AUTOCODE-RL: CLIMBING THE DIFFICULTY PYRAMID WITH VERIFIABLE SYNTHETIC PROBLEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning with verifiable rewards (RLVR) trains a coding agent by rewarding programs that pass tests, so the model learns only from problems it solves some of the time. Competitive-programming problems, however, form a pyramid: easy problems are plentiful and hard ones are rare, so the problems that can still teach a model become scarce as it improves. We propose AutoCode-RL, in which a frozen setter model adjusts the difficulty of existing contest problems for the solver being trained: plentiful lower-rated problems become harder, so that the solver sometimes fails, and problems beyond its reach become easier stepping stones toward them. An automated pipeline builds and validates tests for each rewrite, and the solver learns from whether its programs pass them. Training GPT-OSS-20B on rewrites of Codeforces problems released after the benchmark and base model raises its single-attempt solve rate (pass@1) on LiveCodeBench-Pro by a relative 14% on easy and 20% on medium problems. Our trained model also solves one problem from the hard split, where the base model fails all $1{,}024$ attempts. AutoCode-RL outperforms training on the original problems with the same budget and supervised fine-tuning on a stronger model's correct solutions. To our knowledge, the resulting scores are the highest reported for GPT-OSS-20B on LiveCodeBench-Pro. The simplified problems work as stepping stones: the trained model solves seven of their harder originals that the base model never solves, none of which appeared in training. The same method also improves Qwen3.5-9B.

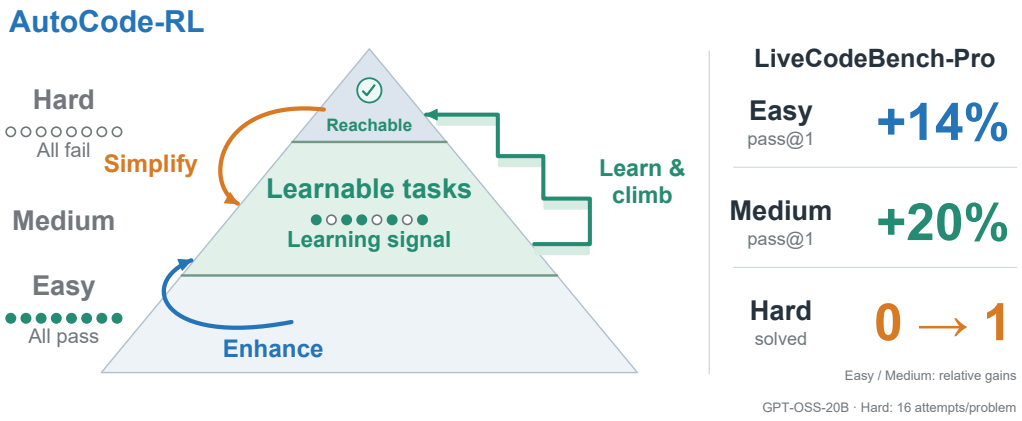


Figure 1: **Learning signals to climb higher.**

1 INTRODUCTION

LLM-based coding agents are now widely used to write and repair software (Jimenez et al., 2024; Yang et al., 2024), but they still fail on tasks that require genuine algorithmic reasoning and can exhaust the token budget before producing code (Wu et al., 2026a). Reinforcement learning with verifiable rewards (RLVR) offers a direct way to improve them: sample a human-written problem from a dataset, let the model write a program, and reward the program if it passes the problem's tests

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

(Shao et al., 2024; DeepSeek-AI, 2025). Since current models already solve most routine exercises, progress needs problems that remain hard for them, which scraping sites such as LeetCode no longer supplies. Competitive programming has long served as such a test of both reasoning and engineering (Li et al., 2022; OpenAI, 2025a): each problem demands an algorithmic insight and an exact, efficient implementation, and its tests separate correct algorithms from plausible but incorrect ones. Even frontier models have not mastered it: when LiveCodeBench-Pro (LCB-Pro) was released, the best model solved none of its hard problems without tools (Zheng et al., 2025). Writing such problems is demanding even for expert setters, so the supply of challenging verifiable tasks limits how far RL can push code reasoning (Liu et al., 2025; Zhou et al., 2025).

Synthetic-data methods use strong models to produce instructions, reference solutions or reasoning traces (Wei et al., 2024; Liu et al., 2025), and self-play methods learn to propose tasks as well as solve them (Zhao et al., 2025; Huang et al., 2025; Liu et al., 2026). Problems from competitive-programming contests such as Codeforces offer task generation a concrete starting point, but they thin out as difficulty rises: the LCB-Pro release we evaluate on labels 475 of its 864 problems easy, 306 medium and only 83 hard. Under typical RL post-training algorithms like GRPO, a problem the solver always or never solves yields no learning signal, so the problems that can still teach an improving solver are the scarcest, and those beyond its reach teach nothing. A task generator should make plentiful easy problems hard enough that the solver sometimes fails, turn out-of-reach ones into stepping stones it sometimes solves, and tie each to a verifiable reward.

We present AutoCode-RL, a decoupled setter–solver framework for competitive-programming RL (Figure 2). A frozen language model acts as the setter: it rewrites lower-rated contest problems into harder ones and higher-rated ones into easier ones, and an automated pipeline builds and checks a test suite for every rewritten problem. A smaller open-weights solver then learns through GRPO from whether its programs pass these tests. The setter constructs a fixed task pool before RL; its reference solutions are used to build and check the tests.

We build FRESH-CF, a new, publicly released corpus from Codeforces contests held between October 2025 and June 2026, after LCB-Pro was collected: from 226 original problems, the pipeline produces 147 rewrites with working test suites, 84 made harder and 63 made easier. Training starts from the released GPT-OSS-20B and uses only this corpus; we evaluate on the 706 LCB-Pro problems with official tests and on LiveCodeBench v6 Hard (Zheng et al., 2025; Jain et al., 2024).

Training only on the rewrites raises pass@1 from 68.0% to 77.2% on the 391 LCB-Pro easy problems and from 18.8% to 22.5% on the 251 medium problems, against 72.0% and 21.2% from training on the original problems with the same budget, a significantly larger gain on easy. The simplified problems act as stepping stones: after training, the solver solves more of the 63 harder originals they came from, none of them in training, with pass@16 rising from 39.7% to 50.8%. It also solves one problem from the hard split, where the base model fails all $1{,}024$ attempts. Learning from execution also beats distillation: the RL model exceeds the pass@1 of every checkpoint of full-parameter SFT on a stronger model's correct solutions. The same method also improves Qwen3.5-9B. Although training contains no mathematics, it also raises GPT-OSS-20B's accuracy on AIME 2024/2025 from 73.5% to 77.3%.

To our knowledge, these are the highest LCB-Pro scores reported for GPT-OSS-20B. The closest prior system, BenchEvolver (Wu et al., 2026b), trains the same backbone with ten times our training tasks and the rollouts its published configuration implies. Our model reaches 74.2% pass@1 on the 96 easy problems it evaluates, above the 72.9% it reports, and under our protocol its released checkpoint trails ours on all 391 easy problems and at every pass@k on the medium subset (§5).

AutoCode-RL reshapes existing problems around the learner: a strong model teaches by setting problems, and the solver discovers its own solutions. Built on AutoCode's verification pipeline, it offers a practical way to keep pushing the capabilities of coding agents once the limited supply of challenging human-written problems runs out.

2 Related Work

Verifiable rewards for reasoning. Outcome-based reinforcement learning trains reasoning models with rewards from automatically checked answers or program execution: DeepSeekMath introduces GRPO (Shao et al., 2024), and DeepSeek-R1 shows that outcome supervision can improve reasoning

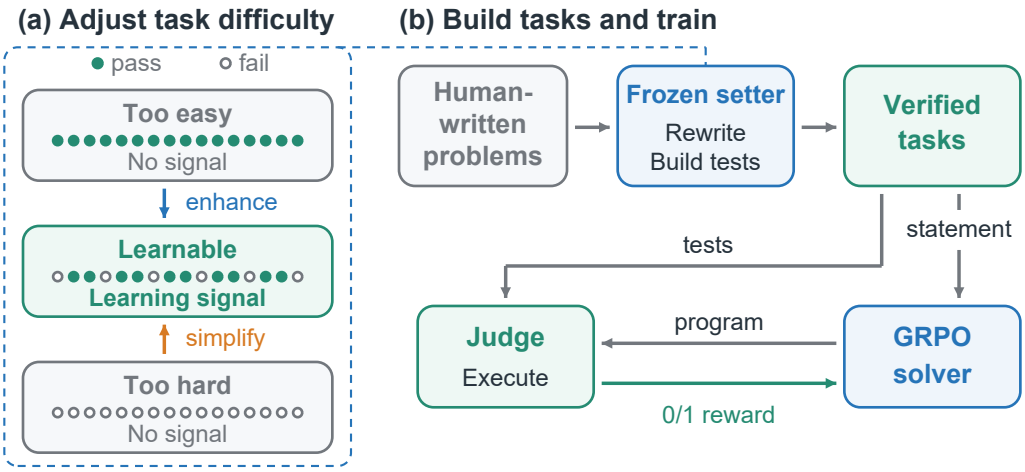


Figure 2: **Overview of AutoCode-RL.**

(DeepSeek-AI, 2025). For code, executable tests provide such supervision, and many papers apply RL with execution feedback to code generation, from early work with unit-test and compiler feedback (Le et al., 2022; Shojaee et al., 2023; Liu et al., 2023; Dou et al., 2024) to recent reasoning models for competitive programming (OpenAI, 2025a; Agentica Team, 2025; Chen et al., 2025; He et al., 2025a). AceCoder, for example, compares learned reward models with direct test-pass rewards on synthesized tests (Zeng et al., 2025). Like much of this work, we reward the solver with a binary execution verdict; our contribution is the construction of the verifiable tasks it trains on.

Synthetic coding data and verification. Synthetic coding-data methods expand training resources by generating problem statements, solutions and tests. Magicoder generates coding instructions from open-source snippets (Wei et al., 2024); rStar-Coder and X-Coder build competitive-programming corpora of synthesized problems, tests and solutions (Liu et al., 2025; Wu et al., 2026a); HardTests synthesizes tests strong enough for difficult problems (He et al., 2025b); and FrontierSmith evolves open-ended problems from competitive-programming ones (He et al., 2026). AutoCode studies LLMs as competitive-programming setters, with repeated validation of statements and tests (Zhou et al., 2025). We build on its test construction and evaluate how the resulting tasks improve a separate solver. Most of this work scales up the amount of data, often for supervised training on a strong model's solutions. AutoCode-RL instead rewrites each human problem toward the difficulty at which the solver still learns, in both directions, and trains only on execution verdicts for the solver's own programs. Closest to our setting, BenchEvolver trains the same GPT-OSS-20B backbone with RL on problems evolved from reference solutions (Wu et al., 2026b); AutoCode-RL scores higher with a tenth of its training tasks (§5).

