BenchEvolver: Frontier Task Synthesis via Solution-Centric Evolution

Yangzhen Wu[*,1] **Aaron J. Li**[*,1] **Wenjie Ma**[†,1] **Li Cao**[2] **Ziheng Zhou**[2]
Mert Cemri[1] **Shu Liu**[1] **Yuran Xiu**[2] **Chenxiao Yan**[2] **Haikun Zhao**[2]
Bin Yu[1] **Ion Stoica**[‡,1] **Dawn Song**[†,1]
[1]University of California, Berkeley
[2]Institute for Interdisciplinary Information Sciences, Tsinghua University
yangzhen_wu@berkeley.edu aaronjli@berkeley.edu

Project Page | Code | Dataset

Abstract

The rapid progress of frontier large language models has led to widespread benchmark saturation, limiting the ability of existing datasets to differentiate model capabilities or provide useful training signal. For instance, on LiveCodeBench, frontier models achieve over 99% Pass@1 on easy splits and exceed 90% on average across difficulty levels [1–3]. We introduce BenchEvolver, a solution-centric evolutionary framework that automatically transforms existing coding problems into substantially harder variants. By evolving reference solutions and grounding task generation in executable semantics, BenchEvolver enables scalable construction of diverse, challenging, and verifiable tasks. Applied to LiveCodeBench and SciCode, BenchEvolver produces substantially harder tasks while preserving validity and reference correctness. We further curate LIVECODEBENCH-PLUS, a 91-problem benchmark on which frontier-model Pass@1 ranges from 27.5% to 62.6%, restoring discrimination among strong coding models. Importantly, evolved tasks remain challenging even for the models that generate them: in a generation–training cycle, reinforcement learning on self-generated evolved tasks improves the generating model's held-out coding performance beyond training on the original seed pool. Together, these results show that benchmark evolution can both refresh frontier evaluation and supply reusable training challenges, providing a task-generation mechanism for recursive self-improvement.

1 Introduction

Recent advances in frontier large language models (LLMs) are rapidly saturating widely used evaluation benchmarks, reducing their ability to meaningfully measure progress or provide useful training signal. On competitive coding benchmarks such as LiveCodeBench (LCB) [1], state-of-the-art models now achieve over 99% Pass@1 on the newest easy split and exceed 90% on average across difficulty levels. As models continue to improve, static benchmarks inevitably lose discriminative power, creating an increasing need for new tasks that are both sufficiently challenging and reliably verifiable. Constructing such tasks manually, however, requires substantial expert effort and is difficult to scale.

[*]Equal contribution.
[†]Core contributor.
[‡]Equal advising.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

A natural alternative is to use LLMs themselves to generate new evaluation and training data. Existing methods have made substantial progress in synthesizing coding instructions, solutions, reasoning traces, and complete programming tasks [4–11]. However, two challenges remain for using synthetic data to refresh saturated benchmarks. First, many approaches follow a teacher–student paradigm in which a stronger model generates or verifies data for weaker models, rather than producing tasks that remain challenging for the generator itself. Second, generating a difficult-looking instruction is not sufficient for benchmark construction: a usable benchmark item requires a coherent statement, a correct reference solution, and tests that jointly define the intended task. These requirements make self-challenging task generation substantially harder than ordinary instruction synthesis.

We focus on executable coding tasks, where correctness can be grounded in program execution, and introduce BenchEvolver, a solution-centric evolutionary framework for transforming saturated problems into harder, verifiable variants. Instead of first generating a new statement and subsequently attempting to recover a consistent solution and test suite, BenchEvolver evolves the computation underlying an existing task. Conditioned on the parent problem and its reference implementation, the Proposer introduces a substantive algorithmic change to the reference solution and then derives a corresponding statement and tests around the evolved computation. An independent Evaluator checks consistency among the statement, reference solution, tests, and execution harness before measuring difficulty through actual target-model performance. Candidates are retained only when they remain valid while becoming empirically harder for the target models. Memory over previous successes and failures further guides evolution toward new algorithmic directions and away from repeated or invalid mutations.

This solution-centric design provides two complementary benefits. First, it offers a scalable mechanism for refreshing saturated evaluations. We apply BenchEvolver to LiveCodeBench [1] and SciCode [12], covering competitive programming and scientific function-level coding. Across both settings, the evolved problems remain executable and well specified while substantially reducing target-model pass rates. Controlled comparisons further show that solution-centric generation improves task yield and validity over problem-centric alternatives, while memory-guided evolution improves over independent mutations. We use the resulting tasks to construct LIVECODEBENCH-PLUS, a difficulty-upgraded benchmark that restores clear performance separation among frontier coding models.

Second, evolved tasks can provide useful training signal for the models that generate them. Using `gpt-oss-20b` as both the evolver and the model being trained, we show that reinforcement learning on evolved tasks improves held-out coding performance beyond training on the original seed pool alone. This connects benchmark evolution to recursive self-improvement: a model constructs challenges that expose its current weaknesses, learns from those challenges, and can then use its improved capabilities to generate the next round of tasks. Our experiments demonstrate one complete generation–training cycle, providing evidence for a central component of this process: self-generated challenges can improve the model that produced them.

Our contributions are summarized as follows:

- We introduce BenchEvolver, a solution-centric evolutionary framework that transforms saturated executable coding tasks into harder variants grounded in reference implementations and verifiable tests.
- We demonstrate that BenchEvolver produces valid, diverse, and substantially harder tasks across LiveCodeBench and SciCode, and show through controlled comparisons and human evaluation that the gains reflect meaningful algorithmic evolution rather than superficial complexity.
- We construct LIVECODEBENCH-PLUS, a 91-problem difficulty-upgraded benchmark that provides substantially greater discrimination among frontier coding models.
- We demonstrate a generation–training cycle toward recursive self-improvement: a model generates harder tasks from reliably solved source problems and learns from them through reinforcement learning, achieving stronger held-out coding performance than training on the original seed pool alone.

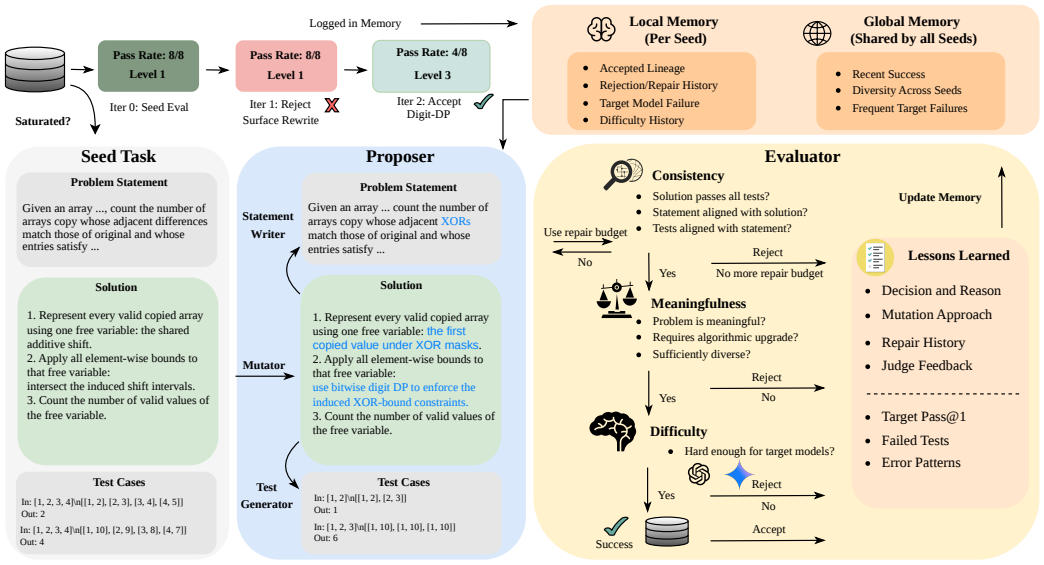


Figure 1: Overview of BenchEvolver. Starting from a saturated seed, the Proposer evolves its reference solution and derives a corresponding statement and tests. The Evaluator verifies consistency and measures target-model difficulty. Accepted tasks become new parents, while outcomes from both accepted and rejected proposals are stored in memory to guide later evolution.

2 Related Work

Our work connects synthetic coding data, self-guided improvement, and evolutionary search; we provide a more comprehensive discussion in Appendix A. Prior work synthesizes code instructions, reasoning traces, repository-level bug-fix tasks, competitive-programming problems, and RL curricula to improve code models [4, 5, 7–11, 13]. A complementary line of work uses model-in-the-loop data generation to target current model weaknesses, including self-challenging tool-use agents, solver–conjecturer self-play, code-centric data synthesis, and adversarial co-evolution between code and test generators [14–18]. Finally, LLM-based evolutionary methods have been used for prompt optimization, program discovery, algorithm search, and self-evolving agents [19–25]. BenchEvolver differs in the object being evolved: rather than generating auxiliary supervision or optimizing solutions for a fixed task, it evolves complete executable benchmark items—statements, reference solutions, and tests—selected by empirical target-model failure. This turns inference-time search [26] into reusable tasks for both frontier-model evaluation and subsequent model improvement.

3 Method

We introduce BenchEvolver, an evolutionary framework for transforming saturated executable programming tasks into harder, verifiable variants. The framework has three components: a *Proposer* that evolves the computation underlying a task, an *Evaluator* that checks task consistency and measures empirical difficulty, and a *Memory* that guides subsequent proposals using previous successes and failures. Figure 1 provides an overview.

3.1 Problem Setting

We consider executable programming tasks

$$I = (S, C, T, E),$$

where S is the problem statement, C a reference implementation, T a hidden test suite, and E an execution harness that determines whether a submitted program passes the task. BenchEvolver

applies to settings with (i) an executable reference solution and (ii) an automatic execution-based correctness signal.

The framework separates a shared evolution procedure from a small domain-specific validation interface. The Proposer, memory, difficulty measurement, selection rule, and repair loop are shared across domains. The execution harness E and the consistency check used to validate a newly generated task are instantiated for each benchmark. For example, LiveCodeBench uses stdin–stdout execution and brute-force triangulation, whereas SciCode uses function-level assertions and a statement-faithfulness check. Thus, benchmark differences affect how candidate correctness is verified, not the evolution or difficulty-selection mechanism.

Let $\Pi = \{\pi_j\}_{j=1}^{J}$ denote a target-model panel and K the number of attempts per model. We measure task difficulty behaviorally using

$$a(I; \Pi, K) = \frac{1}{JK} \sum_{j=1}^{J} \sum_{k=1}^{K} \mathbf{1}\Big[E(\hat{C}_{j,k}, T) = 1\Big],$$

where $\hat{C}_{j,k} \sim \pi_j(S)$. Lower empirical pass rate corresponds to higher difficulty. Difficulty is therefore determined by executable model performance, rather than by the generator or an LLM difficulty judge.

3.2 Proposer: Solution-Centric Evolution

The Proposer evolves the computation of an existing task before constructing its new specification. Importantly, evolution is *not* performed from the solution in isolation: the mutator conditions on both the parent statement and reference solution. For parent $I_p = (S_p, C_p, T_p, E)$ and memory context h,

$$C' \sim q_\theta(\cdot \mid S_p, C_p, h), \qquad S' \sim w_\theta(\cdot \mid S_p, C', h), \qquad T' \sim g_\theta(\cdot \mid S', C', h).$$

The resulting candidate is $I' = (S', C', T', E)$.

The mutator is instructed to introduce a *dominant algorithmic lift*: a computational change that makes the parent solution strategy insufficient. Such changes may require, for example, a stronger asymptotic algorithm, a new data structure, or a different mathematical or structural formulation. This constraint is intended to steer generation toward substantive computational changes rather than longer statements, adversarial formatting, or arbitrary edge cases.

The evolved reference implementation C' anchors the remaining artifacts but does not by itself establish task validity. Public outputs and test answers are obtained by executing C', while the statement and tests are subsequently checked independently by the Evaluator. The Proposer therefore proposes complete executable candidates; it does not decide whether they are valid or sufficiently difficult.

3.3 Evaluator: Consistency, Quality, and Difficulty

The Evaluator separates three questions that can otherwise be conflated: whether a task is *executable and internally consistent*, whether its difficulty is *meaningful rather than artificial*, and whether it is *empirically harder*.

