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]

Abstract

The rapid progress of frontier large language models has led to widespread benchmark saturation, limiting the ability of existing datasets to distinguish model capabilities or provide useful training signal. For example, on LiveCodeBench, frontier models achieve over (99%) Pass@1 on easy splits and exceed (90%) on average across difficulty levels. Constructing sufficiently challenging new datasets, however, typically requires substantial human effort.

We introduce BenchEvolver, a solution-centric evolutionary framework that automatically transforms existing programming problems into harder variants. Rather than generating problems from scratch, BenchEvolver evolves reference solutions through structured transformations, then derives corresponding statements and tests from the evolved solutions. Grounding generation in executable semantics enables scalable construction of diverse, difficult tasks with verifiable correctness. Across LiveCodeBench and SciCode, BenchEvolver produces substantially harder tasks while preserving validity, reference correctness, and diversity. These tasks remain challenging even for the generating model, enabling self-improvement.

RL on evolved tasks also improves held-out coding performance: for gpt-oss-20b, seed+evolved training yields +8.7 and +8.3 Pass@1 gains on LCB v6 Hard and LCB-Pro Easy, exceeding seed-only gains by 70.7% and 34.8%, respectively. BenchEvolver thus converts saturated benchmarks into both stronger evaluations and reusable training signal.

[*]Equal contribution [1]University of California, Berkeley [2]Institute for Interdisciplinary Information Sciences, Tsinghua University. Correspondence to: Yangzhen Wu <yangzhen_wu@berkeley.edu>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

1. Introduction

Recent advances in frontier large language models (LLMs) have led to rapid saturation of widely used evaluation benchmarks, limiting their ability to meaningfully measure progress or guide further improvement. On competitive coding benchmarks such as LiveCodeBench (Jain et al., 2024), state-of-the-art models now achieve over 99% pass rate on the newest easy split and exceed 90% on average across difficulty levels. As a result, these benchmarks provide diminishing discriminative power between models and offer little gradient for training or analysis. This phenomenon is not unique to coding: even when absolute saturation levels differ, static evaluations across reasoning (Hendrycks et al., 2020; Cobbe et al., 2021), scientific problem solving (Rein et al., 2023; Tian et al., 2024), and agentic tasks (Jimenez et al., 2023; Merrill et al., 2026) increasingly lose discriminative power as frontier models improve. Consequently, continued progress increasingly depends on the availability of new, more challenging, and reliably verifiable datasets.

Human construction of new benchmarks and datasets is expensive and difficult to scale, creating a bottleneck for continuous model improvement. A natural alternative is synthetic data generation, where LLMs are used to curate new tasks for evaluation and training. Existing methods have made substantial progress in generating synthetic questions from seed data (Luo et al., 2023; Xu et al., 2025; Wei et al., 2024b;a; Ahmad et al., 2025; Pham et al., 2025; Sancaktar et al., 2026). However, many of these pipelines follow an asymmetric teacher–student paradigm: a strong model synthesizes, filters, or verifies data that is then used to train or evaluate weaker models. In addition, much of the synthesis operates at the instruction level, improving prompt diversity and surface complexity without necessarily changing the underlying solution structure or providing explicit control over task difficulty. Moving from synthetic instructions to reusable benchmarks therefore requires generating complete executable tasks, where the statement, reference solution, and tests are jointly valid. Recent work has begun to address this setting, but correctness and test validity are often ensured through stronger models, external validation, or human verification (Zhou et al., 2025a). Consequently, existing pipelines do not directly address the self-challenging setting required for open-ended improvement: generating

tasks that are valid, verifiable, difficulty-controlled, and hard for the generator itself. Without this property, synthetic data generation remains primarily a way to distill stronger models into weaker ones, rather than a mechanism by which frontier models can expose their own weaknesses and improve through training on self-generated challenges.

Toward this end, we propose BenchEvolver, a solution-centric evolutionary framework for transforming saturated programming tasks into harder yet verifiable problems. Rather than generating a new problem statement first, BenchEvolver evolves the reference solution itself, using the evolved solution as an executable oracle from which the statement, examples, and tests are derived. Each accepted mutation must change the solution structure enough to make the parent algorithm insufficient, and candidates are accepted only after independent consistency checks and empirical difficulty evaluation against a target model panel. This design measures task difficulty empirically rather than heuristically, and enables frontier models to generate challenges that expose their own weaknesses without relying on a strictly stronger teacher.

We apply our framework to LiveCodeBench (LCB) (Jain et al., 2024), a competitive-programming benchmark, and SciCode (Tian et al., 2024), a research-oriented scientific coding benchmark. Across both domains, BenchEvolver generates valid evolved tasks at scale and substantially reduces target-model pass rates. We further show that the solution-centric design outperforms a problem-centric generation baseline, that memory-guided evolution improves over independent one-step mutations, and that the evolved tasks can support closed-loop self-improvement through reinforcement learning. In particular, using `gpt-oss-20b` as both the evolver and the target model, we find that training on evolved problems improves held-out coding performance more than training on the original seed problems alone, and that the combined seed+evolved mixture performs best. Across the two informative held-out evaluation settings we study, the combined mixture yields a 70.7% and 34.8% larger improvement over the seed-only RL baseline, respectively. These results suggest that evolved tasks are not merely harder evaluation items, but can also provide useful training signal for improving the same model family that generated them.

Our contributions are summarized as follows:

- We introduce BenchEvolver, a solution-centric framework that upgrades saturated coding tasks into harder variants grounded in executable reference solutions and verifiable tests.
- We show that BenchEvolver works across both competitive-programming and scientific-coding domains. On LiveCodeBench and SciCode, across multiple evolver and target models, BenchEvolver produces valid and diverse evolved tasks that substantially reduce pass rates, including for the evolver itself.
- We provide initial evidence that self-challenging task evolution can support model improvement. For `gpt-oss-20b`, training on seed+evolved tasks or evolved tasks alone improves held-out coding performance more than training on the original seed set, indicating that evolved tasks can serve as reusable RL signal rather than only as harder benchmark items.

2. Method

We introduce BenchEvolver, a closed-loop framework for evolving saturated programming benchmarks into harder, verifiable tasks. The framework follows three principles: *generate in solution space*, *verify by independent consistency checks*, and *select by empirical model failure*. Given seed tasks, a *Proposer* constructs candidate evolutions, an *Evaluator* validates them and measures target-model difficulty, and a *Memory* module feeds past successes and failures back into search. Figure 1 provides an overview.

2.1. Self-Challenging Problem Evolution

We consider a benchmark $\mathcal{D} = \{I_i\}$ of executable programming tasks. Each task is represented as

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

where S is the natural-language statement, C is a reference implementation, T is a hidden test suite, and E is an execution harness. The harness is the only domain-specific component: for competitive-programming tasks, E runs code on stdin and checks stdout; for scientific coding tasks, E executes function-level submissions against assertion tests and domain-specific oracle artifacts when available. The rest of the framework—mutation, writing, verification, difficulty measurement, and memory-guided search—is shared across domains.

Our goal is to construct an evolved benchmark $\mathcal{D}'$ whose tasks are well-posed, verifiable, empirically harder than their seeds, and topically and algorithmically diverse. Crucially, this construction should not require a model stronger than the target panel; otherwise, the setting reduces to teacher–student distillation rather than self-challenging generation.

We define difficulty behaviorally: each target model receives multiple attempts, and an attempt succeeds only if the generated program passes all hidden tests. The average success rate across models and attempts is the empirical pass rate, with lower pass rate indicating higher difficulty. Thus, hardness is measured by executable model failure rather than assigned by the generator or an LLM judge.