Self-play and task generation. Self-play methods co-evolve task proposers and solvers, using solver feedback to shape the next round of tasks. Absolute Zero trains one model to propose and solve tasks validated by an executor (Zhao et al., 2025); R-Zero co-evolves a Challenger and a Solver through tasks near the solver's capability boundary (Huang et al., 2025); SPADE has one model write executable, multi-turn environments and learn to act in them (Liu et al., 2026); and Self-play SWE-RL trains a software agent to inject and repair increasingly complex bugs (Wei et al., 2025). Training the proposer online adds a second learning problem, extra solver rollouts to score proposed tasks, and coordination between the two roles, and the coupling can degrade the curriculum: proposers can collapse onto a narrow set of problems (Dineen et al., 2026), and reward signals can struggle to guide training stably without effective task filtering (Pu et al., 2026). AutoCode-RL takes a simpler route: it prompts a frozen setter once, before training, to make easy problems harder and hard ones easier, then trains the solver with a standard single-model RL loop on the fixed pool. The pool can be reused across runs, and every task is anchored to a human problem. This simple setup already trains a stronger solver than the human problems it starts from (§5).

Task set	Seeds	Seed rating (Codeforces)	Tasks
Originals	226	1200–2600	226
Enhanced	93	1200–1600	84
Simplified	64	2200–2600	63
Synthetic (enhanced + simplified)			147

Table 1: **Composition of FRESH-CF.**

3 METHOD

3.1 SETTER–SOLVER DECOMPOSITION

AutoCode-RL separates task construction from solver training (Figure 2). A frozen setter receives a human-written problem and its reference solution and writes a new problem, a new reference solution and an explanation. It is applied in two modes. Enhancement asks for a harder problem that changes at least one condition and requires a different solution approach; simplification asks for an easier practice problem that still requires a substantive algorithm. Both modes request constraints, examples and a C++ reference solution (prompts in Appendix A). The solver is a separate, smaller model trained with GRPO using the resulting problem statements and execution verdicts.

Why two operators. GRPO scores each response by how far its reward lies above or below the mean of a group of 16 responses to the same problem, so a problem the solver always or never solves contributes no gradient. Enhancement therefore targets lower-rated seeds, which the solver would otherwise solve every time, and simplification targets higher-rated seeds it would otherwise never solve, turning them into stepping stones toward the original (Figure 2a). Section 6 measures the resulting shifts in difficulty and algorithmic content.

3.2 FROM STATEMENTS TO VERIFIABLE REWARDS

A rewritten statement becomes an RL task only once it has a test suite that it has been checked against. The setter writes demanding problems, so we adopt AutoCode's test-construction pipeline, which reports 98.7% agreement with official verdicts on LLM submissions to 720 Codeforces problems (Zhou et al., 2025), and use GPT-5.5 with medium reasoning effort for both rewriting and test construction. The pipeline validates each rewrite in several rounds. It generates three candidate input validators and three output checkers together with unit tests for them, compiles all six, and keeps the best-scoring validator and checker; six test-input generators, written against `testlib`, then produce inputs, and the validator discards any that violate the stated constraints; finally, the reference solution produces the expected outputs and must itself pass the finished suite when the package is registered with the judge. A rewrite that fails any of these checks is discarded (9 of 93 enhancements and 1 of 64 simplifications), so every task the solver sees has a statement, a reference solution and tests that agree with one another. Original problems receive tests through the same procedure (§5). Appendix A gives the construction details.

3.3 A FRESH TRAINING CORPUS

FRESH-CF draws its seeds from 75 Codeforces contests held between October 3, 2025 and June 30, 2026, after the LCB-Pro collection window, which ends in September 2025. No training problem or contest appears among the 864 LCB-Pro problems, and seed selection also excludes LiveCodeBench v6. Table 1 gives the composition: enhancement is applied to the lower-rated seeds and simplification to the higher-rated ones, and nearly every rewrite yields a working test suite. The 147 rewrites form the synthetic corpus of our main runs. The corpus, tests and code are publicly released (Reproducibility statement).

3.4 SOLVER TRAINING

The solver interacts with each task in a single turn: it reads the statement, reasons, and returns one C++ program, with no tool calls, feedback or retries. The judge compiles the program and runs every

Training problems	N	All problems with tests		Apr–Jun 2025 subset	
		easy (391)	medium (251)	easy (96)	medium (56)
none (GPT-OSS-20B)	–	68.0	18.8	65.0	9.8
AutoCode-RL	147	**77.2**	**22.5**	**74.2**	**14.3**
Originals only	226	72.0	21.2	69.5	12.7
Originals + synthetic	373	74.8	21.4	73.2	14.2
BenchEvolver (released)	1,466	73.5	21.9	72.9	12.2

Table 2: **LCB-Pro pass@1 (%) at the final checkpoints.** Each problem has 4 samples on full splits and 16 on subsets. AutoCode-RL is trained on the 147 synthetic tasks alone; BenchEvolver is its released checkpoint re-evaluated in the same setting.

test of the task; the reward is 1 if all tests pass and 0 otherwise, and a response with no program, including one cut off at the response cap, receives 0.

Each run starts from the released GPT-OSS-20B checkpoint (OpenAI, 2025b) and updates all parameters with GRPO (Shao et al., 2024). We train one model on the 147 synthetic tasks and one on the 226 originals with identical settings: 80 steps, each sampling 8 problems and 16 responses per problem at temperature 1.0, for 10,240 rollouts per run; Adam with a constant learning rate of 10^{-6}, gradient clipping at 1.0, policy-ratio clipping at 0.2 below and 0.28 above the ratio of one, and no KL penalty or entropy bonus. Training runs in the slime framework (THUDM, 2025) on 16 H200 GPUs; Appendix C gives the parallelism settings and the run on all 373 problems.

4 Experimental protocol

Held-out benchmarks. Our primary benchmark is LCB-Pro (Zheng et al., 2025). We use every problem with official tests: 391 of its 475 easy problems, 251 of its 306 medium problems and 64 of its 83 hard problems, which we call Easy391, Medium251 and Hard64. We also report the 96 easy and 56 medium problems of LCB-Pro's April–June 2025 subset (Easy96 and Medium56), because BenchEvolver (Wu et al., 2026b) reports results on Easy96. We use C++, the benchmark's tests, checker and resource limits, and temperature 1.0. Easy391 and Medium251 use 4 samples per problem; Easy96, Medium56 and Hard64 use 16 (Appendix C). Pass@1 is the fraction of samples that pass, averaged over problems; pass@k estimates the chance that at least one of k samples passes, using the unbiased estimator of Chen et al. (2021).

Our secondary benchmark is LiveCodeBench v6 Hard (Jain et al., 2024), with 16 samples per problem: its 60 AtCoder problems, which read from standard input, are judged in C++ at AtCoder's standard 2-second time limit, and its 20 LeetCode problems, which ask for a single function, are judged in Python. Appendix C compares this setup with the official all-Python harness. We also evaluate the trained models, with no further training, on MATH-500 and AIME 2024/2025 (§5).

Appendices C and D provide additional evaluation details and uncertainty estimates.

5 Results

Learning from verified synthetic problems. Table 2 shows the accuracy of each model on LCB-Pro. We observe that training on only the 147 synthetic tasks improves held-out accuracy on every split: pass@1 rises by 9 points on the full easy split and by 4 on the full medium split, and by 10 points on LiveCodeBench v6 Hard (Appendix C). Importantly, the synthetic run is the strongest model in the table. It outperforms the run on the 226 human-written originals it was derived from, the runs on unions of both, and BenchEvolver's released checkpoint, which was trained on ten times as many tasks.

Accuracy across sampling budgets. A recurring concern with RLVR is that it sharpens a model without extending it: RL raises the chance that a single attempt succeeds, but given enough attempts the base model solves as many problems or more (Yue et al., 2025). If AutoCode-RL only sharpened

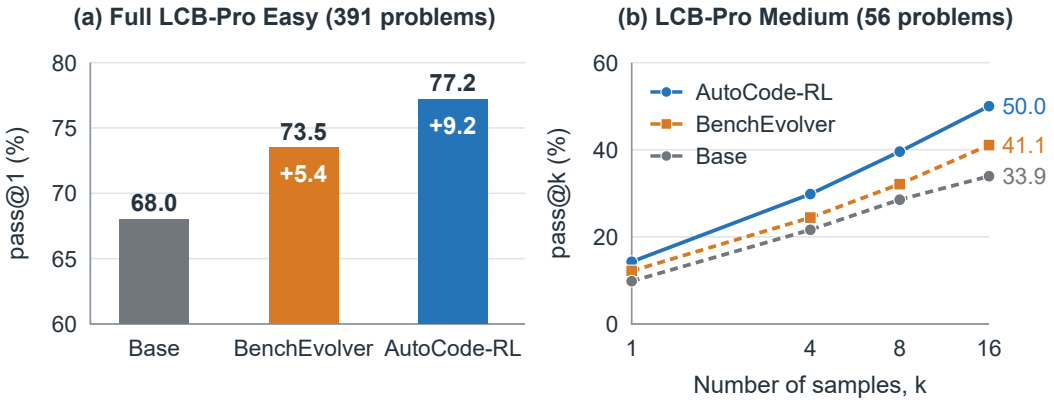


Figure 3: **LCB-Pro accuracy across sampling budgets.**

Training configuration	Tasks	Steps	Tasks/step	Samples/task	Rollouts
AutoCode-RL	147	80	8	16	10,240
BenchEvolver (released)	1,466	100	64	16	102,400†

Table 3: **Training data and rollout budgets.** †Computed from BenchEvolver's published configuration for its released checkpoint (Appendix F).

the solver, its advantage over the base should shrink as the number of attempts k grows. Figure 3 (right) shows the opposite on Medium56: the gap widens from 4.5 points at pass@1 to 16 points at pass@16, and within 16 attempts the trained solver solves ten problems the base never solves while losing one. This suggests that training on tasks placed at the edge of the solver's ability extends the set of problems it can solve, rather than only reweighting solutions it could already find.

First successes on hard. The hard split provides the most striking evidence. The base model does not pass a single one of its 1,024 samples (64 problems $\times$ 16), and neither does the model trained on the originals. The synthetic-trained model passes on two hard problems, one in each of two separate 16-sample evaluations (Codeforces 2046F1 and 2006E), each program passing all 60 official tests. Appendix E reproduces the first program and a recorded reasoning trace.

A second backbone. To check that the effect is not specific to one model family, we train Qwen3.5-9B (Qwen Team, 2026), a model with a thinking mode, on the same synthetic tasks with the same binary reward for 60 GRPO steps (Appendix J). The pattern repeats: with 16 samples per problem under its recommended decoding, Easy96 pass@1 rises from 62.6 to 69.4 and Medium56 pass@16 from 21.4 to 33.9, and the gain again grows with the sampling budget.