Executable consistency. A candidate must first pass an independent consistency check between its statement, reference solution, tests, and execution harness. For LiveCodeBench, we automatically triangulate among the evolved reference solution, a brute-force solver synthesized from the statement alone, and a statement-only output oracle. Output disagreements provide concrete counterexamples and trigger targeted repair; brute-force timeouts on large instances are treated as infeasibility rather than disagreement. For SciCode, where brute force is generally inappropriate, an independently generated statement-only solution is executed against the candidate tests; low agreement indicates that the statement does not sufficiently specify the intended computation and triggers revision or rejection. Full procedures are given in Appendix C.

These checks are automatic and benchmark-specific only through the validation interface. They do not assume that brute-force verification is available for every programming domain.

Task quality. Consistency alone does not guarantee a useful benchmark item. The Evaluator therefore rejects candidates whose apparent difficulty arises from ambiguity, underspecified conventions, adversarial I/O, contrived edge cases, or near-duplicate reskins of existing tasks. We distinguish this quality filtering from executable correctness; complementary post-hoc agentic and human audits are reported in Section 4.

Empirical difficulty. Only candidates that pass consistency and quality checks are evaluated by the target panel. Each target model receives K independent attempts, and difficulty is computed using the empirical pass rate above. A candidate is accepted only when it achieves the configured difficulty gain over its seed and current parent while remaining within an informative, solvable regime. Accepted candidates may then become parents for subsequent mutations.

Target-model failure alone is insufficient for acceptance: candidates must first pass the consistency and quality checks. These checks reduce the risk of mistaking invalid or underspecified tasks for genuinely difficult ones; independent post-hoc audits assess the remaining errors.

3.4 Memory-Guided Search

Memory makes evolution adaptive rather than independent repeated sampling. Each seed maintains a *local memory* containing both accepted and rejected attempts, including the proposed mutation idea, validation or rejection reason, repair history, target-model pass rates, and observed model errors. As the lineage grows, these records are summarized into a running digest together with a small window of recent attempts, which is provided to the Proposer on subsequent iterations. This encourages the search to avoid previously failed directions and build on successful or near-successful mutations.

A *global memory* stores compact summaries of accepted mutations across seeds, including their algorithmic idea and achieved difficulty. These summaries provide diversity context to the Proposer so that new lineages do not repeatedly rediscover the same transformation. Reused mutation families are also subject to a stricter difficulty requirement, providing diversity pressure at selection time rather than relying only on prompting.

Memory never determines validity or difficulty: validity remains execution-grounded, and difficulty remains measured from target-model pass rates. It only changes which candidate mutations are explored and how repetitive proposals are treated.

Together, these components implement solution-centric evolutionary search over executable tasks: proposals modify the underlying computation, consistency and quality checks help distinguish genuine difficulty from invalidity, empirical model failure provides the selection signal, and memory guides search across iterations. The resulting tasks can be used either as refreshed evaluation items or as training data for subsequent model improvement. Full pseudocode and implementation details are provided in Appendix B.

4 Experiments

We evaluate BenchEvolver along three dimensions. First, we study whether it can generate valid and empirically harder tasks across two executable coding domains: competitive programming and scientific coding, with diversity further assessed through human evaluation. Second, we construct LIVECODEBENCH-PLUS, a challenging benchmark for frontier coding models. Third, we test whether evolved tasks provide useful reinforcement- learning signal for improving the same model that generates them.

4.1 Task-Evolution Evaluation

We evaluate whether BenchEvolver can generate valid and empirically harder tasks across two executable coding settings: competitive programming and scientific coding. We consider a *lightweight* target tier consisting of GPT-5.4-mini and Gemini-3-Flash, and a *frontier* tier consisting of GPT-5.4 and Gemini-3.1-Pro. Full evolution configurations are provided in Appendix E.

Method	Target	Evolver	Easy	Medium	Hard	Audit Pass (O/G)	Agree.
AutoCode	Light.	GPT-5.4-mini	12/22	11/23	6/20	73.7/69.7	93.4
Problem-Centric	Light.	GPT-5.4-mini	15/22	11/23	10/20	84.9/83.6	93.2
Memory-Free	Light.	GPT-5.4-mini	13/22	15/23	9/20	88.6/89.9	96.2
BenchEvolver	Light.	GPT-5.4-mini	**17/22**	**16/23**	**12/20**	**97.7/95.4**	95.4
BenchEvolver	Light.	Gemini-3-Flash	18/22	14/23	16/20	94.9/90.9	91.9
BenchEvolver	Light.	Claude-Sonnet-4.6	14/22	15/23	7/20	95.3/90.7	93.0
BenchEvolver	Frontier	Gemini-3.1-Pro	8/10	9/10	6/10	97.9/93.6	91.5

Table 1: LiveCodeBench-v6 evolution yield and post-hoc audit results. Easy, Medium, and Hard report the number of seeds yielding at least one accepted evolved problem. Audit Pass reports the fraction of accepted problems passing an independent post-hoc audit by Claude Code with Opus-4.8 (O) and Codex with GPT-5.6-sol (G); Agree. is their binary agreement rate. Bold indicates the best result within the matched GPT-5.4-mini ablation comparison.

4.1.1 Competitive Programming: LiveCodeBench

For the controlled LiveCodeBench-v6 study, we sample 65 seed problems spanning easy, medium, and hard difficulty levels. We use GPT-5.4-mini, Gemini-3-Flash, and Claude-Sonnet-4.6 as evolvers under the lightweight target tier. For frontier-target evolution, we additionally use Gemini-3.1-Pro on 10 saturated seeds from each difficulty level. All candidates must first pass the executable consistency checks in Appendix C.1 before target-model difficulty is measured.

Solution-centric generation and memory improve evolution. We compare methods along two axes: problem-centric versus solution-centric generation, and memory-free versus memory-guided evolution. AUTOCODE [11] denotes our implementation of the closest directly comparable problem-centric synthesis baseline without iterative memory. PROBLEM-CENTRIC adds the same memory mechanism used by BenchEvolver while retaining problem-centric generation, whereas MEMORY-FREE uses solution-centric generation without memory.

Under the same GPT-5.4-mini evolver, solution-centric generation improves both evolution yield and post-hoc audit pass rates over problem-centric alternatives, while memory further improves yield over the memory-free variant. The audit scores in Table 1 are independent post-hoc evaluations rather than acceptance criteria; task acceptance uses the executable consistency checks in Section 3.3.

Evolved tasks are substantially harder. For post-hoc difficulty evaluation, we re-evaluate seed and evolved problems using $k = 8$ independent attempts per model. Across LiveCodeBench configurations, evolution reduces Pass@1 by 32.4 percentage points on average. The decrease is consistent across target models, evolvers, and model providers. Detailed per-model results and difficulty plots are provided in Appendix F. We further evaluate transfer to models outside the selection panel in Section 4.2.

Human evaluation. A blinded study with six competitive-programming experts further finds that evolved tasks are substantially harder and more novel while remaining clear, with broader algorithmic coverage than their source problems. Full results are reported in Appendix G.

4.1.2 Scientific Coding: SciCode

We next evaluate whether the same solution-centric evolution procedure transfers to scientific function-level coding. We select 30 self-contained SciCode subproblems spanning 15 main problems from the validation split; 27 are saturated under the lightweight target tier and 28 under the frontier tier. Because SciCode does not naturally admit brute-force validation, candidate consistency is checked using the statement-faithfulness procedure in Appendix C.2.

Despite its different task format and validation procedure, SciCode exhibits the same pattern: BenchEvolver achieves high independent audit pass rates while reducing Pass@1 by 28.9 percentage points on average under the $k = 8$ post-hoc evaluation. Detailed results are provided in Appendix F. This provides evidence that the shared solution-centric evolution and empirical selection procedure transfers across two distinct executable coding settings, while allowing the consistency validator to adapt to each benchmark.

Target Tier	Evolver Model	Evolved Seed Fraction	Audit Pass (O)	Audit Pass (G)	Agree.
Lightweight	GPT-5.4-mini	24/27 (88.9%)	89.8%	91.8%	89.8%
Lightweight	Gemini-3-Flash	26/27 (96.3%)	90.6%	88.7%	94.3%
Lightweight	Claude-Sonnet-4.6	8/27 (29.6%)	100.0%	93.3%	93.3%
Frontier	Gemini-3.1-Pro	17/28 (60.7%)	97.0%	97.0%	100.0%

Table 2: SciCode evolution yield and post-hoc audit results. Evolved Seed Fraction reports the fraction of saturated seeds yielding at least one accepted evolved problem. Audit Pass reports the fraction of accepted problems passing an independent post-hoc audit by Claude Code with Opus-4.8 (O) and Codex with GPT-5.6-sol (G); Agree. is their binary agreement rate.

Evolution efficiency. Across the broader evolution runs, 1,002 candidates from 220 seeds reach target-model evaluation and 294 are accepted, giving a 29.3% proposal-level acceptance rate. BenchEvolver produces 1.34 accepted tasks per seed on average, with 71% of seeds yielding at least one accepted task. With four parallel workers, evolution takes approximately 15–46 minutes per seed and costs roughly $1.5 with lightweight evolvers and $10 with the frontier evolver.

4.2 LIVECODEBENCH-PLUS: A Benchmark for Frontier Coding Models

We next apply BenchEvolver to saturated LiveCodeBench-v6 problems to construct a difficulty-upgraded benchmark for frontier coding models, which we call LIVECODEBENCH-PLUS.

Evolution and filtering. Among 52 Medium problems evaluated against the lightweight target tier, 31 are saturated and 23 yield at least one accepted evolved problem. Among 80 Hard problems evaluated against the frontier tier, 57 are saturated and 43 yield at least one accepted evolved problem. We evolve the Medium split using Gemini-3-Flash and the Hard split using Gemini-3.1-Pro, following the same configurations as Section 4.1. This produces 35 validated evolved Medium tasks and 55 evolved Hard tasks.

To construct a high-quality evaluation set, each evolved problem must first pass the executable validation pipeline in Appendix C.1 and is then reviewed by human evaluators. We retain problems satisfying two additional criteria:

- **Quality gate**: a quality and novelty score of at least 3 on a 1–5 scale under our competitive-programming review rubric.
- **Difficulty range**: aggregate Pass@1 across the evaluation suite between 0.05 and 0.75, excluding near-zero-pass tasks with little comparative signal and tasks that are too easy to discriminate among strong models.

After filtering, 64 evolved problems remain: 20 from the Medium split and 44 from the Hard split. We combine them with 27 difficult original LCB-v6 problems satisfying the same difficulty ceiling, yielding the final 91-problem LIVECODEBENCH-PLUS benchmark.

Difficulty, discrimination, and ranking consistency. We evaluate the resulting tasks on eight models from multiple providers. Table 3 shows that evolution substantially increases difficulty across all evaluated models. Averaged across models, Pass@1 drops from 96.5% to 69.6% on the Medium split and from 87.0% to 45.7% on the Hard split, corresponding to absolute decreases of 26.8 and 41.3 percentage points, respectively. The decrease holds for every model in the evaluation suite, including models not used during evolution. On the final LIVECODEBENCH-PLUS benchmark, Pass@1 ranges from 27.5% to 62.6%, yielding a wide 35.1-point spread across the evaluated model suite despite substantial saturation on the source problems.

Importantly, the increased difficulty does not discard the broad model ordering of the source benchmark. After aggregating the Medium and Hard source-seed scores weighted by their numbers of source problems, the model ranking on LIVECODEBENCH-PLUS has a Spearman rank correlation of $\rho = 0.76$ with the corresponding source-set ranking across the eight evaluated models. The top four models retain the same ordering, while several lower-ranked models are reordered. Thus, LIVECODEBENCH-PLUS preserves substantial ranking consistency with the source tasks while exposing additional differences among models in the harder regime.

