE produces the strongest overall results. On TSP, the RL-trained methods have similar mean scores, while all methods reach near-optimal solutions at Best@16. On Set Cover, CCE leads in mean, median, P75, and every Best@B column. It likewise achieves the strongest PMS results and, on Game 2048, the highest mean and Best@B performance for both reported metrics.

Among the direct-inference baselines, Qwen2.5-Coder is the strongest competitor, particularly at the upper percentiles on Game 2048. Nevertheless, CCE exceeds both direct baselines in mean and every reported Best@B metric on all four tasks, indicating that training improves the overall program distribution as well as its upper tail.

Effect of Continuous Feedback. Comparing CCE with CURE isolates continuous feedback within the same co-evolutionary pipeline. CCE obtains a higher mean on TSP (0.629 vs. 0.613), Set Cover (0.550 vs. 0.504), and PMS (0.693 vs. 0.600), as well as higher mean merge score (46.1 vs. 24.4) and valid steps (5.63 vs. 3.72) on Game 2048. The gains also hold across every reported Best@B value on Set Cover, PMS, and Game 2048, supporting the value of preserving graded differences rather than thresholding them into pass/fail outcomes.

Effect of Coder-Tester Co-evolution. Comparing CCE with coder-only training isolates the contribution of the test-generation role. CCE improves mean performance on every task, with the largest differences on Set Cover, PMS, and Game 2048; the two methods are closer on TSP. This indicates that ranking-based test rewards provide a useful additional training signal through the shared model. Generated-test validity and ranking agreement are analyzed in Appendix H.

Case Study: Binary Feedback Collapses Valid but Different Heuristics. Table 6 shows a Set Cover case study (problem setting in Appendix B.2) illustrating why binary feedback can be too coarse. These three programs, A, B, and C, have substantially different solution quality. Binary feedback assigns the same Binary Pass rate (*Bin. Pass*) of 1.0 to all three programs, while our continuous scores (*Con. Score*) can distinguish better programs from worse ones. Specifically, the strongest program (A) selects the set with maximum newly covered elements and breaks ties by lower cost, while the weakest program (C) effectively selects the last set that improves coverage. Appendix E provides more details, including the corresponding code excerpts.

Table 6: Set Cover case study on hidden tests (the test set). Three programs, A, B, and C, have substantially different solution quality. Binary feedback assigns the same Binary Pass rate (*Bin. Pass*) of 1.0 to all three programs, while our continuous scores (*Con. Score*) can distinguish better programs from worse ones.

Program	Strategy	Con. Score	Bin. Pass
A	Max coverage + tie-break	0.9763	1.0000
B	Max coverage only	0.8781	1.0000
C	Last improving set	0.6811	1.0000

More Case Studies. Additionally, in Appendix F, we provide a qualitative Set Cover case study to illustrate how CCE's continuous feedback can lead to programs with stronger strategies compared to myopic strategies produced by CURE.

5 Conclusion

We propose CCE, a continuous coder-tester co-evolution framework for graded code generation. CCE is motivated by the observation that solver and policy programs are evaluated by quality, not only by correctness. It therefore trains generated tests as surrogate evaluators: a test is rewarded when its scores preserve the reliability ordering induced by reference tests. This objective uses only execution scores, avoids ad hoc thresholding, and recovers the CURE unit-tester reward when outcomes are binary. Experiments on TSP, Set Cover, PMS, and Game 2048 show that continuous co-evolution improves the trained sampling distribution over coder-only RL and CURE-style binary co-evolution, with the largest gains in regimes where validity and solution quality separate.

AI USE STATEMENT

We used generative AI tools to assist with drafting and polishing portions of the manuscript, identifying potentially relevant literature, and preparing experimental and evaluation code. AI assistance in research execution was limited to code preparation; the authors determined the research questions, methodology, experimental design, and interpretation of results. The authors reviewed all AI-assisted text and code, verified cited literature against the original sources, and tested AI-assisted code before use. We did not use generative AI to generate synthetic datasets or reported experimental results. The authors independently checked the mathematical arguments and take full responsibility for the final content of this work, including all AI-assisted text, claims, code, and other artifacts.

REFERENCES

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*, 2023.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4571–4585, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*, 2025.

Baizhou Huang, Shuai Lu, Xiaojun Wan, and Nan Duan. Enhancing large language models in coding through multi-perspective self-consistency. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1429–1450, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Xue Jiang, Yihong Dong, Mengyang Liu, Deng Hongyi, Tian Wang, Yongding Tao, Zhi Jin, Wenpin Jiao, and Ge Li. CODERL+: Improving code generation via reinforcement with execution semantics alignment. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3608–3622, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.164.

Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. Reveal: Self-evolving code agents via reliable self-verification. *arXiv preprint arXiv:2506.11442*, 2025.

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

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 15964–15978, Suzhou, China, November 2025a. Association for Computational Linguistics. ISBN 979-8-89176-335-7.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. CodePRM: Execution feedback-enhanced process reward model for code generation. In

Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 8169–8182, Vienna, Austria, July 2025b. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.428.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223. PMLR, 21–27 Jul 2024.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*, 2023.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. Dynamic scaling of unit tests for code reward modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6917–6935, 2025.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/S41586-023-06924-6.

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

Longwen Wang, Yirui Liu, Xuan'er Wu, Xiaohui Hu, Yuankai Fan, Kaidong Yu, Qizhen Weng, Wei Xi, and Xuelong Li. Beyond binary: Turning partial success into dense verifiable rewards for reinforcement learning in code generation, 2026.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning. *arXiv preprint arXiv:2506.03136*, 2025.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 43571–43608. Curran Associates, Inc., 2024. doi: 10.52202/079017-1381.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12023–12040, 2025.

A PROOFS

Proposition A.1 (Efficient Computation of the Pairwise Reward). *After sorting candidates by reliability, the empirical generated-test reward can be computed in linear time in N, including when reliability ties are present. When all candidate reliabilities are distinct, the reward in Eqn. 2 admits the closed form*

$$\widehat{R}_{\tau_k} = \frac{2}{N(N-1)} \sum_{r=1}^{N} (2r - N - 1) S_{\sigma(r),k}.$$

Proof. For a fixed rank r, the score $S_{\sigma(r),k}$ appears with a positive sign whenever it is the higher-reliability element in a pair. This happens for the $r-1$ lower-ranked candidates. It appears with a negative sign whenever it is the lower-reliability element in a pair, which happens for the $N-r$ higher-ranked candidates. Therefore, its total coefficient is

$$(r-1) - (N-r) = 2r - N - 1.$$

Substituting these coefficients into the pairwise sum gives the stated expression.

If ties are present, partition the candidates into groups $G_1, \ldots, G_q$ of equal reliability, ordered from lower to higher reliability, and process them in this order. For the current group G_t, let $m = |G_t|$ and $B = \sum_{j \in G_t} S_{j,k}$. Let $L_t = \bigcup_{u<t} G_u$ contain all candidates in preceding, strictly lower-reliability groups, with $c = |L_t|$ and $A = \sum_{i \in L_t} S_{i,k}$. The pairs whose higher-reliability endpoint lies in G_t contribute

$$\sum_{j \in G_t} \sum_{i \in L_t} (S_{j,k} - S_{i,k}) = cB - mA$$

to the numerator and add cm strictly ordered pairs to the denominator. Summing these denominator increments over all groups gives the total number of pairs having unequal reliability. A pair between G_t and a later, higher-reliability group is intentionally deferred: after updating $c \leftarrow c + m$ and $A \leftarrow A + B$, the candidates in G_t belong to the preceding set when that later group is processed. Hence every pair from two distinct reliability groups is counted exactly once, when its higher-reliability group is processed, while within-group tied pairs are never counted.