Data efficiency and comparison with BenchEvolver. How does this compare with prior work? BenchEvolver (Wu et al., 2026b) trains the same GPT-OSS-20B backbone with GRPO, but on ten times as many tasks and, by its published configuration, ten times as many rollouts (Table 3). With a tenth of that budget, our endpoint reaches 74.2 pass@1 on Easy96, above the 72.9 BenchEvolver reports there; to our knowledge, this is the highest LiveCodeBench-Pro score reported for GPT-OSS-20B. Evaluated in the same setting, its released checkpoint also trails ours on Easy391 and Medium251 and at every pass@k on Medium56 (Table 2, Figure 3, Appendix F). This is compelling given that BenchEvolver evolves its tasks from 880 seed problems, four times as many as the 226 we rewrite.

Rewritten versus original problems. We hypothesize that the advantage comes from placing problems where the solver can learn. Two experiments test this hypothesis. The first compares the rewrites with the human-written problems they came from. The runs on the 147 rewrites and on the 226 originals share every setting, from optimizer and batch size to the 80 steps and 10,240 rollouts, and differ only in the problems. The rewrites win on every split (Table 2), by 5 points on Easy391,

	Easy96	Medium56	
	pass@1	pass@1	pass@16
Base	64.7	7.5	32.1
SFT@Step8†	68.4	11.2	39.3
SFT@Step16	65.0	9.2	37.5
SFT@Step24	63.0	8.4	35.7
SFT@Step32	58.3	6.3	28.6
AutoCode-RL	**74.2**	**14.3**	**50.0**

Table 4: **SFT checkpoints and the RL endpoint.** † marks the selected SFT checkpoint.

Checkpoint	MATH-500	AIME 2024	AIME 2025	AIME 24+25
GPT-OSS-20B	95.2	74.2	72.9	73.5
AutoCode-RL	**96.5**	**77.3**	**77.3**	**77.3**
Originals only	96.3	76.0	74.2	75.1

Table 5: **Zero-shot mathematics pass@1 (%).**

despite being fewer. Adding the originals back does not help either: the run on all 373 problems spends 75% more rollouts (Appendix C) and still lands below the rewrites alone, and a 226-problem union that pairs the rewrites with the 79 originals lacking a rewrite, at the same budget, does no better (72.3 on Easy96 and 12.7 on Medium56, against 74.2 and 14.3). Fewer problems placed where the solver can learn thus beat more problems that it either always or never solves, which supports the hypothesis; §6 examines what the rewriting changes.

Verifiable rewards versus teacher imitation. The second experiment compares learning from verifiable rewards with supervised fine-tuning on a teacher's correct solutions and reasoning. Kimi K2.7 (Moonshot AI, 2026) solves 331 of the 373 fresh-CF problems, and we fine-tune all parameters of GPT-OSS-20B on its correct solutions, reasoning included, for 295 of them (Table 4; details in Appendix C).

Table 4 shows that learning from verifiable rewards wins by a wide margin. The best SFT checkpoint adds 3.7 points to Easy96 pass@1; AutoCode-RL adds more than twice that, and given 16 attempts it solves 50% of Medium56 against SFT's 39%. Further SFT only hurts: held-out teacher loss keeps falling while accuracy declines and Medium56 truncation rises from 7% to 66% (Appendix C), which we read as a mismatch between the teacher's reasoning style and the smaller solver's own. Learning from verifiable rewards on its own attempts yields higher held-out accuracy than imitating the teacher's correct solutions and reasoning.

Transfer to mathematical reasoning. We also evaluate whether the gains transfer from code to mathematical reasoning. Table 5 shows gains on all three sets: after code-only training, pass@1 on MATH-500 rises from 95.2 to 96.5 and on AIME 2024 and 2025 from 73.5 to 77.3, with gains in both years, and the trained solver reaches the response cap less often than the base. Training on the originals moves both in the same direction. Code RL alone thus improves mathematical problem solving, and the synthetic problems gain more on AIME.

6 Analysis of generated tasks

What does the setter change when it rewrites a problem? Every task analyzed here passed the validation pipeline of §3.2; we characterize what survived it by difficulty and algorithmic content (Figure 4), then evaluate the trained solver on the harder originals behind the simplified problems (Figure 5).

Difficulty moves in the requested direction. GPT-5.5, at low reasoning effort, assigns every problem algorithm tags from a fixed list of 19 and a Codeforces-style difficulty rating, seeing each statement on its own. Enhanced rewrites gain 285 rating points over their seeds on average and

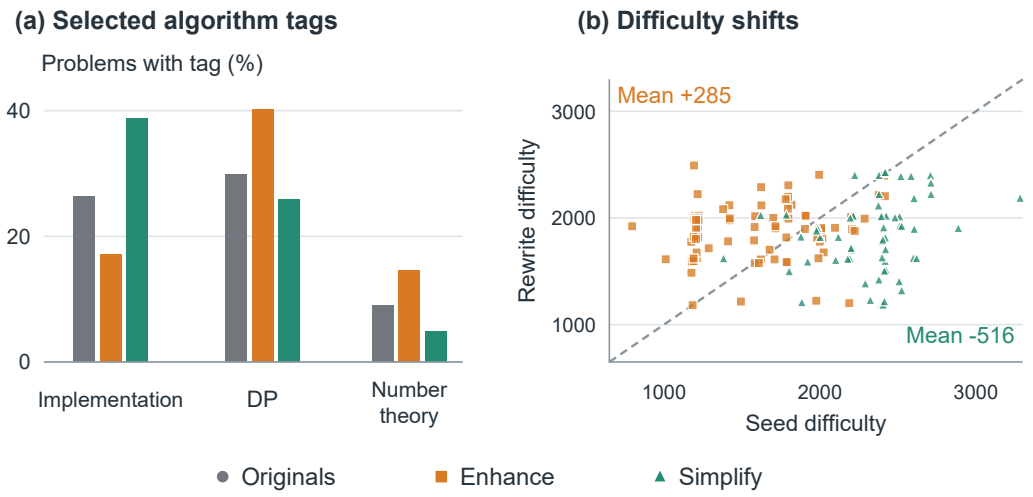


Figure 4: **Algorithmic content and difficulty of generated tasks.** (a) Algorithm-tag prevalence by pool. (b) Estimated rewrite versus seed difficulty; the diagonal marks equality.

simplified rewrites lose 516, with 89% of simplifications rated strictly below their seed (Figure 4(b)). The algorithmic mix is consistent with these difficulty shifts: compared with the originals, the enhanced pool has more dynamic-programming and number-theory tasks and fewer implementation-tagged tasks, while the simplified pool shows the reverse pattern (Figure 4(a)). Harder tasks thus demand more algorithmic reasoning and simpler ones more direct implementation, as the requested difficulty implies.

Related tasks with changed algorithmic content. Rewrites remain semantically related to their seeds while changing their algorithmic content. Their algorithm tags overlap more with their own seed's than with an unrelated seed's, yet only 3% retain exactly the same tag set. Difficulty adaptation thus accompanies changes in algorithmic requirements; Appendix B provides the tag and statement-similarity analyses.

Case studies of problem rewriting. Appendix H presents two seed–rewrite pairs, one for each operator. The enhancement keeps the seed's counting rule but asks for it over many range queries, so a direct count no longer suffices and the solution needs an efficient data structure. The simplification adds a constraint that reveals the seed's key observation, removing the search for an optimal sequence of operations while keeping a large counting problem. In the run on all 373 problems, the solver's success rate moves from 96% on the seed to 50% on the enhanced rewrite, and from 0% to 29% on the simplified one, so both rewrites demand new algorithmic work at a level where the solver sometimes succeeds (Table 17).

7 Discussion

AutoCode-RL combines algorithmic challenge with accessible learning opportunities: enhancement adds demands to lower-rated seeds, while simplification makes higher-rated seeds more approachable (Figure 4). Both supply frequent reward contrasts in the combined-corpus run. Enhanced tasks retain mixed passing and failing 16-response groups at 73.0%; simplification raises this proportion from 35.4% on its originals to 62.1%, providing more opportunities to learn from contrasting outcomes.

Rewriting may also adjust the reasoning needed to solve a task. An exploratory single-model review of 24 random seed–rewrite pairs, 12 per operator, with solver outcomes withheld, found greater demands on key observations in 9 enhanced pairs and lower demands in 10 simplified pairs. Such changes may improve the fit to the solver's capabilities, alongside human-rated difficulty, given the human–model differences reported by LiveCodeBench-Pro (Zheng et al., 2025). Figure 5 shows transfer after training on both types of rewrites: pass@16 on the 63 untrained originals behind simplified tasks

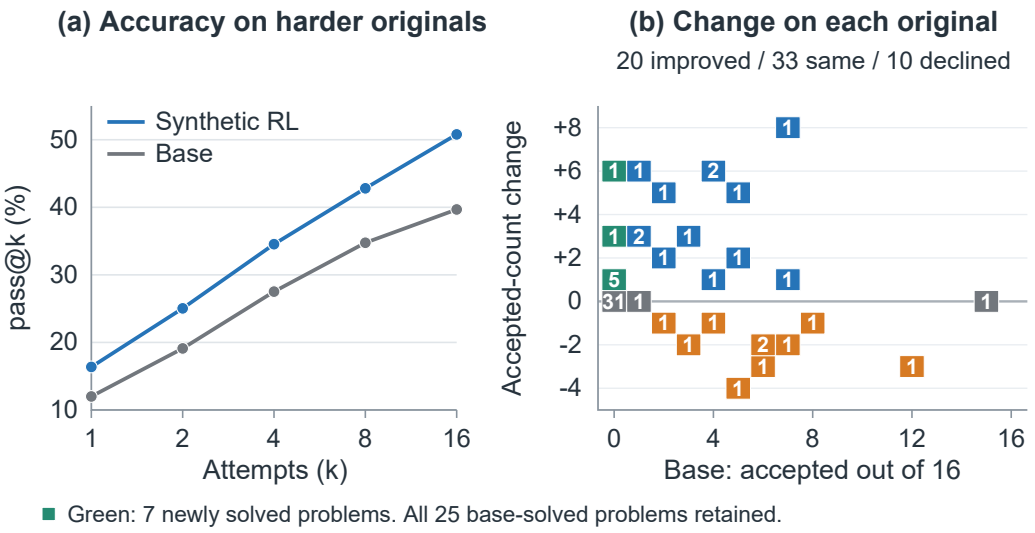


Figure 5: **Evaluation on the originals of simplified tasks.** (a) Pass@k from 16 samples per problem. (b) Change in accepted count versus the base count; cells count problems. Blue/green: improvement; orange: decline; grey: unchanged. Green marks newly solved problems.

rises from 39.7% to 50.8%, with seven newly solved problems and none lost (Appendix I). This is consistent with simplification making intermediate insights accessible while retaining useful structure, potentially offering more targeted preparation for harder originals than unrelated tasks of similar difficulty.

