

-Coder: Advancing Competitive Programming with Synthetic Tasks, Solutions, and Tests

Jie Wu^{1,*} Haoling Li^{1,*} Xin Zhang^{2,*†‡} Jiani Guo¹ Jane Luo² Xuwei Yang¹
Steven Liu² Yangyu Huang² Ruihang Chu^{1,‡} Scarlett Li² Yujiu Yang¹

¹Tsinghua University ²Microsoft

*Equal Contribution †Project Lead ‡Corresponding Author

{wujie24, li-hl23, yang.yujiu}@mails.tsinghua.edu.cn

Abstract

Competitive programming remains challenging for code LLMs. Despite recent progress, many training pipelines still depend on scarce real-world data, raising concerns about scalability and near-duplicate benchmark contamination. In this paper, we examine whether synthetic training artifacts can support the complete SFT-to-RL cycle for competitive programming: no real-world tasks, solutions, or test cases are directly used for post-training. To this end, we synthesize tasks, verified solutions, and reliable test cases that serve as reward signals for reinforcement learning. To improve reward reliability, we introduce a dual-verification strategy that reduces noise in both selected solutions and test outputs. Using this high-quality data, we train the X-Coder model series. X-Coder-14B achieves 67.5% avg@8 on LiveCodeBench v5 and 63.4% on v6, outperforming its base model by over 40 points. Further analysis provides practical insights into synthetic post-training, highlighting the value of diverse tasks, verified long-CoT supervision, and code-centric reinforcement. Our data and models are released at github.com/JieWu02/X-Coder.

1 Introduction

As Code LLMs advance, they have demonstrated remarkable proficiency in basic programming tasks. Benchmarks such as HumanEval (Chen et al., 2021; Liu et al., 2023a) and MBPP (Austin et al., 2021) are largely considered solved. However, the frontier of code intelligence has shifted towards competitive programming. This domain is exemplified by platforms like Codeforces and benchmarks like LiveCodeBench (Jain et al., 2024), requiring

Work done during the internships of Jie Wu, Haoling Li, Jiani Guo, Jane Luo, and Steven Liu at Microsoft Research.

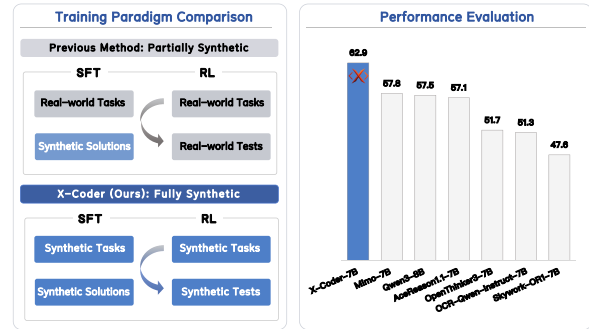


Figure 1: Left: To address persistent data scarcity in competitive programming, we build a scalable alternative by training Code LLMs with synthetic data. Right: Avg@8 results on LiveCodeBench v5. X-Coder achieves significant performance gains on competitive programming using synthetic data.

deep algorithmic reasoning and complex problem-solving capabilities.

To bridge this gap, recent success in reasoning models has been driven by the utilization of extensive reasoning data, both for supervised fine-tuning (SFT) and reinforcement learning (RL) (Guo et al., 2025; Hugging Face, 2025; Luo et al., 2025). However, this data-driven scaling faces a critical bottleneck: the scarcity of high-quality training data. The finite pool of real-world competitive programming tasks is insufficient to sustain the scaling laws of reasoning models. Furthermore, existing collections (Hendrycks et al., 2021; Li et al., 2022) are heavily exhausted, raising concerns about near-duplicate benchmark contamination and overfitting. This dilemma poses a critical question: Can models reach expert-level reasoning without reusing real-world tasks, solutions, or tests in post-training?

Answering this question is non-trivial. Generating high-quality synthetic data for competitive pro-

programming is significantly harder than for general-purpose programming, facing unique challenges across three dimensions:

(i) Tasks must be both solvable and challenging. Synthesizing competitive-level tasks often yields ill-defined or trivial instances, failing to provide the rigorous complexity needed to train expert models. (ii) High-quality SFT demands logically sound solutions. Validating these without ground-truth test cases is inherently risky, potentially polluting the training set with incorrect reasoning paths. (iii) RL necessitates reliable reward signals. Weak or insufficient synthetic tests fail to catch incorrect solutions, providing noisy reward signals that can mislead the reinforcement learning process.

Given these challenges, off-the-shelf synthesis frameworks (Wang et al., 2025a; Wei et al., 2024) fail to generalize to competitive programming and yield suboptimal results. In this work, we validate the hypothesis that synthetic training artifacts can drive expert-level code reasoning. In our setting, no real-world task, solution, or test case is directly used in SFT or RL. The pipeline nevertheless uses real data indirectly: abstract competition-related features are extracted from 10k TACO snippets without reusing task content, and teacher LLMs may have learned from real-world corpora. We investigate the key factors governing synthetic code reasoning data and establish a high-quality foundation through dual verification.

To validate this setting, we train the X-Coder model series using only synthesized artifacts under an SFT-then-RL paradigm. As shown in Figure 1, X-Coder achieves significant performance gains on the challenging LiveCodeBench v5 and v6.

Through extensive evaluation, we further distill key insights into what makes synthetic data effective, shedding light on synthetic data scaling, task diversity, verified long-CoT supervision, and the role of code-centric reinforcement learning.

Our contributions are as follows:

(1) We formulate SFT-then-RL from synthetic training artifacts for competitive programming. We examine whether Code LLMs can be post-trained entirely from synthetic tasks, solutions, and tests while accounting for indirect dependence on seed features and teachers.

(2) We develop the X-Coder model series. X-Coder-14B achieves 67.5% on LiveCodeBench v5 and 63.4% on v6, outperforming its base model by over 40 points. X-Coder also generalizes across model families and diverse benchmarks.

(3) Through extensive experiments and ablations, we analyze synthetic data scaling, task diversity, long-CoT supervision, verification reliability, and RL initialization, offering practical guidance for future work on synthetic post-training. We release related models and data.

2 Related Work

Data Synthesis for Code Generation. Synthetic code data has been studied as a way to alleviate the scarcity of high-quality programming tasks. Existing methods improve code data through instruction tuning and evolution (Yu et al., 2024; Luo et al., 2024; Xu et al., 2024; Liu et al., 2025a), coder-reviewer interaction (Sun et al., 2025), concept composition (Wei et al., 2024), and feature-based problem synthesis (Wang et al., 2025a). Recent work moves beyond instruction-code pairs by constructing verifiable question-solution-test triplets or by using multi-round validation and cross-verification (Xu et al., 2025; Zhou et al., 2026). These studies demonstrate the value of synthetic code data and executable verification. However, existing pipelines either target general code generation or still directly use real-world tasks in post-training. In contrast, our SFT and RL artifacts contain only synthetic tasks, solutions, and tests, although their construction is indirectly seeded by real-data features and teacher models.

Test Generation and Verification. Executable tests are important for code reasoning because they provide correctness checks and training feedback for RL. Prior work has studied LLM-based test generation, feedback refinement, solution-based validation, and reward construction for coder RL (He et al., 2025b; Wang et al., 2025b; Cai et al., 2026; Fu et al., 2025a; Zeng et al., 2025). Other studies evaluate test-suite quality, fault exposure, verifier accuracy, and coverage, including proactive bug discovery through repository-level test generation, or explore grammar-guided, efficiency-oriented, generator-based, and adversarial testing (Yang et al., 2025b; Ma et al., 2026; Cao et al., 2026; Liu et al., 2026; Sung et al., 2025; Peng et al., 2025; Shi et al., 2026; Hort and Moonen, 2025). These works highlight both the value and the difficulty of reliable executable tests. Our work shares the motivation of scalable test synthesis, but uses generated tests mainly as RL reward signals in an SFT-then-RL pipeline built from synthetic training artifacts. We use dual verification to reduce noise

in synthesized tests.

Post-training for Competitive Programming. Existing approaches include supervised fine-tuning on real-world tasks or their variants (Labs, 2025; Guha et al., 2025; Liu et al., 2025a), preference-based code alignment that emphasizes functional differences between correct and incorrect programs (Wu et al., 2025), reinforcement learning with executable or GRPO-like reward optimization (Shao et al., 2024b; He et al., 2025a; Luo et al., 2025; Fu et al., 2025b), and hybrid SFT-then-RL pipelines that often mix coding and mathematical reasoning data (Liu et al., 2025b; Xiaomi et al., 2025; Su et al., 2025). Despite their strong performance, these methods still rely heavily on finite real-world competitive-programming corpora. Our work instead studies whether synthetic tasks, solutions, and executable tests can sustain the complete SFT-then-RL cycle, where synthetic tests provide reward signals for code-centric RL.

3 Synthesis of Competition-Level Data

We adapt the feature-based synthesis framework (Wang et al., 2025a) to the competitive programming domain, establishing a comprehensive methodology to construct tasks that support both SFT and RL stages. As illustrated in Figure 2, this process addresses the unique challenges of synthesis through four key steps: (i) generating novel and challenging problems (with the capacity for easy scaling in quantity); (ii) constructing diverse and comprehensive test inputs for each problem (including boundary and stress tests); (iii) producing high-quality candidate solutions; and (iv) employing a dual-verification strategy that cross-checks solutions with test cases to yield more accurate test outputs and more reliable solutions.

(i) Task Generation. While EpiCoder (Wang et al., 2025a) introduced feature-based synthesis for general coding tasks, we extend this approach with three key improvements specifically tailored for the complexity of competitive programming. First, instead of relying on broad definitions of features, we explicitly extract and evolve competition-related features from 10k code snippets in the TACO dataset (Li et al., 2023) using GPT-4o-0513 (detailed in Appendix B.1). Second, formulating competitive scenarios from a rich feature tree is non-trivial, as LLMs often oversimplify complex prompts into trivial cases, thereby reducing both diversity and difficulty. To address this, we

adopt a two-stage process that separates feature selection from scenario formulation: first, selecting mutually consistent features for meaningful composition; and second, formulating hint-free tasks that demand genuine reasoning. This two-stage approach significantly outperforms single-step generation (Appendix Table 11). We further incorporate one-shot prompting to improve task understanding and instruction-following. Third, we adapt the synthesis method to support multi-style task generation, covering Codeforces¹-style tasks (standard input/output with narrative-rich contexts), LeetCode-style² tasks (starter code with pre-defined function signatures), and AtCoder³-style tasks (concise specifications with minimal explanations), thereby enhancing task diversity. Examples of the task generation process are provided in Appendix B.2, together with difficulty estimates on generated tasks in Appendix B.3 and style comparisons in Appendix B.2.3.

(ii) Test Input Generation. Obtaining sufficient and accurate test cases is a formidable challenge. Problems from competitive programming platforms often do not provide test cases, or only provide a limited number, due to platform constraints. This results in insufficient quantity, difficulty, and coverage of test cases during RL training. To address the inherent scarcity of test cases in synthesized data, we investigate two complementary methods for generating the input component of the test case. The *Prompting-based* method instructs the LLM to interpret the problem’s input constraints and directly generate multiple test inputs, covering both standard cases and edge-case instances. The *Tool-based* method leverages CYaRon⁴, a dedicated test case generation library, enabling the LLM to construct test inputs by invoking functions documented within the library after understanding the problem. For each task, we generate a set of n test case inputs $[x_1, x_2, \dots, x_n]$. Test input generation is detailed in Appendix D, with a comparison in Appendix D.2.