After all groups are processed, dividing the accumulated numerator by the accumulated denominator gives the tie-aware empirical reward; if the denominator is zero, meaning that all candidates are tied, the reward is defined as 0. Each candidate enters the cumulative count and sum once, proving the claimed $O(N)$ time after sorting. □

Theorem A.1 (CURE as a Special Case of CCE). *Suppose each generated test τ induces a binary pass/fail score,*

$$\mathrm{score}(s;\tau) = \begin{cases} 1, & \textit{if } s \textit{ passes } \tau, \\ 0, & \textit{otherwise}. \end{cases}$$

Let $c_s \in \{0,1\}$ denote whether a program s is correct, and let $c_\tau \in \{0,1\}$ denote whether a generated test τ is correct. Under the binary execution model of CURE (Wang et al., 2025),

$$\Pr(\mathrm{score}(s;\tau) = 1) = \begin{cases} 1, & c_s = 1,\ c_\tau = 1, \\ 0, & c_s = 1,\ c_\tau = 0, \\ p_{01}, & c_s = 0,\ c_\tau = 1, \\ p_{00}, & c_s = 0,\ c_\tau = 0, \end{cases}$$

where p_{01} is the probability that an incorrect program passes a correct test, and p_{00} is the probability that an incorrect program passes an incorrect test. Let

$$p_\tau = \Pr(c_\tau = 1)$$

be the probability that a generated test is correct. Then under the notation $p_\tau = p_u$, our CCE's pairwise generated-test reward recovers CURE's unit-tester reward

$$R_\tau = p_\tau(1 - p_{01}) - (1 - p_\tau)p_{00}.$$

Proof. Take two candidate programs a and b such that $I(a) > I(b)$. In the binary setting, this corresponds to treating a as correct and b as incorrect. Since the induced scores are binary,

$$\text{score}(a;\tau) - \text{score}(b;\tau) \in \{-1, 0, 1\}.$$

Define the two events

$$A^+ = \{\text{score}(a;\tau) = 1,\ \text{score}(b;\tau) = 0\},$$
$$A^- = \{\text{score}(a;\tau) = 0,\ \text{score}(b;\tau) = 1\}.$$

Therefore, the pairwise generated-test reward can be written as

$$R_\tau = \Pr(A^+ \mid I(a) > I(b)) - \Pr(A^- \mid I(a) > I(b)).$$

The first event occurs when the generated test is correct and the incorrect program fails it, which has probability $p_\tau(1 - p_{01})$. The second event occurs when the generated test is incorrect, the correct program fails it, and the incorrect program passes it, which has probability $(1-p_\tau)p_{00}$. Substituting these two probabilities gives

$$R_\tau = p_\tau(1 - p_{01}) - (1 - p_\tau)p_{00}.$$

This matches the CURE unit-tester reward with $p_\tau = p_u$, completing the proof. □

B TASK EXAMPLES

This appendix gives compact examples of the task interfaces used in our experiments. For TSP, Set Cover, and PMS, a candidate solution is a complete Python program that reads from standard input and writes to standard output. For Game 2048, a candidate solution defines a function `next_move(board: list[int]) -> str`, which is repeatedly called by the evaluator during a rollout.

B.1 TSP

In TSP, the first input line gives the number of cities n. The following n lines give the city coordinates. Cities are indexed from 0 to $n - 1$ in input order. A valid program prints a route containing all city indices exactly once, and the route must start at city 0. The evaluator adds the return edge from the last city back to city 0.

Example Input.

```
4
0 0
1 0
1 1
0 1
```

Example Program.

```
import math

n = int(input())
pts = [tuple(map(float, input().split())) for _ in range(n)]

route = [0]
unused = set(range(1, n))
while unused:
    cur = route[-1]
    nxt = min(unused, key=lambda j:
              (pts[cur][0] - pts[j][0])**2 +
              (pts[cur][1] - pts[j][1])**2)
```

```
    route.append(nxt)
    unused.remove(nxt)

print(*route)
```

Example Output.

```
0 1 2 3
```

The parser reads the first n integers from the program output. The route is invalid if it does not start at 0, omits a city, repeats a city, or contains an index outside $[0, n-1]$.

B.2 SET COVER

In Set Cover, the first input line contains two integers n and m, where n is the universe size (i.e., the number of elements) and m is the number of sets. Universe elements are indexed from 1 to n. Each of the next m lines has the format

```
cost elem_1 elem_2 ...  elem_t.
```

There is no separate field for t. A valid program prints m binary decisions, where decision i indicates whether set i is selected.

Example Input.

```
5 4
3 1 2
4 2 3
2 4 5
6 1 3 5
```

Example Program.

```
n, m = map(int, input().split())
sets = []
for _ in range(m):
    row = list(map(int, input().split()))
    cost, elems = row[0], set(row[1:])
    sets.append((cost, elems))

covered = set()
take = [0] * m
while len(covered) < n:
    best, best_score = -1, -1
    for i, (cost, elems) in enumerate(sets):
        if take[i]:
            continue
        gain = len(elems - covered)
        score = gain / max(cost, 1)
        if score > best_score:
            best, best_score = i, score

    if best == -1 or best_score <= 0:
        break

    take[best] = 1
    covered |= sets[best][1]

print(*take)
```

Example Output.

```
1 1 1 0
```

This selects sets 1, 2, and 3 in the printed decision vector, covering all elements $\{1, 2, 3, 4, 5\}$. The evaluator reads the first m integers from the output. Outputs are invalid if fewer than m integers are printed, if any of the first m values is not 0 or 1, or if the selected sets do not cover the universe.

B.3 Parallel Machine Scheduling (PMS)

In PMS, the first input line gives the number of identical machines m, the second line gives the number of jobs n, and the third line gives the n job processing times as positive integers. Jobs are indexed from 0 to $n-1$ in input order and machines from 0 to $m-1$. A valid program prints n integers, where the i-th value is the index of the machine that job i is assigned to. The makespan is the largest total load over the m machines, and the score is the reference makespan divided by the achieved makespan, clipped to $[0, 1]$.

Example Input.

```
2
4
5 4 3 2
```

Example Program.

```
def assign_jobs(m, n, p):
    # Sort jobs by processing time in descending order
    jobs = sorted(zip(p, range(n)), key=lambda x: x[0], reverse=True)
    machines = [0] * m  # Initialize machines with zero processing time
    assignment = [-1] * n  # Initialize assignment list
    for i, (_, job) in enumerate(jobs):
        min_load_index = machines.index(min(machines))
        assignment[job] = min_load_index
        machines[min_load_index] += p[job]
    return assignment

m = int(input())
n = int(input())
p = list(map(int, input().split()))

assignment = assign_jobs(m, n, p)
print(' '.join(map(str, assignment)))
```

Example Output.

```
0 1 1 0
```

This output assigns jobs 0 and 3 to machine 0 and jobs 1 and 2 to machine 1. Both machines have total load 7, so the resulting makespan is 7. The evaluator reads the first n integers from the output. Outputs are invalid if fewer than n integers are printed or if any of these values falls outside $[0, m-1]$.

B.4 Game 2048

Game 2048 is a single-player sliding-tile game on a 4×4 board. At each step, the player chooses one direction, and all tiles slide as far as possible in that direction. Adjacent tiles with the same value merge into a single tile with doubled value; for example, two tiles with value 2 merge into one tile with value 4. The merge contributes to the game score. After a valid move, the evaluator deterministically spawns one or more new tiles in empty cells. A move is valid only if it changes the board.

For Game 2048, the generated program is not a standard-input program. Instead, it must define:

```
def next_move(board: list[int]) -> str:
    ...
```

The board is a flattened 4×4 list in row-major order. Each entry stores the base-2 logarithm of the tile value: 0 denotes an empty cell, 1 denotes tile 2, 2 denotes tile 4, and so on. The function must return one of:

```
"UP", "DOWN", "LEFT", "RIGHT"
```

The evaluator strips surrounding whitespace, is case-insensitive, and also accepts the single-letter aliases `"U"`, `"D"`, `"L"` and `"R"`; any other return value terminates the rollout as invalid.

For Game 2048, the test generator produces initial boards as generated test cases. Figure 4(a) shows an example of a generated initial board, together with a one-step transition example. In the actual program interface, each board is passed as a flattened list of log-tile values rather than as an image.