Part of the benefit may also come from more efficient use of the training budget. Over the same 80 updates and 10,240 rollouts, rewrite-only training generates 182.35M response tokens versus 223.14M for originals-only training, an 18.3% reduction, while achieving stronger final performance (Table 2). Its truncation rate is also lower (20.5% versus 45.1%), consistent with more attempts completing within the response budget and supplying useful feedback. Mixing originals back in reduces practice on rewrites: the 373-problem run assigns them 442 groups versus 640 in rewrite-only training, despite a larger total rollout budget (§5). Task suitability and concentrated practice may therefore contribute jointly to the gains.

8 Conclusion

Hard problems are scarce, and those beyond a learner's reach teach it nothing. A strong model can teach a weaker one by showing solutions or by setting problems. AutoCode-RL sets them: it rewrites existing contest problems, making easy ones harder and out-of-reach ones easier, so that the solver succeeds on some attempts and can learn from its own programs, and it checks every rewritten problem by execution. Trained on such problems, GPT-OSS-20B becomes a stronger solver than when trained on the originals they came from with the same rollout budget. It also solves more of the harder originals it never saw, improves on competition mathematics and outperforms fine-tuning on a stronger model's solutions. Setting problems within a learner's reach is a compact and effective way to teach code reasoning.

Limitations

Our experiments were limited by compute. A full-parameter RL run of GPT-OSS-20B occupies 16 H200 GPUs throughout, so we trained each configuration once, with a single seed, and did not sweep RL hyperparameters. The same cost confined the study to two backbones of at most 20B parameters and task pools of a few hundred problems; variance across seeds and the behavior of larger solvers and larger synthetic corpora remain to be measured.

AI USE STATEMENT

Language models generate problem variants, reference solutions, and test artifacts in the proposed pipeline, provide the supervised baseline's demonstrations, and annotate problem difficulty and algorithmic tags for the analysis. AI assistants were also used to polish the manuscript's writing.

ETHICS STATEMENT

The work uses public competitive-programming tasks and machine-generated variants. Generated programs are evaluated in resource-limited execution environments. Deterministic judging removes subjective model grading from the training loop. Any redistribution of statements or tests must respect the source datasets' licenses and contest platforms' terms. The trained models share the general benefits and risks of code-generation systems.

REPRODUCIBILITY STATEMENT

We have released the complete training corpus at `https://huggingface.co/datasets/anonymous1926/autocode-fresh-cf`: all 373 problems with their statements, reference programs, checkers and 17,583 generated test pairs, the GRPO prompts of the three training arms, and the SFT comparison data. The code is available at `https://anonymous.4open.science/r/AutoCodeRL-6A6A/`: the generation pipeline, the local judge and verifiable reward, the GRPO and SFT recipes and the evaluation tools. Section 3 describes problem construction, the corpus and its temporal separation from the benchmarks, and solver training, and Section 4 and Appendix C specify evaluation subsets, sample counts, checkpoint selection and judging. Appendix D reports paired uncertainty estimates, and Appendix E reproduces the first accepted hard-split program.

REFERENCES

Agentica Team. DeepCoder-14B-Preview: A fully open-source 14b coder at O3-mini level, 2025. URL `https://www.together.ai/blog/deepcoder`. together.ai / agentica-project blog.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yang Chen, Zhuolin Yang, Zihan Liu, Chankyu Lee, Peng Xu, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. AceReason-Nemotron: Advancing math and code reasoning through reinforcement learning. *arXiv preprint arXiv:2505.16400*, 2025.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Jacob Dineen, Aswin RRV, Zhikun Xu, and Ben Zhou. Vocabulary dropout for curriculum diversity in LLM co-evolution. In *COLM*, 2026. URL `https://arxiv.org/abs/2604.03472`. arXiv:2604.03472.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4571–4585, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.251. URL `https://aclanthology.org/2024.acl-long.251/`.

Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, et al. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*, 2025a.

Runyuan He, Qiuyang Mang, Shang Zhou, Kaiyuan Liu, Hanchen Li, Huanzhi Mao, Qizheng Zhang, Zerui Li, Bo Peng, Lufeng Cheng, et al. FrontierSmith: Synthesizing open-ended coding problems at scale. *arXiv preprint arXiv:2605.14445*, 2026.

Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. HardTests: Synthesizing high-quality test cases for LLM coding. *arXiv preprint arXiv:2505.24098*, 2025b. URL https://arxiv.org/abs/2505.24098.

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. *arXiv preprint arXiv:2508.05004*, 2025.

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.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Hynek Kydlíček. Math-Verify: Math verification library. Software. URL https://github.com/huggingface/Math-Verify. Accessed September 26, 2026.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*, 2022.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *ICLR*, 2024.

Bo Liu, Simon Yu, Yiding Jiang, Ao Qu, Andrew Zhao, Zichen Liu, Junsu Kim, Zijian Zhou, Seungone Kim, Tongzheng Ren, et al. Spade: Self-play in adaptive synthetic executable environments. *arXiv preprint arXiv:2608.19197*, 2026.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*, 2023.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. rStar-Coder: Scaling competitive code reasoning with a large-scale verified dataset. *arXiv preprint arXiv:2505.21297*, 2025.

Mike Mirzayanov. Testlib: A library for developing competitive programming problems, 2014. https://github.com/MikeMirzayanov/testlib.

Moonshot AI. Kimi K2.7. Model release, 2026. URL https://www.kimi.com/resources/kimi-k2-7-code.

OpenAI. Competitive programming with large reasoning models. *arXiv preprint arXiv:2502.06807*, 2025a.

OpenAI. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025b.

Sophia Xiao Pu, Zhaotian Weng, Chengzhi Liu, Jayanth Srinivasa, Gaowen Liu, William Yang Wang, and Xin Eric Wang. Survive or collapse: The asymmetric roles of data gating and reward grounding in self-play RL. *arXiv preprint arXiv:2605.22217*, 2026. URL https://arxiv.org/abs/2605.22217.

Qwen Team. Qwen3.5-9B. https://huggingface.co/Qwen/Qwen3.5-9B, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*, 2023.

THUDM. slime: An LLM post-training framework for RL scaling. `https://github.com/THUDM/slime`, 2025.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play SWE-RL. *arXiv preprint arXiv:2512.18552*, 2025.

Yuxiang Wei et al. Magicoder: Empowering code generation with OSS-Instruct. In *ICML*, 2024.

Jie Wu et al. X-Coder: Advancing competitive programming with fully synthetic tasks, solutions, and tests. *arXiv preprint arXiv:2601.06953*, 2026a.

Yangzhen Wu, Aaron J. Li, Wenjie Ma, Li Cao, Ziheng Zhou, Mert Cemri, Shu Liu, Yuran Xiu, Chenxiao Yan, Haikun Zhao, Bin Yu, Ion Stoica, and Dawn Song. BenchEvolver: Frontier task synthesis via solution-centric evolution. *arXiv preprint arXiv:2606.01286*, 2026b. URL `https://arxiv.org/abs/2606.01286`.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? *arXiv preprint arXiv:2504.13837*, 2025.

Huaye Zeng et al. AceCoder: Acing coder RL via automated test-case synthesis. In *ACL*, 2025.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 Embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025. URL `https://arxiv.org/abs/2506.05176`.

Andrew Zhao et al. Absolute Zero: Reinforced self-play reasoning with zero data. In *NeurIPS*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL `https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html`.

Zihan Zheng, Zerui Cheng, Zeyu Shen, Shang Zhou, Kaiyuan Liu, Hansen He, Dongruixuan Li, Stanley Wei, Hangyi Hao, Jianzhu Yao, Peiyao Sheng, Zixuan Wang, Wenhao Chai, Aleksandra Korolova, Peter Henderson, Sanjeev Arora, Pramod Viswanath, Jingbo Shang, and Saining Xie. LiveCodeBench Pro: How do olympiad medalists judge LLMs in competitive programming? *arXiv preprint arXiv:2506.11928*, 2025.

Shang Zhou, Zihan Zheng, Kaiyuan Liu, Zeyu Shen, Zerui Cheng, Zexing Chen, Hansen He, Jianzhu Yao, Huanzhi Mao, Qiuyang Mang, Tianfu Fu, Beichen Li, Dongruixuan Li, Wenhao Chai, Zhuang Liu, Aleksandra Korolova, Peter Henderson, Natasha Jaques, Pramod Viswanath, Saining Xie, and Jingbo Shang. AutoCode: LLMs as problem setters for competitive programming. *arXiv preprint arXiv:2510.12803*, 2025. URL `https://arxiv.org/abs/2510.12803`.

A Problem construction prompts

The setter receives a human problem statement and a reference C++ solution. Both rewrite modes request a new statement, a new reference solution, and an explanation. The solver's RL input contains the resulting problem statement; the reference and explanation are used for construction and checking.

Enhance instruction.

> You are an experienced competitive programming problem setter. Please modify at least one condition of the problem to design a new, more difficult problem. This new problem must require a different solution approach from the original and should not be solvable using heuristic methods. The problem must be of such quality and novelty that it could be accepted on a platform like Codeforces, or at the very least serve as a valuable training exercise. In the problem statement, please include time and space constraints and provide sample test cases. Furthermore, please write an accepted C++ solution, along with a complexity analysis, a proof of correctness, and a clear explanation.

Simplify instruction.

> You are an experienced competitive programming problem setter. Please modify at least one condition of the problem to design a new, but easier problem for practice purposes. The original problem is considered as too difficult, you should simplify it while remains moderate in difficulty ($\sim$2000 codeforces rating). You are also allowed to give moderate tips in problem statement. This new problem must require a different solution approach from the original and should not be solvable using heuristic methods. The problem must be of such quality and novelty that it could be accepted on a platform like Codeforces, or at the very least serve as a valuable training exercise. In the problem statement, please include time and space constraints and provide sample test cases. Furthermore, please write an accepted C++ solution, along with a complexity analysis, a proof of correctness, and a clear explanation.

Section 6 measures the resulting difficulty shifts.

Executable verification artifacts. Subsequent calls produce candidate input validators, output checkers, test-input generators, and unit tests for the validators and checkers. We compile three validator and three checker candidates and select the best-scoring of each on the generated unit tests. Six input generators use the `testlib` library (Mirzayanov, 2014). Generated inputs are filtered by the selected validator; the reference program produces corresponding answers. A task is retained for RL only after its package is built and its reference is accepted by that package.

Construction stage sequence. The implementation proceeds through seven stages: freeze the seed list; propose an enhanced or simplified task; parse it into a canonical record with a stable identifier; generate checker, validator and input-generator artifacts; compile and test the artifacts in a container; package complete input–answer pairs with the selected checker and execution limits; and register the package with the judge for an end-to-end reference acceptance check. Selection and packaging reuse the AutoCode construction pipeline (Zhou et al., 2025). The training interface exposes each accepted task's statement and deterministic verifiable reward to the solver. All 226 originals, 84 of 93 enhanced and 63 of 64 simplified attempts pass this end-to-end check.