(iii) Candidate Solutions Generation. For each task, we generate multiple candidate solutions using advanced open-source reasoning LLMs, obtaining m answers $[A^1, A^2, \dots, A^m]$. We verify that each candidate solution includes a complete chain-of-thought reasoning process and a syntactically

¹<https://codeforces.com/>

²<https://leetcode.com/>

³<https://atcoder.jp/>

⁴<https://github.com/luogu-dev/cyaron>

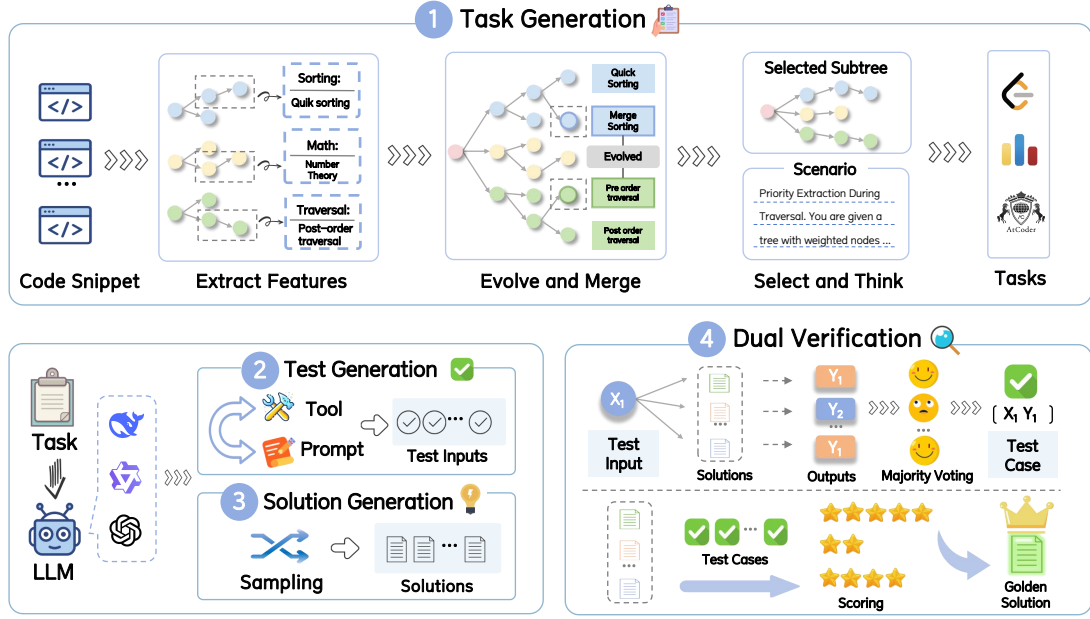


Figure 2: **Overview of the data synthesis framework.** It begins with feature extraction and evolution to construct a domain-specific feature tree. We generate multi-style competitive tasks by formulating scenarios from compatible feature sets. Finally, we synthesize solutions and test inputs, employing a dual-verification strategy to ensure solution correctness and test accuracy, yielding a high-quality data foundation for post-training.

valid Python code implementation, ensuring the absence of syntax errors through static analysis based on Abstract Syntax Tree (AST) parsing. Filtering details are in Appendix C.1.

(iv) Dual-Verification of Solutions and Test Cases. We adopt a dual-verification strategy. Step 1 of this strategy extends the principle of self-consistency (Wang et al., 2022) by applying majority voting across candidate solutions from multiple LLMs, which mitigates model-specific biases and enhances generalization, thereby yielding a reliable test output for each input. Step 2 then identifies the top-performing candidate solution by incorporating test case difficulty weighting alongside a hold-out validation set.

Step 1: Verification of Test Cases via Consensus Voting. First, we establish a preliminary ground truth for each test case input. For a given input x_i , we execute all candidate solutions to obtain a set of outputs $\{y_i^1, y_i^2, \dots, y_i^m\}$, where $y_i^j = A^j(x_i)$. A provisional ground truth output $\hat{y}_i$ is determined via majority voting:

$$\hat{y}_i = \underset{y}{\operatorname{argmax}} \sum_{j=1}^m \mathbb{I}(y_i^j = y) \quad , \quad (1)$$

where $\mathbb{I}(\cdot)$ is the indicator function. Empirical evaluation on the TACO dataset demonstrates that this voting strategy achieves 94.7% labeling ac-

curacy with 8 sampled solutions (see analysis in Appendix E.3 and Table 17). This yields a candidate test set $\mathcal{T}_{\text{candidate}} = \{(x_1, \hat{y}_1), \dots, (x_n, \hat{y}_n)\}$. Crucially, we posit that not all test cases are of equal importance; boundary or edge cases are critical for robust evaluation. We therefore introduce a weighting function $w(x_i) \rightarrow w_i$ that assigns a higher score to more challenging test cases. The weight w_i is determined by combining semantic-based heuristics (e.g., assigning higher weights to boundary and stress tests explicitly requested in the prompt) and size-based metrics (e.g., input file size as a proxy for computational complexity). Detailed weighting criteria are provided in Appendix E.2.

Step 2: Verification of Solutions via Weighted Evaluation and Hold-out Validation. For reliable golden-solution selection, we use 50% of $\mathcal{T}_{\text{candidate}}$ as the hold-out validation set $\mathcal{T}_{\text{val}}$ and the remainder as the weighted test suite $\mathcal{T}_{\text{golden}}$.

Each A^j is scored on $\mathcal{T}_{\text{golden}}$.

$$A'_{\text{golden}} = \underset{A^j \in \{A^1, \dots, A^m\}}{\operatorname{argmax}} \sum_{(x_i, \hat{y}_i) \in \mathcal{T}_{\text{golden}}} w_i \cdot \mathbb{I}(A^j(x_i) = \hat{y}_i) \quad (2)$$

The final confirmation of A'_{golden} depends on its performance on the unseen hold-out set $\mathcal{T}_{\text{val}}$. We verify that A'_{golden} also achieves the highest (or a competitively high) unweighted accuracy on $\mathcal{T}_{\text{val}}$ relative to other candidates. This additional valida-

tion step ensures that the selected solution is not merely overfitted to the specifics of the weighted test cases but demonstrates superior, generalizable correctness. The detailed algorithm is provided in Appendix E.1.

We ensure task solvability by using GPT-5 (high reasoning effort) as a proxy solver, filtering out approximately 36.9% of tasks where it achieves a zero pass rate on the voted test cases. We posit that if such a capable model fails completely, the task likely suffers from ambiguity or underspecification. In a human audit of 150 randomly sampled SFT tasks, 90.0% were well-defined, while golden-solution correctness and voted test-label accuracy were both 87.3%; Appendix E.4.2 reports the protocol, residual noise, and filtering fix. The pass-rate distribution of the remaining tasks is shown in Table 23 (Appendix E.5).

Ultimately, this process yields a verified solution A_{golden} and a comprehensive test suite $\mathcal{T}_{golden}$ for every generated task q . We leverage these high-quality synthetic assets to drive post-training for Code LLMs: (q, A_{golden}) pairs for supervised fine-tuning, and $(q, \mathcal{T}_{golden})$ for reinforcement learning via GRPO algorithm (Shao et al., 2024b).

4 Experiment

Data and Training. We adopt GPT-o3-mini (OpenAI, 2025) for task formulation, DeepSeek-R1-0528 (DeepSeek-AI, 2025) and Qwen3-235B-A22B-Thinking-2507 (Yang et al., 2025a) for solution sampling, and R1-0528 for test case generation. Statistics for SFT datasets are provided in Appendix C.2. For SFT, we set the learning rate at $5e-5$, with a global batch size of 128 for 8 epochs. For RL, reward is the fraction of passed tests (Appendix A.1). The program executes in an isolated Redis sandbox for concurrent code testing (Appendix A.4). Training configurations and costs are supplemented in Appendix A.3.

Evaluation. We evaluate X-Coder on Live-CodeBench v5 and v6 (Jain et al., 2024). Avg@8 is the mean pass rate over 8 independent completions per problem. We use temperature 0.6, top-p 0.95, top-k 20, and a maximum generation length of 32,768 tokens. We further examine whether synthetic training transfers across model families and benchmarks in Section 5.

4.1 Main Results

As shown in Table 1, during the SFT stage, X-Coder achieves avg@8 scores of 60.3/53.5 on Live-CodeBench v5/v6 with the 7B model, and 64.2/59.1 with the 14B model. After RL, the scores improve to 62.9/55.8 and 67.5/63.4.

4.2 SFT Experiments and Analysis

Scaling Laws Hold for Synthetic Data. During the SFT stage, we investigate a central question: Can the SFT dataset be effectively scaled, and along which dimension should it be scaled more favorably? To explore this, we are inspired by AceReason-Nemotron 1.1 (Liu et al., 2025b) and expand the SFT dataset from two distinct perspectives: increasing the number of unique tasks and enlarging the number of solutions per task. We design six subsets (v1–v6): v1–v4 increase the number of unique tasks (32k, 64k, 128k, and 200k unique prompts, each with 1 solution), while v5–v6 expand the number of solutions per task (16k unique prompts with 4 solutions, and 8k unique prompts with 8 solutions). The results in Figure 3 reveal a promising scaling trend: performance improves monotonically as the dataset size increases (v1 $\rightarrow$ v4), rising steadily from 43.7% to 62.7%.

Task Diversity Matters More than Solution Diversity. The performance hierarchy v2 (64k $\times$ 1) > v5 (16k $\times$ 4) > v6 (8k $\times$ 8) shows that, under a fixed compute budget, scaling the number of unique tasks is consistently more effective than increasing solutions per task. This suggests that broader algorithmic coverage across diverse problems drives stronger generalization than repeated exposure to variations of the same problem.

Synthetic Data Offers a Strong Alternative. We conduct a token-matched comparison with Open-CodeReasoning (Ahmad et al., 2025), the largest reasoning dataset built upon real-world competitive programming tasks. As shown in Table 2, our 25k $\times$ 8 setting is slightly weaker than Open-CodeReasoning in the overall score. This is mainly because it uses fewer unique prompts and allocates more tokens to multiple solutions of the same tasks, leading to weaker coverage of easy and common problem patterns.

However, this setting already performs better on the medium and hard splits, suggesting that our synthetic tasks provide effective supervision for more challenging reasoning. When we scale along the more effective dimension, namely increasing the

Table 1: Cross-paper overview on LiveCodeBench. Metrics, base models, inference settings, and data sizes follow the source papers or model cards and are not directly comparable; Table 2 provides a controlled comparison, and Appendix Table 9 reevaluates public checkpoints under avg@8. †: OpenThinker3 and rStar-Coder mix real and synthetic data. X-Coder uses only synthetic training artifacts.