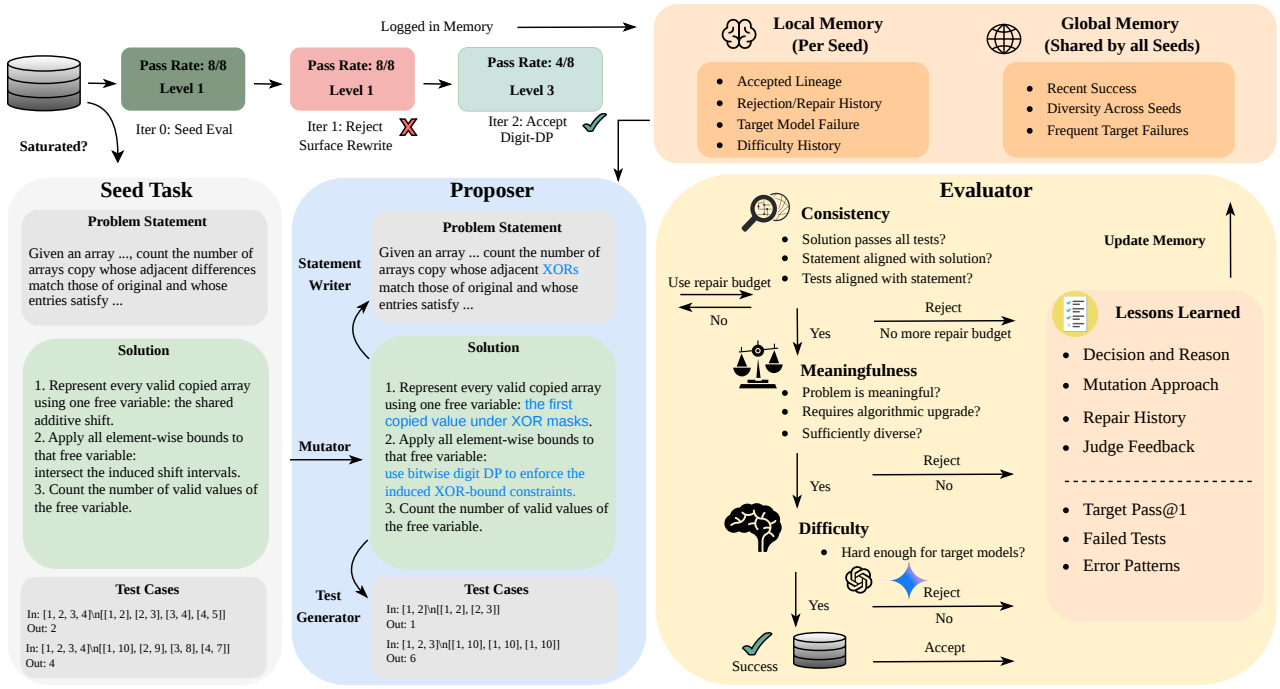


Figure 1. Overview of BenchEvolver. Starting from a saturated seed task, the proposer first mutates the reference solution and derives a new statement and tests; then the evaluator filters candidates for validity, diversity, and difficulty; memory is updated to include evolution outcomes with reasons, and accepted candidates become new parents.

2.2. Proposer: Solution-Centric Task Generation

The Proposer constructs candidate tasks by evolving solutions rather than statements. Conventional synthetic problem generation often begins with a new statement, after which the system must infer or verify whether a correct solution and tests exist. This statement-first direction is fragile in the self-challenging setting: when the same model proposes and verifies a task, ambiguous specifications, hidden assumptions, and superficial complexity can pass undetected. BenchEvolver instead follows a solution-first pipeline:

$$C \longrightarrow C' \longrightarrow I' = (S', C', T', E),$$

where the parent reference solution C is first mutated into an evolved solution C', and the statement S' and tests T' are then derived around this evolved solution under the benchmark's fixed execution harness E.

This design makes difficulty a property of the task's computation rather than its surface form. The Proposer must introduce a *dominant algorithmic lift*: a substantive change to the solution structure that makes the parent approach insufficient. In this way, evolved tasks become harder because they require stronger algorithms, not because they contain longer statements, adversarial formats, or obscure edge cases.

Each proposal is produced by first mutating the parent reference implementation into an evolved solution C', together with a concise explanation of the new algorithmic idea and why the parent solution fails. The Proposer then derives a natural-language statement S', public examples, and hidden tests around C'. Public examples and expected test outputs are materialized by executing the evolved reference solution, anchoring the task in executable behavior. For stdin–stdout tasks, tests are organized into small, medium, large, and stress regimes; for scientific function-level tasks, they are assertion-style tests executed by the domain harness.

The Proposer is conditioned on feedback from previous attempts, including parent difficulty, accepted mutations, rejection reasons, target-model error patterns, and global diversity signals. It does not decide whether a candidate is valid or difficult; its role is to propose a complete executable task, while acceptance is left to the Evaluator.

2.3. Evaluator: Verification and Empirical Selection

The Evaluator ensures that a candidate is both valid and genuinely challenging. Validity means that the statement, reference solution, tests, and execution harness define the same task; difficulty is measured only after this consistency is established.

For validation, BenchEvolver uses benchmark-specific checks rather than relying on a single judge. In competitive programming, it triangulates among the evolved reference solution, a statement-only brute-force solver, and a statement-only public-output oracle. Since these witnesses observe different information, their disagreements help identify whether the issue lies in the reference, the brute-force solver, the public outputs, or the specification. For scientific coding, where brute-force solvers are less applicable, BenchEvolver uses a statement-faithfulness check: an independently generated solution from the statement is run against the candidate tests to assess whether the written task sufficiently determines the intended computation. Full validation and repair details are given in Appendix C.

After validation, each target model receives multiple attempts, and an attempt succeeds only if it passes all hidden tests. The resulting pass rate is mapped to the same difficulty scale used for seed tasks. A candidate is accepted only if it improves over the seed difficulty, and optionally over the parent difficulty, making selection empirical rather than judge-assigned.

Finally, the Evaluator filters out false difficulty: ambiguous wording, misleading I/O, underspecified constraints, unnatural edge cases, or near-duplicate reskins. Localized failures are routed to bounded repair; candidates that cannot be made consistent within the repair budget are rejected. Thus, accepted tasks must be both executable and empirically hard.

2.4. Memory-Guided Evolution

Memory turns BenchEvolver from repeated sampling into adaptive search. Each seed maintains a local memory of its lineage, including accepted mutations, failed attempts, validation issues, target-model pass rates, and observed error patterns. This history is summarized and fed back to the Proposer, helping later mutations avoid repeated failures and focus on directions that have exposed model weaknesses.

BenchEvolver also maintains a global memory across seeds. This memory records accepted and attempted mutation families throughout the run, encouraging different lineages to explore distinct algorithmic directions rather than rediscovering the same lift under different surface forms. It is also used in selection: when a mutation family has already succeeded elsewhere, a new candidate from that family must provide a larger difficulty gain to be accepted. Diversity is therefore enforced both through generation context and through the acceptance rule.

Together, local and global memory give BenchEvolver a lightweight evolutionary structure. Mutation comes from solution-centric proposal; selection comes from validation and empirical target-model failure; inheritance comes from accepted lineages; and diversity is maintained through memory. The resulting tasks form a hard, verified distribution that can serve both as an evolved benchmark and as reinforcement-learning data, enabling a closed loop in which models generate challenges, train on executable rewards, and return to generate harder tasks.

The full pseudocode of BenchEvolver is provided in Appendix B.

3. Experiments

We evaluate BenchEvolver in two different coding domains: competitive programming and research-oriented scientific coding. The former uses LiveCodeBench, a collection of stdin–stdout problems evaluated by hidden tests; the latter uses SciCode, a benchmark of function-level scientific programming tasks requiring domain-specific routines. Together, these settings test whether BenchEvolver can generate valid, diverse, and empirically harder tasks across both competition-style algorithm design and scientific code generation.