	Medium (23 seeds / 35 evolved)			**Hard** (43 seeds / 55 evolved)			**LCB-Plus**
Model	Seed	Evolved	Δ	Seed	Evolved	Δ	(91 problems)
GPT-5.5	100.0	80.0	-20.0	97.1	62.3	-34.8	62.6
GPT-5.4	98.9	74.3	-24.6	94.8	49.7	-45.1	54.1
GPT-5.4-mini	95.7	59.3	-36.4	79.7	21.7	-58.0	29.6
Gemini-3.1-Pro	100.0	78.6	-21.4	96.5	56.8	-39.7	59.1
Gemini-3.5-Flash	95.7	73.6	-22.1	87.8	50.5	-37.3	50.0
Gemini-3-Flash	88.0	67.1	-20.9	82.0	43.2	-38.8	40.1
DeepSeek-V4-Pro	95.7	57.1	-38.6	83.7	23.2	-60.5	27.5
Qwen-3.7-Max	97.8	67.1	-30.7	74.4	58.6	-15.8	47.5

Table 3: Pass@1 (%) on LiveCodeBench-v6 source seeds, their evolved variants, and the 91-problem LIVECODEBENCH-PLUS benchmark. Δ denotes the absolute seed-to-evolved change. Results use $k = 4$ attempts per model. GPT models use medium reasoning effort, Gemini models use adaptive reasoning, and DeepSeek-V4-Pro uses high thinking mode; all API calls use a 600-second timeout.

4.3 Self-Improvement through Reinforcement Learning

The results above show that BenchEvolver can transform saturated problems into verified tasks that expose failures of current models. We next ask whether these evolved tasks also provide useful training signal for improving the model that generated them. Specifically, we study whether training on evolved problems improves held-out performance and whether these problems complement the original seed distribution.

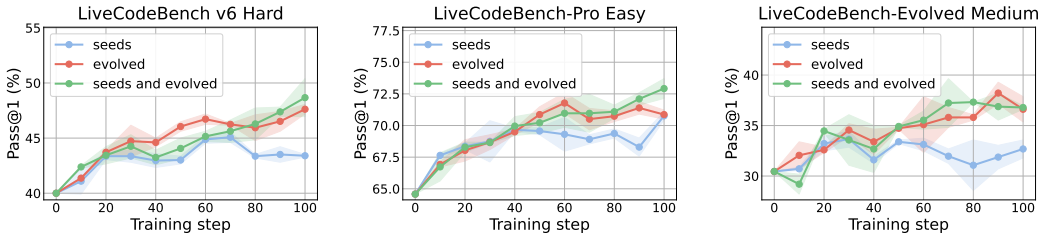
Setup. We use `gpt-oss-20b` as both the evolver and the model being trained. Our seed pool contains 880 LiveCodeBench v1–v5 problems released before January 2025, while LiveCodeBench v6 and LiveCodeBench-Pro are held out for evaluation. We apply BenchEvolver only to seeds that the base model solves correctly in all five initial attempts, treating these as empirically saturated problems. For each eligible seed, we run up to 10 evolution iterations and accept at most two evolved problems, requiring accepted tasks to lower the model's empirical accuracy while retaining a nonzero success rate. This yields 586 evolved problems from 404 successfully evolved source seeds.

We compare three training conditions: the 880 original seeds, the 586 evolved problems, and their union of 1,466 problems. Comparing the original seeds with the union tests whether evolved data improves the practical training mixture; evolved-only training tests whether the generated tasks provide useful training signal on their own. All conditions use the same `gpt-oss-20b` initialization and GRPO configuration in Tinker, with 64 problems per batch, 16 rollouts per problem, and two independent random seeds. We train for 100 RL steps, with maximum output lengths of 24K tokens during training and 30K tokens during evaluation.

We evaluate on LiveCodeBench v6 Hard (80 problems) and LiveCodeBench-Pro Easy (96 problems), two held-out splits within an informative difficulty range for the base model. We additionally evaluate on LCB-EVOLVED Medium, an independently constructed evolved evaluation set generated with Gemini-3-Flash rather than `gpt-oss-20b`. This split is not used in RL training and tests transfer to tasks produced by a different evolver. We estimate Pass@1 by averaging the correctness of 16 samples per problem. Full training details are provided in Appendix D.

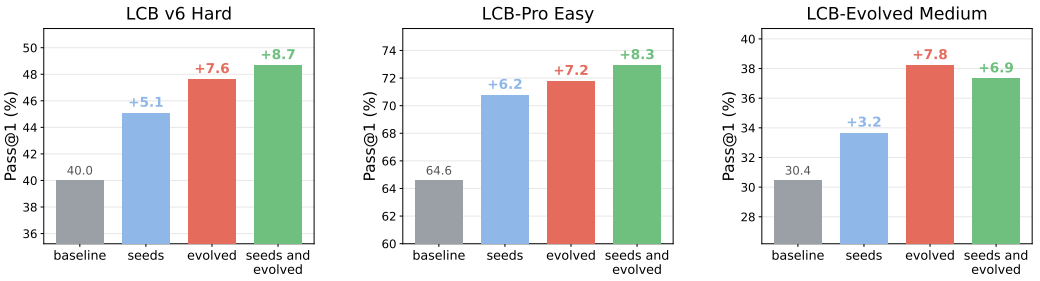
Evolved tasks improve held-out coding performance. Figure 2 reports the full training trajectories, and Figure 3 summarizes the peak observed performance of each training condition. On LCB v6 Hard, seed-only training improves Pass@1 from 40.0% to a peak of 45.1%, whereas evolved-only training reaches 47.6% and the seed+evolved mixture reaches 48.7%. These correspond to additional gains of 2.5 and 3.6 percentage points, respectively, over the seed-only peak. The same pattern holds on LCB-Pro Easy: seed-only training improves from 64.6% to 70.8%, while evolved-only and seed+evolved training reach 71.8% and 72.9%.

The advantage of augmentation is also present at the fixed 100-step endpoint. On LCB v6 Hard, seed+evolved training reaches 48.7%, compared with 43.4% for seed-only training; on LCB-Pro Easy, the corresponding results are 72.9% and 70.8%. Thus, the benefit of adding evolved tasks is not limited to selecting favorable checkpoints. Together, these results suggest that evolved problems



(a) LCB v6 Hard (b) LCB-Pro Easy (c) LCB-EVOLVED Medium

Figure 2: Pass@1 versus RL training step for three training data conditions. Curves show the mean and shaded regions show ± one standard deviation over two independent random seeds. Step 0 corresponds to the base model. Pass@1 is estimated using 16 samples per problem. LCB-EVOLVED Medium is constructed independently using Gemini-3-Flash as the evolver and is not used in RL training.



(a) LCB v6 Hard (b) LCB-Pro Easy (c) LCB-EVOLVED Medium

Figure 3: Peak observed Pass@1 for each training condition. For each evaluation set and condition, we select the checkpoint with the highest mean across two random seeds. Labels show gains over the base model in percentage points. Peaks may occur at different training steps; the y-axes are truncated.

provide complementary training signal beyond the original seed pool, with the combined mixture achieving the strongest observed performance on both public held-out benchmarks.

Transfer to an independently evolved benchmark. The same training runs also improve performance on LCB-EVOLVED Medium, whose tasks are generated by Gemini-3-Flash. The base model achieves 30.45% Pass@1, and seed-only training reaches a peak of 33.66%. In contrast, evolved-only training reaches 38.22%, improving over the base model by 7.77 percentage points and over the seed-only peak by 4.56 points. The seed+evolved mixture reaches 37.32%.

Unlike the public held-out benchmarks, evolved-only training achieves the highest observed peak on this evaluation set. This pattern suggests that the relative benefit of retaining the original seeds depends on the evaluation distribution: the combined mixture performs best on the public benchmarks, while evolved-only training is particularly effective on evolved challenges. Because the latter evaluation set uses a different evolver, these results provide evidence that the utility of self-generated training data extends beyond tasks produced by the training evolver itself.

From saturated problems to useful training signal. One plausible explanation follows from the group-relative RL objective. When all rollouts for a problem receive identical rewards, their group-relative advantages are zero, so the group provides no reward-differential policy-gradient signal. Problems that the model already solves reliably are more likely to produce such groups. By selecting harder variants with nonzero empirical success rates, BenchEvolver aims to produce tasks on which successes and failures coexist, creating opportunities for informative relative advantages. Establishing this mechanism directly would require measuring reward variation during training; the present experiments establish the downstream utility of the evolved data.

Toward recursive self-improvement. These results demonstrate a complete generation–training cycle: `gpt-oss-20b` uses inference-time computation to construct verified challenges from problems it already solves reliably, and reinforcement learning on those challenges improves its held-out coding performance. BenchEvolver therefore provides a task-generation mechanism for recursive self-improvement, in which an updated model can become the evolver for the next round and construct new tasks targeted to its remaining weaknesses. Our experiments establish the benefit of one such cycle; sustained improvement across repeated cycles remains an open question. Evolved tasks thus serve both as evaluation items that reveal model weaknesses and as reusable training data for addressing them.

5 Conclusion and Future Work

Conclusion. We introduced BenchEvolver, a solution-centric framework that evolves executable reference solutions and derives corresponding statements and tests to construct harder coding tasks. Experiments on LiveCodeBench and SciCode show that the resulting tasks increase empirical difficulty while achieving high post-hoc audit pass rates. We further construct LIVECODEBENCH-PLUS, a 91-problem benchmark for frontier coding models, and demonstrate that self-generated evolved tasks improve held-out performance through reinforcement learning. These results connect benchmark renewal with model improvement: tasks that expose current model failures can also provide reusable training signal.

Limitations. Execution-based consistency checks and model-assisted audits do not guarantee semantic correctness, and some accepted tasks fail post-hoc review. Task difficulty and benchmark composition also depend on the models and sampling budgets used during selection. Our self-improvement experiments cover one model, a single generation–training cycle, and two random seeds per condition. They match rollout counts and training steps rather than total generated tokens, and do not establish the behavior of repeated self-improvement cycles.

Future work. A central next step is to extend the demonstrated generation–training cycle into recursive self-improvement. After each training round, the updated model would generate new challenges from tasks it now solves reliably, adapting the training distribution as its capabilities change. Studying this process requires tracking held-out performance across rounds, accounting for both task-generation and training costs, and controlling task difficulty and diversity to avoid repeatedly targeting a narrow set of weaknesses.

A complementary direction is to maintain versioned benchmarks through periodic task generation, validation, and human auditing, with documented selection panels and held-out evaluations. Together, these directions connect adaptive evaluation with an ongoing source of training challenges for improving models.

References

[1] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[2] OpenAI. Introducing gpt-5.5. `https://openai.com/index/introducing-gpt-5-5/`, 2026.

[3] DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

[4] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

[5] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions, 2025. URL `https://arxiv.org/abs/2304.12244`.

[6] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct, 2024. URL `https://arxiv.org/abs/2312.02120`.

[7] Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. Selfcodealign: Self-alignment for code generation. *Advances in Neural Information Processing Systems*, 37:62787–62874, 2024.

[8] Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning: Advancing data distillation for competitive coding. *arXiv preprint arXiv:2504.01943*, 2025.

[9] Minh VT Pham, Huy N Phan, Hoang N Phan, Cuong Le Chi, Tien N Nguyen, and Nghi DQ Bui. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *arXiv preprint arXiv:2504.14757*, 2025.

[10] Cansu Sancaktar, David Zhang, Gabriel Synnaeve, and Taco Cohen. A deep dive into scaling rl for code generation with synthetic data and curricula. *arXiv preprint arXiv:2603.24202*, 2026.

[11] Shang Zhou, Zihan Zheng, Kaiyuan Liu, Zeyu Shen, Zerui Cheng, Zexing Chen, Hansen He, Jianzhu Yao, Huanzhi Mao, Qiuyang Mang, et al. Autocode: Llms as problem setters for competitive programming. *arXiv preprint arXiv:2510.12803*, 2025.

[12] Minyang Tian, Luyu Gao, Shizhuo D Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, et al. Scicode: A research coding benchmark curated by scientists. *Advances in Neural Information Processing Systems*, 37:30624–30650, 2024.

[13] Jie Wu, Haoling Li, Xin Zhang, Jiani Guo, Jane Luo, Steven Liu, Yangyu Huang, Ruihang Chu, Scarlett Li, and Yujiu Yang. X-coder: Advancing competitive programming with fully synthetic tasks, solutions, and tests. *arXiv preprint arXiv:2601.06953*, 2026.

[14] Yifei Zhou, Sergey Levine, Jason Weston, Xian Li, and Sainbayar Sukhbaatar. Self-challenging language model agents. *arXiv preprint arXiv:2506.01716*, 2025.