Model	Base Model	SFT	RL	Size	Data	Task	Metric	LCB v5	LCB v6
SFT Approaches									
Bespoke-Stratos (Labs, 2025)	Qwen2.5-Instruct (Qwen et al., 2025)	✓	✗	7B	17k	Real	pass@1	16.2	8.57
Qwen2.5-Coder-7B-Instruct (Hui et al., 2024)	Qwen2.5-Coder-7B-Base	✓	✗	7B	-	Real	-	17.5	21.7
Qwen2.5-Coder-14B-Instruct (Hui et al., 2024)	Qwen2.5-Coder-14B-Base	✓	✗	14B	-	Real	-	24.0	23.4
OpenThinker3 (Guha et al., 2025)	Qwen2.5-Instruct	✓	✗	7B	1.2m	Mixed†	-	51.7	40.8
OlympicCoder (Hugging Face, 2025)	Qwen2.5-Coder-Instruct (Hui et al., 2024)	✓	✗	7B	100k	Real	-	40.9	19.3
OCR-Qwen-Instruct (Ahmad et al., 2025)	Qwen2.5-Instruct	✓	✗	7B	736k	Real	avg@64	51.3	44.5
rStar-Coder (Liu et al., 2025a)	Qwen2.5-Coder-Instruct	✓	✗	7B	580K	Mixed†	avg@16	57.3	-
Qwen3-8B (Yang et al., 2025a)	Qwen3-8B-Base	✓	✗	8B	-	Real	-	57.5	48.4
RL Approaches									
Skywork-OR1 (He et al., 2025a)	R1-Distilled-Qwen (DeepSeek-AI, 2025)	✗	✓	7B	124k	Real	avg@32	47.6	40.0
DeepCoder-Preview (Luo et al., 2025)	R1-Distilled-Qwen	✗	✓	14B	24k	Real	pass@1	57.9	48.5
ARReal-boba² (Fu et al., 2025b)	R1-Distilled-Qwen	✗	✓	14B	24k	Real	avg@32	58.1	56.7
SFT-then-RL Approaches (Stage 1)									
AceReason1.1-SFT (Liu et al., 2025b)	Qwen2.5-Math (Yang et al., 2024)	✓	✗	7B	2.2M	Real	avg@8	51.2	-
MiMo-SFT (Xiaomi et al., 2025)	MiMo-Base	✓	✗	7B	500k	Undisclosed	avg@8	52.3	45.5
Klear-Reasoner-SFT (Su et al., 2025)	Qwen3-Base (Yang et al., 2025a)	✓	✗	8B	1.5m	Real	avg@8	58.5	49.6
X-Coder-Qwen2.5-7B-SFT	Qwen2.5-Coder-7B-Instruct	✓	✗	7B	200k	Syn	avg@8	60.3$\pm$2.5	53.5$\pm$1.7
X-Coder-Qwen2.5-14B-SFT	Qwen2.5-Coder-14B-Instruct	✓	✗	14B	200k	Syn	avg@8	64.2$\pm$1.7	59.1$\pm$2.5
SFT-then-RL Approaches (Stage 2)									
AceReason1.1	AceReason1.1-SFT	✓	✓	7B	-	Real	avg@8	57.2	52.1
MiMo	MiMo-SFT	✓	✓	7B	130k	Undisclosed	avg@8	57.8	49.3
Klear-Reasoner	Klear-Reasoner-SFT	✓	✓	8B	106k	Real	avg@8	61.6	53.1
X-Coder-Qwen2.5-7B-RL	X-Coder-Qwen2.5-7B-SFT	✓	✓	7B	40k	Syn	avg@8	62.9$\pm$1.8	55.8$\pm$1.9
X-Coder-Qwen2.5-14B-RL	X-Coder-Qwen2.5-14B-SFT	✓	✓	14B	40k	Syn	avg@8	67.5$\pm$1.2	63.4$\pm$2.6

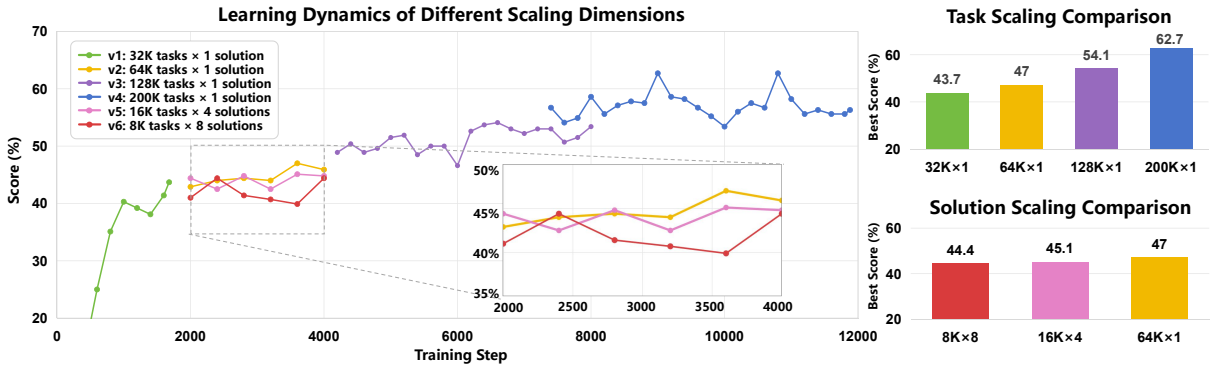


Figure 3: Left: Data scaling on LiveCodeBench v5. Right: Scaling unique tasks and scaling solutions per task.

Table 2: Head-to-head Comparison with OpenCodeReasoning on LCB v5 using Qwen2.5-Coder-7B-Instruct.

Model	Avg. Pass@1	Pass@8	Easy	Medium	Hard
OpenCodeReasoning-Qwen (28k prompts x ~26 solutions)	53.6	69.0	95.2	67.0	21.8
X-Coder-7B-SFT (25k prompts x 8 solutions)	52.5	74.9	82.0	67.4	27.7
X-Coder-7B-SFT (200k prompts x 1 solution)	60.3	77.0	96.8	73.3	37.8

number of unique prompts, X-Coder-7B-SFT with 200k $\times$ 1 data outperforms OpenCodeReasoning by 6.7 points. This result is consistent with our scaling analysis, which shows that task diversity matters more than the number of solutions per task. Synthetic artifacts reduce near-duplicate contamination risk (Appendix F), but seed features and teachers retain real-data dependence.

4.3 RL Experiments and Analysis

After RL, X-Coder-14B improves from 64.2/59.1 to 67.5/63.4 on LiveCodeBench v5/v6, and the 7B

model improves from 60.3/53.5 to 62.9/55.8.

Code reinforcement is a complementary stage after SFT. While SFT accounts for most of the performance jump, RL consistently extracts additional gains from execution rewards.

SFT Absorbs Most Gains, but RL Still Provides Non-trivial Improvements. Table 3 decomposes the contributions of SFT and RL across Qwen Instruct, Qwen Base, and Llama backbones. A consistent pattern emerges: SFT provides the dominant first-order improvement, while RL further improves the already strong SFT models. This sug-

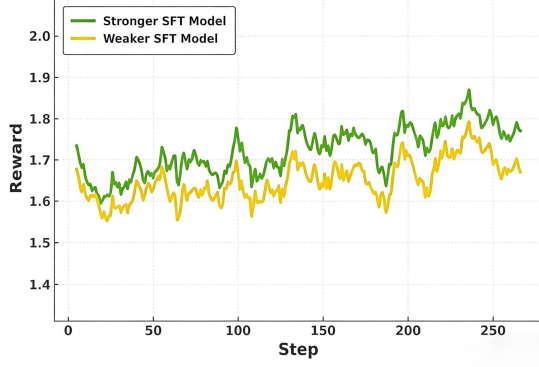


Figure 4: Reward dynamics of different SFT models.

gests that verified long-CoT supervision captures most of the performance gain, but does not exhaust the optimization potential of the model. RL should not be viewed as a minor polishing step. Instead, it serves as a complementary optimization stage that uses executable feedback to refine behaviors that are difficult to fully learn from imitation alone.

Strong SFT Initializers Enable More Effective RL. The reward dynamics in Figure 4 further support this view, showing that stronger SFT initializers lead to more effective RL optimization. Starting from a stronger SFT model places the policy in a more promising region of the solution space, making executable rewards more informative and easier to exploit. With the same RL setup, the stronger SFT initializer earns higher rewards.

5 Generality

We test whether gains from synthetic training artifacts transfer beyond one model and benchmark.

Model Generality. We apply the same pipeline to Qwen2.5-Coder-7B-Instruct, Qwen3-8B-Base, Llama-3.1-8B-Instruct, and Qwen3.5-4B⁵. Synthetic training consistently transfers across model families and generations (Table 3). Qwen3.5-4B (32k) improves from 39.7/37.0 to 65.6/58.8 after SFT and RL; at 80k, it gains 4.2/6.0 points over the original on v5/v6.

Benchmark Generality. To assess whether X-Coder generalizes beyond LiveCodeBench, we evaluate on LiveBench (White et al., 2025), Codeforces (Penedo et al., 2025), and MBPP-Plus (Liu et al., 2023b). As shown in Table 4, X-Coder achieves the highest average score across all three benchmarks, confirming broadly transferable code reasoning ability.

⁵huggingface.co/Qwen/Qwen3.5-4B.

Table 3: Avg@8 results of SFT (200k) and RL (40k) across backbones. Parentheses denote the inference-token budget. The best result in each column is bold.

Model	LCB v5		LCB v6	
	Ori.	SFT RL	Ori.	SFT RL
Qwen2.5-Coder-7B-Instruct	17.5	60.3	62.9	21.7 53.5 55.8
Qwen3-8B-Base	16.4	59.4	64.0	15.4 55.4 56.5
Llama-3.1-8B-Instruct	11.8	42.7	47.8	14.3 35.9 39.4
Qwen3.5-4B (32k)	39.7	63.0	65.6	37.0 54.6 58.8
Qwen3.5-4B (80k)	63.1	63.2	67.3	55.8 55.3 61.8

Table 4: Avg@8 results across LiveBench, Codeforces, and MBPP+.

Model	LiveBench	Codeforces	MBPP+	Avg
X-Coder-14B-RL	82.7	48.3	73.0	68.0
X-Coder-14B-SFT	75.5	45.6	73.4	64.8
X-Coder-7B-RL	70.3	38.1	71.8	60.1
X-Coder-7B-SFT	68.0	38.4	72.7	59.7
AceReason1.1-7B	68.2	32.8	75.0	58.7
Klear-Reasoner-8B-SFT	68.0	34.0	72.9	58.3
MiMo-7B-RL	66.7	35.1	70.7	57.5
Qwen3-8B	63.3	35.5	73.0	57.3
MiMo-7B-SFT	56.8	29.3	68.1	51.4
OCR-Qwen-Instruct	53.9	23.3	70.1	49.1
Skywork-OR1-7B	49.2	27.7	66.2	47.7
DeepCoder-14B-Preview	60.4	35.6	45.2	47.0
OlympicCoder-7B	38.3	18.4	64.9	40.5
Qwen2.5-Coder-7B-Instruct	39.6	6.6	62.7	36.3

6 Analysis

Dual-verification is Critical for Quality. We employ a dual-verification strategy to mitigate noise in stochastic sampling. Table 5 confirms its efficacy: training on verified solutions (64k tasks, Qwen2.5-Coder-7B-Instruct) outperforms raw solutions by 6.4 points. Verifying 200k samples requires 1.6M CoT trajectories and 24M executions, but this is a one-time data-construction cost. Raw-solution training remains a lower-cost alternative (Li et al., 2025; Gandhi et al., 2025). Detailed costs, human validation, and correlated-error analyses are provided in Appendix E.3.

Long-CoT Tasks Provide a Practical Data Selection Signal. To investigate data utilization efficiency, we evaluate three 50k-task selection strategies from a 200k-task pool: difficulty-based selection using GPT-4o ratings, rationale-based selection using DeepSeek-R1-0528 reasoning length,

Table 5: Raw vs. verified solutions on LCB v5.

Dataset	Size	LCB v5
Raw Solution	64k	47.0
Verified Solution	64k	53.4

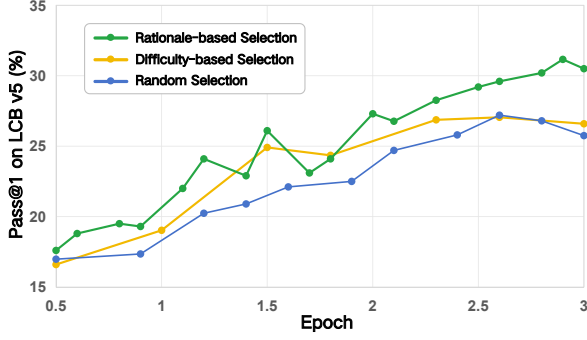


Figure 5: Comparison of data selection.