B Additional analysis of generated tasks

Difficulty and algorithmic content. GPT-5.5, frozen and configured with low reasoning effort, assigns problems a rating and algorithm tags. Figure 6 shows prevalence for 12 algorithm tags. Seed–rewrite comparisons use the same annotator and match a variant to the original problem through its identifier. The difficulty shifts in Figure 4 are therefore paired estimates on one scale. The requested direction holds on average, with harder variants gaining about 285 rating points and simpler variants losing about 516.

Relationship to human seeds. Algorithm-tag Jaccard similarity, $|A \cap B|/|A \cup B|$, averages 0.36 for matched seed–rewrite pairs, versus 0.20 for unrelated seeds and 0.18 for unrelated seed–rewrite

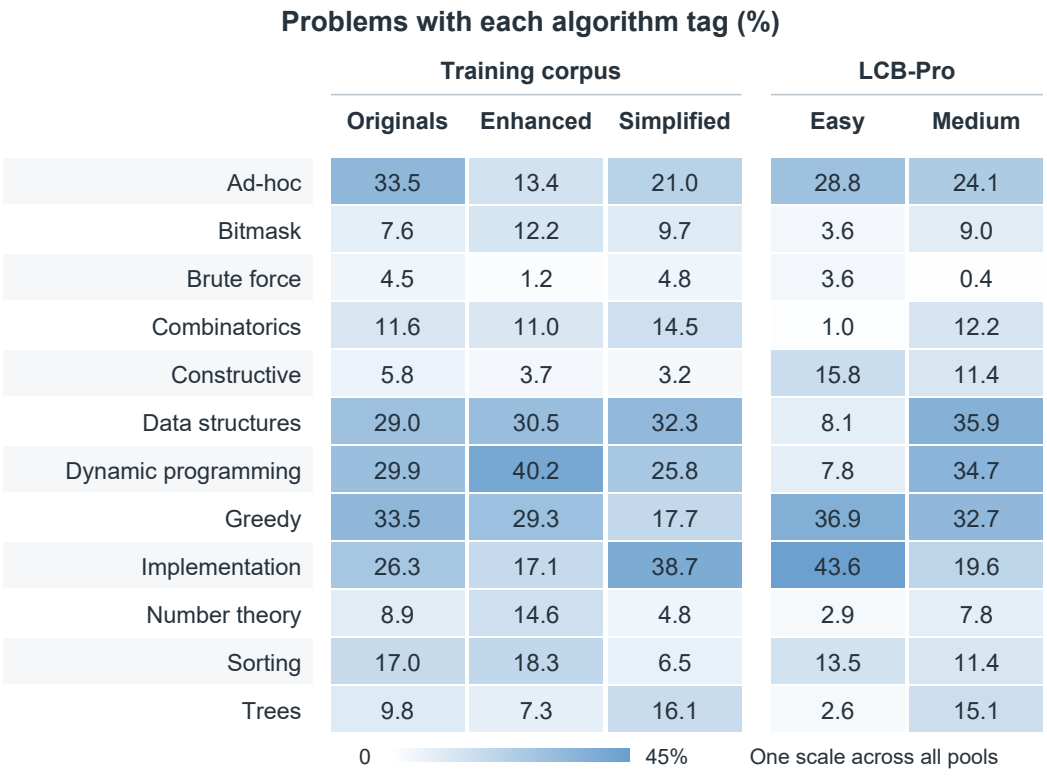


Figure 6: Algorithm-tag prevalence in training and evaluation pools. Cells show percentages on a shared color scale.

pairs. Only 3% of matched pairs have identical tag sets. Mean within-pool similarity ranges from 0.18 to 0.22 across the training pools and the held-out benchmark.

Statement similarity to the seed. We encode statements with Qwen3-Embedding-0.6B (Zhang et al., 2025) and compute cosine similarity in the normalized embedding space. The median seed–rewrite similarity is 0.86 for enhancement and 0.93 for simplification, compared with 0.44 for unrelated seeds.

C ADDITIONAL EVALUATION DETAIL

Training infrastructure. The synthetic-only and originals-only runs use 16 H200 GPUs with tensor parallelism and expert parallelism both set to 4 in slime (THUDM, 2025); the policy weights are synchronized to the rollout engines after every step, so sampling stays on-policy.

Training on the combined corpus. The run on all 373 original and synthetic problems uses 70 steps, 16 problems per step and 16 responses per problem, with a 26.5K-token response cap. This gives 17,920 rollouts, 75% more than the 10,240 used by each of the synthetic-only and originals-only runs, which use 80 steps, 8 problems per step and a 30K-token response cap.

Checkpoint selection. RL results use each run's final checkpoint, termed its endpoint. SFT uses the checkpoint selected on held-out problems by the rule given below.

Subsets and repeated evaluations. The 96-problem easy subset is identified by LCB-Pro's public `quater_2025_4_6` field; the 56-problem medium subset uses the same quarter. The 80-problem LiveCodeBench v6 Hard set is the hard portion of the v6 release file. Table 2's Easy391 and Medium251 results average four samples per problem. Every Easy96 and Medium56 result in the main text comes from a single 16-sample evaluation of each final checkpoint, run after training; the pass@k results use the same samples. During training we also evaluated Easy96 every 10 steps with four samples per problem. These in-training measurements appear only in Table 7.

Sampling budget. The LCB-Pro evaluations use a 32.5K response cap; training uses 30K for the synthetic and originals runs and 26.5K for the run on all 373 problems. We retain every sampled attempt, including responses that reach the token cap. RL evaluations use the sampler embedded in the training framework; the SFT study uses standalone SGLang (Zheng et al., 2024) with the same prompts, judge, temperature and response cap.

Pass@k on Medium56. Table 6 gives the 16-sample evaluation behind Figure 3. The synthetic endpoint solves 28 of 56 problems at least once in 16 attempts, ten of them problems the base model never solves.

k	base	synthetic	synthetic − base
1	9.8	**14.3**	+4.5
4	21.6	**29.8**	+8.2
8	28.5	**39.6**	+11.0
16	33.9	**50.0**	+16.1

Table 6: Medium56 pass@k (%) for the base and synthetic endpoint, estimated from 16 samples per problem.

Training trajectories. Easy96 accuracy fluctuates during training in all three runs, rising and falling with the average response length; Table 7 records it every 10 steps.

step	0	10	20	30	40	50	60	70	80
Originals + synthetic (373)	63.5	65.5	69.3	69.3	73.7	72.4	74.0	71.4	–
Synthetic (147)	63.5	65.4	69.5	68.5	70.1	72.7	62.2	73.2	74.7
Originals (226)	63.5	71.4	61.5	62.0	65.4	65.1	69.5	69.3	68.8

Table 7: Easy96 pass@1 (%) during training, with 4 samples per problem. Step 10 covers 74% of Easy96 for the combined run and 75% for the synthetic run. "–" denotes no checkpoint.

Teacher trajectories and full-parameter SFT. Kimi K2.7 provides accepted solutions and reasoning for 331 of the 373 problems. Kimi rewrites 157 long traces as shorter structured analyses; the prompt requests fewer than 1,500 words while retaining the final program. The remaining 174 traces are unchanged. A family-disjoint split assigns 295 problems from 193 families to training and 36

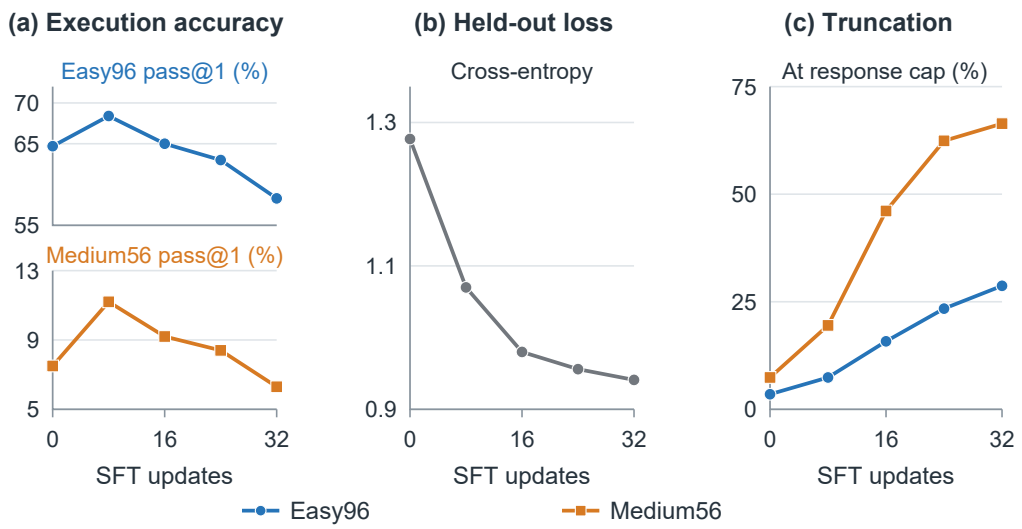


Figure 7: **SFT training dynamics.** Left: Easy96 and Medium56 pass@1 (16 samples per problem). Center: held-out teacher loss. Right: responses reaching the 32,500-token cap. Update 0 is the base model.

problems from 20 families to development; a family contains an original problem and its generated variants. The training split contains 169 originals, 71 enhanced and 55 simplified problems. Held-out pass@1 (36 problems, 4 samples each) selects among updates 8, 16, 24 and 32, with ties broken toward the earlier update. All reasoning and answer tokens are supervised in Harmony, GPT-OSS's chat format, with loss averaged over the batch's supervised tokens; trajectories are neither truncated nor packed.

We update all 20.9B parameters using FSDP on four GB200 GPUs, with BF16 computation and FP32 master weights and optimizer states. AdamW uses $\beta = (0.9, 0.95)$, $\epsilon = 10^{-8}$, weight decay 0.01 on matrix weights and gradient clipping at 1.0. Learning rates for the first three updates are 5×10^{-8}, 2.75×10^{-7} and 5×10^{-7}, followed by cosine decay to 5×10^{-8} at update 32. Batch size 32 yields 1,024 trajectory presentations and 7,255,776 supervised tokens. The model and data-shuffle seeds are 1234 and 42. The base and four checkpoints use SGLang 0.5.19 in the same GB200 environment, with 16 samples per benchmark problem; their programs are judged against the benchmark's official packages. SFT gains use this separately evaluated base.

Training dynamics and accounting. Figure 7 summarizes SFT's execution accuracy, held-out teacher loss and response truncation. The development-selected update 8 raises pass@1 from 64.7 to 68.4 on Easy96 and from 7.5 to 11.2 on Medium56. Further updates reduce execution accuracy even as teacher loss falls from 1.070 to 0.941; Medium56 truncation rises from 19.5% to 66.4%. At the selected checkpoint, Medium56 pass@16 is 39.3, compared with 50.0 for the RL endpoint (Table 9). SFT uses 7.26M supervised tokens; RL generates 182.35M response tokens over 80 updates.