[15] Luke Bailey, Kaiyue Wen, Kefan Dong, Tatsunori Hashimoto, and Tengyu Ma. Scaling self-play with self-guidance. *arXiv preprint arXiv:2604.20209*, 2026.

[16] Qiushi Sun, Jinyang Gong, Lei Li, Qipeng Guo, and Fei Yuan. Codeevo: Interaction-driven synthesis of code-centric data through hybrid and iterative feedback. *arXiv preprint arXiv:2507.22080*, 2025.

[17] Ruixiang Zhang, Richard He Bai, Huangjie Zheng, Navdeep Jaitly, Ronan Collobert, and Yizhe Zhang. Embarrassingly simple self-distillation improves code generation. *arXiv preprint arXiv:2604.01193*, 2026.

[18] Aozhe Wang, Yuchen Yan, Nan Zhou, Zhengxi Lu, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. Code-a1: Adversarial evolving of code llm and test llm via reinforcement learning. *arXiv preprint arXiv:2603.15611*, 2026.

[19] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

[20] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv e-prints*, pages arXiv–2309, 2023.

[21] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[22] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[23] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[24] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

[25] Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, et al. Evolver: Self-evolving llm agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079*, 2025.

[26] Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*, 2024.

[27] Andrew Zhao, Yiran Wu, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *Advances in Neural Information Processing Systems*, 38:105816–105879, 2026.

[28] Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data, 2026. URL `https://arxiv.org/abs/2508.05004`.

[29] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

[30] Parshin Shojaee, Ngoc-Hieu Nguyen, Kazem Meidani, Amir Barati Farimani, Khoa D Doan, and Chandan K Reddy. Llm-srbench: A new benchmark for scientific equation discovery with large language models. *arXiv preprint arXiv:2504.10415*, 2025.

[31] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, et al. Barbarians at the gate: How ai is upending systems research. *arXiv preprint arXiv:2510.06189*, 2025.

[32] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

[33] Jiachen Jiang, Tianyu Ding, and Zhihui Zhu. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. *arXiv preprint arXiv:2602.02919*, 2026.

[34] Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, et al. Pacevolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026.

[35] Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, et al. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

[36] Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

[37] Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: an open source evolutionary coding agent for algorithmic discovery and optimization, 2026. URL `https://arxiv.org/abs/2510.14150`.

[38] Thinking Machines Lab. Tinker, 2025. URL `https://thinkingmachines.ai/tinker/`.

A Additional Related Work

Synthetic coding tasks. Synthetic data generation has become a central approach for improving and evaluating LLMs' coding capabilities, especially as human-written programming tasks are expensive to collect and curate at scale. Early work primarily synthesizes instruction-following data: WizardCoder and WizardLM evolve seed instructions into more complex variants [4, 5], while SelfCodeAlign generates instruction data for code alignment [7]. Other work distills reasoning traces for competitive programming [8], synthesizes repository-level bug-fix tasks [9], or constructs synthetic curricula for RL-based code generation [10]. More recent systems move from instruction synthesis to complete executable tasks. AutoCode uses LLMs as competitive-programming problem setters, generating new statements together with reference solutions and tests [11]; X-Coder similarly studies fully synthetic competitive-programming data for training code reasoning models [13]. These methods demonstrate that synthetic coding data can improve scale and coverage. However, most of them are still organized as data-generation pipelines for downstream models: they generate instructions, tasks, solutions, or curricula, but do not explicitly require the generated tasks to challenge the generator itself.

Self-play and self-guided improvement. A complementary line of work studies model-in-the-loop data generation, where the model creates new supervision targeted to its evolving capabilities. Self-Challenging Agents instantiate this idea for tool-use agents, generating verifiable tasks and training an executor through RL feedback [14]. Self-Guided Self-Play studies a solver–conjecturer loop for formal theorem proving, introducing a guide role to avoid degenerate or uninformative conjectures [15]. More closely related in motivation, Absolute Zero trains a single model to propose and solve self-generated code-reasoning tasks, using execution both to construct valid tasks and to verify solver outputs [27]. R-Zero instead co-evolves separate Challenger and Solver models from zero external data: the Challenger generates mathematical questions near the Solver's capability boundary, while Solver self-consistency provides pseudo-labels for subsequent training [28]. In code generation, related work studies iterative code-centric data synthesis through Coder–Reviewer feedback [16], self-improvement from model-generated solutions [17], and adversarial co-evolution between code-generation and test-generation models [18]. Together, these works show that models can generate renewable training signal by targeting their own current weaknesses.

Our work shares this self-challenging motivation, but targets a different object: complete benchmark items evolved from existing coding problems. Rather than generating free-form training questions from scratch, BenchEvolver mutates an executable reference solution of a real seed task and derives a new statement and test suite around the evolved computation. Each accepted task is therefore required to be well specified, executable under the original benchmark harness, and empirically harder for a panel of target models, including the evolver itself. This design makes the output suitable not only as self-generated RL data, but also as an evolved benchmark for evaluating frontier coding models.

Self-evolving algorithms for LLM agents. Evolution-based algorithms have emerged as a general in-context strategy for solving difficult but verifiable optimization problems. They typically combine LLM-generated mutations with automatic evaluators, using feedback from candidate performance to guide subsequent generations. This paradigm has been applied to prompt and context optimization [19–21, 29], program and algorithm discovery [22, 23, 30, 31], and adaptive evolutionary search, including diversity-driven program evolution, meta-evolution, and long-horizon progress-aware optimization [32–36]. Related work also studies automated or self-evolving agent systems [24, 25] and open-source evolutionary coding agents [37].

Our framework adopts the same closed-loop search principle, but applies it to benchmark generation rather than solution optimization. Instead of evolving better prompts, programs, or agents for a fixed objective, BenchEvolver evolves the objective itself: harder executable coding tasks with reference solutions and tests. This shift is important for self-improvement. In conventional evolutionary optimization, the evaluator is fixed and the goal is to find a better solution; in BenchEvolver, the evaluator is used to discover new tasks that expose the model's current weaknesses, turning inference-time search [26] into reusable benchmark and training data.

B Pseudocode for BenchEvolver

Notation for Algorithm 1. Each task is $I = (S, C, T, E)$, consisting of a statement, reference code, hidden tests, and execution harness. The panel $\Pi = \{\pi_j\}_{j=1}^{J}$ contains the target solvers, and $\hat{C}_{j,k} \sim \pi_j(S)$ is the k-th solution attempt from solver π_j. The map ϕ converts lower empirical accuracy to higher difficulty level. The operators q_θ, w_θ, and g_θ are the mutator, statement writer, and test generator. For seed I_i, L_i is the accepted lineage, m_i stores local history such as accepted/rejected mutations, repairs, scores, and target error patterns, and G stores accepted mutation ideas across seeds for diversity guidance. The context $h_{i,b}$ packages the local and global memories together with the seed and parent difficulty levels. The flags r and q indicate whether the repaired candidate passes consistency validation and quality assessment, respectively. The feedback records F_{val}, F_{qual}, and F_{eval} contain validation outcomes, quality judgments, and target-model execution results.

C Validation and Repair Details

C.1 LiveCodeBench brute-force triangulation

For LiveCodeBench, BenchEvolver applies a benchmark-specific self-validation stack before target-model evaluation. It synthesizes an independent brute-force solver from the statement alone and checks public examples using a three-way vote among the reference solution, the brute-force solver, and a natural-language oracle that sees only the problem statement. Agreement passes the candidate; disagreements trigger targeted repair of the brute-force solver, reference solution, or task specification. The vetted brute-force solver is then run on generated hidden tests where feasible, using concrete output disagreements as counterexamples for repair while treating large-case timeouts as brute-force infeasibility. Candidates are rejected if the validation stack cannot resolve the inconsistency within a shared repair budget of three attempts, with each repair action, regardless of type, consuming one attempt.

C.2 SciCode statement-faithfulness validation

SciCode tasks do not naturally support brute-force validation because they are scientific function-level problems with assertion-based tests and domain-specific conventions. We therefore use a best-of-N statement-faithfulness check. After generating the statement, reference solution, and hidden assertion tests, the evaluator model solves the task from the statement alone. Each alternate solution is executed against the generated tests, and we keep the best pass rate over N attempts. We set a threshold of 0.5, and candidates below this threshold are treated as underspecified; the pipeline revises the statement and reruns the check against the same tests. If the revised task still fails within the shared repair budget, the candidate is rejected. This gate is intended to detect specification gaps rather than certify scientific correctness. Same-model self-play may share numerical blind spots with the reference solution, but if a capable solver cannot reproduce the expected behavior from the statement after several attempts, the task likely omits an important convention, assumption, or return-contract detail.

D Training Details

We fine-tune the open-weight `openai/gpt-oss-20B` model with on-policy reinforcement learning on LiveCodeBench-style coding problems. Training is performed through the Tinker RL service [38] with LoRA adapters on the policy; the reference policy and the sampler share the same base weights. All rollouts execute generated programs in an isolated cloud sandbox so that the reward signal is grounded in real test-case outcomes rather than a learned reward model.

D.1 Optimization and Model Configuration

The policy uses LoRA adapters of rank $r = 32$ over the attention and MLP projections of the base model. We optimize with a constant learning rate of $\eta = 1 \times 10^{-5}$ and a single optimizer substep per batch. We do not apply an explicit KL penalty to the reference policy ($\beta_{\mathrm{KL}} = 0$). Conversations are formatted with the `gpt_oss_medium_reasoning` chat renderer, which preserves the model's native thinking–answer structure.

Algorithm 1 BenchEvolver: Solution-Centric Evolution

Require: Seed tasks $D = \{I_i = (S_i, C_i, T_i, E_i)\}_{i=1}^{n}$; target panel $\Pi = \{\pi_j\}_{j=1}^{J}$; attempts per model K; proposal budget B; difficulty mapping ϕ; minimum seed-level gain Δ

Ensure: Accepted evolved tasks D'

Definitions:

$V_E(C, T) = 1$ iff program C passes all tests T under harness E.

EVALUATE(I, Π, K) samples K solutions from each $\pi_j(S)$ and returns their mean pass rate and execution feedback.

$D' \leftarrow \varnothing$; global memory $G \leftarrow \varnothing$
for each seed $I_i = (S_i, C_i, T_i, E_i) \in D$ **do**
  $(a_i, F_i) \leftarrow$ EVALUATE(I_i, Π, K)
  $\ell_i \leftarrow \phi(a_i)$; $\ell(I_i) \leftarrow \ell_i$
  Accepted lineage $L_i \leftarrow [\,]$; local memory $m_i \leftarrow \varnothing$
  for $b = 1, \ldots, B$ **do**
Select the current parent and construct memory context.
    if L_i is empty **then**
      $I_p \leftarrow I_i$
    else
      $I_p \leftarrow$ LAST(L_i)
    end if
    Unpack $I_p = (S_p, C_p, T_p, E_i)$
    $h_{i,b} \leftarrow$ BUILDCONTEXT$(m_i, G, \ell_i, \ell(I_p))$
Evolve the solution, then derive the statement and tests.
    $C' \sim q_\theta(\cdot \mid S_p, C_p, h_{i,b})$
    $S' \sim w_\theta(\cdot \mid S_p, C', h_{i,b})$
    $T' \sim g_\theta(\cdot \mid S', C', h_{i,b})$
    $I' \leftarrow (S', C', T', E_i)$
Check consistency with bounded repair, then assess quality.
    $(I', r, F_{\text{val}}) \leftarrow$ VALIDATEANDREPAIR(I')
    if $r = 0$ **then**
      $m_i \leftarrow$ RECORDREJECTION$(m_i, I', F_{\text{val}})$
      continue
    end if
    $(q, F_{\text{qual}}) \leftarrow$ CHECKQUALITY$(I', h_{i,b})$
    if $q = 0$ **then**
      $m_i \leftarrow$ RECORDREJECTION$(m_i, I', F_{\text{qual}})$
      continue
    end if
Measure difficulty only after validation and quality checks.
    $(a', F_{\text{eval}}) \leftarrow$ EVALUATE(I', Π, K)
    $\ell' \leftarrow \phi(a')$
    if $\ell' \geq \ell(I_p)$ **and** $\ell' \geq \ell_i + \Delta$ **then**
      $\ell(I') \leftarrow \ell'$
      $D' \leftarrow D' \cup \{I'\}$; append I' to L_i
      $m_i \leftarrow$ RECORDACCEPTANCE$(m_i, I', a', F_{\text{eval}})$
      $G \leftarrow$ UPDATEGLOBALMEMORY(G, I', a')
    else
      $m_i \leftarrow$ RECORDREJECTION$(m_i, I', a', F_{\text{eval}})$
    end if
  end for
end for
return D'

Component	Hyperparameter	Value
Base model	`model_name`	`openai/gpt-oss-20B`
LoRA	`lora_rank`	32
Renderer	`renderer_name`	`gpt_oss_medium_reasoning`
Optimizer	`learning_rate`	1×10^{-5}
	`kl_penalty_coef`	0.0
	`num_substeps`	1
Rollouts	`group_size` (G)	16
	`groups_per_batch` (B)	64
	`seed`	42 and 43
Context (train)	`max_tokens`	24,000
	`max_trajectory_tokens`	26,500
Context (eval)	`test_max_tokens`	30,000
	`test_max_trajectory_tokens`	32,500
Environment	`sandbox_backend`	`modal`
	per-test `timeout`	6 s
Reward shaping	`format_coef` (λ)	0.1
	`context_overflow_reward`	-0.1

Table 4: Training configuration used for all RL runs. Train/eval context caps differ because the policy needs more room at evaluation time to produce a valid final answer when reasoning chains are longest.