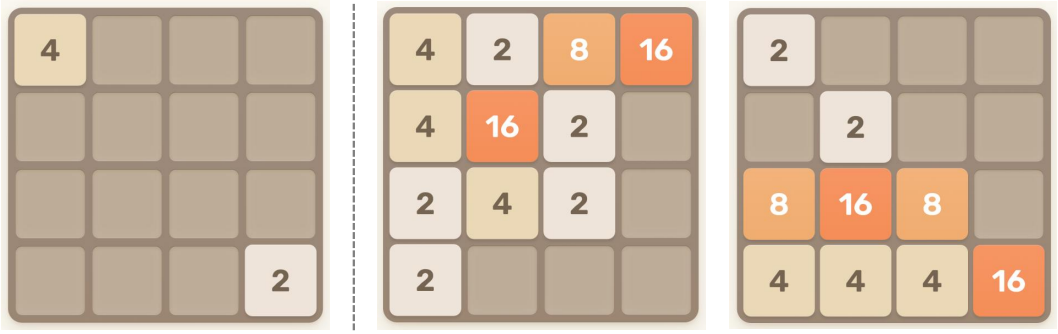
For the transition example in Figure 4(b), the board before the move is represented as (using the exponent x in 2^x):

```
[2, 1, 3, 4,
 2, 4, 1, 0,
 1, 2, 1, 0,
 1, 0, 0, 0]
```

Example Program.

```
def next_move(board: list[int]) -> str:
    def merge_line(line):
        vals = [x for x in line if x != 0]
        out, gain = [], 0
        i = 0
        while i < len(vals):
            if i + 1 < len(vals) and vals[i] == vals[i + 1]:
                out.append(vals[i] + 1)
                gain += 2 ** (vals[i] + 1)
                i += 2
            else:
                out.append(vals[i])
                i += 1
        return out + [0] * (4 - len(out)), gain

    def simulate(direction):
        grid = [board[i:i+4] for i in range(0, 16, 4)]
        gain = 0
        if direction in ["LEFT", "RIGHT"]:
            new = []
            for row in grid:
                line = row if direction == "LEFT" else row[::-1]
                merged, g = merge_line(line)
                if direction == "RIGHT":
                    merged = merged[::-1]
                new.append(merged)
                gain += g
        else:
            new = [[0] * 4 for _ in range(4)]
            for c in range(4):
                col = [grid[r][c] for r in range(4)]
                line = col if direction == "UP" else col[::-1]
                merged, g = merge_line(line)
                if direction == "DOWN":
                    merged = merged[::-1]
                for r in range(4):
                    new[r][c] = merged[r]
                gain += g
```



(a) Generated initial board. (b) Board before a DOWN move. (c) Board after the DOWN move.

Figure 4: Examples for the Game 2048 task. The test generator produces initial boards as generated test cases, as illustrated in **(a)**. Panels **(b)** and **(c)** show a one-step transition: the policy returns `DOWN`; the evaluator applies the slide-and-merge rule, and then deterministically spawns one or more new tiles.

```
        flat = sum(new, [])
        changed = flat != board
        empty = flat.count(0)
        return changed, gain, empty

    best_move, best_score = "UP", (-1, -1)
    for move in ["UP", "DOWN", "LEFT", "RIGHT"]:
        changed, gain, empty = simulate(move)
        if changed and (gain, empty) > best_score:
            best_move, best_score = move, (gain, empty)
    return best_move
```

Assume the program returns `"DOWN"` on the example board in Figure 4(b). The evaluator slides tiles downward and merges adjacent equal-valued tiles. It then applies the deterministic tile-spawning rule, producing the post-move board shown in Figure 4(c).

The evaluator repeatedly calls `next_move` until the maximum rollout length is reached, no legal moves remain, or the rollout terminates due to the policy output. An empty or non-compilable policy, a missing `next_move` function, a non-string return value, an invalid direction string, or a runtime exception is treated as an invalid rollout and receives zero reward. If the policy returns a syntactically valid direction that leaves the board unchanged, the rollout terminates and receives the partial score accumulated up to that point.

C Task-Specific Scoring Details

This section provides additional details on how task-specific program outputs are converted into scores. For TSP, Set Cover, and PMS, invalid, infeasible, or unparsable outputs are assigned score 0 during training reward construction, and valid outputs are normalized relative to a reference value that is the exact optimum for TSP and Set Cover. For Game 2048, invalid policies or invalid rollouts receive zero score, while a legal move that leaves the board unchanged terminates the rollout with the partial accumulated score.

TSP. Each TSP instance consists of a set of two-dimensional points. A candidate program outputs integers; the evaluator uses the first n parsed integers as the candidate tour. This tour is interpreted as a permutation of city indices. We compute the Euclidean tour length $L(\pi, q)$ for the candidate tour π. Let $L^*(q)$ denote the optimal tour length for instance q. The normalized score is

$$\text{score}(\pi, q) = \frac{L^*(q)}{L(\pi, q)}.$$

Tours with missing cities, repeated cities, invalid indices, or unparsable outputs receive score 0. Scores are clipped to $[0, 1]$.

Set Cover. Each Set Cover instance contains a universe of elements and a collection of sets with associated costs. A candidate program outputs integers; the evaluator uses the first m parsed integers as the set-selection vector. A solution is feasible if this vector is binary and the selected sets cover the entire universe. Let $c(x, q)$ be the total selected set cost and let $c^*(q)$ be the optimal feasible cover cost for instance q. For feasible solutions, the normalized score is

$$\text{score}(x, q) = \frac{c^*(q)}{c(x, q)}.$$

Infeasible selections, invalid binary vectors, or unparsable outputs receive score 0. Scores are clipped to $[0, 1]$.

PMS. Each PMS instance consists of m identical machines and n jobs with integer processing times $p_1, \ldots, p_n$. A candidate program outputs integers; the evaluator uses the first n parsed integers as the assignment a, where $a_i \in \{0, \ldots, m-1\}$ is the machine of job i. The makespan is the load of the busiest machine,

$$M(a, q) = \max_{0 \le k < m} \sum_{i:\, a_i = k} p_i,$$

and, writing $M^{\text{ref}}(q)$ for the reference makespan of instance q, the normalized score is

$$\text{score}(a, q) = \frac{M^{\text{ref}}(q)}{M(a, q)}.$$

Assignments with fewer than n parsed integers, or with any index outside $\{0, \ldots, m-1\}$, receive score 0. Scores are clipped to $[0, 1]$, so a program that beats the reference is scored as 1 rather than above 1. Training and evaluation instances are constructed so that a perfectly balanced assignment exists (Appendix D); for these instances the reference makespan is the exact optimum. For instances proposed by the test generator no such construction is available, and the reference is obtained by the longest-processing-time rule followed by local improvement, which upper-bounds the optimum.

Game 2048. For Game 2048, a candidate program implements a policy through the interface

```
next_move(board: list[int]) -> str.
```

The board is represented as a length-16 row-major list of log-tile values, where 0 denotes an empty cell and a value $k > 0$ denotes tile value 2^k. The evaluator rolls out the policy from an initial board using deterministic game transitions and a fixed deterministic tile-spawning rule. At each step, the policy must return one of `UP`, `DOWN`, `LEFT`, or `RIGHT`, with single-letter aliases also accepted.

Empty programs, compilation failures, missing `next_move` functions, non-string return values, invalid direction strings, or runtime exceptions are treated as invalid rollouts and receive score 0. If the policy returns a syntactically valid direction that does not change the board, the rollout terminates and receives the partial score accumulated before that stalled move. Rollouts also terminate when no legal move remains or when the maximum rollout length is reached.

Unlike the combinatorial optimization tasks, Game 2048 does not have an instance-level exact optimum. We therefore compute a task-specific raw rollout score,

$$R(o, q) = \alpha \cdot \text{steps}(o, q) + \beta \cdot \text{maxTileLog}(o, q) + \gamma \cdot \text{mergeScore}(o, q),$$

where steps is the number of valid moves executed, maxTileLog is the base-2 logarithm of the largest tile reached, and mergeScore is the total score gained from merges. We use $\alpha = 1.0$, $\beta = 10.0$, and $\gamma = 0.1$.