Complete SFT measurements. Table 8 gives development accuracy and trajectory loss; Table 9 reports the full pass@k grid, and Table 10 reports response lengths and truncation.

LiveCodeBench v6 Hard and the evaluation language. Hard AtCoder problems are calibrated to compiled-language time limits, so we evaluate the 60 AtCoder problems in C++ at the AtCoder-standard 2-second limit with our training prompt and judge, and the 20 LeetCode function-completion problems in Python. On the full 80-problem set the synthetic endpoint raises pass@1 from 42.6 to 52.7; on the AtCoder portion from 40.7 to 47.9 (Table 11). On the 60 AtCoder problems, base-model accuracy is 13.3 points higher under this setup than under the official all-Python harness. Over 20% of Python generations time out, compared with about 2% in C++ (2.2% base, 1.7% trained). Python-only evaluation therefore understates accuracy on these tasks.

SFT update	Dev accepted /144	Dev pass@1	Dev loss
0	80	55.6	1.277
8†	81	56.3	1.070
16	73	50.7	0.980
24	70	48.6	0.956
32	58	40.3	0.941

Table 8: Development execution accuracy (4 samples per problem) and token-averaged teacher loss. † marks the checkpoint selected by execution accuracy.

Checkpoint	@1	@2	@4	@8	@16
Easy96					
Base	64.7	76.5	84.4	90.2	94.8
SFT update 8†	68.4	78.7	86.0	90.5	92.7
SFT update 16	65.0	76.0	84.0	89.3	91.7
SFT update 24	63.0	72.9	80.2	86.0	90.6
SFT update 32	58.3	70.0	78.2	85.1	90.6
AutoCode-RL (RL)	74.2	83.5	89.4	92.2	92.7
Medium56					
Base	7.5	11.8	17.9	25.2	32.1
SFT update 8†	11.2	17.0	23.7	30.7	39.3
SFT update 16	9.2	14.2	20.7	28.3	37.5
SFT update 24	8.4	12.7	18.3	25.8	35.7
SFT update 32	6.3	10.0	15.3	21.8	28.6
AutoCode-RL (RL)	14.3	21.4	29.8	39.6	50.0

Table 9: Easy96 and Medium56 pass@k (%) from 16 samples per problem. † marks the development-selected SFT checkpoint.

	Easy96		Medium56	
SFT update	Mean tokens	Cap (%)	Mean tokens	Cap (%)
0	8,941	3.5	14,608	7.4
8	10,682	7.4	19,082	19.5
16	13,631	15.8	24,503	46.1
24	16,020	23.4	26,706	62.4
32	17,739	28.7	27,335	66.4

Table 10: SFT evaluation response lengths (16 samples per problem). Cap (%) is the share reaching 32,500 tokens.

LCB v6 Hard, language-matched	AtCoder (60)	all (80)
base GPT-OSS-20B	40.7	42.6
AutoCode-RL (synthetic)	**47.9**	**52.7**
trained − base	+7.2	+10.1

Table 11: LiveCodeBench v6 Hard pass@1 (%) from 16 samples per problem: AtCoder in C++ and LeetCode in Python.

D UNCERTAINTY

The intervals below are 95% percentile intervals from 4000 paired bootstrap resamples over problems. Pairing retains the common problem set across checkpoints; reported contrasts are computed before rounding. Intervals are pointwise, without a multiple-comparison correction, and condition on one fixed checkpoint per training run (Section 4).

Main performance (§5, Table 2). On Easy96, the synthetic checkpoint gains $+9.1$ [+6.1, +12.3] over the base and the originals-only model $+4.5$ [+1.9, +7.3]. Full-easy improvement is $+9.2$ [+6.9, +11.6]; full-medium improvement is $+3.7$ [+1.2, +6.3].

Rewritten versus original problems (§5). Synthetic minus originals is $+4.6$ [+2.1, +7.4] on Easy96 and $+5.2$ [+3.1, +7.4] on full easy; original-only full-easy improvement over the base is $+4.0$ [+1.7, +6.4]. The 373-problem union minus the base is $+8.2$ [+4.7, +12.1] on Easy96 and $+4.4$ [+1.1, +7.9] on Medium56. The 226-problem union minus the base is $+7.2$ [+4.3, +10.4] on Easy96 and $+2.9$ [+0.3, +6.1] on Medium56; its difference from the synthetic run is -6.0 [-12.0, -0.3] at Medium56 pass@4.

Pass@k (§5, Table 6). Synthetic minus base in the 16-sample evaluation:

k	**medium** (56)
1	$+4.5$ [+1.6, +7.6]
4	$+8.2$ [+3.0, +13.5]
8	$+11.0$ [+3.6, +18.8]
16	$+16.1$ [+5.4, +26.8]

Released checkpoint comparison (§5, Tables 14 and 15). On the full Easy391, synthetic minus BenchEvolver is $+3.8$ [+1.6, +5.9] at pass@1 and $+3.8$ [+1.3, +6.6] at pass@4; BenchEvolver minus the base is $+5.4$ [+3.1, +7.9] at pass@1. On Medium251, BenchEvolver minus the base is $+3.1$ [+0.7, +5.5]. On Medium56, synthetic minus base is $+4.5$ [+1.6, +7.6] at pass@1 and $+16.1$ [+5.4, +26.8] at pass@16. Paired directly, synthetic minus BenchEvolver is $+2.1$ [+0.1, +4.2] at pass@1 and $+5.4$ [+0.1, +11.0] at pass@4. In Python, its training language, BenchEvolver gains $+4.5$ [+1.8, +7.6] at pass@1 over its Python base.

SFT (§5). The following differences compare the SFT checkpoint chosen on held-out problems with the base, as evaluated in the SFT study, and with the RL endpoint. All three use 16 samples per problem; differences are percentage points.

Contrast	**Subset**	**pass@1**
SFT minus base	Easy96	$+3.7$ $[+1.4, +6.1]$
SFT minus base	Medium56	$+3.7$ $[+1.3, +6.5]$
SFT minus RL	Easy96	-5.7 $[-8.3, -3.2]$
SFT minus RL	Medium56	-3.1 $[-5.7, -0.9]$

Mathematical transfer (§5, Table 5). Pass@1, synthetic minus base: $+1.3$ [+0.4, +2.4] on MATH-500 and $+3.8$ [+0.5, +7.4] on the 60 AIME problems. Originals minus base: $+1.1$ [+0.4, +2.0] on MATH-500.

Language and execution setup (§C, Table 11). On the 60 AtCoder problems in C++, synthetic minus base is $+7.2$ [+4.5, +10.1]; on all 80 problems under the language-matched setup, $+10.1$ [+7.3, +13.3]. Moving the AtCoder problems from the Python harness to C++ changes base accuracy by $+13.3$ [+7.8, +19.5] and synthetic accuracy by $+22.4$ [+14.6, +30.7].

Transfer to harder originals (§7). On the 63 original parents of simplified tasks, synthetic minus base is $+4.37$ [+0.89, +7.94] percentage points at pass@1 and $+11.11$ [+3.17, +19.05] at pass@16. Both models are sampled 16 times on each of the 63 problems. The intervals reflect variation across these problems for the two fixed checkpoints (Appendix I).

Second backbone (§5, Table 19). Qwen3.5-9B endpoint minus its base on Easy96 is $+6.8$ [+4.6, +9.0] at pass@1, $+6.6$ [+3.9, +9.6] at pass@4, $+7.0$ [+3.7, +11.0] at pass@8 and $+8.3$ [+3.1, +14.6] at pass@16; on Medium56, $+3.1$ [+1.2, +5.5], $+6.3$ [+2.5, +11.1], $+8.3$ [+3.3, +14.1] and $+12.5$ [+5.4, +21.4].

E ACCEPTED PROGRAMS ON LCB-PRO HARD

LiveCodeBench-Pro lists 83 hard problems; official test packages are available for 64. We evaluate these in C++, with 16 samples per problem, temperature 1, a 32.5K evaluation response cap, and the benchmark's tests, checker and resource limits. In the initial 1,024 samples per checkpoint, the base produces no accepted program and the synthetic checkpoint produces one accepted program on 2046F1. Their pass@16 values are therefore 0.0 and 1.6. The accepted sample was independently re-graded in a separate process and passes all 60 tests; a rejected sample of the same problem re-grades to Wrong Answer, $0/60$.

A second, independent 16-sample pass of the synthetic checkpoint over the same 64 problems again produces exactly one accepted program, on a different problem, 2006E (*Iris's Full Binary Tree*; $60/60$ tests), and the originals-only checkpoint produces none in its own 1,024 samples (542 of which reach the response cap, against 181 and 154 for the two synthetic passes and 89 for the base). Pooled over the two synthetic passes (32 samples per problem), 2 of the 64 problems are solved.

Further sampling on 2046F1. The base produces no accepted program in 128 additional samples on 2046F1 (104 Wrong Answer, 21 No Code, 3 Compile Error). A further run of the synthetic checkpoint on the same problem, with its reasoning recorded, yields another accepted program ($60/60$ tests, independently re-graded $60/60$). In outline, its 106K-character trace first casts the construction as a reverse process (deleting one Y, D and X at a time while keeping adjacent letters distinct), tests the obvious "consecutive blocks of three" construction against a hand-made counterexample (`YDYXDX`) and rejects it, derives the two necessary conditions (equal letter counts, no equal adjacent letters), proposes peeling letters greedily from the two ends of the string with an interior deletion as fallback, and spends most of its length on whether the recorded deletion positions can be replayed in reverse as insertion positions. It concludes that they cannot, then re-simulates the small example and finds that they can, and only then writes the program. The resulting program is simpler than the first accepted one (no backtracking over insertion orders, no Fenwick tree) and passes all 60 official tests.

The second problem. *Iris's Full Binary Tree* (2006E). A rooted tree is built by adding vertices $2, \ldots, n$ one at a time, each attached to an earlier vertex; after each addition, output the smallest d such that the current tree can be extended to a full binary tree of depth d (or -1 if none exists), with n up to $5 \cdot 10^5$ per test and the sum of n bounded. The accepted program maintains the tree's eccentricities under insertions with a priority queue keyed on eccentricity.