D.2 RL Objective and Batch Construction

For each task τ in a training batch, we sample $G{=}16$ independent trajectories $\{x^{(g)}\}_{g=1}^{G}$ from the current policy π_θ. A batch consists of $B{=}64$ such groups, yielding $B \cdot G{=}1024$ trajectories per gradient step. We use group-relative advantages: for each task, the scalar reward of every trajectory is centered against the mean reward of its own group,

$$A_\tau^{(g)} \;=\; r(x_\tau^{(g)}) \;-\; \frac{1}{G}\sum_{g'=1}^{G} r(x_\tau^{(g')}), \tag{1}$$

so that easy tasks (where most rollouts pass) and hard tasks (where most rollouts fail) contribute meaningful gradient signal without an additional value baseline. Trajectories are then trained against the policy-gradient loss with clipped importance weights, in the standard on-policy form used by Tinker's RL trainer.

D.3 Reward Design

Rewards are computed once per trajectory, after the entire conversation has finished. The grader extracts the *last* fenced code block from the final assistant turn and submits it to the cloud sandbox along with the task's LiveCodeBench-format tests.

Let $c \in \{0, 1\}$ indicate whether the extracted program passes *every* test case under a per-test wall-clock limit of $T{=}6$ s, and let $f \in \{0, 1\}$ indicate whether the response contains at least one fenced code block. The episode reward is

$$r \;=\; c \;+\; \lambda\,(f - 1), \qquad \lambda = 0.1. \tag{2}$$

Equation 2 has three intended properties. (i) A fully correct solution receives $r{=}1$. (ii) An incorrect but well-formatted attempt receives $r{=}0$. (iii) A response without an extractable code block receives $r{=}-0.1$, providing a small shaping signal that prevents the policy from collapsing into pure chain-of-thought without ever emitting code. We deliberately keep λ small so that format shaping never dominates the correctness signal.

Sandbox grading. Stdin/stdout problems are run as a real Python subprocess inside the sandbox, matching the LiveCodeBench harness; functional problems use the in-process call-based path. Output

comparison applies, in order, exact-match (after stripping trailing whitespace), numeric match with relative tolerance 10^{-6}, and a set-of-tokens fallback for unordered outputs. Per-test failures (Wrong Answer, Time Limit Exceeded, Runtime Error) all count as $c{=}0$, regardless of which test failed first.

Trajectory-level penalties. Trajectories whose token budget is exhausted before a final answer is produced receive $r_{\text{overflow}} = -0.1$, identical in magnitude to the format penalty. This prevents the policy from learning to stall indefinitely in the reasoning channel.

D.4 Training Dynamics

Figure 4 reports the on-policy training reward and average response length for the three RL data mixtures. Seed-only training starts with higher reward and shows smaller subsequent gains, whereas evolved-only training starts lower and continues to improve. The combined mixture generally lies between these two conditions in training reward.

These differences are consistent with the intended difficulty of the evolved data. However, average reward alone does not determine the fraction of groups with nonzero relative advantages, so these curves do not directly establish the mechanism discussed in Section 4.3.

Training on evolved problems also produces longer responses than seed-only training. The experiments therefore match training steps, problems per batch, and rollouts per problem, rather than total generated-token budgets.

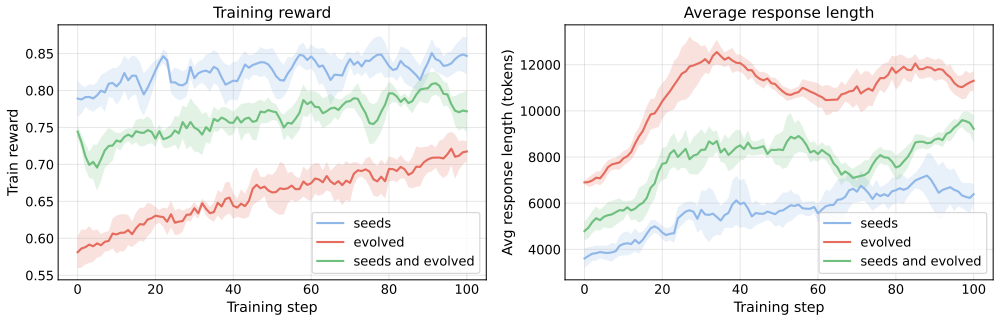


Figure 4: Training reward (left) and average response length in tokens (right), for the three data conditions (seeds, evolved, seeds and evolved). Solid lines are the mean of two random seeds per condition; shaded bands are ± 1 standard deviation across the two seeds. A boxcar smoother of width 5 is applied to both the mean and standard-deviation curves. The reward panel reflects Eq. 2; the maximum attainable per-trajectory reward is 1.0 and the minimum is -0.1.

D.5 Compute and Reproducibility

Each RL run takes approximately 40 hours and costs about \$800 in Tinker credits. All runs use the Tinker service for policy execution and gradient updates, while the client-side training loop runs on a single CPU host. Code execution is delegated to Modal-hosted sandboxes, so throughput is primarily determined by generated-program execution and test-case latency rather than local compute. We use two independent random seeds for each data mixture and keep the hyperparameters fixed across all runs.

E Evolution Configurations and Hyperparameters

The default configurations for our LiveCodeBench and SciCode experiments are summarized in Table 5. The table reports the model settings, acceptance criteria, repair and validation budgets, test-generation parameters, and memory mechanisms used by our evolution pipeline.

Config name	Purpose	LCB	SciCode
Model and evaluation			
`target_eval_k`	Attempts per target model for evolved problems.	4	4
`temperature`	Sampling temperature for generation.	0.8	0.8
`timeout`	LLM request timeout in seconds.	600	600
Difficulty and acceptance			
`allowed_seed_levels`	Seed difficulty levels allowed as starting points.	[1,2]	[1,2]
`accept_min_level_gain`	Minimum difficulty-level gain over the seed.	1	1
`accept_target_level`	Stop once this accepted difficulty level is reached.	5	5
`max_iters_per_seed`	Maximum evolution iterations per seed.	10	10
`max_accepted_per_seed`	Maximum accepted evolved items per seed.	3	3
`no_improve_patience`	Stop evolution after consecutive non-accepted iterations.	4	4
Mutation and repair			
`max_candidate_repairs`	Shared repair budget per candidate.	3	3
`spec_retry_attempts`	Statement/spec revision attempts.	2	2
`enable_test_repair`	Allow regenerating weak or malformed tests.	true	true
`test_repair_attempts`	Test repair attempts.	2	2
SciCode-specific faithfulness checks			
`faithfulness_check_N`	Alt-solver attempts for statement-faithfulness check.	–	3
`faithfulness_min_pass_rate`	Best-of-N pass-rate floor for a faithful statement.	–	0.5
`faithfulness_max_repair`	Spec-revision retries on faithfulness failure.	–	2
Test generation and execution			
`program_gen_small_inputs`	Number of small generated inputs.	5	2
`program_gen_medium_inputs`	Number of medium generated inputs.	5	2
`program_gen_large_inputs`	Number of large generated inputs.	5	2
`program_gen_stress_inputs`	Number of stress generated inputs.	3	–
`execution_timeout`	Per-test reference/target execution timeout.	6.0	30.0
`max_output_bytes`	Maximum captured stdout bytes.	1000000	1000000
Memory and diversity			
`memory_raw_window`	Raw recent iteration records retained in local memory.	10	5
`memory_digest_recent_k`	Recent raw records shown alongside digest.	3	3
`judge_near_duplicate_check`	Judge rejects near-duplicates within a seed lineage.	true	true
`global_memory_max_entries`	Max global accepted entries shown to mutator.	50	20
`global_memory_max_chars`	Character cap for global memory block.	6000	6000

Table 5: Default evolution configurations for LCB and SciCode.

F Additional Difficulty Evaluation

We provide the full post-hoc difficulty evaluation for the task-evolution experiments in Section 4.1. To obtain a more reliable estimate of empirical difficulty, we re-evaluate each seed and evolved problem using $k = 8$ independent attempts per model. These evaluations are performed after task generation and are not used for candidate acceptance.

F.1 LiveCodeBench

Table 6 and Figure 5 report the complete LiveCodeBench-v6 results. Across all target-model and evolver configurations, evolved problems reduce Pass@1 by 32.4 percentage points on average relative to their source seeds. The decrease is consistent across target models and evolvers, including cases where the evolver itself is evaluated on its generated tasks.

The largest drops occur for Claude-Opus-4.6, but the direction is consistent across all evaluated configurations. Importantly, the difficulty increase is also observed when the evolver itself is evaluated on the generated tasks, showing that the effect is not limited to a stronger-model-to-weaker-model generation setting.

Target Model	Evolver Model	Seed	Evolved	Δ
GPT-5.4-mini	GPT-5.4-mini	98.1	73.4	-24.6
GPT-5.4-mini	Gemini-3-Flash	98.2	59.8	-38.3
GPT-5.4-mini	Claude-Sonnet-4.6	97.9	78.9	-19.0
GPT-5.4	Gemini-3.1-Pro	100.0	71.0	-29.0
Gemini-3-Flash	GPT-5.4-mini	94.4	75.4	-19.0
Gemini-3-Flash	Gemini-3-Flash	91.9	71.8	-20.1
Gemini-3-Flash	Claude-Sonnet-4.6	96.2	80.1	-16.1
Gemini-3.1-Pro	Gemini-3.1-Pro	97.8	85.3	-12.5
Claude-Opus-4.6	GPT-5.4-mini	75.0	23.0	-52.0
Claude-Opus-4.6	Gemini-3-Flash	74.2	14.6	-59.6
Claude-Opus-4.6	Claude-Sonnet-4.6	85.1	44.5	-40.6
Claude-Opus-4.6	Gemini-3.1-Pro	76.6	18.4	-58.3

Table 6: LiveCodeBench-v6 post-hoc difficulty evaluation using $k = 8$ independent attempts per model. Seed and Evolved report Pass@1 (%); Δ is the absolute change from the original seed problems to their evolved variants.

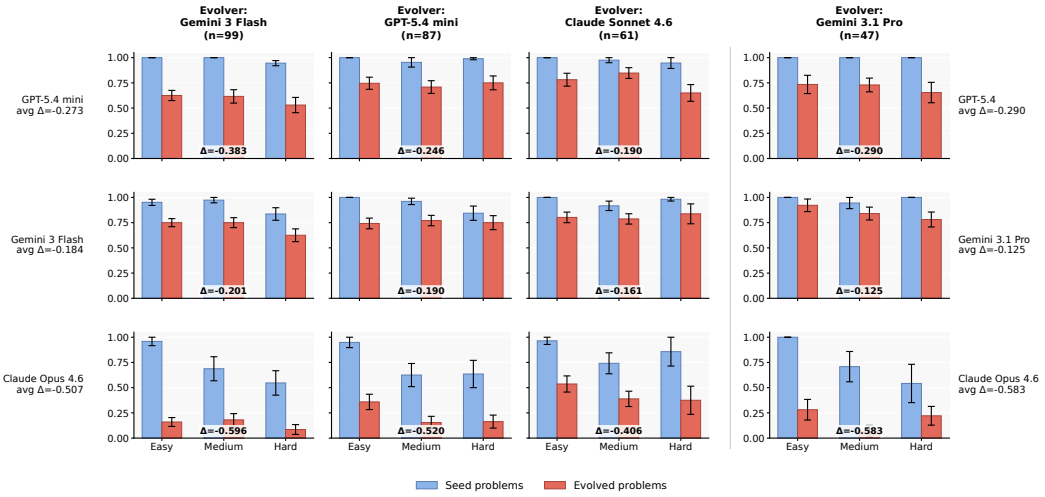


Figure 5: Pass@1 on original LiveCodeBench-v6 seed problems and their evolved variants using $k = 8$ attempts per model. Evolved tasks consistently reduce performance across target models and evolvers.