During training, raw Game 2048 scores are converted into bounded scores using a random-gated normalization. For each board q, we estimate a random-policy threshold $T_{\text{rand}}(q)$ as the mean raw reward over 32 random rollouts on the same board:

$$T_{\text{rand}}(q) = \frac{1}{32} \sum_{r=1}^{32} R_{\text{rand}}^{(r)}(q).$$

The random policy uniformly samples from the legal moves at each state and is rolled out with the same deterministic game dynamics and tile-spawning rule.

For a sampled policy o_i, let

$$e_i(q) = R(o_i, q) - T_{\text{rand}}(q), \qquad e_{\max}(q) = \max_j \max(e_j(q), 0).$$

Let $f = 0.02$ denote the random-baseline boundary. If $R(o_i, q) \le T_{\text{rand}}(q)$, we assign a shaped below-threshold score

$$s_i(q) = f \cdot \frac{R(o_i, q)}{\max(T_{\text{rand}}(q), 10^{-8})}.$$

Thus policies below the random baseline receive scores in $[0, f]$, rather than being collapsed to a single value. If $R(o_i, q) > T_{\text{rand}}(q)$, the above-threshold excess is normalized within the current rollout group and blended with an action-validity term:

$$s_i(q) = f + (1 - f)\left(0.8\frac{e_i(q)}{e_{\max}(q)} + 0.2v_i(q)\right),$$

where $v_i(q) = 1.0$ for a normal rollout and 0.6 for a stalled legal move. Only these two values can occur in this branch: an invalid or failed execution is assigned raw reward 0, which never exceeds $T_{\text{rand}}(q)$, so such rollouts always fall in the below-threshold case and receive score 0. If no sampled policy exceeds the random threshold for a board, the below-threshold shaping still preserves differences among partial rollouts. This normalization gives a bounded training signal without requiring an exact optimum.

For final evaluation on Game 2048, we report raw rollout metrics rather than treating the training-normalized score as the primary result. In particular, we report metrics such as valid rollout steps and merge score, which directly reflect policy quality in the game. This differs from TSP and Set Cover, where normalized scores relative to an optimal solution are the primary evaluation metrics.

D Additional Training Details

Training Pools and Reference Tests. For each task, we construct a training pool of 256 instances and a held-out in-training evaluation pool of 64 instances. At each optimization iteration, we sample reference instances from the training pool to compute the reference score matrix used for coder rewards and generated-test rewards (64 instances for TSP, Set Cover, and PMS, and 32 for Game 2048). TSP instances contain 6-8 cities and use exact optimal tour lengths for normalization. Set Cover instances have universe size 5-7 and 6-9 sets, and use exact optimal cover costs for normalization. PMS instances contain 108–156 machines and 138–387 jobs, with optimal makespans between 222 and 520. Each instance is built by drawing a target makespan and partitioning it into the per-machine job lists, so a perfectly balanced assignment exists by construction and its makespan is the exact optimum used for normalization. Game 2048 uses 4×4 initial boards and the random-gated training normalization described in Appendix C.

Rollout Sampling and Filtering. During sampling, we use vLLM with temperature 1.0 and top-$p = 1.0$. Each training iteration targets up to 64 candidate programs and up to 64 generated tests for reward construction. To obtain sufficiently many usable samples after filtering, we first sample 128 candidate programs and 128 generated tests. Generated programs that cannot be parsed as executable Python programs and generated tests that do not match the task format are treated as invalid.

We also remove duplicate usable samples based on their execution behavior. For candidate programs, two programs are considered duplicated if they obtain the same nonzero score vector across the evaluated tests. For generated tests, two tests are considered duplicated if they induce the same nonzero score vector across the sampled programs. Failed samples with all-zero scores are retained as negative examples rather than removed as duplicates. A small replay buffer stores previously useful programs and discriminative tests; fresh samples are used when they provide enough valid programs and discriminative tests, and replayed samples are used to supplement the batch otherwise.

Binary-Feedback Thresholding. For the CURE binary-feedback baseline, we convert each program-test score into a binary pass/fail indicator using an instance-specific threshold:

$$B(o, q) = \mathbb{1}[\text{score}(o, q) > \tau(q)]\,.$$

The comparison uses a strict inequality. The coder reward is then the average pass rate over reference tests,

$$R_o^{\text{bin}} = \frac{1}{|\mathcal{Q}_{\text{ref}}|} \sum_{q \in \mathcal{Q}_{\text{ref}}} B(o, q),$$

followed by the same group-relative reward normalization used by the continuous setting.

For TSP, $\tau(q)$ is the mean normalized score of 32 random tours sampled with the instance seed, where each random tour starts at city 0 and randomly permutes the remaining cities. For Set Cover, $\tau(q)$ is the mean normalized score of 32 random binary set-selection vectors sampled with the instance seed, where infeasible selections receive score 0. For Game 2048, we use the random-gated normalization described in Appendix C; a rollout passes if and only if its normalized training score exceeds the floor value 0.02, which corresponds to exceeding the random-policy threshold for that board.

Task-Specific Batch Composition. For Game 2048, we additionally retain up to 16 invalid or zero-reward code samples as negative examples in the code-update batch. This stabilizes training by discouraging policies that frequently produce invalid programs or degenerate rollouts. This supplementation affects only the composition of the update batch and does not change the task-specific scoring or reward definitions.

Policy Updates. The shared model is updated in two role-specific phases. We first update on code-generation responses using coder rewards. We then recompute old log-probabilities for test-generation responses and update the same model using generated-test rewards. Rewards for the two roles are normalized separately before policy optimization. In reward construction, we use the population standard deviation and set $\epsilon_{\text{norm}} = 10^{-8}$ in Eqn. 3. An all-zero reward group therefore has zero advantages; a nonzero constant-reward group is omitted because it contains no relative learning signal. The GRPO data loader reapplies same-prompt group-relative centering and scaling with a 10^{-6} numerical stabilizer and rescales the resulting advantages by their overall population standard deviation. These operations are performed within each role-specific dataset and never mix coder and test-generator rewards. For coder-only settings, the test-generation phase is skipped.

We use a GRPO-style clipped policy-gradient objective with clipping range 0.2. The KL-regularization coefficient of the clipped policy objective is $\beta = 0.01$, and $\widehat{D}_{\text{KL}}$ uses the unbiased low-variance estimator $\exp(\Delta) - \Delta - 1$ with $\Delta = \log \pi_{\theta_{\text{old}}}(o) - \log \pi_\theta(o)$, averaged over unmasked response tokens and taken against the policy at the start of the current role-specific phase. The optimizer is AdamW with learning rate 10^{-6}, first- and second-moment decay rates 0.9 and 0.95, no weight decay, and gradient clipping at 1.0. We use one policy-update epoch per iteration.

Numerically Stable Reliability Computation. For numerical stability, when computing $I(\cdot)$, we sort candidates using the log-domain reliability

$$\ell(s_j) = \sum_{l=1}^{T} \log(S^*_{j,l} + \delta),$$

where $\delta = 10^{-8}$. This preserves the ordering of the multiplicative reliability for positive scores while treating zero-score failures as low-reliability outcomes. In our scoring design, valid feasible outputs receive positive scores, whereas invalid or infeasible executions receive a score of 0; thus, the multiplicative reliability mainly penalizes programs that fail on reference instances rather than collapsing the ordering among valid suboptimal programs. When any reference score $S^*_{j,l}$ is 0, the product reliability $I(s_j)$ becomes 0 and ignores the scores on all other reference tests. This can create large tie groups among programs that fail at least one reference test. The log-domain reliability $\ell(s_j)$ with δ keeps the contributions from the remaining reference tests, giving a finer ordering among these programs. Candidates with exactly equal $\ell(s_j)$ form a tie group; pairs within that group are excluded and hence do not affect the generated-test reward.