Table 6: Long-CoT vs. short-CoT.

	Epoch	LCB v5	LCB v6
Short-CoT	3	35.0	29.3
	8	43.1	37.6
	Δ	+8.1	+8.3
Long-CoT	3	42.9	36.0
	8	60.3	53.5
	Δ	+17.4	+17.5

and random selection. Solutions are generated by Qwen3-235B-A22B-Instruct-2507, and models are trained for 3 epochs. As shown in Figure 5, tasks that induce longer CoT are more valuable training data for competitive programming, providing a practical heuristic for efficient data pruning in resource-constrained settings.

The Long-CoT Tradeoff: Higher Quality, Slower Convergence. CoT length matters: longer CoTs improve performance but raise training cost and slow convergence. We demonstrate this by training Qwen2.5-Coder-7B-Instruct on solutions generated by DeepSeek-R1-0528 (long-CoT) and Qwen3-235B-A22B-Instruct-2507 (short-CoT) for an identical set of 200k tasks.

As shown in Table 6, the long-CoT approach achieves a 17.2% absolute gain. This gain justifies the increased computational demand, which manifests as a slower convergence requiring 8–10 epochs compared to the 2–3 epochs needed for short-CoT data.

Synthetic Data Improves Test Time Scaling. We compare the pass@k performance in Figure 6. X-Coder-7B-RL improves over its foundation by 51.3 points in pass@16 and matches Qwen3-8B with 8 $\times$ fewer rollouts. It also shows a larger pass@1-to-pass@16 gap than Qwen3-8B, suggesting more diverse reasoning trajectories.

Reasoning Errors Remain the Bottleneck. Table 7 shows that wrong answers dominate across

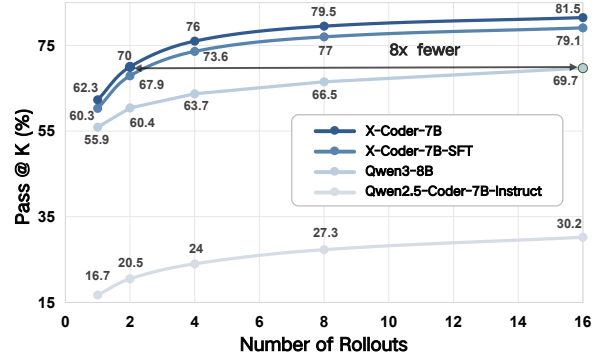


Figure 6: Test-time scaling performance.

Table 7: Distribution of failure cases on LCB v5. Base refers to Qwen2.5-Coder-7B-Instruct.

Error Type	Base	+SFT	+RL
Wrong Answer	194.6 $\pm$ 10.7	69.6 $\pm$ 3.7	67.9 $\pm$ 4.9
Context Exhaustion	6.5 $\pm$ 8.2	21.9 $\pm$ 3.7	11.8 $\pm$ 3.9
Time Limit Exceeded	18.1 $\pm$ 4.1	13.7 $\pm$ 3.3	11.5 $\pm$ 2.6
Memory Limit Exceeded	0.0 $\pm$ 0.0	0.0 $\pm$ 0.0	0.17 $\pm$ 0.4
Signature Mismatch	0.0 $\pm$ 0.0	0.0 $\pm$ 0.0	1.0 $\pm$ 0.8
Syntax Error	0.0 $\pm$ 0.0	0.0 $\pm$ 0.0	8.3 $\pm$ 2.2

models, making algorithmic reasoning the main bottleneck. The next most frequent failures are context-window exhaustion and time limit errors. The former occurs when the model exhausts its 32k-token budget and emits no complete executable solution. Future work should target deeper reasoning, concise reasoning control, and execution efficiency.

Additional details on baselines, feature evolution, and qualitative reasoning behaviors are provided in Appendices A.5, B.1, and G.1.

7 Conclusion

In this work, we demonstrate that synthetic training artifacts can support the complete SFT-to-RL cycle without directly using real-world tasks, solutions, or tests. Our pipeline produces solvable yet challenging tasks, verified long-CoT solutions, and executable tests, with dual verification reducing noise in selected solutions and test outputs. X-Coder-14B achieves 67.5% avg@8 on LiveCodeBench v5 and 63.4% on v6, outperforming its base model by over 40 points, while additional results transfer to newer and cross-family backbones. Our analysis highlights the value of task diversity, verified long-CoT supervision, and code-centric reinforcement learning. Our code and data are publicly available.

Limitations

In this paper, we demonstrate that synthetic tasks, solutions, and tests can effectively support competitive-programming post-training. Our claim concerns these training artifacts, not the whole provenance chain: TACO-derived abstract features seed task synthesis, and teacher models may encode knowledge learned from real-world corpora. Frontier teachers are not strictly necessary (Appendix A.6), but teacher quality affects downstream gains.

The limitations are twofold. First, we focus on self-contained competitive-programming tasks. This setting captures core algorithmic reasoning and supports clean executable verification, but our conclusions may not directly extend to broader scenarios, such as repository-level generation and debugging (Luo et al., 2026), feature implementation, command-line environments, and visual coding tasks.

Second, dual verification is imperfect. Human auditing finds 12.7% residual error in both golden solutions and voted labels before deterministic filtering, and correlated teacher errors can survive consensus. Lower-confidence samples are more tolerable in SFT than in RL, where wrong labels directly shape rewards; we therefore reserve more strongly verified data for RL. Test synthesis, feedback refinement, and adversarial testing may improve reliability and curb reward hacking.

Acknowledgements

This work was partly supported by the National Natural Science Foundation of China (Grant No. 62576191), the Shenzhen Science and Technology Program (ZDCY20250901103533010).

References

- Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. 2025. [Opencodereasoning: Advancing data distillation for competitive coding](#). *Preprint*, arXiv:2504.01943.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *Preprint*, arXiv:2108.07732.
- Jianfeng Cai, Jinhua Zhu, Ruopei Sun, Kangwen Zhao, Dongyun Xue, Mingxiao Feng, Wengang Zhou, and Houqiang Li. 2026. [Codecontests-o: Powering llms via feedback-driven iterative test case generation](#). *Preprint*, arXiv:2601.13682.
- Yuhan Cao, Zian Chen, Kun Quan, Ziliang Zhang, Yu Wang, Xiaoning Dong, Yeqi Feng, Guanzhong He, Jingcheng Huang, Jianhao Li, Yixuan Tan, Jiafu Tang, Yilin Tang, Junlei Wu, Qianyu Xiao, Can Zheng, Shouchen Zhou, Yuxiang Zhu, Yiming Huang, and Tianxing He. 2026. [Can llms generate reliable test case generators? a study on competition-level programming problems](#). *Preprint*, arXiv:2506.06821.
- Mark Chen, Jerry Tworek, Heewoo Jun, and Qiming Yuan et al. 2021. [Evaluating large language models trained on code](#). *Preprint*, arXiv:2107.03374.
- Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. 2024. [Process supervision-guided policy optimization for code generation](#). *CoRR*, abs/2410.17621.
- DeepSeek-AI. 2025. [Deepseek-rl: Incentivizing reasoning capability in llms via reinforcement learning](#). *Preprint*, arXiv:2501.12948.
- Jia Fu, Xinyu Yang, Hongzhi Zhang, Yahui Liu, Jingyuan Zhang, Qi Wang, Fuzheng Zhang, and Guorui Zhou. 2025a. [Klear-codetest: Scalable test case generation for code reinforcement learning](#). *Preprint*, arXiv:2508.05710.
- Wei Fu, Jiakuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. 2025b. [Areal: A large-scale asynchronous reinforcement learning system for language reasoning](#). *Preprint*, arXiv:2505.24298.
- Kanishk Gandhi, Ayush K Chakravarthy, Anikait Singh, Nathan Lile, and Noah Goodman. 2025. [Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective STars](#). In *Second Conference on Language Modeling*.
- Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvama, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. 2025. [Openthoughts: Data recipes for reasoning models](#). *Preprint*, arXiv:2506.04178.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma,

- Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Jujie He, Jiakai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, Siyuan Li, Liang Zeng, Tianwen Wei, Cheng Cheng, Bo An, Yang Liu, and Yahui Zhou. 2025a. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*.
- Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. 2025b. *Hardtests: Synthesizing high-quality test cases for llm coding*. *Preprint*, arXiv:2505.24098.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, and et al. 2021. Measuring coding challenge competence with APPS. In *NeurIPS Datasets and Benchmarks*.
- Max Hort and Leon Moonen. 2025. *Codehacks: A dataset of adversarial tests for competitive programming problems obtained from codeforces*. *Preprint*, arXiv:2503.23466.
- Hugging Face. 2025. *Open r1: A fully open reproduction of deepseek-r1*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. *Qwen2.5-coder technical report*. *Preprint*, arXiv:2409.12186.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. *Livecodebench: Holistic and contamination free evaluation of large language models for code*. *Preprint*, arXiv:2403.07974.
- Bespoke Labs. 2025. Bespoke-stratos: The unreasonable effectiveness of reasoning distillation. <https://www.bespokelabs.ai/blog/bespoke-stratos-the-unreasonable-effectiveness-of-reasoning-distillation>. Accessed: 2025-01-22.
- Nathan Lambert. 2026. *Reinforcement learning from human feedback*. *Preprint*, arXiv:2504.12501.
- Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xi-angxi Mo, Eric Tang, Sumanth Hegde, Kourosh Hakhamaneshi, Shishir G. Patil, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. *Llms can easily learn to reason from demonstrations structure, not content, is what matters!* *Preprint*, arXiv:2502.07374.
- Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS ’23*, Red Hook, NY, USA. Curran Associates Inc.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. *Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation*. In *Thirty-seventh Conference on Neural Information Processing Systems*.
- Steven Liu, Jane Luo, Xin Zhang, Aofan Liu, Hao Liu, Jie Wu, Ziyang Huang, Yangyu Huang, Yu Kang, and Scarlett Li. 2026. *Testexplora: Benchmarking llms for proactive bug discovery via repository-level test generation*. *Preprint*, arXiv:2602.10471.
- Yifei Liu, Li Lina Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. 2025a. *rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset*. *Preprint*, arXiv:2505.21297.
- Zihan Liu, Zhuolin Yang, Yang Chen, Chankyu Lee, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2025b. *Acereason-nemotron 1.1: Advancing math and code reasoning through sft and rl synergy*. *Preprint*, arXiv:2506.13284.
- Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, Hao Sun, Qi Chen, Scarlett Li, and Mao Yang. 2026. *Rpg: A repository planning graph for unified and scalable codebase generation*. *Preprint*, arXiv:2509.16198.
- Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. Deep-coder: A fully open-source 14b coder at o3-mini level. Notion Blog.
- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. *Wizardcoder: Empowering code large language models with evol-instruct*. In *The Twelfth International Conference on Learning Representations*.