F.2 SciCode

Table 7 and Figure 6 report the corresponding SciCode results. Evolved problems reduce Pass@1 by 28.9 percentage points on average despite the different function-level task format and validation procedure. The same qualitative pattern holds across target models and evolvers.

Together, the LiveCodeBench and SciCode results show that the observed difficulty increase is robust to a larger evaluation sample and persists across models, evolvers, and executable coding formats. The $k = 8$ evaluation is used only for post-hoc measurement; candidate acceptance follows the evolution configuration specified in Appendix E.

G Human Evaluation: Full Breakdown

This appendix provides the detailed protocol and results of the human evaluation, including distributions of expert ratings and the algorithm-category breakdown for seed and evolved problems.

Review protocol. We use a blinded expert-review protocol to reduce positional, model, and post-hoc filtering biases. Six competitive-programming experts (Codeforces grandmaster / IOI / ICPC level) review groups of 2 to 4 anonymized problems with opaque identifiers and are not told which problem is the seed, which are evolved variants, or which model generated them. Reviewers certify

Target Model	Evolver Model	Seed	Evolved	Δ
GPT-5.4-mini	GPT-5.4-mini	91.1	67.2	-23.9
GPT-5.4-mini	Gemini-3-Flash	88.2	52.0	-36.2
GPT-5.4-mini	Claude-Sonnet-4.6	88.8	77.5	-11.3
GPT-5.4	Gemini-3.1-Pro	98.1	76.4	-21.7
Gemini-3-Flash	GPT-5.4-mini	95.6	39.8	-55.8
Gemini-3-Flash	Gemini-3-Flash	96.6	65.2	-31.4
Gemini-3-Flash	Claude-Sonnet-4.6	96.3	83.8	-12.5
Gemini-3.1-Pro	Gemini-3.1-Pro	99.1	70.8	-28.3
Claude-Sonnet-4.6	GPT-5.4-mini	89.9	34.5	-55.4
Claude-Sonnet-4.6	Gemini-3-Flash	91.1	49.5	-41.6
Claude-Sonnet-4.6	Claude-Sonnet-4.6	90.0	81.3	-8.7
Claude-Opus-4.6	Gemini-3.1-Pro	94.0	73.6	-20.4

Table 7: SciCode post-hoc difficulty evaluation using $k = 8$ independent attempts per model. Seed and Evolved report Pass@1 (%); Δ is the absolute change from the original seed problems to their evolved variants.

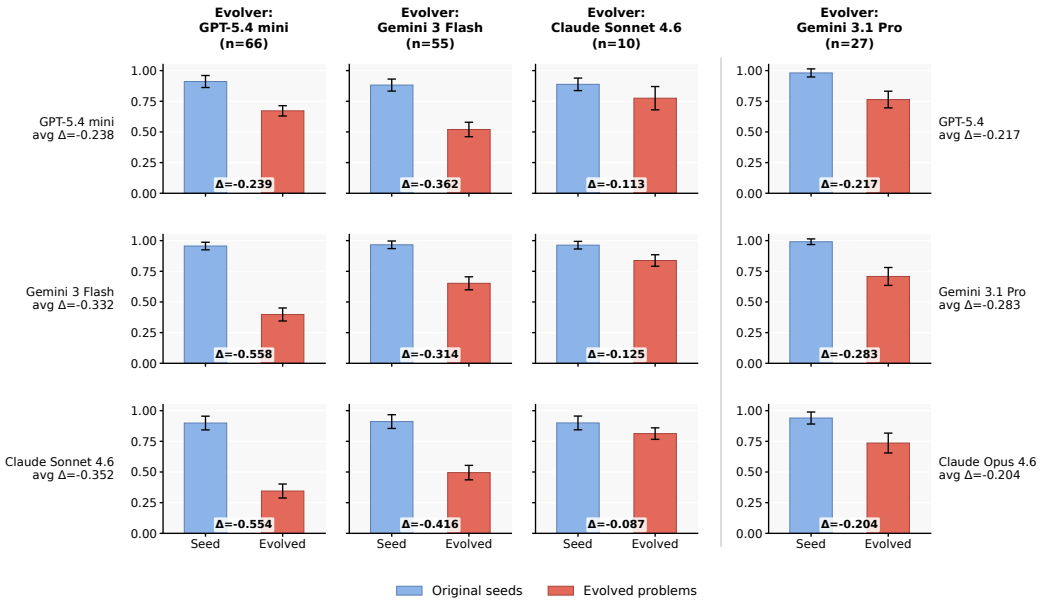
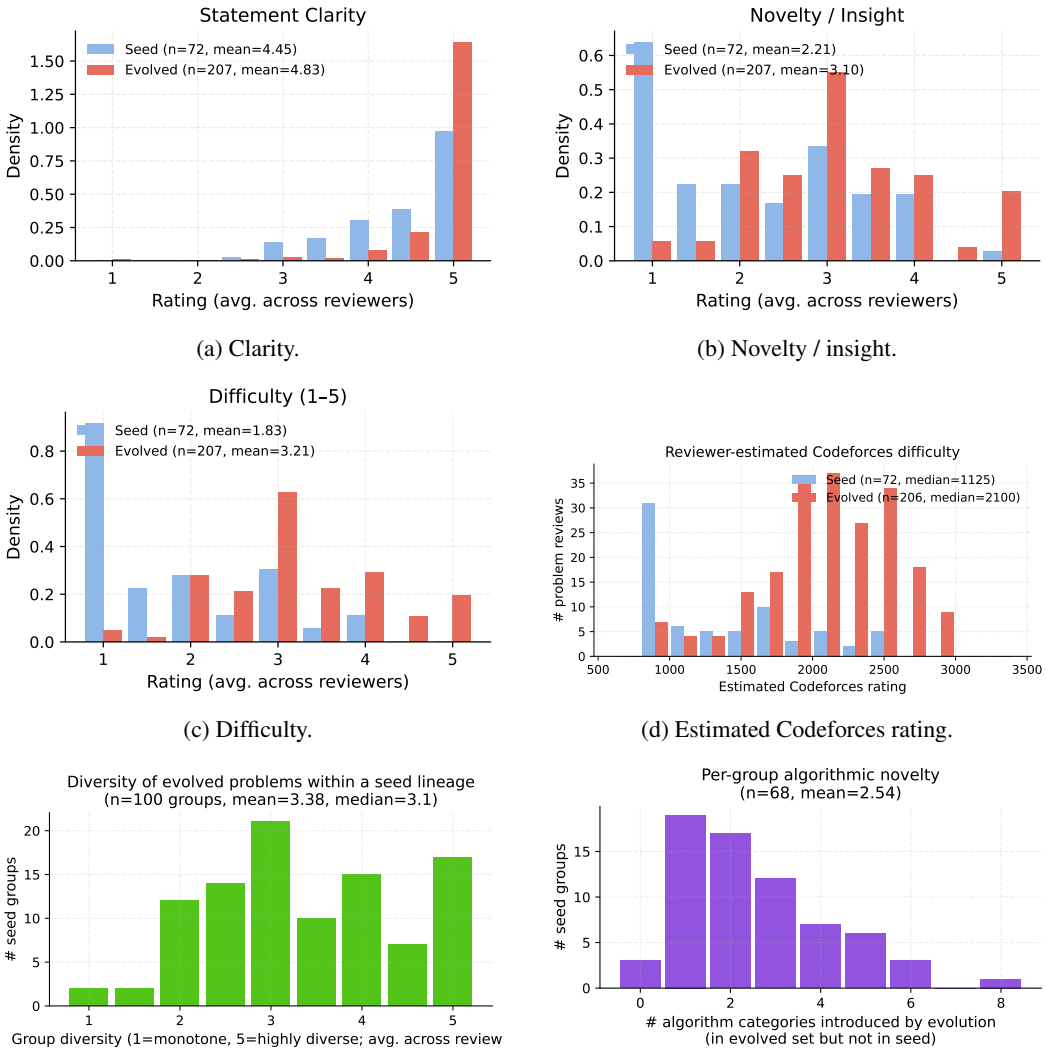


Figure 6: Pass@1 on original SciCode seed problems and their evolved variants using $k = 8$ attempts per model. Evolved tasks consistently reduce performance across target models and evolvers.

that they do not use generative AI during evaluation. For each problem, they rate clarity, novelty, and difficulty on a 1–5 scale, estimate a Codeforces rating, list required algorithms and data structures, and provide short written justifications. Groups are assigned to two reviewers when possible: 65% of problems receive two independent reviews. We aggregate numeric ratings by averaging per problem, and aggregate algorithm tags by taking the union after normalizing synonyms and variants into a controlled vocabulary. This protocol makes the reported statistics primarily reflect problem-level variation rather than a single reviewer or post-hoc selection.

Results. Figure 7 shows a consistent shift from the seed problems to the evolved problems. The evolved problems are rated substantially harder (panel c; $1.83 \rightarrow 3.21$), and the estimated Codeforces ratings (panel d) move from a concentrated pupil/specialist range around 1100 to a broader distribution centered near 2100. They are also judged more novel (panel b; $2.21 \rightarrow 3.10$), with far fewer cases rated as close variants of the seed. Importantly, this increase in difficulty does not come from making the statements less clear: clarity improves from 4.45 to 4.83, with most evolved problems rated 4 or 5. Finally, panels e–f and Figure 8 show that the evolved problems broaden the algorithmic coverage: 95.6% of lineages introduce at least one new algorithmic category, with 2.54 new categories per group on average, and the total number of observed categories increases from 19 to 30. Overall,



(a) Clarity.

(b) Novelty / insight.

(c) Difficulty.

(d) Estimated Codeforces rating.

(e) Group-level diversity.

(f) New categories per group.

Figure 7: Human-evaluation distributions. The evolved problems are rated as more novel and more difficult than their seeds, with median estimated Codeforces rating increasing from 1125 to 2100. At the group level, evolved lineages have mean diversity $3.38/5$ and introduce an average of 2.54 new algorithm categories per group, with at least one new category in 95.6% of groups.

the human study supports the main claim that BenchEvolver produces problems that are harder and diverse while remaining well specified.