Tie-Aware Generated-Test Reward. The main paper presents the reward under distinct candidate reliabilities. The implementation handles ties using the log-domain reliability $\ell(s_j)$ above. Let σ sort candidates such that

$$\ell(s_{\sigma(1)}) \le \ell(s_{\sigma(2)}) \le \cdots \le \ell(s_{\sigma(N)}),$$

and define only the strictly ordered pairs as

$$\mathcal{P}_\sigma = \left\{ (i, j) : 1 \le i < j \le N,\ \ell(s_{\sigma(i)}) < \ell(s_{\sigma(j)}) \right\}.$$

The implemented empirical reward is

$$\widehat{R}^{\mathrm{tie}}_{\tau_k} = \frac{1}{|\mathcal{P}_\sigma|} \sum_{(i,j)\in\mathcal{P}_\sigma} \left(S_{\sigma(j),k} - S_{\sigma(i),k}\right). \tag{4}$$

If $\mathcal{P}_\sigma$ is empty, we set $\widehat{R}^{\mathrm{tie}}_{\tau_k} = 0$. Thus, equally reliable candidates do not contribute a preferred direction. When all reliabilities are distinct, $|\mathcal{P}_\sigma| = N(N-1)/2$, and Eqn. 4 reduces exactly to Eqn. 2. After sorting, the numerator and denominator can still be computed in $O(N)$ time by processing equal-reliability groups and maintaining the cumulative count and score sum of all preceding groups.

Systems. We use gradient checkpointing, FlashAttention, mixed precision depending on hardware support, and DeepSpeed ZeRO stage 3 for multi-GPU runs. Final evaluation is performed on 256 hidden instances generated with a fixed seed and shared across all compared methods; each method samples 256 programs for final evaluation.

E Set Cover Case Study Details

This section provides code excerpts for the Set Cover case study in Table 6. The examples are selected from the final checkpoint of CCE on the hidden Set Cover evaluation set. All three programs are valid on all hidden tests and pass the binary threshold on every instance, but their continuous scores differ substantially.

Program A: Maximum New Coverage with Cost Tie-Break. Program A obtains a mean continuous score of 0.9763 and binary pass rate 1.0000. It greedily selects the set that covers the largest number of currently uncovered elements, using lower cost only as a tie-breaker.

```
def cover(universe_elements, sets):
    selected = [0] * len(sets)
    covered_elements = set()
    while covered_elements != universe_elements:
        max_coverage = (0, 0)
        for i, (cost, elements) in enumerate(sets):
            if selected[i] == 0 and len(elements - covered_elements) > 0:
                coverage = len(elements - covered_elements)
                if coverage > max_coverage[0]:
                    max_coverage = (coverage, i)
                elif coverage == max_coverage[0]:
                    if cost < sets[max_coverage[1]][0]:
                        max_coverage = (coverage, i)
        selected[max_coverage[1]] = 1
        covered_elements.update(sets[max_coverage[1]][1])
    return selected
```

Program B: Maximum New Coverage without Cost Tie-Break. Program B obtains a mean continuous score of 0.8781 and binary pass rate 1.0000. It also greedily selects sets by newly covered elements, but ignores set costs. As a result, it remains valid but produces more expensive covers.

```
def weighted_set_cover(n, sets):
    taken = [0] * len(sets)
    covered = set()
    while len(covered) < n:
        max_improvement = -1
        best_set = -1
        for i, (cost, elements) in enumerate(sets):
            if taken[i]:
                continue
            new_covered = covered.union(elements)
            improvement = len(new_covered) - len(covered)
            if improvement > max_improvement:
                max_improvement = improvement
                best_set = i
        taken[best_set] = 1
```

```
        covered = covered.union(sets[best_set][1])
    return taken
```

Program C: Last Improving Set. Program C obtains a mean continuous score of 0.6811 and binary pass rate 1.0000. Although the code appears to track the cheapest improving set, it never compares the current cost with `min_cost`. It therefore keeps overwriting `min_index` and effectively selects the last set in input order that covers at least one new element.

```
def select_sets(n, m, sets):
    taken = [0] * m
    covered = set()
    cost = 0

    while covered != set(range(1, n + 1)):
        min_cost = float('inf')
        min_index = -1
        for i, (c, s) in enumerate(sets):
            if taken[i]:
                continue
            covered_update = covered | s
            if len(covered_update) > len(covered):
                min_cost = c
                min_index = i
        if min_index == -1:
            raise Exception("No set found to cover any new element.")
        taken[min_index] = 1
        covered |= sets[min_index][1]
        cost += sets[min_index][0]

    return taken
```

This case illustrates why binary feedback can be too coarse for graded optimization tasks. Once programs learn to produce valid covers, a binary threshold against a random baseline may assign the same pass label to heuristics with substantially different cover costs. Continuous scoring preserves these differences.

F Case Study for the Set Cover Task: CCE versus CURE

This section compares one generated Set Cover program from CCE with one generated program from the CURE baseline. Both programs use the same high-level greedy skeleton: repeatedly select one set, update the covered elements, and output a binary set-selection vector. The main difference is the selection criterion used to choose the next set.

Aggregate Behavior. Both programs are shown in Table 7. Both produce valid covers on all 256 hidden Set Cover instances (test cases). However, the CCE program obtains a higher normalized objective score.

Table 7: Aggregate hidden-test performance of the two Set Cover programs used in the case study.

Source	Mean Score	Median Score	Valid Rate
CCE	0.9589	1.0000	1.0000
CURE	0.8591	0.8750	1.0000

Across the 256 hidden instances, the CCE program obtains a higher score on 117 instances, ties on 137 instances, and obtains a lower score on only 2 instances. The mean score difference is 0.0998.

Strategy Learned by CCE. The CCE program chooses the set with the smallest cost per newly covered element:

```
for i, (cost, elements) in enumerate(sets):
    uncovered = elements - covered
```

```
    if uncovered:
        cost_per_elem = cost / len(uncovered)
        if cost_per_elem < min_cost:
            min_cost = cost_per_elem
            selected_set = i
```

This is the standard weighted Set Cover greedy rule. It directly trades off the additional coverage gained by a set against its cost.

Strategy Learned by CURE. The CURE program has a similar loop structure, but prioritizes low absolute cost before coverage:

```
for i in range(m):
    if i in selected_sets:
        continue

    elements = set(sets[i][1:])
    new_covered = elements - covered_elements

    if len(new_covered) > 0:
        cost = sets[i][0]
        if cost < min_cost or (
            cost == min_cost and
            len(new_covered) > max_new_covered
        ):
            min_cost = cost
            max_new_covered = len(new_covered)
            best_set = i
```

This rule is valid and often reasonable, but it is more myopic (i.e., short-sighted): a cheap set that covers only one new element can be preferred over a more expensive set that covers many new elements.

Concrete Hidden Instance. Consider the following hidden Set Cover instance:

```
5 7
3 5
10 1 2 3 4
5 2 5
8 3
9 1 5
10 4
12 1
```

The universe is $\{1, 2, 3, 4, 5\}$, and the seven candidate sets are:

Set id	Cost	Elements
0	3	$\{5\}$
1	10	$\{1, 2, 3, 4\}$
2	5	$\{2, 5\}$
3	8	$\{3\}$
4	9	$\{1, 5\}$
5	10	$\{4\}$
6	12	$\{1\}$

The exact optimum has cost 13, selecting sets 0 and 1:

```
1 1 0 0 0 0 0
```

The CCE program outputs the optimal selection:

```
1 1 0 0 0 0 0
```

with cost $3 + 10 = 13$ and normalized score $13/13 = 1.0000$. This occurs because the ratio rule first selects set 1, which covers four currently uncovered elements at cost 10, giving cost per new element 2.5. It then selects set 0 to cover the remaining element 5.

The CURE binary-feedback program outputs:

```
1 1 1 1 1 0 0
```

with cost $3 + 10 + 5 + 8 + 9 = 35$ and normalized score $13/35 = 0.3714$. This happens because the cheap-first rule repeatedly chooses the lowest-cost set that covers at least one new element, producing a feasible but much more expensive cover.