For each benchmark, we consider two target tiers. The *lightweight* tier consists of GPT-5.4-mini and Gemini-3-Flash, while the *frontier* tier consists of GPT-5.4 and Gemini-3.1-Pro. We exclude Claude models from the target pool because they perform slightly worse on both benchmarks; including them could cause evolved tasks to overfit to Claude-specific failure modes rather than reflect difficulty across the entire target model class.

3.1. Competitive Programming: LiveCodeBench

For LiveCodeBench, we randomly sample 65 seed problems spanning easy, medium, and hard difficulty levels from v6, due to the high cost of running multi-model evolution and target evaluation. We use GPT-5.4-mini, Gemini-3-Flash, and Claude-Sonnet-4.6 as evolvers for the lightweight-target setting. To synthesize tasks that remain challenging even for the strongest target models, we additionally select 10 saturated problems from each difficulty level and use Gemini-3.1-Pro as the evolver.

We use the LiveCodeBench-specific brute-force triangulation protocol described in Appendix C.1 to validate candidates before target-model evaluation.

Synthesis yield and validity. Table 1 shows that BenchEvolver produces evolved tasks with high yield and validity across seed difficulty levels and evolver models. Compared with the problem-centric baseline, BenchEvolver succeeds on substantially more seeds and produces candidates with much higher post-hoc validity. This supports our solution-first design: committing first to executable algorithmic logic makes it easier to recover a coherent statement and test

Method	Target Tier	Evolver Model	Evolved Seed Fraction			Validity
			Easy	Medium	Hard	Avg.
Problem-Centric	Lightweight	GPT-5.4-mini	12/22 (54.5%)	11/23 (47.8%)	6/20 (30.0%)	79.3%
Memory-Free	Lightweight	GPT-5.4-mini	13/22 (59.1%)	15/23 (65.2%)	9/20 (45.0%)	86.2%
BenchEvolver	Lightweight	GPT-5.4-mini	17/22 (77.3%)	16/23 (69.6%)	12/20 (60.0%)	97.7%
BenchEvolver	Lightweight	Gemini-3-Flash	18/22 (81.9%)	14/23 (60.9%)	16/20 (80.0%)	89.9%
BenchEvolver	Lightweight	Claude-Sonnet-4.6	14/22 (63.6%)	15/23 (65.2%)	7/20 (35.0%)	93.4%
BenchEvolver	Frontier	Gemini-3.1-Pro	8/10 (80.0%)	9/10 (90.0%)	6/10 (60.0%)	96.7%

Table 1. LiveCodeBench-v6 evolution yield and validity by seed difficulty. Easy, Medium, and Hard report the fraction of completed seeds for which at least one accepted evolved problem is generated. Validity is the mean post-hoc validity rate of accepted evolved problems judged by Claude Code Opus 4.7. Problem-Centric and Memory-Free are ablations using GPT-5.4-mini as the evolver.

suite, whereas statement-first generation more often produces tasks that are underspecified, inconsistent, or difficult to validate. The memory-free ablation also underperforms BenchEvolver, indicating that iterative evolution is important beyond one-shot mutation. Accepted candidates provide useful lineage context, allowing later proposals to build on successful algorithmic lifts, avoid repeated failure modes, and explore harder variants. Overall, the results suggest that the proposer–evaluator–memory loop is not merely improving sample efficiency, but is essential for reliably converting saturated seeds into valid evolved problems.

Evolved tasks are empirically harder. Figure 3 shows that accepted evolved tasks substantially reduce target-model pass rates relative to their original seeds across both lightweight and frontier models. The effect is consistent across seed difficulty splits and across different evolver models, suggesting that the generated tasks are not narrow artifacts of a single model or prompt configuration. Importantly, evolved tasks remain challenging even when the target is the evolver itself: each evolver exhibits a clear pass-rate drop on the problems it generated. This is the key distinction from teacher–student synthetic data generation. BenchEvolver is not simply using a stronger model to create data for weaker models; it constructs verified tasks that push against the current capability frontier of the generator itself. Together with the high validity in Table 1, these results show that BenchEvolver can produce tasks that are both trustworthy enough for evaluation and difficult enough to serve as self-challenging training signal.

Human evaluation: algorithmic diversity. We complement the executable evaluation in Section 2.3 with a blind human study: six competitive-programming experts (Codeforces master / IOI / ICPC level) reviewed 100 evolved seed lineages spanning 72 distinct LiveCodeBench seeds and 207 distinct evolved problems, listing on each problem the algorithms and data structures it requires. Each problem received 1.87 reviewers on average; we aggregate by unioning the normalized algorithm tags. Here we focus on whether BenchEvolver broadens the algorithmic surface area beyond the seed pool. The full breakdown of clarity, novelty, difficulty and validity is reported in Appendix F.

As shown in Figure 2, the seed pool is dominated by a single algorithmic regime—*Search/simulation* alone accounts for 32.7% of seed-tag mentions—whereas the evolved set has *no* dominant category and instead spreads mass across many advanced data structures and algorithmic regimes that are essentially absent from the seeds (e.g. HLD/LCT, AC automaton, polynomial / matrix methods). Overall, the number of distinct algorithmic categories triggered grows from 19 (seed) to 30 (evolved), and within each lineage 95.6% of seed groups exhibit at least one category the seed itself never required, with a mean of 2.54 new categories per group. BenchEvolver therefore does not just produce harder versions of the same task; it broadens the algorithmic surface area, which is the property required if the evolved set is to function both as a benchmark and as RL training signal exposing a varied set of model weaknesses.

3.2. Self-improvement through Reinforcement Learning

The results above show that BenchEvolver can transform saturated seed problems into harder variants that better distinguish frontier models for evaluation. We next ask whether these evolved problems are not only harder benchmarks, but also useful training signal. This is the central promise of self-challenging data generation: if a model can evolve problems that remain valid, diverse, and difficult for itself, then the resulting tasks should expose weaknesses that can be directly optimized through reinforcement learning.

To test this, we use `gpt-oss-20b` as both the evolver and the target model. We take 880 LiveCodeBench v1–v5 problems released before January 2025 as the seed pool and hold out LiveCodeBench v6 and LiveCodeBench-Pro as evaluation sets. We run BenchEvolver on seeds that `gpt-oss-20b` solves correctly in all 5 evaluation attempts, i.e., problems saturated for the model. For each seed, we allow up to 10 evolution iterations and accept at most 2 evolved problems; a candidate is accepted if its empirical accuracy drops below the seed accuracy while remaining nonzero, so that the evolved task is challenging but not degenerate. In total, 404 seeds produce accepted evolutions,

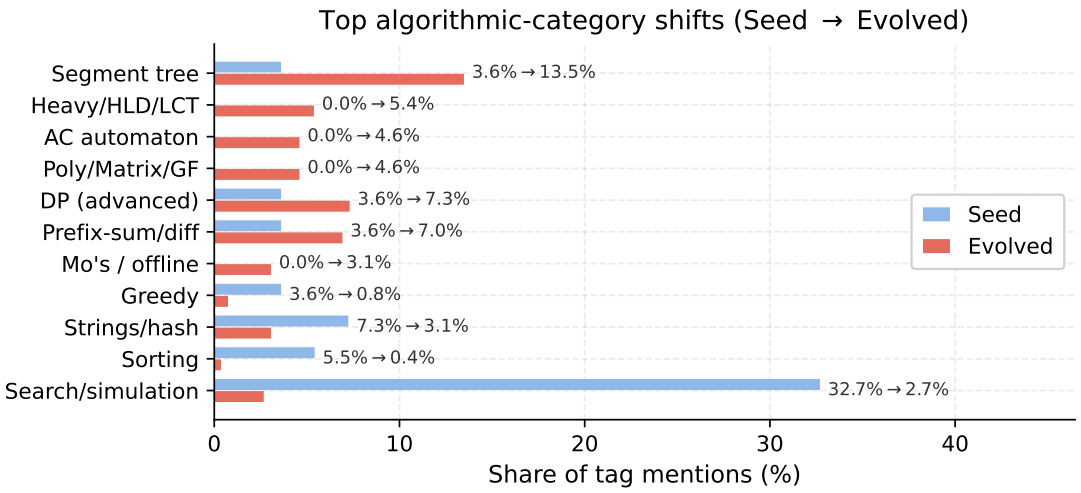


Figure 2. Top eleven algorithm/data-structure categories ordered by absolute seed→evolved share shift. Numbers next to each bar pair show seed-share → evolved-share. Full category breakdown is in Appendix F.