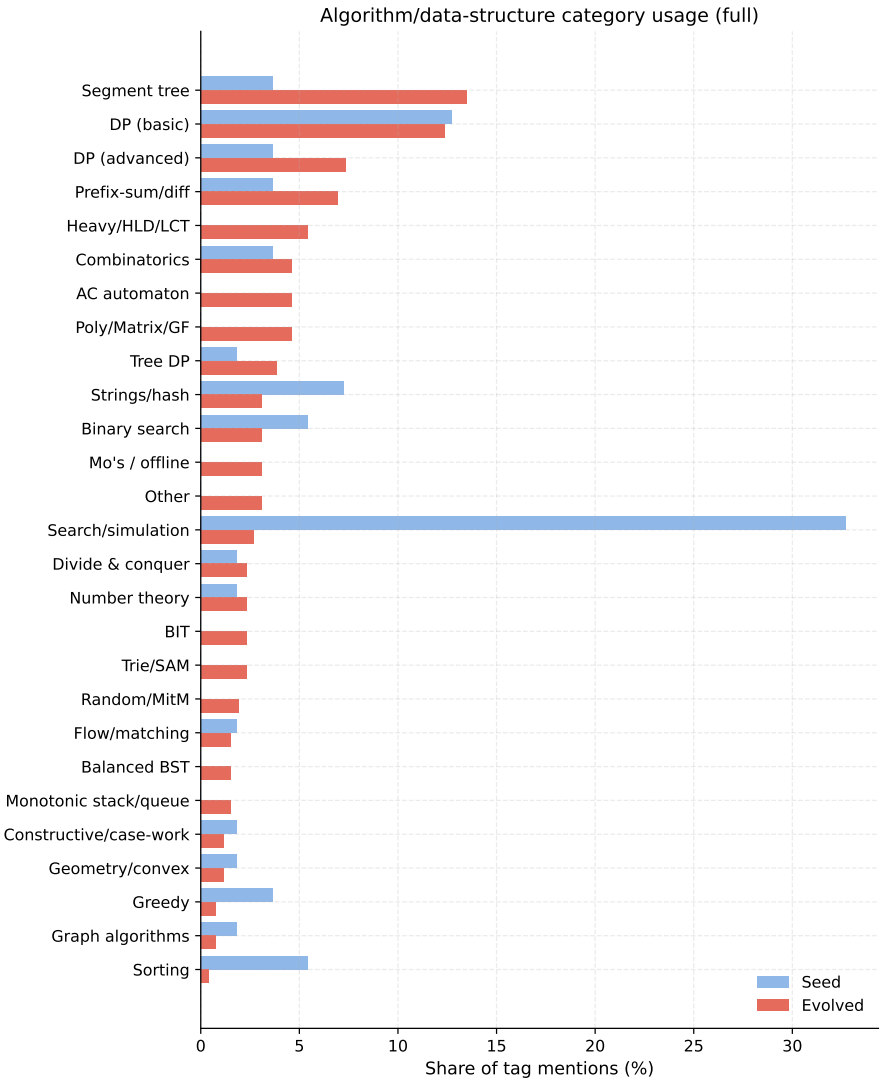


Figure 8: Full algorithm-category distribution for seed and evolved problems, including all categories with at least three combined mentions. Shares are computed over algorithm-tag mentions within each condition.

H Examples of Evolved Benchmark Items

In this section, we provide examples of successfully evolved frontier tasks for LiveCodeBench and SciCode, and compare them with their seed problems.

H.1 LiveCodeBench Examples

LCB Seed: Copy Arrays

decision/counting · O(N)

TASK

Given `original` and interval bounds $\texttt{bounds}[i] = [u_i, v_i]$, count arrays `copy` such that

$$\texttt{copy}[i] - \texttt{copy}[i-1] = \texttt{original}[i] - \texttt{original}[i-1], \qquad 1 \le i < n,$$

with $u_i \le \texttt{copy}[i] \le v_i$ for all i. The key observation is

$$\texttt{copy}[i] = \texttt{original}[i] + d,$$

so feasibility reduces to intersecting valid intervals for one global offset d.

CONSTRAINTS

$$2 \le n \le 10^5$$
$$1 \le \texttt{original}[i] \le 10^9$$
$$1 \le u_i \le v_i \le 10^9$$

EXAMPLES

```
original=[1,2,3,4]; bounds=[[1,2],[2,3],[3,4],[4,5]]   → 2
original=[1,2,3,4]; bounds=[[1,10],[2,9],[3,8],[4,7]]  → 4
original=[1,2,1,2]; bounds=[[1,1],[2,3],[3,3],[2,3]]   → 0
```

Evolved Problem: XOR-Linked Sequence

digit DP · O(N bits)

TASK

The evolved task replaces additive differences with XOR differences:

$$\texttt{copy}[i] \oplus \texttt{copy}[i-1] = \texttt{original}[i] \oplus \texttt{original}[i-1], \qquad 1 \le i < n.$$

Count non-negative $x = \texttt{copy}[0]$ such that all entries satisfy their bounds. Since

$$\texttt{copy}[i] = x \oplus p_i, \qquad p_i = \texttt{original}[i] \oplus \texttt{original}[0],$$

the task becomes counting x such that

$$u_i \le x \oplus p_i \le v_i \qquad \text{for all } i.$$

Unlike arithmetic intervals, these XOR-induced feasible sets are not contiguous.

CONSTRAINTS

$$0 \le n \le 10^5$$
$$0 \le \texttt{original}[i] \le 10^9$$
$$0 \le u_i \le v_i \le 10^9$$

EXAMPLES

```
original=[1,2]; bounds=[[1,2],[2,3]]                   → 1
original=[1,2]; bounds=[[1,1],[3,3]]                   → 0
original=[1,2,3]; bounds=[[1,10],[1,10],[1,10]]        → 6
```

Mutation and algorithmic lift. The seed problem has a single additive offset, so it is solved by interval intersection in $O(N)$ time. The evolved variant changes the relation to XOR, yielding constraints of the form $u_i \le x \oplus p_i \le v_i$. Because XOR ranges are not contiguous, simple interval intersection fails, and a bitwise digit-DP or trie-style method is needed.

LCB Seed: Uniqueness

whole array · O(N)

TASK

Given N people with values A_i, find the label of the person whose value is unique among all people and is numerically largest. If no value is unique, output -1.

CORE STRUCTURE

Count global frequencies, then select the maximum value with frequency one and return its index.

CONSTRAINTS

$$1 \le N \le 3 \times 10^5, \qquad 1 \le A_i \le 10^9.$$

EXAMPLES

```
A=[2,9,9,7,9,2,4,5,8]                  → 9
A=[1e9,1e9,998244353,998244353]        → -1
```

Evolved #1: Maximum Unique Value Query

subarray queries

TASK

Given an array A and Q range queries (L, R), output the index of the largest value that appears exactly once in $A[L..R]$. If no such value exists, output -1.

ALGORITHMIC LIFT

The uniqueness test becomes range-dependent: each query has its own frequency profile, requiring offline or data-structural reasoning rather than one global pass.

CONSTRAINTS

$$1 \le N, Q \le 5 \times 10^5, 1 \le A_i \le 10^9.$$

EXAMPLES

```
A=[10,10,20,20]; q=(1,4)               → -1
A=[1,2,3,2,1]; q=(1,5),(1,2)           → 3,2
A=[2,9,9,7,9,2,4,5,8]; q=(1,9)         → 9
```

Evolved #2: Maximum Unique Path Value

tree paths

TASK

Given a tree with node values W_i, each query gives nodes u, v. On the path between them, output the node index whose value is largest among values appearing exactly once on that path. If none exists, output -1.

ALGORITHMIC LIFT

The range-query setting is lifted from arrays to tree paths, so queries depend on path decomposition and tree topology rather than contiguous intervals.

CONSTRAINTS

$$1 \le N, Q \le 10^5, \qquad 0 \le W_i \le 10^9.$$

The input graph is a tree.

EXAMPLES

```
edge 1-2; W=[10,10]; q=(1,2)           → -1
chain 1-2-3; W=[10,20,10];             → 2
q=(1,3)
chain; W=[1,2,3,2,1];                  → 3,2
q=(1,5),(1,2)
```

Suite-level mutation pattern. The suite preserves the semantic target: identify the largest value that is unique in the relevant scope and return its index. The scope is progressively generalized from the whole array, to a subarray, and finally to a tree path.

H.2 SciCode Examples

SciCode Seed: 4th-Order Runge-Kutta Integrator

forward simulation

TASK

Implement a classical fourth-order Runge-Kutta (RK4) integrator from scratch for a driven damped pendulum. The routine takes a derivative function, an initial state, and fixed time-step information, and returns the full state-space trajectory over n integration steps.

FUNCTION INTERFACE

```
def runge_kutta_4th_order(
    f, state, t0, dt, n, g, L, beta, A, alpha
):
    """Run an RK4 integrator for pendulum motion."""
```

INPUT / OUTPUT

Input: derivative function `f`, initial state `state = [theta, omega]`, initial time `t0`, step size `dt`, number of steps `n`, and pendulum parameters `g, L, beta, A, alpha`.
Output: `trajectory`, a 2D NumPy array of shape `(n+1, 2)` containing the simulated states.

EXAMPLE TEST

```
state0 = np.array([0.1, 0.0])
traj = runge_kutta_4th_order(
    pendulum_derivs, state0,
    0.0, 0.01, 100,
    9.81, 1.0, 0.1, 0.0, 0.0
)
assert traj.shape == (101, 2)
```

Evolved: Fit ODE Trajectory with Damped Gauss-Newton

inverse problem

TASK

Implement a nonlinear inverse solver that estimates both the unknown initial state and unknown ODE parameters from sparse observations. The solver must simulate the ODE trajectory using fixed-step RK4 between observation times and fit the unknown variables by damped Gauss-Newton with finite-difference Jacobians and backtracking line search.

FUNCTION INTERFACE

```
def fit_ode_trajectory_gauss_newton(
    f, state_guess, param_guess, t_obs, observations,
    steps_per_unit=100, max_iter=15, tol=1e-10,
    fd_eps=1e-7, ridge=1e-10
):
    """Fit an ODE trajectory by damped
Gauss-Newton."""
```

INPUT / OUTPUT

Input: ODE right-hand side `f`, guesses for the initial state and parameters, observation times `t_obs`, observed trajectory array `observations`, and numerical optimization hyperparameters.
Output: a tuple containing the fitted initial state, fitted parameter vector, and simulated trajectory evaluated at the observation times.

EXAMPLE TEST

```
def exp_growth(x, t, theta):
    return np.array([theta[0] *
x[0]])
t_obs = np.array([0.0, 0.2, 0.55,
1.1])
obs = np.exp(-0.8 * t_obs)[:,
None] * 2.5
x_hat, theta_hat, traj =
fit_ode_trajectory_gauss_newton(
        exp_growth,
        np.array([1.0]),
        np.array([-0.3]),
        t_obs,
        obs,
        steps_per_unit=200
    )
assert traj.shape == obs.shape
```

Mutation and algorithmic lift. The seed task performs forward simulation for a fixed ODE using the classical RK4 integrator. The evolved task preserves the ODE-trajectory setting but turns a direct numerical integration task into a full nonlinear parameter-estimation pipeline.

SciCode Seed: Compute the n-Tangle of an Even-Qubit Pure State

quantum states

TASK

Implement a function that computes the n-tangle of an even-numbered n-qubit pure state $|\psi\rangle$.
The input `psi` is a one-dimensional array of length 2^n representing the state vector in the computational basis. You may assume that n is even.

FUNCTION INTERFACE

```
def n_tangle(psi):
    """
    Compute the n-tangle of an even-n-qubit pure
state.
    """
```

INPUT / OUTPUT

Input: `psi : np.ndarray`, a one-dimensional array of length 2^n representing a pure quantum state.
Output: `tangle : float`, the n-tangle of the input state.

EXAMPLE TESTS

```
MaxEnt = np.array([1, 0, 0, 1]) /
np.sqrt(2)
assert np.allclose(
n_tangle(MaxEnt), 1.0 )
product_state = np.kron(
np.array([0, 1]), np.array([0.8,
0.6]) )
assert np.allclose(
n_tangle(product_state), 0.0 )
```

Evolved: Maximum Pairwise CHSH Bell Value

pairwise optimization

TASK

Implement a function that computes the largest CHSH Bell value attainable by any pair of qubits in an n-qubit quantum state.
The input may be either a pure-state vector or a density matrix, and both real and complex inputs are allowed.

FUNCTION INTERFACE

```
def max_pairwise_chsh(state):
    """
    Compute the maximum CHSH Bell value over all
    two-qubit subsystems.
    """
```

INPUT / OUTPUT

Input: `state : np.ndarray`, either a state vector of length 2^n or a density matrix of shape $(2^n, 2^n)$.
Output: `value : float`, the largest CHSH Bell value among all distinct pairs of qubits.

EXAMPLE TESTS

```
bell = np.array( [1, 0, 0, 1],
dtype=float ) / np.sqrt(2)
assert np.isclose(
max_pairwise_chsh(bell), 2 *
np.sqrt(2) )
ghz = np.zeros(8, dtype=float)
ghz[0] = ghz[7] = 1 / np.sqrt(2)
assert np.isclose(
max_pairwise_chsh(ghz), 2.0 )
```

Mutation and algorithmic lift. The seed task computes a single global entanglement quantity for an even-qubit pure state. The evolved task instead optimizes over all two-qubit subsystems, searching for the largest attainable CHSH Bell value. This shifts the problem from evaluating one global scalar functional to analyzing many candidate subsystems and comparing their pairwise nonlocal correlations.