Interpretation. This example illustrates a qualitative distinction that binary pass/fail supervision does not directly express. Both programs output feasible covers on this instance, so both would receive positive binary feedback. Our CCE's continuous feedback, however, distinguishes the cheaper cover from the more expensive feasible cover. The CCE sample reflects this distinction by using a cost-per-new-element criterion, which captures a finer-grained optimization signal than simply producing any valid cover.

G Program Validity Rates

Table 8 reports program validity rates on hidden tests. For each method, we sample $K = 256$ programs and evaluate each program on $H = 256$ hidden instances, producing a $K \times H$ validity table. The reported validity rate is the average over both sampled programs and hidden instances. Thus, it measures how often a sampled program produces a valid output on a hidden instance, rather than whether the program is syntactically valid in isolation.

Validity is task-specific. For TSP, a valid output must parse as a tour starting at city 0 and containing each city exactly once. For Set Cover, a valid output must parse as a binary set-selection vector that covers the universe. For PMS, a valid output must parse as an assignment of every job to a machine index within range. For Game 2048, a valid policy must implement `next_move(board)` and produce a nonzero valid rollout under the evaluator. Generated-test validity is measured separately as the fraction of sampled test cases that can be parsed into valid task instances.

Table 8: Program validity rates on hidden tests. Values are percentages; higher is better.

Type	Method	TSP	Set Cover	PMS	Game 2048
RL-Trained	Coder-Only	65.9	42.7	**98.9**	40.8
	CURE	65.1	64.3	92.6	**50.3**
	CCE (Ours)	**66.7**	**68.0**	96.9	46.5
Direct	Qwen2.5-7B-Instruct	56.5	19.3	71.8	33.9
	Qwen2.5-Coder-7B-Instruct	61.8	12.9	72.7	41.3

H Generated-Test Diagnostics

We further analyze the generated tests produced by the test-generation role at the final training iteration. This diagnostic is most directly interpretable for one-shot optimization tasks, where each generated test is itself an optimization instance and can be used to compare candidate programs on the same instance. We therefore report this analysis for TSP, Set Cover, and PMS.

For each generated test case τ, we execute the sampled programs on both the reference test set and on τ. Let r_i denote the reference score of program i, computed as its average score on the reference tests, and let $s_i(\tau)$ denote the score of program i on generated test τ. We evaluate generated tests using four diagnostics. Valid rate measures the fraction of sampled generated tests that can be parsed into valid task instances. Discriminative rate measures the fraction of generated tests that are valid and induce non-constant, nonzero scores across sampled programs. Spearman correlation measures the rank correlation between $\{r_i\}$ and $\{s_i(\tau)\}$. We also report normalized rank-distance agreement,

defined as

$$\mathrm{RankAgree}(\tau) = 1 - \frac{\frac{1}{n}\sum_{i=1}^{n} |\mathrm{rank}(r_i) - \mathrm{rank}(s_i(\tau))|}{n/2},$$

where larger values indicate closer agreement between the reference-based ranking and the ranking induced by the generated test. Spearman and RankAgree are averaged over the valid generated tests only, so that how often a test is parseable (the Valid column) and how informative a parseable test is (the two ranking columns) are reported as separate quantities.

Table 9: Generated-test diagnostics at the final training iteration. Valid rate and discriminative rate measure whether generated tests are parseable and useful for separating candidate programs. Spearman and RankAgree measure agreement between the generated-test ranking and the reference-test ranking of candidate programs. Higher is better for all metrics.

Task	Method	Valid	Disc.	Spearman	RankAgree
TSP	CURE	100.0	100.0	0.667	0.684
	CCE	100.0	100.0	**0.690**	**0.690**
Set Cover	CURE	90.6	90.6	0.894	**0.853**
	CCE	**93.8**	**93.8**	**0.904**	0.847
PMS	CURE	**46.9**	**46.9**	0.562	0.664
	CCE	34.4	34.4	**0.797**	**0.815**

Table 9 reports the four diagnostics. Among the tests that parse, CCE attains the higher Spearman correlation on all three tasks, and the higher RankAgree on TSP and PMS; on Set Cover the two methods are within 0.006 of each other on RankAgree. This supports the role of the reliability-based reward: a parseable generated test from CCE provides an evaluation signal that is better aligned with reference-based program quality. The Valid column separates a second effect. On TSP and Set Cover both methods parse at high rates, but on PMS only 34.4% of CCE's generated instances parse, below CURE's 46.9%. PMS instances are the hardest of the three to emit in a well-formed way, and the reliability reward does not by itself improve the parse rate.

I Cross-Task Transfer Evaluation

As an exploratory diagnostic, we evaluate each trained checkpoint on the three tasks on which it was not trained. The goal is to characterize how task-specific the learned program distribution is, rather than to claim general cross-task transfer.

Protocol. Each checkpoint is evaluated with the target task's prompt, parser, and scoring function using 256 sampled programs and 256 hidden test instances. Diagonal entries are in-distribution evaluations, and off-diagonal entries are cross-task evaluations. Within each column, all checkpoints share the same hidden-test set; Game 2048 uses the 500-move rollout budget throughout. The table reports one checkpoint and one program-sampling round per setting, so we treat differences as descriptive rather than as stable method rankings.

Findings. Table 10 reports the resulting matrix. In this evaluation, the in-distribution checkpoint has the highest mean for Set Cover and PMS within every method block, which is consistent with task-specific effects. On Game 2048 the same holds for Coder-Only and CCE, but not for CURE, whose Set-Cover-trained checkpoint reaches a higher merge score than its own Game 2048 checkpoint (29.3 versus 24.4). TSP is a further exception: the PMS-trained checkpoint has a higher sampled mean than the TSP-trained checkpoint for all three methods. Off-diagonal scores are generally nonzero, indicating that some sampled programs remain executable and obtain nonzero scores under a different task prompt. However, because program-level validity and quality vary across sampling rounds, we do not interpret these values as evidence of reliable positive transfer, a stable cross-task ordering, or the absence of forgetting.

Table 10: Exploratory cross-task evaluation. Rows indicate the training task and columns the evaluation task; each cell is the mean score over 256 programs and 256 hidden instances. The Game 2048 column reports the mean merge score instead: the Game 2048 training normalization rescales above-threshold performance within the current rollout group (Appendix C), so it is not comparable across checkpoints. Its diagonal entries therefore coincide with the Game 2048 rows of Table 4. In-distribution entries are shown in bold.

Method	Train \ Eval	TSP	Set Cover	Game 2048	PMS
Coder-Only	TSP	**0.621**	0.172	18.7	0.404
	Set Cover	0.606	**0.327**	23.8	0.450
	Game 2048	0.567	0.147	**27.0**	0.400
	PMS	0.712	0.146	24.8	**0.604**
CURE	TSP	**0.613**	0.241	19.6	0.456
	Set Cover	0.559	**0.504**	29.3	0.466
	Game 2048	0.560	0.143	**24.4**	0.413
	PMS	0.755	0.136	12.8	**0.600**
CCE (Ours)	TSP	**0.629**	0.164	11.1	0.442
	Set Cover	0.567	**0.550**	20.0	0.501
	Game 2048	0.556	0.183	**46.1**	0.411
	PMS	0.712	0.156	12.5	**0.693**

J Exploratory Extension: Distribution-Aware Program Selection

This section explores a possible use of continuous test feedback beyond its role in the training rewards of CCE. Given several candidate programs and a finite set of selection tests, the standard rule ranks programs by their empirical mean score. Continuous outcomes also reveal dispersion and tail behavior. We first quantify the information lost through binarization, then study a moment-based risk-sensitive score and its assumptions. A final case study illustrates how this score can change the selected program when generated selection tests differ from hidden tests. Note that these results are exploratory, and we do not claim that a variance penalty universally improves selection.