The first accepted program. *Yandex Cuneiform* (easy version). A string over $\{\mathrm{Y}, \mathrm{D}, \mathrm{X}\}$ is a cuneiform if it can be built from the empty string by repeatedly inserting one copy of each letter such that no two adjacent letters are ever equal. Given a string of length $n < 2 \cdot 10^5$, $n \bmod 3 = 0$, decide whether it is a cuneiform and, if so, output a valid sequence of insertions. The program below works backwards: it peels one Y, one D and one X per layer from the final string, keeping per-letter sets of currently removable positions (a letter is removable when its two neighbors differ), backtracking over the six removal orders with an undo log, and using a Fenwick tree to recover the insertion index of each removed letter in the shrinking string. The layers are then reversed into an insertion sequence. It is reproduced verbatim, comments included.

```
#include <bits/stdc++.h>
using namespace std;

struct Fenwick {
    int n;
    vector<int> bit;
    Fenwick(int n=0){init(n);}
    void init(int _n){ n=_n; bit.assign(n+1,0);}
    void add(int idx,int val){ // idx: 0-based, adds to position idx
        for(int i=idx+1;i<=n;i+=i&-i) bit[i]+=val;
    }
    int sum(int idx){ // sum of first idx elements (0..idx-1)
```

```
        int res=0;
        for(int i=idx;i>0;i-=i&-i) res+=bit[i];
        return res;
    }
};

int charType(char c){
    if(c=='Y') return 0;
    if(c=='D') return 1;
    return 2;
}
char typeChar(int t){ return t==0?'Y':(t==1?'D':'X'); }

struct UndoInfo{
    int idx, prev, next;
};

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int T;
    if(!(cin>>T)) return 0;
    while(T--){
        string s;
        cin>>s;
        int n=s.size();
        if(n%3!=0){ cout<<"NO\n"; continue;}
        int cnt[3]={0,0,0};
        bool ok=true;
        for(char c:s){
            int t=charType(c);
            cnt[t]++;
        }
        int k=n/3;
        if(cnt[0]!=k||cnt[1]!=k||cnt[2]!=k){ cout<<"NO\n"; continue;}
        for(int i=1;i<n;i++){
            if(s[i]==s[i-1]){ ok=false; break;}
        }
        if(!ok){ cout<<"NO\n"; continue;}
        vector<int> type(n);
        for(int i=0;i<n;i++) type[i]=charType(s[i]);

        vector<int> prev(n,-1), next(n,-1);
        for(int i=0;i<n;i++){
            if(i>0) prev[i]=i-1;
            if(i+1<n) next[i]=i+1;
        }
        vector<char> alive_char(n);
        vector<bool> alive(n,true);
        Fenwick bit(n);
        for(int i=0;i<n;i++){
            bit.add(i,1);
        }
        vector< set<int> > rem(3);
        auto updateRemovable=[&](int i){
            if(i==-1) return;
            int t=type[i];
            bool condition = (prev[i]==-1 || next[i]==-1 || s[prev[i]]!=s[next[i]]);
            if(condition) rem[t].insert(i);
            else rem[t].erase(i);
        };
        for(int i=0;i<n;i++) updateRemovable(i);

        vector< vector<pair<int,int>> > opsLayers; // each layer insertion pairs (type,pos)
        bool feasible=true;
        for(int layer=0; layer<k; layer++){
            bool found=false;
            // 6 permutations
            array<array<int,3>,6> perms = {{
                {{0,1,2}},{{0,2,1}},{{1,0,2}},{{1,2,0}},{{2,0,1}},{{2,1,0}}
            }};
            for(const auto &perm:perms){
                vector<UndoInfo> undoLog;
                vector<pair<int,int>> delPairs;
                bool fail=false;
                for(int t:perm){
                    if(rem[t].empty()){ fail=true; break;}
                    int idx=*rem[t].begin();
                    // get position
                    int pos=bit.sum(idx);
                    // record deletion
```

```
                    undoLog.push_back({idx,prev[idx],next[idx]});
                    delPairs.push_back({t,pos});
                    // remove
                    rem[t].erase(idx);
                    if(prev[idx]!=-1) next[prev[idx]] = next[idx];
                    if(next[idx]!=-1) prev[next[idx]] = prev[idx];
                    alive[idx]=false;
                    bit.add(idx, -1);
                    updateRemovable(prev[idx]);
                    updateRemovable(next[idx]);
                }
                if(fail){
                    // rollback
                    for(int ii=undoLog.size()-1; ii>=0; --ii){
                        UndoInfo u=undoLog[ii];
                        int idx=u.idx;
                        int p=u.prev; int q=u.next;
                        alive[idx]=true;
                        if(p!=-1) next[p]=idx;
                        if(q!=-1) prev[q]=idx;
                        prev[idx]=p;
                        next[idx]=q;
                        bit.add(idx,1);
                        updateRemovable(p);
                        updateRemovable(q);
                        updateRemovable(idx);
                    }
                }else{
                    // success
                    // reverse delPairs to get insertion order
                    vector<pair<int,int>> insPairs;
                    for(int i=delPairs.size()-1;i>=0;--i) insPairs.push_back(delPairs[i]);
                    opsLayers.push_back(insPairs);
                    found=true;
                    break;
                }
            }
            if(!found){ feasible=false; break;}
        }
        if(!feasible){ cout<<"NO\n"; continue;}
        cout<<"YES\n";
        cout<<s<<"\n";
        vector<string> tokens;
        for(auto &layer: opsLayers){ // layers already from last to first? We processed
     from first to last? Actually we processed layers from first to last, because each time
     we removed 3 letters, starting from final string. So opsLayers[0] corresponds to last
     layer. Need to reverse to output from first to last.
        }
        // reverse operations layers
        reverse(opsLayers.begin(), opsLayers.end());
        for(auto &layer: opsLayers){
            for(auto &p:layer){
                tokens.push_back(string(1,typeChar(p.first)));
                tokens.push_back(to_string(p.second));
            }
        }
        for(size_t i=0;i<tokens.size();++i){
            if(i) cout<<" ";
            cout<<tokens[i];
        }
        cout<<"\n";
    }
    return 0;
}
```

F Comparison with BenchEvolver: data and training budgets

Training pools. BenchEvolver's RL study uses 880 LiveCodeBench v1–v5 seeds released before January 2025, 586 evolved problems, and their 1,466-problem union (Wu et al., 2026b). Our solver trains on 147 rewritten fresh Codeforces problems, a tenth of BenchEvolver's combined pool and a quarter of its evolved problems alone. BenchEvolver's reported 72.9% on Easy96 comes from training on the combined pool; for budget accounting we use its released checkpoint from that configuration (run 01, step 100).

Quantity	AutoCode-RL (synthetic)	BenchEvolver (combined pool)
Solver backbone	GPT-OSS-20B	GPT-OSS-20B
Task generator	Frozen GPT-5.5	GPT-OSS-20B
Parameter updates	Full-parameter GRPO	GRPO, LoRA rank 32
Distinct training tasks	147	1,466
Endpoint steps	80	100
Tasks sampled per step	8	64
Rollouts per task	16	16
Rollouts per step	128	1,024
Total task draws	640	6,400†
Total rollouts	10,240	102,400†
Learning rate	10^{-6}	10^{-5}
Training response cap	30K	24K output / 26.5K trajectory
Reward	Binary execution	Execution + format/overflow penalties

Table 12: Training configurations and per-run budgets, excluding data generation and evaluation. †Derived from the published configuration and released checkpoint step.

Rollout accounting. A rollout is one sampled solver response to a task (Table 12). Our budget is $80 \times 8 \times 16 = 10{,}240$ responses. Applying BenchEvolver's published batch size to its step-100 checkpoint gives $100 \times 64 \times 16 = 102{,}400$ scheduled rollouts, ten times ours. This comparison measures task and rollout counts.

Reported Easy96 results. Table 13 compares results on the same 96-problem Easy subset, using BenchEvolver's published numbers directly. BenchEvolver reports the highest value, over training, of the mean across two training seeds, estimated with 16 samples per problem; our result is the endpoint, also with 16 samples per problem. The reported peak is not tied to the released step-100 checkpoint used for budget accounting.

System / source	RL tasks	Base pass@1	Trained pass@1	Gain
BenchEvolver (reported)	1,466	64.6	72.9	+8.3
AutoCode-RL (this work)	147	65.0	**74.2**	**+9.1**

Table 13: Reported Easy96 pass@1 (%): BenchEvolver (Wu et al., 2026b) and this work, each under its own evaluation and checkpoint-selection protocol.

Re-evaluating BenchEvolver on Easy391 and Medium251. Every BenchEvolver number we measure in this paper comes from a single checkpoint: its released run 01, step 100, from the seeds-and-evolved configuration, at a fixed revision of its public repository. BenchEvolver reports no score on Easy391. We sampled this checkpoint four times per problem on a standalone SGLang server, with the prompt, C++ judge, temperature and 32.5K response cap used for all our models; the base and synthetic rows are the four-sample evaluations of Table 2. AutoCode-RL leads at pass@1, pass@2 and pass@4 and solves 352 problems at least once, against 337 for BenchEvolver and 336 for the base (Table 14); paired intervals are in Appendix D. On Medium251, under the same four-sample protocol, BenchEvolver reaches 21.9 pass@1, 29.0 pass@2 and 35.1 pass@4 and solves 88 of the 251 problems at least once, against 18.8 pass@1 for the base and 22.5 for AutoCode-RL (Table 2).

Checkpoint	Accepted	pass@1	pass@2	pass@4	solved ≥ 1
GPT-OSS-20B	1,064/1,564	68.0	78.5	85.9	336/391
BenchEvolver (released)	1,149/1,564	73.5	81.6	86.2	337/391
AutoCode-RL (synthetic)	**1,208/1,564**	**77.2**	**85.3**	**90.0**	**352/391**

Table 14: **Easy391 checkpoint comparison under a shared protocol.** Pass@k is in percent, with 4 samples per problem; Accepted counts responses and solved ≥ 1 counts problems.

Re-evaluating BenchEvolver on Medium56. BenchEvolver reports no LCB-Pro Medium score, so we evaluated the same released checkpoint on Medium56 (Figure 3, right): identical prompts, C++ judge, 32.5K response cap, temperature 1.0 and 16 samples per problem, with truncated responses counted as failures exactly as during training (Table 15). AutoCode-RL leads at every k and solves 28 problems to BenchEvolver's 23; paired directly, the synthetic endpoint leads by $+2.1$ at pass@1 and $+5.4$ at pass@4 (Appendix D). Scored in Python, its training language, the released checkpoint reaches 10.6 at pass@1, compared with 12.2 in C++. Our checkpoints are sampled by the evaluator embedded in the training framework and the released checkpoint by a standalone SGLang server. The base model scores 9.8 pass@1 and 33.9 pass@16 through the training-framework evaluator, against 8.7 and 35.7 through SGLang.

Checkpoint	pass@1	pass@4	pass@16	solved ≥ 1
GPT-OSS-20B	9.8	21.6	33.9	19/56
BenchEvolver (released)	12.2	24.4	41.1	23/56
AutoCode-RL (synthetic)	**14.3**	**29.8**	**50.0**	**28/56**

Table 15: **Medium56 checkpoint comparison under a shared protocol.** Pass@k is in percent, with 16 samples per problem.

Interpretation. A tenth of the tasks and rollouts suffices to exceed the reported result and, under one protocol, to lead the released checkpoint on the full Easy391 and at every pass@k on Medium56 (Tables 14 and 15).