I Examples of Evolution Trajectory

In this appendix section, we append example trajectories of Gemini-3-Flash on LiveCodeBench Easy split, shown in Figure 9.

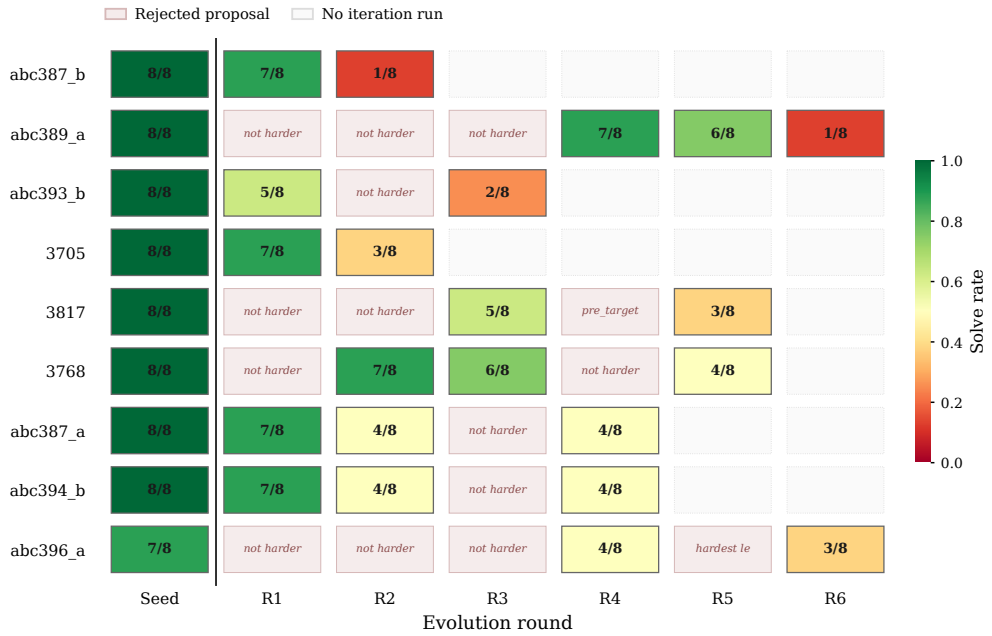


Figure 9: **Example evolution trajectories produced by BenchEvolver.** Each row shows one seed problem, labeled by question ID. The leftmost column reports the seed solve rate, and subsequent columns (R1, R2, . . .) show accepted evolution rounds in chronological order. Each cell reports passes/attempts, pooled across all target models, where attempts equals the number of target models times `target_eval_k`; cell color encodes solve rate, with red indicating harder problems and green indicating easier ones. The dark path traces the accepted lineage within each row, highlighting the monotonic decrease in solve rate, i.e., the monotonic increase in empirical difficulty. Faint pink cells denote rejected proposals: *not harder* means the candidate did not exceed the required difficulty level, and *judge* means it failed final LLM-judge review. Light gray cells indicate that no further iteration was run. Overall, the trajectories show that BenchEvolver moves near-saturated seeds into a useful difficulty regime, while the rejection mechanism filters candidates that are too easy or invalid.

J Prompt Templates for BenchEvolver

In this section, we provide representative prompt templates used by the main generation components in BenchEvolver. These include the solution mutator, which proposes a harder reference solution; the statement writer, which converts the solution into a complete problem statement; and the test generator, which produces validated tiered test inputs for evaluation.

Solution Mutator

```
You are an expert competitive programmer and benchmark designer.
Your job is to mutate an existing solution into a structurally different,
more complex solution code for a harder problem. Return ONLY valid JSON.

You are mutating a seed problem into a harder variant.

=== PARENT ===
Problem statement:
{parent_problem}

Solution code:
{parent_solution}

=== DIFFICULTY CONTEXT ===
Seed difficulty level: {seed_difficulty_level}

Parent difficulty level: {parent_difficulty_level}

Best level reached for this seed:
{best_level_reached}
```

```
Your goal:
reach level {target_level_goal} or higher

=== EVOLUTION MEMORY ===
What worked, what failed, and why:
{memory_context}

=== ALREADY ACCEPTED ACROSS ALL SEEDS ===
Accepted ideas from this run. Propose a dominant algorithmic idea that is
clearly distinct from every entry below.

{global_accepted_context}

=== TARGET MODEL WEAKNESSES ===
Reasoning errors to exploit:
{target_weaknesses}

=== TASK ===
Produce a NEW problem whose difficulty lift is real, natural, genuine, and
algorithmic.

The parent information may be shown in MINIMAL-SPEC mode. If so, treat the
parent solution code as the primary source of truth and use the compact semantic
scaffold only to recover intended semantics and I/O. Do not stay close to the
original wording or story.

Your goal is to move the problem from difficulty level {parent_difficulty_level}
toward level {target_level_goal}. Use the evolution memory above to guide your
mutation strategy.

Keep free mutation mode, but use that freedom to search for a SINGLE DOMINANT
algorithmic lift.

=== Additional Guidance ===
{Examples of valid and invalid mutations}
{Output constraints}
```

Problem Statement Writer

```
You are a competitive programming problem setter.
Given a solution and sample input/output pairs, write a clear, natural problem statement.
Return ONLY valid JSON.

Solution code:
{solution_code}

Mutation notes:
{mutation_notes}

Algorithm used: {approach}

Sample input/output pairs:
{sample_io}

Write a competitive-programming-style problem statement for this solution.
Requirements:
- The FINAL solution code is the primary source of truth. Use mutation notes only as hints; if they conflict with the
      code, ignore the notes and follow the code.
- Natural, unambiguous prose (as in LeetCode/Codeforces/AtCoder)
- Include: problem description, input format, output format, constraints, and sample input/output (embed the provided
      sample I/O pairs as "Sample Input / Sample Output" sections in the statement body)
- Do NOT hint at the algorithm or data structure used
- If sample I/O pairs are provided, the statement must be consistent with ALL of them
- If no sample I/O is provided, infer a valid and unambiguous input/output spec from the solution code
- Answer must be uniquely determined by the input. If the underlying task admits multiple equally valid answers under
      different deterministic strategies (e.g., merging equal adjacent values, choosing among tied minima/maxima,
      selecting any valid partition, returning any optimal subset), you MUST spell out the exact tie-breaking rule the
      solution uses (e.g., "choose the leftmost such pair", "return the smallest index", "output the lexicographically
      smallest sequence"). Infer the tie-breaking rule by reading the reference solution code. Do not guess and do not
      paper over the ambiguity.
- Do NOT write blanket phrases like "any one may be chosen", "the answer is uniquely determined regardless of tie-
      breaking", or "any valid answer is accepted". If the answer truly does not depend on tie-breaking, prove it by
      stating the concrete invariant (e.g., "the sum is invariant under order of operations because addition is
      associative"); if you cannot state such an invariant, treat the task as ambiguous and add an explicit tie-
      breaking rule.
- Every input variable must have explicit bounds in the constraints section appropriate to its type (e.g. integer range,
       string length, number of elements). Read them from the solution code if unsure
- Use plain text for all math
- Do NOT use any LaTeX or math markup of any kind
- For every operation type, copy the exact token order from the code's parser. Never swap argument order between the
      description, input format, and samples.
- Use standard line-based competitive-programming formatting only. Do not invent dual formats, optional parsing
      conventions, or ambiguous line structure.
- Do not claim stronger constraints than the code supports. If the code is only correct for a simpler/static version,
      do not write a dynamic/online statement.
- If unsure, use fewer public examples rather than inventing a risky one.
```

Test Generator

```
You are a senior competitive-programming test-data engineer.
You write standalone Python 3 programs that generate deterministic, valid test inputs for a given problem.

Your validate() function is the SOLE source of truth for input legality, and the caller performs no secondary format
      check. Any input you emit that passes validate() will be fed straight to the reference solution and the reference
      's stdout will be frozen as ground truth. A too-permissive validate() therefore silently corrupts the benchmark;
      a too-strict validate() starves the suite of inputs.

Your programs must:
  - use a caller-controlled random seed so the same CLI args always produce the same inputs,
  - enforce every declared constraint via validate() exactly as implied by the problem statement AND the reference
      solution,
  - partition generation into four explicit tiers defined BEHAVIORALLY (see user message): small / medium / large /
      stress,
  - retry internally on validation failure before giving up on an input,
  - emit a single JSON object on stdout and nothing else.

Return ONLY valid JSON wrapping the program.

Problem statement:
{problem_statement}

Reference solution (authoritative for input format, constraints, and semantics):
{solution_code}

Sample input/output pairs (use these to infer the canonical format):
{sample_io}

Your task: write a SINGLE self-contained Python 3 program that generates test inputs for this problem.

{Tier Definitions}
```

K Reproducibility and Asset Licenses

Code and configurations. We released the code used to run our evolution pipeline, including the configuration files for LiveCodeBench and SciCode experiments. The released repository will include scripts for seed selection, candidate generation, validation, target-model evaluation, and result aggregation. We will also provide the default hyperparameters used in our experiments, including the target model panels, number of solver attempts, repair budgets, validation thresholds, test-generation settings, and stopping criteria.

Benchmarks and datasets. Our experiments use existing public benchmark assets, including LiveCodeBench and SciCode. We cite the original benchmark papers and repositories in the main text. For LiveCodeBench, we use the Version 6 split and report results by seed difficulty. For SciCode, we use the validation split and select the subset of subproblems described in Section 4. We use these assets only for research evaluation and follow their corresponding licenses and terms of use.

Licenses. We use LiveCodeBench under its MIT License and SciCode under the Apache License 2.0. We cite the original benchmark papers and repositories and follow the corresponding licenses and terms of use. For LiveCodeBench, we use the public benchmark data linked from the official repository; for SciCode, we use the Hugging Face dataset released under Apache-2.0. Any released generated tasks, tests, logs, and metadata from our work will include appropriate attribution and license information compatible with the underlying assets.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the paper's actual contributions: a solution-centric evolution framework, empirical evaluation on LiveCodeBench and SciCode, validation/ablation results, and RL evidence for self-improvement. Claims are limited to executable coding domains and are supported by the experiments reported in the paper.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.

- The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
- The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
- It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: It's in Section 5.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: This is an empirical paper.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

Justification: They are discussed in Appendix D and E.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The project page, code repository, and dataset links are provided on the first page. Appendix K describes the released assets and their licenses.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: They are specified both in the core of the paper and in appendix.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Error bars are plotted in all the figures. Training uses different seeds.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: We present our usage of Tinker compute in Appendix D.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: The research conforms to the NeurIPS Code of Ethics. The work studies synthetic generation and evaluation of executable coding tasks, does not involve sensitive personal data or deployment on human subjects, and all human evaluation was limited to expert annotation of programming problems with appropriate anonymity and consent.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: Since our study focuses on synthetic coding tasks and benchmark construction, we expect the direct societal impact to be limited.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release a new general-purpose foundation model, scraped Internet dataset, image generator, or sensitive data. Released artifacts are limited to coding-task data, validation scripts, and experiment checkpoints for reproducibility, all in the executable programming domain. We therefore do not identify high-risk misuse requiring additional access safeguards.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

Justification: All used data and models are specified in the paper, while licenses in Appendix K.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [No]

    Justification: We provide partial code as supplementary material, but the complete evolved datasets and full asset documentation are not included at submission time. We will open-source the datasets, code, validation scripts, and documentation after publication.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [Yes]

    Justification: The paper includes an expert human evaluation of generated programming problems. We provide the annotation protocol, task instructions, rating criteria. All annotators were informed of the task purpose and compensated for their work.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [No]

    Justification: The paper includes an expert human evaluation of programming problems, but we did not obtain IRB approval or an equivalent review before the study. The evaluation involved minimal risk: participants reviewed coding tasks and provided technical ratings, without collecting sensitive personal data. We disclose the evaluation protocol and participant instructions in Appendix G.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are a core component of the proposed framework. The paper describes how LLMs are used for solution mutation, problem-statement generation, test generation, validation via evaluator/oracle roles, and target-model evaluation. We also report the model families used as evolvers, evaluators, and target solvers in the experimental setup.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.