J.1 Continuous Feedback Preserves Distinguishing Information

Let $\tau \sim \mathcal{D}$ be a random test drawn from a common test distribution $\mathcal{D}$. For each candidate program s, define

$$X_s = X(s, \tau) \in [0, 1], \qquad P_s = (X(s, \cdot))_{\#}\mathcal{D}.$$

Thus P_s is the score distribution of program s under the test distribution $\mathcal{D}$, and P_a and P_b denote the corresponding distributions for candidates a and b under the same $\mathcal{D}$. Larger scores indicate better performance. Write $\mu_s = \mathbb{E}_{P_s}[X_s]$ and $\sigma_s^2 = \mathrm{Var}_{P_s}(X_s)$. A binary evaluator applies a measurable threshold map $g_t(x) = \mathbf{1}\{x > t\}$ and observes only the pushforward distribution $(g_t)_{\#}P_s$.

Proposition J.1 (Information loss under binarization). *For any two candidate programs a and b such that $P_a \ll P_b$ and $D_{\mathrm{KL}}(P_a \| P_b) < \infty$, and for any measurable binarization g_t,*

$$D_{\mathrm{KL}}(P_a \| P_b) \geq D_{\mathrm{KL}}((g_t)_{\#}P_a \,\|\, (g_t)_{\#}P_b)\,. \tag{5}$$

Equality holds if and only if the binary outcome is sufficient for distinguishing the two score distributions; otherwise, the inequality is strict.

Proof. The binary outcome is a measurable function of the continuous score. Eqn. 5 is therefore the data-processing inequality for KL divergence. Under the stated absolute-continuity and finite-divergence assumptions, equality holds if and only if the likelihood ratio dP_a/dP_b is measurable with respect to the σ-algebra generated by g_t. Equivalently, the likelihood ratio cannot vary within either threshold cell, up to P_b-null sets. □

Corollary J.1 (Asymptotic sample-efficiency consequence). *Consider the simple hypotheses $H_0 : X_i \overset{\text{i.i.d.}}{\sim} P_a$ and $H_1 : X_i \overset{\text{i.i.d.}}{\sim} P_b$. Type-I error means rejecting H_0 under P_a, while type-II error*

means accepting H_0 under P_b. Fix a type-I error tolerance $\alpha \in (0,1)$, and let $n_{\mathrm{cont}}(\varepsilon)$ and $n_{\mathrm{bin}}(\varepsilon)$ denote the minimum numbers of i.i.d. observations required by the optimal tests based on continuous and binarized scores, respectively, to attain type-II error at most ε. If both divergences in Eqn. 5 are positive, then, as $\varepsilon \downarrow 0$, the Chernoff-Stein lemma gives

$$\begin{aligned} n_{\mathrm{cont}}(\varepsilon) &= \frac{\log(1/\varepsilon)}{D_{\mathrm{KL}}(P_a \| P_b)}(1+o(1)), \\ n_{\mathrm{bin}}(\varepsilon) &= \frac{\log(1/\varepsilon)}{D_{\mathrm{KL}}((g_t)_{\#}P_a \| (g_t)_{\#}P_b)}(1+o(1)). \end{aligned} \tag{6}$$

Consequently,

$$\lim_{\varepsilon \downarrow 0} \frac{n_{\mathrm{cont}}(\varepsilon)}{n_{\mathrm{bin}}(\varepsilon)} = \frac{D_{\mathrm{KL}}((g_t)_{\#}P_a \| (g_t)_{\#}P_b)}{D_{\mathrm{KL}}(P_a \| P_b)} \leq 1. \tag{7}$$

The inequality is strict whenever binarization strictly reduces the KL divergence.

Proof. Apply the Chernoff-Stein lemma to the continuous and binarized observation models, then use Prop. J.1. □

If the binarized divergence is zero while the continuous divergence is positive, the two binarized distributions coincide. For every sample size, a binary-data test with type-I error at most α then has type-II error at least $1-\alpha$, whereas continuous observations can distinguish the pair at an exponential rate. Note that this theoretical result concerns distributional discrimination; it does not imply that every variance-aware selector improves empirical-mean ranking when the target objective is exactly μ_s.

J.2 Risk-Sensitive Scores and a Robustness Bound

To illustrate a further use of score distributions, consider the moment-based, risk-sensitive functional

$$Q_\kappa(s) = \mu_s - \kappa\sigma_s, \qquad \kappa \geq 0. \tag{8}$$

For $\kappa > 0$, Cantelli's inequality gives the lower-tail statement

$$\Pr_{P_s}(X_s \geq Q_\kappa(s)) \geq \frac{\kappa^2}{1+\kappa^2}. \tag{9}$$

This is a distribution-free statement about the score threshold $Q_\kappa(s)$; it is uninformative if that threshold is at or below the minimum possible score.

A related robustness bound follows from the χ^2 neighborhood around P_s defined by

$$\mathcal{U}_\rho(P_s) = \left\{ Q \ll P_s : \mathbb{E}_{P_s}\left[\left(\frac{dQ}{dP_s} - 1\right)^2\right] \leq \rho \right\}. \tag{10}$$

Proposition J.2 (Robust lower bound). *For any score distribution P_s with finite variance,*

$$\inf_{Q \in \mathcal{U}_\rho(P_s)} \mathbb{E}_Q[X_s] \geq \mu_s - \sqrt{\rho}\,\sigma_s. \tag{11}$$

For $\rho > 0$ and $\sigma_s > 0$, equality holds if and only if the density ratio

$$w^*(x) = 1 - \sqrt{\rho}\,\frac{x - \mu_s}{\sigma_s}$$

is nonnegative P_s-almost surely. If $\rho = 0$ or $\sigma_s = 0$, both sides equal μ_s.

Proof. Let $w = dQ/dP_s$. Since $\mathbb{E}_{P_s}[w] = 1$,

$$\mathbb{E}_Q[X_s] - \mu_s = \mathbb{E}_{P_s}[(w-1)(X_s - \mu_s)].$$

Cauchy-Schwarz and the definition of $\mathcal{U}_\rho(P_s)$ lower-bound the right-hand side by $-\sqrt{\rho}\,\sigma_s$, proving Eqn. 11. If w^* is nonnegative, it has expectation one, satisfies the divergence constraint with equality, and attains the bound. Conversely, feasible densities form a weakly compact subset of $L^2(P_s)$, so the infimum is attained. For $\rho > 0$ and $\sigma_s > 0$, equality then requires both the Cauchy-Schwarz and divergence bounds to be tight, which forces $w = w^*$ P_s-almost surely; thus w^* must be nonnegative. □

Setting $\kappa = \sqrt{\rho}$ makes $Q_\kappa(s)$ a lower bound on the worst-case expected score in this ambiguity set, but generally not the exact worst-case value. For example, with $\kappa = 1$, a program with score $X_a = 0.5$ almost surely has $Q_1(a) = 0.5$, whereas a program with score distribution $P_b = 0.9\delta_{0.5} + 0.1\delta_1$ has $Q_1(b) = 0.4$, although b first-order stochastically dominates a. Thus ranking by Q_κ need not respect stochastic dominance when the lower bound is loose; we use it as a risk-sensitive score, not as an exact robust objective.

The radius ρ measures the permitted reweighting of the selection-test distribution. One sufficient condition for applying the bound to deployment is at the test level: if selection tests follow $\mathcal{D}$, deployment tests follow $\mathcal{D}' \ll \mathcal{D}$, and $\chi^2(\mathcal{D}'\|\mathcal{D}) \le \rho$, then data processing for χ^2 divergence gives $X(s,\cdot)_\#\mathcal{D}' \in \mathcal{U}_\rho(P_s)$ for every candidate s, where $P_s = X(s,\cdot)_\#\mathcal{D}$ and χ^2 uses the same convention as in Eqn. 10 (without a factor of $1/2$). This condition does not follow merely from observing selection-test scores. The bound is computed separately for each program, while an actual deployment shift is shared across programs.