yielding 586 evolved problems. We then construct three training sets: the original seed set with 880 problems, the evolved set with 586 problems, and their union with 1,466 problems. We train `gpt-oss-20b` with GRPO using Tinker, running two independent random seeds for each training set under fixed configurations: 64 problems per batch, 16 rollouts per problem, and a maximum output length of 24K tokens for training, 30K for test. For evaluation, we report results on LiveCodeBench v6 Hard (80 problems) and LiveCodeBench-Pro Easy (96 problems), which provide the most informative difficulty range for `gpt-oss-20b`; the remaining splits are either nearly saturated or too difficult to yield meaningful comparisons.

Figure 4 shows that evolved problems provide useful training signal beyond the original seed distribution. On LCB v6 Hard, RL on the 880 original seed problems improves the base model from 40.0% to 45.1%, a $+5.1$ point gain. Training on the 586 evolved problems achieves a larger gain, reaching 47.6% ($+7.6$ points), while the combined seed+evolved mixture performs best at 48.7% ($+8.7$ points). Relative to the all-seeds baseline, evolved-only training yields a 50.0% larger improvement over the base model, and the combined mixture yields a 70.7% larger improvement.

The same trend holds on LCB-Pro Easy. The all-seeds baseline improves from 64.6% to 70.8%, a $+6.2$ point gain, whereas evolved-only training reaches 71.8% ($+7.2$ points) and the seed+evolved mixture reaches 72.9% ($+8.3$ points). Compared with the all-seeds baseline, evolved-only training gives a 16.3% larger improvement over the base model, and the combined mixture gives a 34.8% larger improvement. These gains indicate that evolved problems are not merely harder evaluation items; they provide complementary RL signal that improves generalization to held-out future coding benchmarks.

These results support the value of self-challenging data generation. Importantly, the evolved problems are generated from the subset of seed problems that `gpt-oss-20b` solves correctly in all five attempts. Thus, BenchEvolver specifically targets regions of the training pool where the current model receives little or no corrective signal, and converts them into harder, verified tasks that expose new failure modes. This converts inference-time computation into reusable training data: the model first spends inference compute to discover valid challenges that defeat its current policy, and reinforcement learning then amortizes those challenges into improved parameters. The strongest performance comes from mixing seeds and evolved tasks, suggesting that the two distributions are complementary: the original seeds preserve coverage of the source benchmark, while the evolved problems push the model toward harder algorithmic regimes. This provides evidence that BenchEvolver can serve not only as a benchmark-evolution method, but also as a practical self-play mechanism for model improvement. We provide the training details in Appendix D.

3.3. Scientific Coding: SciCode

For SciCode, we select 30 self-contained subproblems spanning 15 main problems from the validation split. Among them, 27 are saturated by the lightweight target tier and 28 by the frontier target tier, serving as seeds for generating harder scientific-coding variants. Since SciCode does not naturally admit brute-force validation, we instead use a customized statement-faithfulness protocol, described in Appendix C.2.

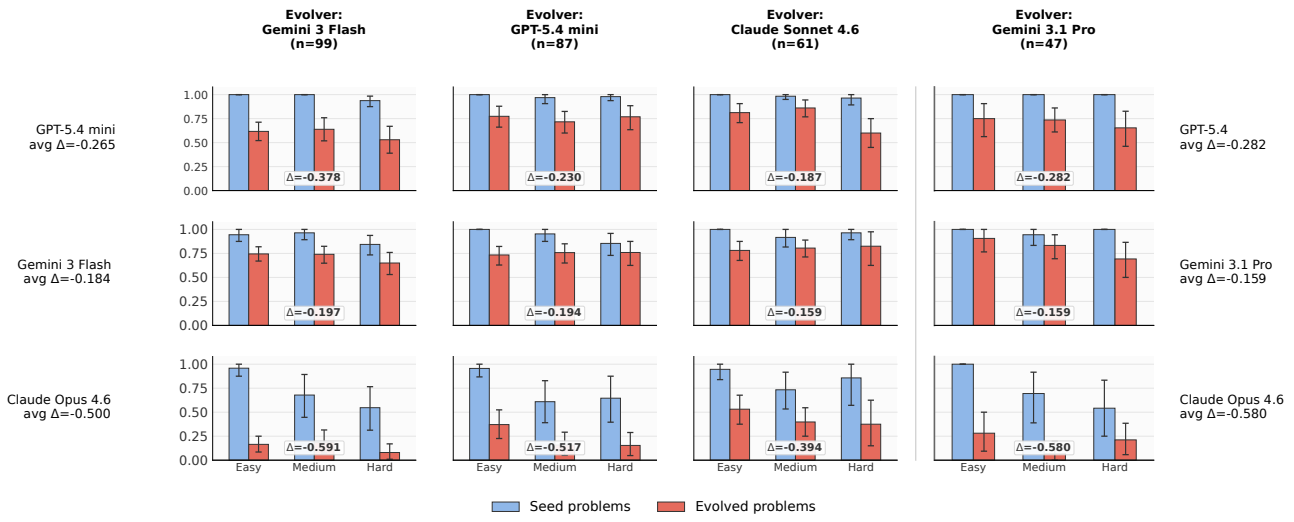
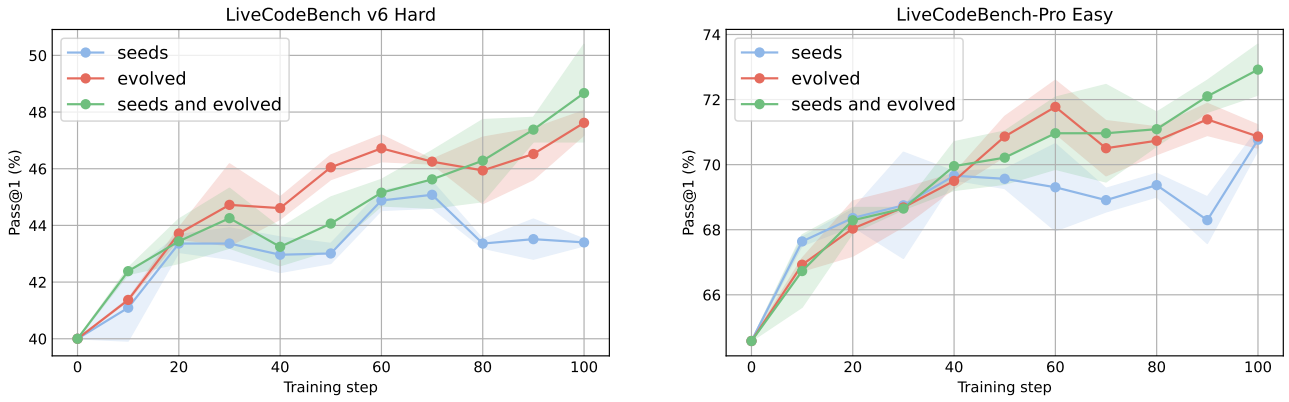


Figure 3. Pass@1 on original LiveCodeBench seed problems versus evolved problems. Each column group corresponds to an evolver model, and each row corresponds to a target model. Within each subfigure, bars show Easy, Medium, and Hard seeds from left to right. Evolved problems consistently reduce target-model pass rates across both lightweight and frontier models ($k = 4$ attempts per model).