G TRANSFER TO MATHEMATICAL REASONING

Protocol. We evaluate the base model and the synthetic and originals endpoints, which share the same training budget, on three public mathematics sets: MATH-500 (the 500-problem subset of MATH used by Lightman et al. 2024), AIME 2024 and AIME 2025 (30 problems each). No model has seen any mathematics during RL; the training corpus consists only of competitive-programming statements graded by execution. Sampling follows the code evaluations exactly: the same checkpoints, a standalone SGLang server, temperature 1.0, medium reasoning effort and a 32.5K response cap, with 4 samples per MATH-500 problem and 16 per AIME problem. A response is graded by `math-verify` (Kydlíček) on the last `\boxed{}` expression of its final channel; a `\boxed{}` that appears only in the reasoning channel does not count, and a response without one is a failure.

	pass@1			**AIME truncated (%)**	
Checkpoint	MATH-500	AIME 2024	AIME 2025	2024	2025
GPT-OSS-20B	95.2	74.2	72.9	5.8	7.1
AutoCode-RL (synthetic)	**96.5**	**77.3**	**77.3**	**4.4**	**4.2**
Originals only	96.3	76.0	74.2	9.6	11.2

Table 16: **Mathematics evaluation after code RL.** Pass@1 (%) uses 4 samples per MATH-500 problem and 16 per AIME problem; AIME truncation is the share reaching the 32.5K response cap.

Reading. Table 16 shows that both trained models improve single-attempt accuracy on every set, and the synthetic model improves it most: +1.3 on MATH-500 and +3.8 on the combined AIME problems, with +3.1 on AIME 2024 and +4.4 on AIME 2025. The synthetic model also reaches the response cap least often on AIME.

H CASE STUDIES OF PROBLEM REWRITING

Two seed–rewrite pairs illustrate how the setter changes the algorithmic work required of the solver. We give condensed statements, worked examples, and the corresponding training outcomes, one pair for each direction of difficulty adjustment.

H.1 ENHANCEMENT: FROM LETTER COUNTS TO INTERVAL QUERIES

Seed: Alphabet City (Codeforces 2181A). There are n street names, each a string of uppercase English letters. The city orders m copies of every name as individual metal letters, but one street's entire order is lost. The delivered letters may be rearranged: make k copies of every other name and at least one copy of the missing name. For each possible missing street, find the largest feasible k, or report -1. There are at most 2×10^5 streets and 5×10^5 letters in their names.

Rewrite: Alphabet Megacity. The missing order becomes a whole interval of streets $[l, r]$, and names use positive integer symbols up to 10^9. For each of up to 2×10^5 independent interval queries, make k copies of every name outside the interval and at least one copy of every name inside it. The total name length remains at most 5×10^5.

A worked example. Let the four names be (AB, A, BC, A) and $m = 4$, writing the integer symbols as letters for readability. The rewrite's example contains these queries:

Missing interval	Constraint determining the answer	Maximum k
$[2, 2]$	Available A: 8; required A: $2k + 1$	3
$[1, 2]$	Available A: 4; required A: $k + 2$	2
$[3, 3]$	The missing name needs C, but none is delivered	-1

What changes algorithmically? Both versions share the same resource constraint. For symbol c, let I_c count its occurrences in the missing names and O_c its occurrences outside them. Feasibility requires

$$kO_c + I_c \le mO_c \quad \Longleftrightarrow \quad (m - k)O_c \ge I_c. \tag{1}$$

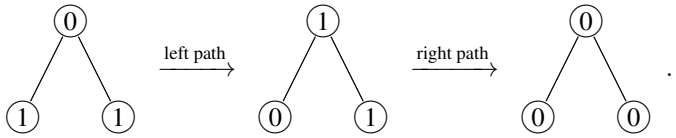
If $I_c > 0$ but $O_c = 0$, reconstruction is impossible. Otherwise, each positive O_c imposes $k \le m - \lceil I_c/O_c \rceil$. The seed can scan 26 letter counts per missing street, after computing global frequencies. The rewrite couples two changes: the missing set is an interval, and the alphabet can grow with the input. With only 26 letters, prefix counts would still handle intervals easily; with many symbols, scanning the entire alphabet per query is too expensive. The generated reference solution compresses symbols, flattens the names, and processes intervals with Mo's algorithm in Hilbert order. As interval boundaries move, it updates symbol frequencies and maintains the maximum required ratio $\lceil I_c/O_c \rceil$ using frequency buckets. The original counting invariant survives, but solving the rewrite requires an efficient way to maintain it across many queries.

H.2 Simplification: from path flips to marked leaves

Seed: Down the Pivot (Codeforces 2217G). Each node of an ordered binary tree has an arbitrary $0/1$ label. An operation flips the labels on any simple path through the root, including the root alone. A tree's cost is the minimum number of operations needed to make every label zero. Given n and k, count the distinct labeled trees with n nodes and cost k, modulo $10^9 + 7$, for $n \leq 10^6$. This combines a minimum-operation problem on each labeled tree with a count over all tree shapes and labelings.

Rewrite: Down the Pivot: Leaf Rule. Leaf labels remain free, but every internal node must equal the XOR of its children's labels. An operation flips a leaf-to-root path. The target is still to count n-node trees of cost k, with the same $n \leq 10^6$ bound. The statement also points out that leaves are the only free labels.

A worked example and the new invariant. The following valid tree is cleared by flipping the left and then the right leaf-to-root path:



Every operation changes exactly one leaf and preserves the XOR rule: at each ancestor, exactly one child changes together with the ancestor. Thus every leaf labeled 1 must be flipped, and flipping each such leaf once is sufficient. The minimum cost is exactly the number of 1-leaves. For a fixed shape with L leaves, the number of cost-k labelings is $\binom{L}{k}$. For example, with $n = 3, k = 1$, the four chain shapes contribute one labeling each and the two-leaf shape contributes two, giving 6. The reference solution evaluates the resulting count over all shapes as $\frac{1}{n}\binom{n}{k}\binom{2(n-k)}{n-1}$ modulo $10^9 + 7$, using precomputed factorials. The rewrite removes the need to infer an optimal path decomposition while retaining a large-scale combinatorial problem.

H.3 Observed training outcomes

	2181A: enhance	**2217G: simplify**
Run on all 373 *problems: every logged attempt*		
Seed	$46/48$ (95.8%)	$0/48$ (0.0%)
Rewrite	$24/48$ (50.0%)	$14/48$ (29.2%)
Run on synthetic tasks: every group of 16*, in order*		
Accepted out of 16	2, 1, 8, 3, 8	2, 7, 10, 7
All rewrite samples	$22/80$ (27.5%)	$26/64$ (40.6%)

Table 17: **Case-study training outcomes.** Fractions count accepted responses over all attempts. Top: pooled groups from the combined run. Bottom: synthetic-run groups of 16 in training order.

Table 17 summarizes the observed outcomes. In the run on all 373 problems, the solver passes almost every attempt on the 2181A seed and none on the 2217G seed, whereas both rewrites produce a mix of passes and failures. In the run on synthetic tasks, every group contains both passing and failing attempts, so each group gives GRPO a nonzero learning signal. Later groups show higher success than the first group in both cases, with substantial fluctuations. These observations illustrate accessible but nontrivial training tasks.

I Transfer from synthetic training to harder originals

We evaluate the original parents of all 63 simplified training tasks, using the explicit seed–variant mappings to fix membership before sampling. Their Codeforces ratings range from 2200 to 2600. The released GPT-OSS-20B base and the synthetic endpoint (update 80) each generate 16 C++ responses per original, giving 1,008 responses per model. Original statements are used verbatim. None of these original problems belongs to the endpoint's 147-task training pool.

Generation and judging. Both models use BF16 weights, identical tokenizers and chat templates, and SGLang 0.5.19 on the same GB200 node. Temperature and top-p are 1.0, reasoning effort is medium, and the response cap is 32,500 tokens. We retain every response, including truncations. Both models use the same FRESH-CF generated test packages, C++ checker and resource limits. Reference checks pass for all 63 packages.

Model	Accepted	@1	@2	@4	@8	@16
Base	121/1008	12.00	19.11	27.52	34.75	39.68
Synthetic RL	165/1008	16.37	25.04	34.54	42.81	50.79

Table 18: **Evaluation on the originals of simplified tasks.** Pass@k is in percent, with 16 samples per problem; Accepted counts passing responses.

From variants back to originals. Pass@k increases at every sampling budget (Table 18). Accepted-sample counts increase on 20 problems, decrease on 10, and remain unchanged on 33. Seven originals have at least one accepted RL response and no accepted base response; every original solved by the base is also solved by the RL endpoint (Figure 5). The newly covered problems are `2156F1`, `2170F`, `2183F`, `2193H`, `2205F`, `2211F`, and `2228E2`. Mean response length is 19,697 tokens for the base and 24,744 for RL; 169 and 217 responses reach the token cap.

Every one of these originals is the source of a simplified training task: the solver trains on the accessible version and is scored on the original.

J A SECOND BACKBONE: QWEN3.5-9B

Training. We train Qwen3.5-9B (Qwen Team, 2026) with its thinking mode enabled on the same 147 synthetic tasks, with the same C++ prompt, judge and binary reward as GPT-OSS-20B. Full-parameter GRPO runs in slime on eight AMD MI300X GPUs with tensor parallelism 2: each step samples 8 problems and 16 responses per problem at temperature 1.0 without top-k or top-p truncation, for 60 steps (7,680 rollouts). Adam uses a constant learning rate of 10^{-6}, gradient clipping at 1.0 and ratio clipping $[0.2, 0.28]$, with no KL penalty or entropy bonus. Thinking traces on these tasks often exceed 40K tokens, so the response cap is 49,152 tokens; a truncated response receives zero reward.

Evaluation. The base model and the step-60 endpoint are evaluated with 16 samples per problem on Easy96 and Medium56, with thinking enabled and the same 49,152-token cap, using the decoding recommended on the model card (temperature 1.0, top-p 0.95, top-k 20, presence penalty 1.5). Both models are served by the same vLLM engine and judged by the same packages as every other evaluation (Table 19); paired intervals are in Appendix D.

Subset	Checkpoint	pass@1	pass@4	pass@8	pass@16	solved ≥ 1	Truncated
Easy96	Qwen3.5-9B	62.6	75.2	78.7	81.2	78/96	35.8%
	+ AutoCode-RL	**69.4**	**81.7**	**85.7**	**89.6**	**86/96**	20.8%
Medium56	Qwen3.5-9B	6.2	14.0	18.3	21.4	12/56	88.3%
	+ AutoCode-RL	**9.4**	**20.4**	**26.6**	**33.9**	**19/56**	72.4%

Table 19: **Qwen3.5-9B before and after synthetic-task RL.** Each problem has 16 samples; truncation denotes responses reaching the 49,152-token cap.

Reading. The pattern matches GPT-OSS-20B: the gain holds at every k on both subsets and grows with k on Medium56, from $+3.1$ at pass@1 to $+12.5$ at pass@16. The endpoint also finishes within the budget far more often.