- Zihan Ma, Taolin Zhang, Maosongcao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. 2026. *Rethinking verification for LLM code generation: From generation to testing*. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.
- OpenAI. 2025. Openai o3-mini: Pushing the frontier of cost-effective reasoning. <https://openai.com/index/openai-o3-mini/>. Accessed: 2025-09-11.
- Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piñeres Lajarán, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. 2025. Codeforces. <https://huggingface.co/datasets/open-rl/codeforces>.
- Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. *Coffe: A code efficiency benchmark for code generation*. Preprint, arXiv:2502.02827.
- Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yaqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. *Qwen2.5 technical report*. Preprint, arXiv:2412.15115.
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. *Proximal policy optimization algorithms*. Preprint, arXiv:1707.06347.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024a. *Deepseekmath: Pushing the limits of mathematical reasoning in open language models*. Preprint, arXiv:2402.03300.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. 2024b. *Deepseekmath: Pushing the limits of mathematical reasoning in open language models*. arXiv preprint arXiv:2402.03300.
- Jingwei Shi, Xinxiang Yin, Jing Huang, Jinman Zhao, and Shengyu Tao. 2026. *Codehacker: Automated test case generation for detecting vulnerabilities in competitive programming solutions*. Preprint, arXiv:2602.20213.
- Zhenpeng Su, Leiyu Pan, Xue Bai, Dening Liu, Guanting Dong, Jiaming Huang, Wenping Hu, Fuzheng Zhang, Kun Gai, and Guorui Zhou. 2025. *Klear-reasoner: Advancing reasoning capability via gradient-preserving clipping policy optimization*. Preprint, arXiv:2508.07629.
- Qiushi Sun, Jinyang Gong, Lei Li, Qipeng Guo, and Fei Yuan. 2025. *Codeevo: Interaction-driven synthesis of code-centric data through hybrid and iterative feedback*. Preprint, arXiv:2507.22080.
- Sicheol Sung, Aditi, Dogyu Kim, Yo-Sub Han, and Sang-Ki Ko. 2025. *Logicase: Effective test case generation from logical description in competitive programming*. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pages 7742–7750. International Joint Conferences on Artificial Intelligence Organization. Main Track.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171.
- Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, et al. 2025a. *Epicoder: Encompassing diversity and complexity in code generation*. arXiv preprint arXiv:2501.04694.
- Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. 2025b. *CodeContests+: High-quality test case generation for competitive programming*. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 5576–5600, Suzhou, China. Association for Computational Linguistics.
- Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and LINGMING ZHANG. 2024. *Selfcodealign: Self-alignment for code generation*. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. 2025. *Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution*. Preprint, arXiv:2502.18449.
- Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Schwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddhartha Venkat Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2025. *Livebench: A challenging, contamination-limited LLM benchmark*. In *The Thirteenth International Conference on Learning Representations*.
- Jie Wu, Haoling Li, Xin Zhang, Xiao Liu, Yangyu Huang, Jianwen Luo, Yizhen Zhang, Zuchao Li, Ruihang Chu, Yujiu Yang, and Scarlett Li. 2025. *Teaching your models to understand code via focal preference alignment*. Preprint, arXiv:2503.02783.
- LLM-Core Xiaomi, :, Bingquan Xia, Bowen Shen, Cici, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, Jinhao Dong, Liang

- Zhao, Peidian Li, Peng Wang, Shihua Yu, Shimao Chen, Weikun Wang, Wenhan Ma, Xiangwei Deng, Yi Huang, Yifan Song, Zihan Jiang, Bowen Ye, Can Cai, Chenhong He, Dong Zhang, Duo Zhang, Guoan Wang, Hao Tian, Haochen Zhao, Heng Qu, Hongshen Xu, Jun Shi, Kainan Bao, Kai Fang, Kang Zhou, Kangyang Zhou, Lei Li, Menghang Zhu, Nuo Chen, Qiantong Wang, Shaohui Liu, Shicheng Li, Shuhao Gu, Shuhuai Ren, Shuo Liu, Sirui Deng, Weiwei Zhuang, Weiwei Lv, Wenyu Yang, Xin Zhang, Xing Yong, Xing Zhang, Xingchen Song, Xinze Xu, Xu Wang, Yihan Yan, Yu Tu, Yuanyuan Tian, Yudong Wang, Yue Yu, Zhenru Lin, Zhichao Song, and Zihao Yue. 2025. *Mimo: Unlocking the reasoning potential of language model – from pretraining to posttraining*. *Preprint*, arXiv:2505.07608.
- Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. *WizardLM: Empowering large pre-trained language models to follow complex instructions*. In *The Twelfth International Conference on Learning Representations*.
- Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. 2025. *KodCode: A diverse, challenging, and verifiable synthetic dataset for coding*. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 6980–7008, Vienna, Austria. Association for Computational Linguistics.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025a. *Qwen3 technical report*. *Preprint*, arXiv:2505.09388.
- An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. 2024. *Qwen2.5-math technical report: Toward mathematical expert model via self-improvement*. *Preprint*, arXiv:2409.12122.
- Zheyuan Yang, Zexi Kuang, Xue Xia, and Yilun Zhao. 2025b. *Can LLMs generate high-quality test cases for algorithm problems? TestCase-eval: A systematic evaluation of fault coverage and exposure*. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1050–1063, Vienna, Austria. Association for Computational Linguistics.
- Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. *Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning*. *Preprint*, arXiv:2312.14187.
- Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhui Chen. 2025. *ACECODER: Acing coder RL via automated test-case synthesis*. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12023–12040, Vienna, Austria. Association for Computational Linguistics.
- 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. 2026. *Autocode: LLMs as problem setters for competitive programming*. In *The Fourteenth International Conference on Learning Representations*.

Appendix

A	Training and Evaluation	13
A.1	Reward Function	14
A.2	Training Dynamics	14
A.3	Training Configs and Costs	14
A.4	Distributed Code Verification	15
A.5	Baselines	15
A.6	Open-Source Teacher Ablation	15
B	Novel Task Synthesis	16
B.1	Feature Extraction and Evolution	16
B.2	Stylized Task Generation for Competitive Programming	17
B.3	Task Difficulty Estimates	19
B.4	Task Diversity Estimates	20
B.5	Comparison with Alternative Synthesis Strategies	20
C	Solution Generation and Quality Assurance	21
C.1	Validation on Solution	21
C.2	SFT Dataset Statistics	21
D	Test Case Generation	21
D.1	Prompting-based Test Generation	21
D.2	Comparison of Test Generation Methods	21
D.3	Tool-based Test Generation	22
E	Dual-verification	23
E.1	Core Algorithm	23
E.2	Test-Case Weighting Criteria	23
E.3	Validation Effectiveness Analysis	23
E.4	Solution Quality Assessment	24
E.5	Task Solvability Analysis	25
F	Data Leakage Analysis	25
G	Case Study	26
G.1	Successful Case	26
G.2	Bad Case	26
G.3	Reward Hacking	27
G.4	Extended Analysis	28

A Training and Evaluation

Supervised Fine-tuning. Given a dataset of task–solution pairs $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, the model with parameters θ is trained by minimizing the negative log-likelihood (NLL) of the target solution y conditioned on the task x :

$$J_{\text{SFT}}(\theta) = -\mathbb{E}_{(x,y) \sim \mathcal{D}} \left[\sum_{t=1}^{|y|} \log \pi_{\theta}(y_t \mid x, y_{<t}) \right]. \quad (3)$$

The loss is applied over full long-CoT trajectories, including both reasoning steps and final code, enabling the model to imitate not only the solutions but also the underlying reasoning patterns.

Reinforcement Learning. Proximal Policy Optimization (PPO) (Schulman et al., 2017) is a widely adopted policy gradient method in Reinforcement Learning from Human Feedback (RLHF) (Lambert, 2026) for LLM due to its balance between exploration and exploitation and its empirical robustness. The method optimizes a policy π_{θ} by using a clipped surrogate objective to limit policy divergence, incorporating a value function to estimate expected rewards, and an entropy term to encourage exploration. The overall objective function for PPO is designed to maximize the policy performance while maintaining stability, and it is typically formulated as minimizing the following:

$$J_{\text{PPO}}(\theta) = \mathbb{E}_{s \sim P(S), a \sim \pi_{\theta}(a|s)} \left[\min \left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A(s, a), \text{clip} \left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A(s, a) \right) \right] \quad (4)$$

where the expectation is computed over states s (drawn from distribution $P(S)$) and actions a (sampled from the current policy $\pi_{\theta}(a \mid s)$), combining the minimum of two terms: (1) the product of the probability ratio $\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ and the advantage function $A(s, a)$, where the advantage function quantifies the relative benefit of taking action a in state s ; and (2) the same product but with the probability ratio clipped to the interval $[1 - \epsilon, 1 + \epsilon]$. Here, ϵ is a hyperparameter governing the magnitude of policy updates. This clipping mechanism effectively constrains excessive policy changes, thereby enhancing training stability.

However, its application to LLMs encounters significant challenges, including substantial computational overhead from maintaining a critic network, which increases memory usage and training time for models with billions of parameters. To address

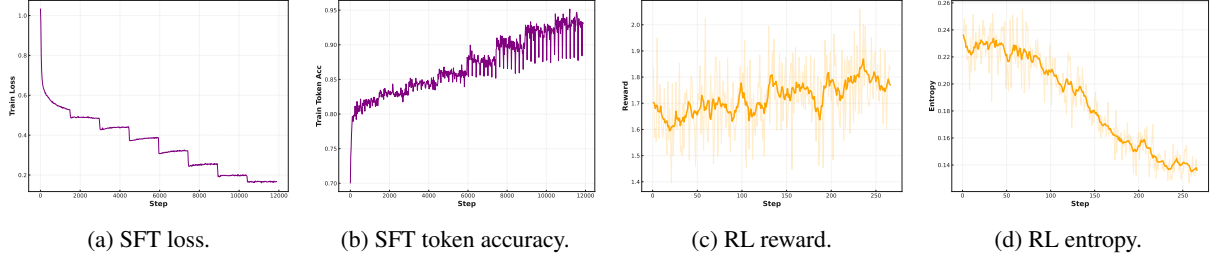


Figure 7: Training dynamics for SFT and RL.

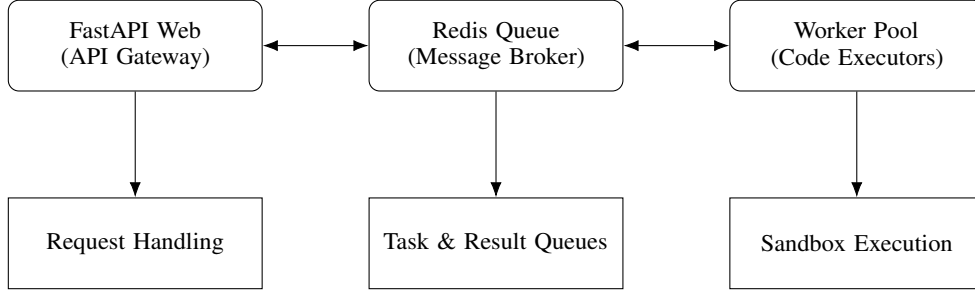


Figure 8: The distributed architecture of the code verification framework.

these limitations, Group Relative Policy Optimization (GRPO) (Shao et al., 2024a) has emerged as an efficient alternative. By eliminating the critic network, GRPO reduces computational and memory demands, estimating advantages directly from rewards of multiple rollouts to the same prompt, thus leveraging the comparative nature of reward models and offering a scalable solution for LLM training. The GRPO objective function is mathematically formulated as an averaged composite expression across multiple rollouts, incorporating policy ratio optimization and KL regularization:

$$J_{\text{GRPO}}(\theta) = \frac{1}{G} \sum_{i=1}^G \frac{1}{|a_i|} \sum_{t=1}^{|a_i|} \left\{ \min \left(\rho_{i,t} \hat{A}_{i,t}, \text{clip}(\rho_{i,t}, 1 - \epsilon, 1 + \epsilon) \hat{A}_{i,t} \right) - \beta D_{\text{KL}}[\pi_{\theta} \parallel \pi_{\text{ref}}] \right\} \quad (5)$$

where $\rho_{i,t} = \frac{\pi_{\theta}(a_{i,t}|s, a_{i,<t})}{\pi_{\theta_{\text{old}}}(a_{i,t}|s, a_{i,<t})}$ denotes the probability ratio of the old and new strategies. G is the number of rollouts per prompt, $|a_i|$ denotes the length of the i -th action sequence, $\hat{A}_{i,t}$ estimates the advantage of action $a_{i,t}$ at timestep t . The clipping is analogous to PPO, and β penalizes deviations from π_{ref} via the KL-divergence term. The objective averages across rollouts and timesteps, combining a clipped probability ratio

(to stabilize updates while leveraging advantage signals) with a KL penalty to balance policy improvement against alignment with the reference policy. This dual mechanism ensures controlled optimization by restricting drastic policy shifts while maintaining coherence with prior behavior.