J.3 General Exponential-Family Estimation

As one possible way to estimate Eqn. 8, suppose the score distribution of each candidate is correctly specified by a regular, identifiable finite-dimensional exponential family

$$p_\eta(x) = h(x)\exp\left(\eta^\top T(x) - A(\eta)\right), \qquad \eta \in \Theta \subseteq \mathbb{R}^d, \tag{12}$$

with $P_s = p_{\eta_s}$ for a true parameter $\eta_s \in \operatorname{int}\Theta$. Define $\mu(\eta) = \mathbb{E}_\eta[X]$, $v(\eta) = \operatorname{Var}_\eta(X)$, and

$$Q_\kappa(\eta) = \mu(\eta) - \kappa\sqrt{v(\eta)}.$$

Given n i.i.d. observed scores for program s, suppose an interior maximum-likelihood estimate $\widehat{\eta}_s$ exists. The direct plug-in selector uses $Q_\kappa(\widehat{\eta}_s)$. To account for parameter-estimation uncertainty, define the likelihood-ratio confidence set

$$\mathcal{C}_s(n,\delta) = \{\eta \in \Theta : nD_{\mathcal{F}}(\widehat{\eta}_s, \eta) \le \beta_d(n,\delta)\}, \tag{13}$$

where

$$D_{\mathcal{F}}(\eta,\eta') = (\eta-\eta')^\top \nabla A(\eta) - A(\eta) + A(\eta')$$

is $D_{\mathrm{KL}}(p_\eta\|p_{\eta'})$. An appropriate finite-sample calibration can ensure $\Pr(\eta_s \in \mathcal{C}_s(n,\delta)) \ge 1-\delta$. Under the standard regularity conditions for Wilks' theorem, including nonsingular Fisher information, the choice $\beta_d = \frac{1}{2}\chi^2_{d,1-\delta}$ gives asymptotic coverage $1-\delta$; it is not in general a finite-sample guarantee. For simultaneous coverage over a fixed number of candidates, one may allocate δ across them.

A conservative distribution-aware score is then

$$\underline{Q}_{\kappa,s}(n,\delta) = \inf_{\eta\in\mathcal{C}_s(n,\delta)} Q_\kappa(\eta). \tag{14}$$

Proposition J.3 (Coverage and consistency). *If* $\Pr(\eta_s \in \mathcal{C}_s(n,\delta)) \ge 1-\delta$, *then*

$$\Pr\Big(\underline{Q}_{\kappa,s}(n,\delta) \le Q_\kappa(\eta_s)\Big) \ge 1-\delta. \tag{15}$$

If, additionally, $\widehat{\eta}_s \to \eta_s$ *in probability,* $\beta_d(n,\delta)/n \to 0$, *the likelihood-ratio sets shrink to* η_s *in probability, and* Q_κ *is continuous at* η_s, *then*

$$\underline{Q}_{\kappa,s}(n,\delta) \xrightarrow{\mathrm{p}} Q_\kappa(\eta_s).$$

Proof. On the coverage event, the true parameter η_s is feasible in Eqn. 14; the infimum can therefore be no larger than $Q_\kappa(\eta_s)$, proving Eqn. 15. Under the remaining assumptions, every parameter in $\mathcal{C}_s(n,\delta)$ converges to η_s. Continuity of Q_κ then implies convergence of the infimum. □

Under their respective modeling assumptions, illustrative plug-in functionals are

$$\begin{aligned} Q_\kappa^{\mathrm{Ber}}(p) &= p - \kappa\sqrt{p(1-p)}, \\ Q_\kappa^{\mathrm{Gauss}}(\mu,\sigma) &= \mu - \kappa\sigma, \\ Q_\kappa^{\mathrm{Beta}}(a,b) &= \frac{a}{a+b} - \kappa\sqrt{\frac{ab}{(a+b)^2(a+b+1)}}. \end{aligned}$$

The Bernoulli form describes binary pass/fail feedback; its variance is determined by the pass rate, and $Q_\kappa^{\mathrm{Ber}}(p)$ need not increase with p when p is small. An all-zero or all-one Bernoulli sample has no finite natural-parameter MLE; in that case, its likelihood-ratio set can instead be written in the mean parameter as $\{p \in [0,1] : n\,\mathrm{kl}(\widehat{p}\|p) \le \beta\}$, where kl is Bernoulli KL divergence with the usual boundary conventions. The Gaussian form is a working approximation on $\mathbb{R}$ that does not enforce $X \in [0,1]$; its Wilks calibration should not be treated as valid under a misspecified Gaussian model, and a constant sample also lacks an interior MLE. The beta family models scores strictly in $(0,1)$, so it requires separate treatment of exact zero and one scores, for example a model with endpoint masses.

J.4 Illustrative Selection with Mismatched Generated and Hidden Tests

We examine how graded test-score dispersion can affect selection when generated and hidden Set Cover tests differ in instance size.

We analyze evaluations of a fixed pool of 256 candidate programs on 64 model-generated tests used for selection (hereafter, selection tests) and 256 hidden tests. The pool was generated by an earlier CURE-baseline checkpoint used only for this exploratory analysis. The 256 hidden tests used here were drawn separately for this analysis and are substantially larger than the hidden tests behind the main results. Accordingly, the absolute values in this subsection are not directly comparable with the main-table results. In this pool, we observed a case in which accounting for score dispersion changes the selected program. Four selection tests are infeasible because their sets do not cover the universe; the remaining 60 yield graded scores in $[0,1]$ for each candidate. For each candidate-subset size $N \in \{4, 16, 64, 256\}$, we compare selection by the mean selection-test score with selection by the plug-in moment score mentioned above

$$\widehat{Q}_\kappa(s) = \widehat{\mu}_s - \kappa\widehat{\sigma}_s, \tag{16}$$

where $\widehat{\mu}_s$ and $\widehat{\sigma}_s$ are the mean and standard deviation of the 60 selection-test scores, and $\widehat{\sigma}_s^2$ is computed with divisor 60. For nonconstant samples, this coincides with the Gaussian maximum-likelihood plug-in; however, we compute it directly from sample moments, without fitting a Gaussian model or using the confidence-set score in Eqn. 14. We report $\kappa = 1.5$ based on a post-hoc sweep over $\kappa \in \{0, 0.1, 0.25, 0.5, 0.75, 1, 1.5, 2, 3\}$. For the full pool ($N = 256$), the hidden-test mean was 0.170 for every tested $\kappa \le 1$ and 0.904 for every tested $\kappa \ge 1.5$, corresponding to a change in the selected program between $\kappa = 1$ and $\kappa = 1.5$. The latter is the smallest value on this grid that changes the full-pool selection. We therefore treat it as a descriptive post-hoc choice rather than a calibrated confidence or robustness bound.

For $N < 256$, each table entry reports the average hidden-test score of the selected program over 1,000 uniformly sampled subsets of N distinct programs from the same fixed pool. Ties in selection scores are resolved by a fixed program order. For $N = 256$, each rule is applied once to the full fixed pool. The hidden oracle selects the program with the highest hidden-test mean within each subset and is unavailable to a practical selector.

Table 11: Set Cover selection within one fixed candidate pool. For $N < 256$, entries average the selected program's hidden-test mean over 1,000 sampled subsets; the $N = 256$ column reports selection from the full pool.

Selection rule	$N = 4$	$N = 16$	$N = 64$	$N = 256$
Empirical mean	0.576	0.730	0.508	0.170
$\widehat{\mu} - 1.5\widehat{\sigma}$	0.592	0.826	0.903	0.904
Hidden oracle	0.629	0.874	0.912	0.930

The selection and hidden test sets also differ substantially in instance size: the selection tests have median universe size 6 and median set count 4, whereas the corresponding medians for the hidden tests are 11 and 22. This mismatch is a property of this subsection only, and it has not been shown to satisfy the test-level χ^2 condition above; Prop. J.2 therefore does not guarantee performance on these hidden tests. In the full pool ($N = 256$), selection by the mean selection-test score chooses a program with mean scores of 0.966 on the selection tests and 0.170 on the hidden tests. The moment score instead selects a program with a hidden-test mean of 0.904. Within this particular pool, the

observed gap between the two rules is larger at larger N (Table 11). This comparison uses one candidate pool and two fixed test sets; the 1,000 subsets vary only the candidate composition within this fixed pool.

This case study illustrates one setting in which continuous score dispersion provides useful information for program selection when generated and hidden tests are mismatched.