(a) LCB v6 Hard

(b) LCB-Pro Easy

Figure 4. Test accuracy across training steps for three RL data mixes (mean ± std over 2 seeds). Step 0 shows the base model. Pass@1 is computed by averaging 16 samples per problem.

Table 2 and Figure 5 show that BenchEvolver generalizes beyond competitive programming to scientific coding tasks. Despite the smaller seed pool and the absence of brute-force oracles, the framework still produces accepted evolved tasks with high validity across evolver models. In particular, GPT-5.4-mini and Gemini-3-Flash evolve the majority of saturated lightweight-tier seeds while maintaining over 90% validity, and Gemini-3.1-Pro also produces valid evolved tasks in the frontier-tier setting. More importantly, the accepted SciCode tasks consistently reduce pass@1 for both lightweight and frontier target models relative to the original seeds. This indicates that the solution-first principle is not specific to stdin–stdout algorithmic problems: by replacing brute-force validation with statement-faithfulness checks and assertion-based execution, BenchEvolver can also evolve scientific function-level tasks whose difficulty comes from more precise computational specifications rather than contest-style input constraints. These results support the broader claim that BenchEvolver is a general executable-task evolution framework, with benchmark-specific validation modules rather than benchmark-specific generation logic.

4. Conclusion and Future Work

4.1. Conclusion

We presented BenchEvolver, a solution-centric framework for turning saturated executable coding tasks into harder, verified challenges. The key idea is to evolve the computation first—by mutating reference solutions—and then recover

Target Tier	Evolver Model	Evolved seed fraction	Valid ratio
Lightweight	GPT-5.4-mini	24/27(88.9%)	91.0%
Lightweight	Gemini-3-Flash	26/27(96.3%)	98.2%
Lightweight	Claude-Sonnet-4.6	8/27(29.6%)	90.0%
Frontier	Gemini-3.1-Pro	17/28(60.7%)	88.9%

Table 2. SciCode evolution achieves high yield and validity, producing accepted evolved tasks for most saturated seeds across target tiers and evolver models.

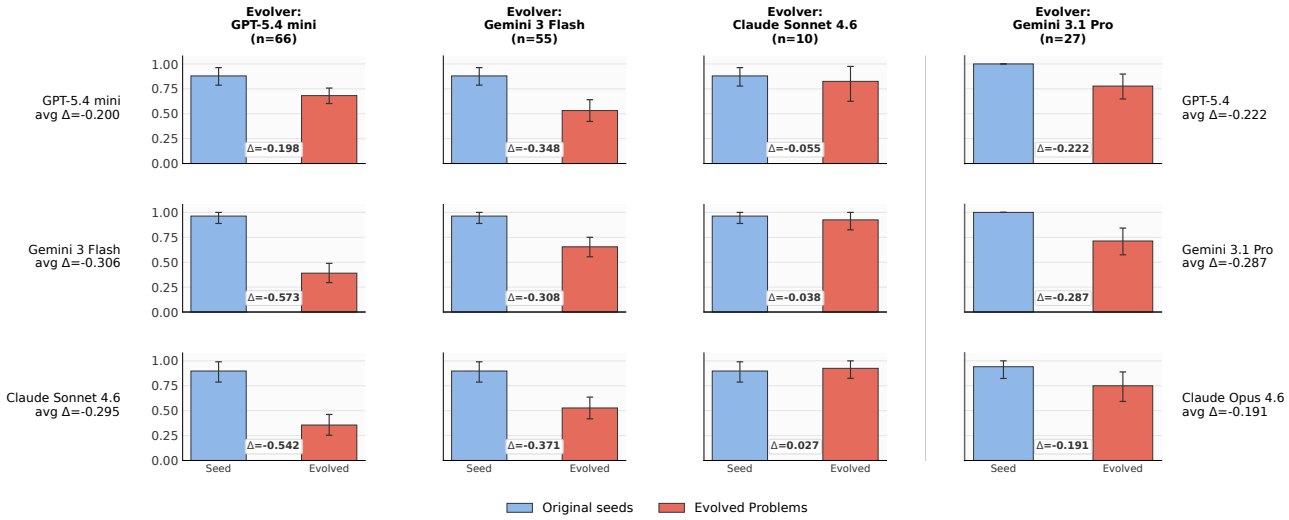


Figure 5. Pass@1 on original SciCode seed problems versus evolved problems. Across both the lightweight and frontier target models, evolved problems substantially reduce model pass rates (k=4 attempts per model).

statements and tests around the evolved solution, keeping task generation grounded in executable semantics. Across LiveCodeBench and SciCode, BenchEvolver produces valid, diverse tasks that substantially reduce target-model pass rates, including for the models that generated them. Building on these results, we curate LIVECODEBENCH-PLUS, a 91-problem benchmark that combines 64 evolved and 27 difficult original LiveCodeBench-v6 tasks and restores meaningful discrimination among frontier coding models. More importantly, evolved tasks are not only useful as harder benchmarks: they also provide actionable training signal. Our RL experiments show that training on evolved problems improves held-out coding performance beyond training on the original seeds alone. This suggests a broader role for benchmark evolution: instead of treating evaluation datasets as static artifacts that inevitably saturate, models can use inference-time computation to discover their own failures, convert those failures into verified training environments, and improve through closed-loop self-play.

4.2. Future Work

Scaling closed-loop RL self-improvement. Our RL experiments instantiate one round of self-improvement: a model evolves saturated problems into harder verified tasks, trains on them with executable rewards, and improves on held-out coding benchmarks. Scaling this into multiple rounds, where each improved model becomes the next evolver, raises challenges in stability, curriculum design, and diversity. Narrow selection may overfit to recurring failure modes, while aggressive selection may yield difficult but uninformative tasks. Principled control of difficulty and task diversity will be essential for scalable self-challenging training.

Toward living benchmarks. Finally, our results suggest that benchmark construction should move beyond static datasets. Any fixed benchmark will eventually saturate as models improve, especially in executable domains where training signal can be extracted from public tasks. Instead of releasing a benchmark as a one-time artifact, future work could maintain a reproducible evolution pipeline that periodically generates, validates, audits, and calibrates new tasks against current frontier models. Such a living benchmark would make evaluation adaptive to model progress while preserving transparency through versioned releases, held-out tests, and documented validation protocols. More broadly, this would align evaluation and training: the same verified tasks that reveal current model failures can also become the environments used to improve future models.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Ahmad, W. U., Narenthiran, S., Majumdar, S., Ficek, A., Jain, S., Huang, J., Noroozi, V., and Ginsburg, B. Opencodereasoning: Advancing data distillation for competitive coding. *arXiv preprint arXiv:2504.01943*, 2025.

Assumpção, H., Ferreira, D., Campos, L., and Murai, F. Codeevolve: an open source evolutionary coding agent for algorithmic discovery and optimization, 2026. URL https://arxiv.org/abs/2510.14150.

Bailey, L., Wen, K., Dong, K., Hashimoto, T., and Ma, T. Scaling self-play with self-guidance. *arXiv preprint arXiv:2604.20209*, 2026.

Cemri, M., Agrawal, S., Gupta, A., Liu, S., Cheng, A., Mang, Q., Naren, A., Erdogan, L. E., Sen, K., Zaharia, M., et al. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

Cheng, A., Liu, S., Pan, M., Li, Z., Wang, B., Krentsel, A., Xia, T., Cemri, M., Park, J., Yang, S., et al. Barbarians at the gate: How ai is upending systems research. *arXiv preprint arXiv:2510.06189*, 2025.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Fernando, C., Banarse, D., Michalewski, H., Osindero, S., and Rocktäschel, T. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Guo, Q., Wang, R., Guo, J., Li, B., Song, K., Tan, X., Liu, G., Bian, J., and Yang, Y. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv e-prints*, pp. arXiv–2309, 2023.

Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Jiang, J., Ding, T., and Zhu, Z. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. *arXiv preprint arXiv:2602.02919*, 2026.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Lab, T. M. Tinker, 2025. URL https://thinkingmachines.ai/tinker/.

Lange, R. T., Imajuku, Y., and Cetin, E. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Liu, S., Agarwal, S., Maheswaran, M., Cemri, M., Li, Z., Mang, Q., Naren, A., Boneh, E., Cheng, A., Pan, M. Z., et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., Shi, L., Shin, J. Y., Walshe, T., Buchanan, E. K., et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J., Mehrabian, A., et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Pham, M. V., Phan, H. N., Phan, H. N., Chi, C. L., Nguyen, T. N., and Bui, N. D. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *arXiv preprint arXiv:2504.14757*, 2025.

Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J., Ellenberg, J. S., Wang, P., Fawzi, O., et al. Mathematical discoveries from program search with large language models. *Nature*, 625 (7995):468–475, 2024.

Sancaktar, C., Zhang, D., Synnaeve, G., and Cohen, T. A deep dive into scaling rl for code generation with synthetic data and curricula. *arXiv preprint arXiv:2603.24202*, 2026.

Shojaee, P., Nguyen, N.-H., Meidani, K., Farimani, A. B., Doan, K. D., and Reddy, C. K. Llm-srbench: A new benchmark for scientific equation discovery with large language models. *arXiv preprint arXiv:2504.10415*, 2025.

Sun, Q., Gong, J., Li, L., Guo, Q., and Yuan, F. Codeevo: Interaction-driven synthesis of code-centric data through hybrid and iterative feedback. *arXiv preprint arXiv:2507.22080*, 2025.

Tian, M., Gao, L., Zhang, S. D., Chen, X., Fan, C., Guo, X., Haas, R., Ji, P., Krongchon, K., Li, Y., et al. Scicode: A research coding benchmark curated by scientists. *Advances in Neural Information Processing Systems*, 37: 30624–30650, 2024.

Wang, A., Yan, Y., Zhou, N., Lu, Z., Lu, W., Xiao, J., Zhuang, Y., and Shen, Y. Code-a1: Adversarial evolving of code llm and test llm via reinforcement learning. *arXiv preprint arXiv:2603.15611*, 2026.

Wei, Y., Cassano, F., Liu, J., Ding, Y., Jain, N., Mueller, Z., de Vries, H., Von Werra, L., Guha, A., and Zhang, L. Selfcodealign: Self-alignment for code generation. *Advances in Neural Information Processing Systems*, 37: 62787–62874, 2024a.

Wei, Y., Wang, Z., Liu, J., Ding, Y., and Zhang, L. Magicoder: Empowering code generation with oss-instruct, 2024b. URL https://arxiv.org/abs/2312.02120.

Wu, J., Li, H., Zhang, X., Guo, J., Luo, J., Liu, S., Huang, Y., Chu, R., Li, S., and Yang, Y. X-coder: Advancing competitive programming with fully synthetic tasks, solutions, and tests. *arXiv preprint arXiv:2601.06953*, 2026.

Wu, R., Wang, X., Mei, J., Cai, P., Fu, D., Yang, C., Wen, L., Yang, X., Shen, Y., Wang, Y., et al. Evolver: Self-evolving llm agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079*, 2025.

Wu, Y., Sun, Z., Li, S., Welleck, S., and Yang, Y. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*, 2024.

Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., Tao, C., Lin, Q., and Jiang, D. Wizardlm: Empowering large pre-trained language models to follow complex instructions, 2025. URL https://arxiv.org/abs/2304.12244.

Yan, M., Peng, B., Coleman, B., Chen, Z., Xie, Z., Chen, S., He, Z., Sachdeva, N., Ye, I., Wang, W., et al. Pacevolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026.

Zhang, Q., Hu, C., Upasani, S., Ma, B., Hong, F., Kamanuru, V., Rainton, J., Wu, C., Ji, M., Li, H., et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

Zhang, R., Bai, R. H., Zheng, H., Jaitly, N., Collobert, R., and Zhang, Y. Embarrassingly simple self-distillation improves code generation. *arXiv preprint arXiv:2604.01193*, 2026.

Zhou, S., Zheng, Z., Liu, K., Shen, Z., Cheng, Z., Chen, Z., He, H., Yao, J., Mao, H., Mang, Q., et al. Autocode: Llms as problem setters for competitive programming. *arXiv preprint arXiv:2510.12803*, 2025a.

Zhou, Y., Levine, S., Weston, J., Li, X., and Sukhbaatar, S. Self-challenging language model agents. *arXiv preprint arXiv:2506.01716*, 2025b.

A. 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 (Luo et al., 2023; Xu et al., 2025), while SelfCodeAlign generates instruction data for code alignment (Wei et al., 2024a). Other work distills reasoning traces for competitive programming (Ahmad et al., 2025), synthesizes repository-level bug-fix tasks (Pham et al., 2025), or constructs synthetic curricula for RL-based code generation (Sancaktar et al., 2026). 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 (Zhou et al., 2025a); X-Coder similarly studies fully synthetic competitive-programming data for training code reasoning models (Wu et al., 2026). 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 current model behavior is used to create new supervision. Self-Challenging Agents instantiate this idea for tool-use agents: the model plays a challenger role to generate verifiable tasks and then trains an executor role using RL feedback (Zhou et al., 2025b). Self-Guided Self-Play studies a solver–conjecturer loop for formal theorem proving, adding a guide role to prevent the conjecturer from collapsing into artificial or unhelpful problems (Bailey et al., 2026). In code generation, related work studies iterative code-centric data synthesis through Coder–Reviewer feedback (Sun et al., 2025), self-improvement from model-generated solutions (Zhang et al., 2026), and adversarial co-evolution between code-generation and test-generation models (Wang et al., 2026). These works share our motivation that static datasets can become saturated, and that generated tasks, tests, or solutions can provide renewable training signal.

Our work differs in the object being evolved and in the role of verification. Rather than generating auxiliary supervision, tool-use tasks, theorem-proving subgoals, or tests for a fixed programming problem, BenchEvolver evolves complete coding benchmark items. Each accepted item must include a natural-language statement, executable reference solution, and valid test suite, and must empirically reduce pass rates of the target model panel. Moreover, while prior self-play systems often need additional mechanisms to prevent degenerate or artificial challenges, BenchEvolver anchors generation in executable solution evolution and filters candidates through independent consistency checks and target-model evaluation. This makes the generated tasks suitable both as harder benchmarks and as RL training environments for closed-loop self-improvement.

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 (Fernando et al., 2023; Guo et al., 2023; Agrawal et al., 2025; Zhang et al., 2025), program and algorithm discovery (Romera-Paredes et al., 2024; Novikov et al., 2025; Shojaee et al., 2025; Cheng et al., 2025), and adaptive evolutionary search, including diversity-driven program evolution, meta-evolution, and long-horizon progress-aware optimization (Lange et al., 2025; Jiang et al., 2026; Yan et al., 2026; Cemri et al., 2026; Liu et al., 2026). Related work also studies automated or self-evolving agent systems (Hu et al., 2024; Wu et al., 2025) and open-source evolutionary coding agents (Assumpção et al., 2026).

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 (Wu et al., 2024) 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 these memories and difficulty levels for proposal step b; $r(I')$ is the validation predicate; and $A(I')$ is the final acceptance predicate.