A.1 Reward Function.

We remove formatting rewards (e.g., enforcing “think” tags), as the SFT model already follows the format, allowing the policy to focus on passing test cases. Given a rollout, the reward R is practiced as:

$$R = \begin{cases} -2, & \text{if no code is extracted or the code fails to compile,} \\ 0, & \text{if the code compiles but passes no test cases,} \\ \frac{5.0 \times \# \text{passed}}{\# \text{total}}, & \text{otherwise.} \end{cases} \quad (6)$$

We adopt a continuous reward setting, as it provides denser supervision than the all-or-nothing alternative and leads to faster convergence (Wei et al., 2025; Dai et al., 2024).

A.2 Training Dynamics.

As shown in Figure 7a and Figure 7b, we present the SFT training curves (loss and token accuracy). Figure 7c and Figure 7d illustrate the RL training curves (reward and entropy).

A.3 Training Configs and Costs

For SFT, we use a learning rate of $5e-5$ with a global batch size of 128 for 8 training epochs. For

RL, the policy models are updated with a global batch size of 512 and a consistent learning rate of $7e-5$, without applying the KL-divergence constraint, and employ a rollout temperature of 1.0 with 8 rollouts for each query. Concretely, training X-Coder on Qwen2.5-Coder-7B-Instruct required 128 H20 Enterprise (96 GB) GPUs for 220 hours during SFT, and 32 H200 GPUs for 7 days to complete 270 update steps during RL.

Data construction is also compute-intensive: verifying 200k samples generates 1.6M long-CoT candidate trajectories and performs approximately 24M sandbox executions. This is a one-time construction expense; it improves LCB v5 by 6.4 points over training on raw solutions (Table 5), while the raw variant remains a lower-cost option. We report trajectory and execution counts rather than a monetary API estimate because price schedules and deployment choices differ across teacher providers.

A.4 Distributed Code Verification

To provide a robust and scalable solution for code validation, we develop a distributed arbitration framework inspired by open-source repository implementations⁶. The system is based on a microservice architecture, comprising a *FastAPI*-based asynchronous API Gateway, a pool of code execution workers in the sandbox and a central *Redis* instance. *Redis* serves as a high-performance message broker and state manager, effectively decoupling the client-facing gateway from the backend computational workers. This architectural choice facilitates independent scaling, deployment, and enhances the overall resilience of the system. **Based on this evaluation framework, we implemented highly concurrent code testing during RL training.** We used batching when submitting tasks to the *Redis* server to achieve high concurrency even with low request rates. This process required the server to distribute all test tasks to different workers, utilizing the CPU power of all participating machines. Figure 8 shows the system diagram of the framework.

The framework’s efficacy is derived from its strategic implementation of *Redis* data structures. Task distribution is managed by a *Sorted Set*, which functions as a time-prioritized FIFO queue; submissions are added with a timestamp score via *ZADD*,

⁶<https://github.com/0xWJ/code-judge.git>

and workers atomically retrieve the next task using *BZPOPMIN*. This approach ensures ordered processing and prevents race conditions. For result transmission, each task is assigned a dedicated *List*, to which a worker pushes the outcome using *RPush*. The API Gateway then performs a blocking pop (*BLPOP*) on this unique list to retrieve the corresponding result efficiently. Furthermore, worker health and presence are monitored using *String* keys with a Time-To-Live (TTL). Workers periodically refresh their key’s TTL as a heartbeat, enabling the system to automatically detect and de-register unresponsive nodes.

The resulting system exhibits several key advantages. The asynchronous, in-memory nature of its core components yields high throughput and low-latency performance. Its design is inherently scalable, as the stateless worker pool can be expanded horizontally to meet computational demand, while native support for *Redis* Cluster addresses data-tier bottlenecks. Finally, the framework’s reliability is bolstered by the atomicity of *Redis* operations and the integrated fault-detection mechanism, ensuring dependable and consistent code verification.

A.5 Baselines

We compare the X-Coder with three categories of baselines: (1) SFT model, e.g., Bespoke-Stratos, OlympicCoder, OCR-Qwen-Instruct, OpenThinker3, Qwen3-8B, and rStar-Coder; (2) RL model, including Skywork-OR1, DeepCoder-14B-Preview, and AReal-boba2-14B; (3) SFT-then-RL model, such as AceReason1.1, Klear-Reasoner, and MiMo-RL.

A.6 Open-Source Teacher Ablation

To test whether the pipeline requires frontier proprietary teachers, we replace task formulation, test generation, and solution generation with Qwen3.5-35B-A3B, using neither GPT-5 filtering nor Qwen3-235B. We compare 32k tasks from this open-source variant against a 32k subsample of the proprietary pipeline, keeping the Qwen2.5-Coder-7B-Instruct student and SFT configuration fixed.

Table 8: Teacher ablation on LCB v5 (avg@8).

Pipeline	Tasks	LCB v5
Base model	–	17.5
Open-source teacher	32k	27.7 (+10.2)
Proprietary teachers	32k	35.9 (+18.4)

Frontier teachers are therefore not necessary for

the pipeline to produce substantial gains, although teacher quality determines the attainable improvement.

B Novel Task Synthesis

Building on EpiCoder, which synthesizes programming tasks through feature-based combinations, we introduce three key improvements to generate more diverse and complex instructions.

First, rather than relying on broad feature definitions, we explicitly extract and evolve competition-related features from 10,000 question–solution pairs in TACO (Li et al., 2023) using GPT-4o-0513 (Appendix B.1). *Second*, we adopt a two-stage process: selecting mutually consistent features and then formulating challenging, hint-free tasks (Appendix B.2). *Third*, we extend the synthesis method to support multi-style generation, covering CodeForces-style tasks (rich narratives with standard I/O), LeetCode-style tasks (starter code with fixed signatures), and AtCoder-style tasks (concise specifications), thereby enhancing task diversity. In Appendix B.3, we further estimate the difficulty of synthesized problems using a trained discriminator.

B.1 Feature Extraction and Evolution

While EpiCoder extracts general-purpose features from raw corpus, we explicitly extract and evolve competitive programming-related feature. Specifically, we design multiple aspect of features that highly relates to competitive programming, such as data structure, algorithm, mathematical, etc.

We improve the extraction process to guide the LLM to focus on competitive programming–related concepts, as follows:

```
Extract features from the provided problem and solution
code related to algorithmic programming, competitive
programming, Leetcode, and Codeforces, following the
requirements for each category below, formatted in JSON
structure.
Responses in the following categories should be concise
and organized in a JSON format surrounded with <begin>
and <end>. Categories may include nested structures if
applicable. Here is an example of the expected format:
<begin>{
  "programming language": [
    "Python"
  ],
  "problem type": [
    "graph traversal"
  ],
  "algorithm": {
    "graph algorithms": [
      "Dijkstra's algorithm",
      "DFS",
      "BFS"
    ],
    "dynamic programming": [
      "Longest Increasing Subsequence",
      "Knapsack Problem"
    ]
  },
  ...
  "implementation logic": ["recursive", "iterative"]
}<end>
```

```
Categories to extract:
1. Programming Language: Note the specific programming
   language used. Example: ["Python", "C++"].
2. Problem Type: Outline the type of problem the code is
   solving. Example: ["graph traversal", "sorting", "dynamic
   programming"].
3. Algorithm: Identify the specific algorithm or method
   being used in the code. This category can include the
   following subcategories:
   3.1 Graph Algorithms: Specify graph algorithms used.
   Example: ["Dijkstra's algorithm", "DFS", "BFS"].
   3.2 Sorting Algorithms: Specify sorting algorithms
   used. Example: ["QuickSort", "MergeSort"].
   ...
4. Data Structures: Describe the primary data structures
   utilized. Example: ["array", "graph", "tree", "heap"].
5. Implementation Logic: Describe the implementation
   logic. Example: ["iterative", "recursive", "bit
   manipulation"].
6. Complexity Analysis: Provide time and space complexity
   of the code if available. Example: ["Time Complexity: O(n
   log n)", "Space Complexity: O(n)"].
7. Optimization Techniques: Specify any optimizations
   applied. Example: ["memoization", "greedy approaches",
   "bitwise operations"].
   ...
Extract as many features as possible and try not to let a
feature appear in multiple categories at the same time.
```

Then we increase the diversity and complexity through evolution along both the breadth and depth dimensions. For example, along the breadth dimension, given an extracted feature such as quicksort, the LLM may evolve new features like bubble sort, even if they were not originally extracted. Along the depth dimension, a concept such as prefix sum can evolve into more advanced variants like difference array or Fenwick tree, reflecting increasing levels of abstraction and difficulty. The overall evolution process is illustrated below.

Table 9: Public checkpoints reevaluated under a unified avg@8 protocol on LCB (8 samples, temperature 0.6, top-p 0.95, and official chat templates). Reported values retain each source’s original metric and are shown only for context.

Model	Reported v5	Reported v6	Avg@8 v5	Avg@8 v6
OlympicCoder-7B	40.9	–	41.2	34.1
OCR-Nemotron-7B	51.3 (avg@64)	44.5	49.6	44.2
Skywork-OR1-7B	47.6 (avg@32)	40.0	46.5	40.5
DeepCoder-14B-Preview	57.9 (pass@1)	48.5	56.5	47.2

Feature Tree Evolution Task:
You are provided with a feature tree represented as a nested JSON structure. Each node in this tree represents a feature or a sub-feature of competitive algorithm programming, with the leaves being the most specific features. Your task is to expand this feature tree both in depth and breadth. Depth expansion means adding more specific sub-features to existing leaves. Breadth expansion means adding more sibling features at the current levels.

Here are some explanations of the features:
(explanations)
The input feature tree will be provided in JSON format, and your output should be a JSON structure that represents the expanded feature tree.
Output Format:
- Expanded Feature Tree: Provide the expanded feature tree as a JSON structure. Surround the json with <begin> and <end>.

Input Feature Tree Example:

```
{
  "algorithm": {
    "sorting": ["quick sort", "merge sort"],
    "tree traversal": ["in-order traversal"]
  },
  ...
}
```

Expanded Feature Tree Example:

```
<begin>
{
  "algorithm": {
    "sorting": {
      "quick sort": ["3-way quick sort",
"dual-pivot quick sort"],
      "merge sort": ["top-down merge sort",
"bottom-up merge sort"],
      "heap sort": []
    },
    "tree traversal": {
      "in-order traversal": ["recursive in-order
traversal", "iterative in-order traversal"],
      "pre-order traversal": [],
      "post-order traversal": [],
      "level-order traversal": [],
    }
  },
  ...
}
<end>
```

Constraints:
1. For breadth expansion, add at least 2 new sibling features to each existing node.
2. For deep expansion, you need to add new sub-features to it, provided that you think the current leaf node has a more fine-grained feature.
3. Focus on generating new and innovative features that are not present in the provided examples.
4. The features are related to competitive algorithm programming.
Please follow the above constraints and expand the feature tree accordingly.

Input:
{features}
Output:
<begin>expanded feature tree<end>

After evolution, we merge features that share common traits into a larger tree, providing a rich pool of features for subsequent task formulation.

B.1.1 Statistics for Feature Extraction and Evolution

We present detailed statistics on feature evolution and data filtering to demonstrate how the process expands feature diversity and yields a high-quality 240k dataset. The statistics of feature extracted and evolved are shown in Table 10.

The evolution strategy is the primary driver of feature-pool breadth: starting from 10k TACO tasks, it expands individual categories by factors of 5.3–9.2 and yields 582,608 features, exceeding the 428,628 features directly extracted from the multi-source OCR corpus. Broader real-data seeds and evolution remain complementary: OCR provides a much stronger initial pool than raw TACO, and future versions can evolve multi-source seeds.

B.2 Stylized Task Generation for Competitive Programming

We design a prompt template to systematically transform extracted features into stylized competitive programming tasks.

Input: a sampled feature tree represented in JSON format.

Output: a feature-role tree (JSON), where each node is assigned roles such as core technique, sub-routine, or constraint, together with an integration strategy (string) that explains how to combine these features into a coherent problem.

To improve instruction-following and task understanding, the template is equipped with a one-shot example that demonstrates how raw features are mapped into roles and integrated into a task.

Table 10: Feature counts from a single-source TACO seed, a larger multi-source OpenCodeReasoning (OCR) corpus, and feature evolution. Evolution from 10k TACO tasks produces more features overall than extraction from 28k multi-source real tasks.

Category	TACO 10k	OCR 28k	TACO Evolved	Growth
Algorithm	27,400	168,274	176,914	×6.46
Data Structures	12,353	54,061	65,104	×5.27
Problem Type	14,134	73,481	130,293	×9.22
Implementation Logic	12,419	61,263	106,157	×8.55
Complexity Analysis	16,124	47,088	90,016	×5.58
Optimization Techniques	1,537	24,461	14,124	×9.19
Total	83,967	428,628	582,608	×6.94

```

You are a professional competitive programming problem
setter. Your task consists of three parts:

Step 1: Tree-Structured Feature Role Explanation
Recursively traverse the provided feature tree.
- For each leaf node, annotate it with a "potential_use"
field describing how this feature is typically used in
competitive programming problems (e.g., input modeling,
optimization, search, handling edge cases, etc.).
- Internal nodes retain their structure for hierarchy.
Output the annotated tree in the same structure, with
every leaf node containing its "potential_use".
Step 2: Subtree Selection for Problem Integration
Based on your role analysis, select a subtree
(tree-structured subset) where all selected leaf features
can be naturally integrated into a single, high-quality
competitive programming problem.
- Only include features that contribute meaningfully to
the same problem idea.
- Internal nodes are included only if they have selected
children.
- For each selected leaf, include only its "feature" name
and "potential_use".
Step 3: Integration Strategy
Briefly describe ("integration_strategy") how the
selected features can be integrated together in a single
problem, focusing on how their combination enables a
meaningful and challenging algorithmic scenario.

Return a JSON object **with exactly this structure** (an
example):
{
  "feature_roles_tree": {
    "algorithm": {
      "search algorithm": {
        "binary search": {
          "recursive binary search": {
            "potential_use": "Used for divide-and-conquer
searching in sorted structures or answer spaces."
          },
          ...
        },
        ...
      },
      ...
    },
    "selected_features_tree": {
      "algorithm": {
        "search algorithm": {
          "binary search": {
            "recursive binary search": {
              "feature": "recursive binary search",
              "potential_use": "Used for divide-and-conquer
searching in sorted structures or answer spaces."
            },
            ...
          },
          ...
        },
        ...
      },
      "integration_strategy": "The problem will require
recursive binary search to efficiently search over a
sorted value space, while bitwise AND operations will be
used to filter candidate solutions according to
constraints. Their combination allows for a problem that
involves searching over sets and optimizing bitwise
criteria."
    },
    ...
  },
  **Available Features (Tree):**
  {features_json}

```

B.2.1 Compatibale Feature Selection

We present a case to examine how model selects compatible features and combine them.

Given a sampled feature tree:

```



```

LLM pairs each feature with *potentially usage* to obtain feature tree with role annotation. For example, LLM will anonote feature “rolling hash” as “Compute hash values for sliding windows in constant time”. These annotations help LLM to aggregate these features based on their potentially usage. For above given feature tree, the feature tree with potential usage looks like:

```

"feature_roles_tree": {
  "algorithms": {
    "graph_algorithms": {
      "shortest_path": {
        "Dijkstra's_algorithm": {
          "potential_use": "Find single-source shortest
paths in weighted graphs with non-negative edges"
        },
        "Floyd_Warshall": {
          "potential_use": "Compute all-pairs shortest
paths with O(n^3) complexity"
        }
      },
      ...
    },
    ...
  }
}

```

LLM then selects a compatible and consistent subtree that can formulate a self-contained competitive programming problem. LLM keeps the features that can be aggregated into selected feature tree, and concluding with an integration strategy, which displays how to combine these features into a unified problem. In this case, LLM selects "Dijkstra's algorithm", "Edmonds-Karp", "segment tree", and "tree DP", and aims to formulate a problem around "dynamic network optimization".

```

"algorithms": {
  "graph_algorithms": {
    "shortest_path": {
      "Dijkstra's_algorithm": {
        "feature": "Dijkstra's algorithm",
        "potential_use": "Primary pathfinding algorithm"
      }
    },
    "network_flow": {
      "Edmonds_Karp": {
        "feature": "Edmonds-Karp",
        "potential_use": "Flow computation with
guaranteed complexity"
      }
    }
  },
  "data_structures": {
    "tree_structures": {
      "segment_tree": {
        "feature": "segment tree",
        "potential_use": "Maintain dynamic edge weights or
capacities"
      }
    }
  },
  ...
  "integration_strategy": "Create a dynamic network
optimization problem where Dijkstra's algorithm finds
shortest paths that are used as augmenting paths in a
modified Edmonds-Karp flow algorithm. Use segment tree to
handle dynamic updates to edge capacities based on flow
history. Apply tree DP on the shortest path tree to
compute optimal flow distributions. This models a
transportation network with time-varying capacities."
}

```

B.2.2 From Feature to Stylized Task

We separate feature selection from task generation, as our initial attempts showed that prompting an LLM to perform both within a single prompt often led it to choose fewer features and produce overly simple problems.

During task generation, LLM receives *selected features tree* and its *integration strategy* to formulate stylized task based on prompt recieved. In this instance, our generated Codeforces problem is shown in Figure 10.

The rationale for above two-stage approach is that a single-step approach is less effective. When performing both steps simultaneously, LLMs tend to oversimplify complex instructions into trivial cases, reducing both diversity and difficulty of the generated task.

To empirically validate this, we generated 32k tasks using the one-step method (feature-tree $\rightarrow$ task) and using proposed “two-stage” method

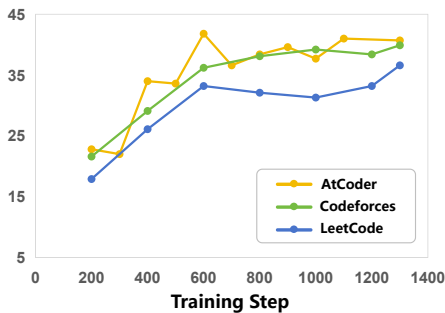


Figure 9: Task style comparison.

Table 11: One-step and two-stage generation.

Generation Method	V5 Score (avg@4)
One-Step (end-to-end)	34.8
Two-Stage	40.1

(feature-tree $\rightarrow$ sub-tree $\rightarrow$ task). The SFT results on LiveCodeBench v5 are as Table 11. The 5.3 gain shows that explicit sub-tree selection and integration is significantly helpful for producing high-quality, challenging tasks and justifies the strategy’s modular design.

B.2.3 Ablation on Task Style

We evaluate the effect of task styles (AtCoder, Codeforces, and LeetCode) by synthesizing three corpora of 32k tasks each (8k unique problems with 4 solutions per problem) from identical input features. For each corpus, solutions are generated with DeepSeek-R1-0528 and used to fine-tune the Qwen2.5-Coder-7B-Instruct. Results are shown in Figure 9. Although AtCoder-style tasks yield slightly higher scores, we adopt Codeforces-style as the predominant format in our demonstration dataset (Codeforces : AtCoder : LeetCode = 70 : 15 : 15), reflecting its prominence as the main-stream competitive-programming platform.

B.3 Task Difficulty Estimates

Judging the difficulty of a synthetic task is challenging. To better capture the difficulty distribution of tasks generated by X-Coder, we adopt a classifier-based approach. Specifically, we add a special classification token to Qwen2.5-Coder-14B-Instruct and fine-tune it to predict the Codeforces rating of 6,246 tasks from the CodeContests dataset with annotated ratings, reserving 5% as a validation set. The fine-tuned model achieves 84% classification accuracy on the validation set. We then use this model to estimate the difficulty of 1,000 tasks generated by the proposed method, obtaining a holistic distribution as shown in Table 12.

The estimated ratings are concentrated at 2400–2600 (90.4%), reflecting our deliberate focus on challenging but learnable problems and likely contributing to weaker gains on the easy split. This should not be conflated with problem diversity: our diversity claim concerns coverage across algorithms, data structures, problem types, and implementation patterns, rather than a balanced rating histogram. Moreover, the 84%-accurate classifier is only a coarse estimator, so we do not treat these

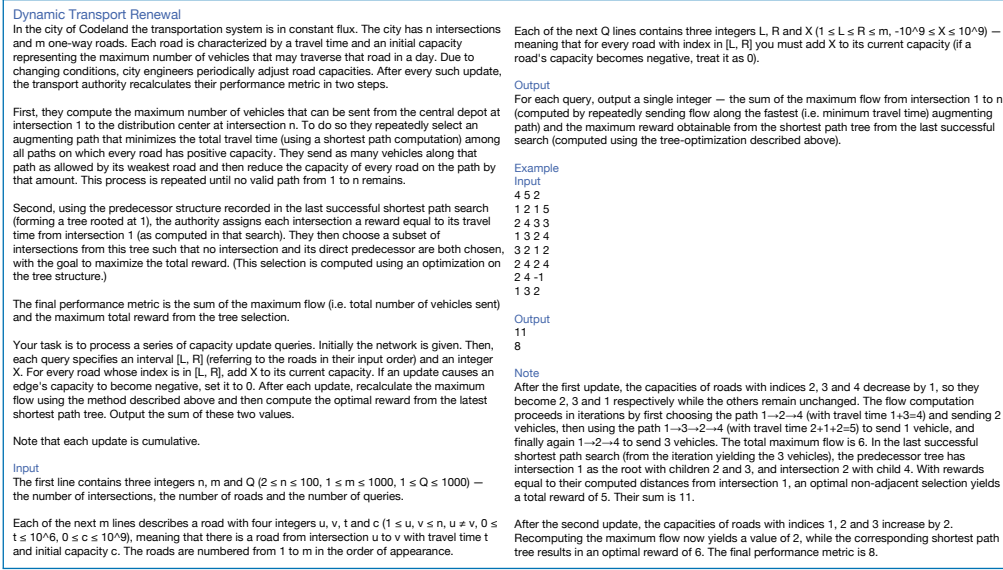


Figure 10: Case for Codeforces-style Problem, featuring rich, imaginable narrative contexts.

Table 12: Difficulty distribution of Codeforces-style ratings. “Original” denotes the annotated distribution from CodeContests, and “Test” denotes 1,000 tasks generated by the proposed method.

CF Rating	Original	Test (Ours)	Original Share	Test Share
1200	623	0	10.0%	0.0%
1400	727	0	11.7%	0.0%
1600	889	0	14.3%	0.0%
1800	840	16	13.5%	1.6%
2000	797	2	12.8%	0.2%
2200	697	47	11.2%	4.7%
2400	665	585	10.7%	58.5%
2600	484	319	7.8%	31.9%
2800	312	12	5.0%	1.2%
3000	233	15	3.7%	1.5%
3200	157	4	2.5%	0.4%
3400	122	0	2.0%	0.0%
Total	6,246	1,000	100%	100%

predicted ratings as ground truth.