Algorithm 1 BenchEvolver: Solution-Centric Evolution

Require: Seed benchmark $\mathcal{D} = \{I_i = (S_i, C_i, T_i, E_i)\}_{i=1}^{n}$, target solver panel $\Pi = \{\pi_j\}_{j=1}^{J}$, attempts per solver K, evolution budget B, minimum level gain Δ
Ensure: Evolved benchmark $\mathcal{D}'$
$V_E(\hat{C}, T) = 1$ iff $\hat{C}$ passes all tests T under harness E.
$a(I; \Pi, K) = \frac{1}{JK} \sum_{j=1}^{J} \sum_{k=1}^{K} V_E(\hat{C}_{j,k}, T)$ and $\ell(I) = \phi(a(I; \Pi, K))$.
$L_i, m_i, G, h_{i,b}, r(I'), A(I')$ are defined in Appendix B.
$\mathcal{D}' \leftarrow \emptyset$, $G \leftarrow \emptyset$
for all $I_i = (S_i, C_i, T_i, E_i) \in \mathcal{D}$ **do**
  $a_i \leftarrow a(I_i; \Pi, K)$, $\ell_i \leftarrow \phi(a_i)$, $L_i \leftarrow [\,]$, $m_i \leftarrow \emptyset$
  for $b = 1, \ldots, B$ **do**
    {PARENT/CONTEXT: latest accepted child, not sampled parent}
    $I_p = (S_p, C_p, T_p, E_p) \leftarrow \mathrm{Last}(L_i)$ if $L_i \neq \emptyset$, else I_i; $h_{i,b} \leftarrow (m_i, G, \ell_i, \ell(I_p))$
    {PROPOSER: evolve solution, then derive statement/tests}
    $C' \sim q_\theta(\cdot \mid C_p, h_{i,b})$; materialize examples by executing C'
    $S' \sim w_\theta(\cdot \mid C')$, $T' \sim g_\theta(\cdot \mid S', C')$; set $I' = (S', C', T', E_i)$
    {VALIDATION: executable consistency with bounded repair}
    $r(I') \leftarrow \mathbf{1}\{\mathrm{Consistent}(S', C', T', E_i)\}$; repair and recompute $r(I')$ if needed
    if $r(I') = 0$ **then**
      record rejection in m_i and continue
    end if
    {DIFFICULTY/SELECTION: target-panel failure after validation}
    $a' \leftarrow \frac{1}{JK} \sum_{j=1}^{J} \sum_{k=1}^{K} V_{E_i}(\hat{C}'_{j,k}, T')$, $\ell' \leftarrow \phi(a')$
    $A(I') \leftarrow r(I') \wedge [\ell' \geq \ell(I_p)] \wedge [\ell' \geq \ell_i + \Delta] \wedge \neg\mathrm{Artificial}(I')$
    if $A(I') = 1$ **then**
      $\mathcal{D}' \leftarrow \mathcal{D}' \cup \{I'\}$; append I' to L_i; update m_i and G
    else
      Record rejection reason and observed failures in m_i
    end if
  end for
end for
return $\mathcal{D}'$

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 (Lab, 2025) 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 η=1×10^{-5} and a single optimizer substep per batch (K=1). We do not apply a KL penalty to the reference policy (β_{KL}=0) and instead rely on the LoRA bottleneck and the group-relative advantages (Sec. D.2) to keep the policy close to the base model. Conversations are formatted with the `gpt_oss_medium_reasoning` chat renderer, which preserves the model's native thinking–answer structure.

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 3. 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 at nucleus sampling temperature defaults. 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, so the policy is not punished for trying. (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 6 reports the on-policy training reward and average response length for the three RL data mixtures. The reward curves start at different levels because the datasets have different difficulty: seed problems are easier for the base policy and therefore begin with higher reward, while evolved problems are intentionally harder and start substantially lower. This gap is expected and reflects the construction of BenchEvolver: evolved tasks are selected to expose failures of the current model rather than to maximize initial reward. Across training, all mixtures improve, but with different dynamics. Seed-only training begins high and quickly saturates, suggesting limited remaining learning signal. Evolved-only training starts from the lowest reward but rises steadily throughout training, indicating that the evolved distribution provides nontrivial gradient signal over many updates. The combined seed+evolved mixture lies between the two at initialization and also improves consistently, balancing easier problems that stabilize training with harder evolved tasks that continue to drive learning.

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.

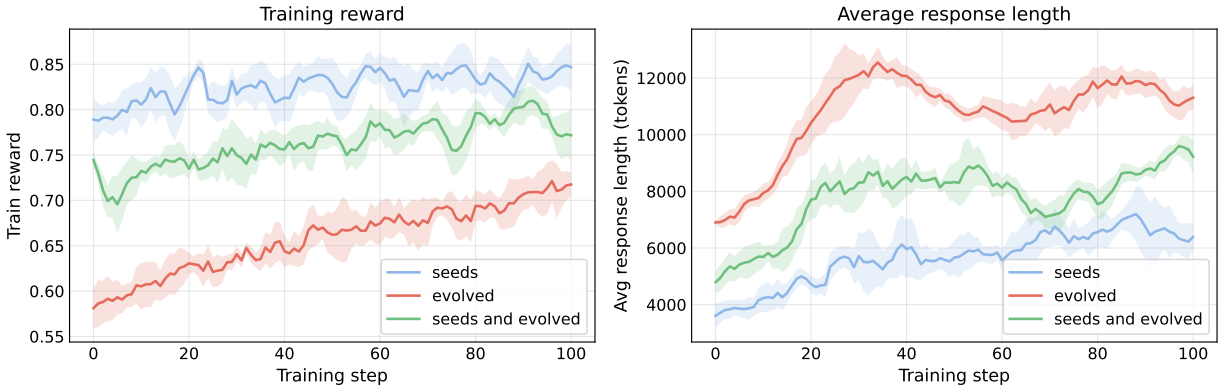


Figure 6. Training reward (left) and average response length in tokens (right), for the three data conditions (seeds, evolved, seeds and evolved). Solid lines are the mean of two random seeds per condition; shaded bands are ± 1 standard deviation across the two seeds. A boxcar smoother of width 5 is 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.

E. Evolution Configurations and Hyperparameters

The default configurations for our LiveCodeBench and SciCode experiments are summarized in Table 4. The table reports the model settings, acceptance criteria, repair and validation budgets, test-generation parameters, and memory mechanisms used by our evolution pipeline.

F. Human Evaluation: Full Breakdown

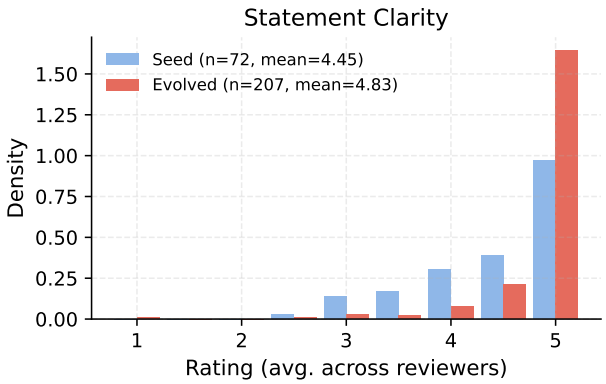
This appendix provides the full human-evaluation breakdown for Paragraph 3.1 and Figure 2, including distributional statistics (Figure 7) and the complete algorithm-category distribution (Figure 8).

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 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, union algorithm tags 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.

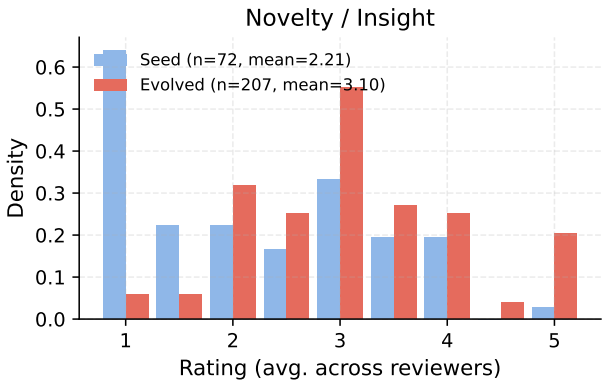
Results. Figure 7 shows a consistent shift from the seed problems to the evolved problems. The evolved problems are rated substantially harder (panel c; $1.83 \rightarrow 3.21$), and the estimated Codeforces ratings (panel d) move from a concentrated pupil/specialist range around 1100 to a broader distribution 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, the human study supports the main claim that BenchEvolver produces problems that are harder and diverse while remaining well specified.

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

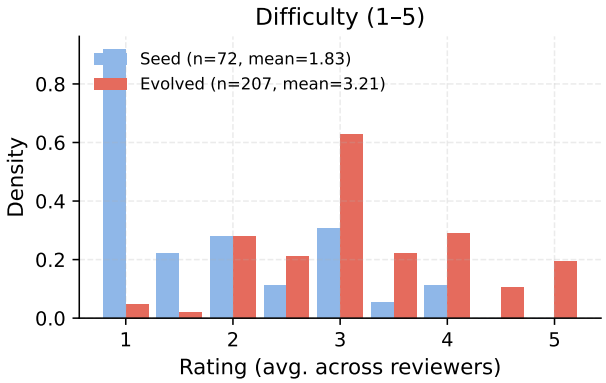
Table 4. Default evolution configurations for LCB and SciCode.



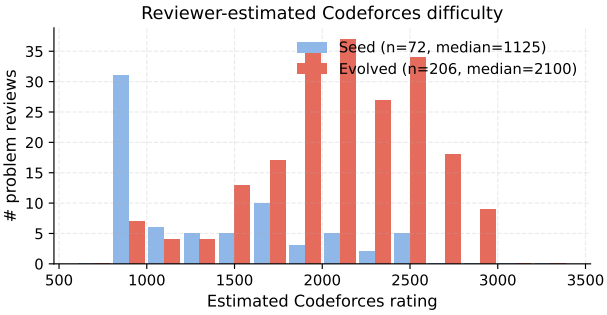
(a) Clarity.



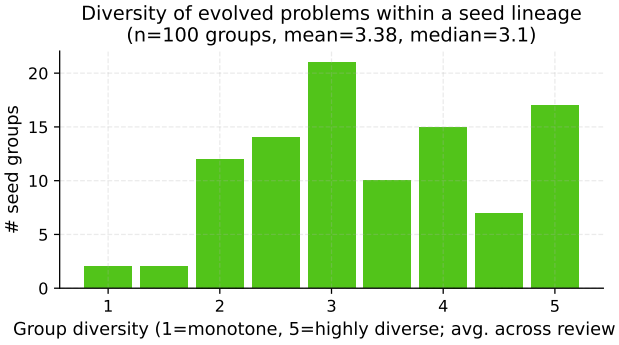
(b) Novelty / insight.



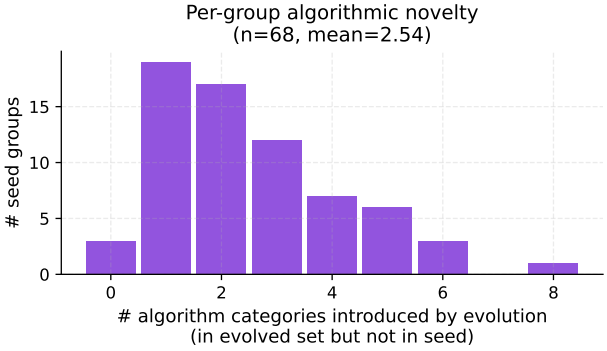
(c) Difficulty.



(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.

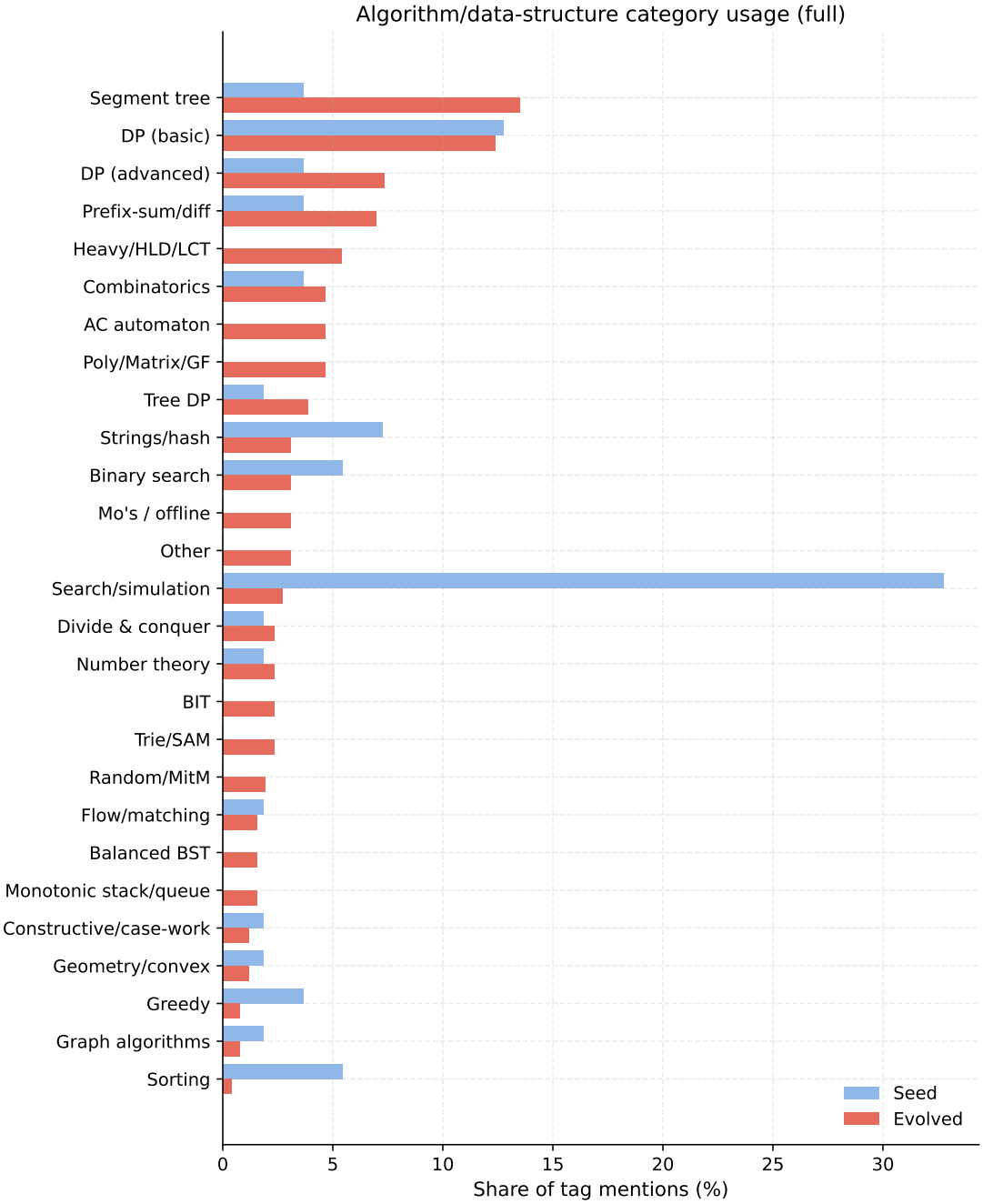


Figure 8. Full algorithm-category distribution for seed and evolved problems, including all categories with at least three combined mentions. The main-body Figure 2 shows the eleven categories with the largest absolute seed-to-evolved share shift.

G. 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.

G.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 $u_i \leq x \oplus p_i \leq v_i$. 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

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

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

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

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

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

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.

G.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.

H. Examples of Evolution Trajectory

In this appendix section, we append example trajectories of Gemini-Flash-3 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.

I. 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}
```

J. Reproducibility and Asset Licenses

Code and configurations. We will release 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 3. 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.