B.4 Task Diversity Estimates

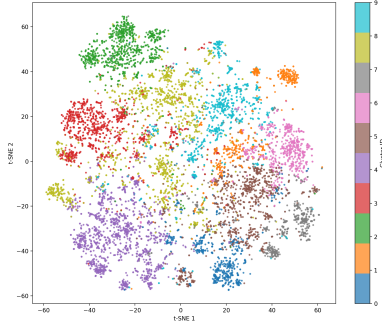
To analyze the diversity of our generated tasks quantitatively, we analyze diversity in the embedding space following the steps below: (i) Embedding: We first embed the tasks into embeddings using *jinaai/jina-embeddings-v2-base-code*, a specialized coding embedding model. (ii) t-SNE Dimensionality Reduction: We apply t-SNE to reduce the embedded data to 2D space. (iii) Clustering: We perform K-means clustering on the t-SNE-reduced data to group the data into 10 clusters and compute the centroids of each cluster. (iv) Inter-cluster Distance Calculation: We calculate the Euclidean distance between cluster centroids. Larger inter-cluster distances indicate greater diver-

sity within the dataset.

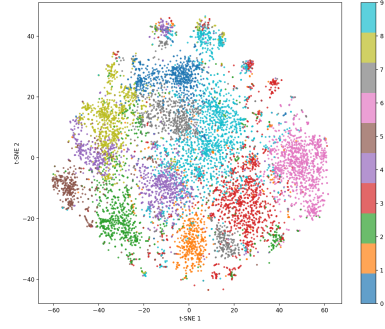
In our datasets (randomly sampled 10k), cluster sizes range 529-1,612 items, average centroid distance 0.613, min 0.369, max 0.760. In Evol-Instruct-Code, the mean centroid distance is 0.507. The visualization results are shown in Figure 11a and Figure 11b. The visualization suggests that the clusters in our dataset are more widely separated compared to those in Evol-Instruct-Code, indicating higher diversity.

B.5 Comparison with Alternative Synthesis Strategies

As presented in Table 13, we compare the proposed strategy against two baselines to validate our methodological choices. First, compared to EpiCoder (using general-purpose prompts), the domain-adapted synthesis achieves a substantial 20.9% improvement. Since this method is built upon EpiCoder’s framework but injects competitive programming-oriented feature extraction and evolution, this gain directly validates the necessity of domain expertise for synthesizing high-complexity tasks. General-purpose synthesis is insufficient for expert-level domains. Second, compared to Self-CodeAlign (Wei et al., 2024) (where we applied the same domain adaptation), the proposed method still yields a 4.6% gain. This suggests that for competitive programming, the feature-based evolution strategy is more effective than concept composition at generating high-quality, diverse, and rigorous training data.



(a) t-SNE visualization of our datasets.



(b) t-SNE of the Evol-Instruct-Code.

Figure 11: Task diversity visualizations in the t-SNE space.

Table 13: Synthetic Data Comparison.

Method	Evolution	Adapted	LCB v5
SelfCodeAlign-10k	✗	✓	27.1
Ours-10k	✓	✓	31.7
EpiCoder-64k	✓	✗	25.4
Ours-64k	✓	✓	46.3

C Solution Generation and Quality Assurance

C.1 Validation on Solution

For tasks with descriptions shorter than 200 tokens, we discard them, as such descriptions are often either too trivial or incomplete. For each generated solution, we ensure quality by (i) removing samples without complete think and answer tags, (ii) rejecting cases where the extracted Python block fails AST validation, (iii) excluding solutions that contain multiple code blocks after the reasoning process, as they hinder reliable solution extraction, and (iv) filtering out samples exceeding 25k tokens to prevent overthinking and to reduce SFT cost caused by sequence padding.

C.2 SFT Dataset Statistics

The overall token length distribution, shown in Table 14, and Figure 12, primarily follows a normal distribution, with a median of 16k.

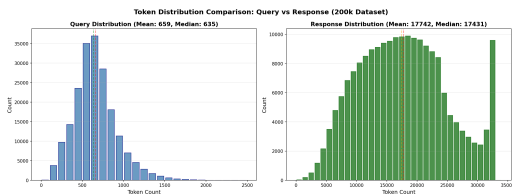


Figure 12: Dataset statistics of the demonstration dataset.

D Test Case Generation

D.1 Prompting-based Test Generation

You are a professional test case generation expert, skilled at designing comprehensive test cases for programming problems. Please generate 15 different test cases for the following programming problem, including edge cases, small-scale, medium-scale, and large-scale test data.

Problem:
{problem_statement}

Requirements:

1. Generate 15 test cases
2. Include edge cases (empty input, minimum values, maximum values, etc.)
3. Include different scales of data (small, medium, large)
4. Each test case should have clear input data
5. Ensure test cases can thoroughly validate the correctness of solutions

Please return in JSON format as follows:

```
{
  "test_cases": [
    {
      "idx": 0,
      "description": "Test case description",
      "input_string": "Input data"
    },
    ...
  ]
}
```

D.2 Comparison of Test Generation Methods

This comparison is scoped to our synthesis pipeline and is not intended as a comprehensive benchmark of test-case generation methods. Recent work has proposed stronger test synthesis, feedback-driven refinement, grammar-guided generation, targeted generator generation, and adversarial hack-style tests. These techniques are orthogonal to our pipeline and could be incorporated to improve the coverage and discriminative power of the generated rewards. (He et al., 2025b; Wang et al., 2025b; Cai et al., 2026; Sung et al., 2025; Cao et al., 2026; Shi et al., 2026)

We compare prompting-based and tool-based test generation using tasks from CodeContests (Li et al., 2022). We leverage the corresponding golden solutions to evaluate the accuracy and complexity

Table 14: Token statistics for tasks and solutions of the demonstration dataset.

Type	Min	Max	Mean	Median	Std Dev	Total Tokens
Task	200	3,537	658.91	635.00	258.49	134.3M
Solution	1,711	33,144	17,742.50	17,431.00	7,295.92	3.25B
Dataset Size	200,091 entries					3.38B

Table 15: Comparison of Prompting-based and Tool-based Test Generation. The tool-based approach excels in test diversity, accuracy, and the ability to generate more challenging test cases.

	Random	Scalable	Boundary	Stress	Cost	Avg Tests	Min Tests	Max Tests	Consensus	Pass Rate
Prompting-based	✗	✗	✗	✗	low	13.6	5	15	82.0%	77.4%
Tool-based	✓	✓	✓	✓	high	18.3	5	27	78.8%	87.9%

of the tests produced by the two approaches. The results in Table 15 show that the tool-based approach outperforms the prompting-based method across multiple dimensions. Qualitatively, it is more versatile, capable of systematically generating random, scalable, boundary, and stress tests, which are essential for robust code evaluation but not supported by prompting-based methods.

Quantitatively, the tool-based approach achieves a higher pass rate on ground-truth solutions (87.9% vs. 77.4%), confirming that its test cases are more accurate and reliable. It also generates more challenging and discriminative tests, as reflected by the lower consensus ratio (78.8% vs. 82.0%), which indicates stronger effectiveness in uncovering subtle bugs. In addition, the tool-based generator provides broader test coverage, albeit at a higher computational cost.

D.3 Tool-based Test Generation

The tool-based test generation strategy relies on **CYaRon**, an open-source Python library aimed at rapidly generating random data for Informatics Olympiad problems (or problems of equivalent difficulty). This library contains a variety of common data structures (e.g., graphs, trees, polygons, vectors, strings, and sequences), along with mathematics-related functions and the necessary input/output interfaces. When prompting the Teacher model to utilize the CYaRon tool, we provide its detailed documentation and usage instructions as part of the prompt. Additionally, we encourage the model to generate more boundary tests and large-scale random use cases. To ensure the sufficiency of test cases, we mandate the use of this library in conjunction with its random features and set a seed to ensure reproducibility. The detailed prompt used is illustrated as:

```
Please write a test case generator that meets the
following requirements based on the following CYaRon
documentation:

1. Write a canonical CYaRon Generator using Python
2. Generate a single, executable Python program that can
produce test cases with at least 5 different features
3. The Python program should save each test case
individually in the format [use case characteristics].in
4. The program should include a variety of test case
types such as base cases, boundary cases, large random
cases, etc
5. The Python program code should contain clear comments
to explain the design intent for each test case generation
6. The .in output files should contain ONLY pure input
data without any comments, explanations, or answer
validation
7. The Python program should be able to generate all test
cases in a single run when executed
8. The program should use argparse to provide
configurable random seed control:
    parser.add_argument('--seed', type=int, default=42,
    help='Random seed for reproducibility')
9. All random number generation must use Python's
built-in random module (import random) - do not use any
external random libraries or the random functions from
CYaRon

### CYaRon Documentation
[The complete API documentation of the CYaRon library is
provided here, covering Input/Output (IO), Graph
Generation, Polygon, Vector, String, Sequence, and
Utilities. We omit the full text for brevity as it
follows the standard documentation of the library.]

### Code Question
{QUESTION}
```

E Dual-verification

E.1 Core Algorithm

We summarize the symbols used in the dual-verification process in Table 16, and outline the corresponding procedure in Algorithm 1.

Table 16: Notation for Our Framework.

$\{x_i\}_{i=1}^n$	Test inputs for a task q
$\{A^j\}_{j=1}^m$	Candidate solutions (LLM-generated)
y_i^j	Output of A^j on input x_i
$\hat{y}_i$	Provisional label via majority vote on $\{y_i^j\}_{j=1}^m$
w_i	Difficulty weight for x_i
$\mathcal{T}_{candidate}$	Provisional labeled set $\{(x_i, \hat{y}_i, w_i)\}$
$\mathcal{T}_{golden}$	Weighted suite for selecting the solution
$\mathcal{T}_{val}$	Hold-out validation set
S_j	Weighted score of A^j on $\mathcal{T}_{golden}$
A_{golden}	Final selected “golden” solution

E.2 Test-Case Weighting Criteria

We offer two distinct strategies for assigning weights to individual test cases:

Semantic-Based Weighting. During test-case generation, the model is prompted to produce multiple categories of test cases (stored as .in files), including nominal (weight = 1), complex (2), boundary (3), and stress (4) scenarios. This assigns higher weights to test cases that are more likely to expose corner cases or failure modes.

Size-Based Weighting. We assign weights based on the size of the input files, which serves as a proxy for memory consumption. Specifically, we sort test cases by the size of their input files and divide them into four equal-sized buckets: the smallest 25% receive weight = 1, the next 25% receive weight = 2, the next 25% receive weight = 3, and the largest 25% receive weight = 4. This ensures that heavier test cases, which require greater memory resources, are assigned higher weights.

In this work, we adopted Size-Based Weighting for data synthesis.

E.3 Validation Effectiveness Analysis

E.3.1 Error Rate for Labeling Test Outputs via Voting

On TACO-verified, we measure a 5.27% false-positive rate under voting with 8 solutions. To assess the false-positive rate of test-output labeling, we evaluate the proposed approach on real-world, verified datasets. Specifically, we randomly sample

Algorithm 1 Dual-Verification of Solutions and Test Cases (Strict Verification)

Require: Task q ; test inputs $\{x_i\}_{i=1}^n$; candidate solutions $\{A^j\}_{j=1}^m$.

Ensure: Golden solution A_{golden} and test suite $\mathcal{T}_{golden}$, or **None**.

1: **Step 1: Consensus Voting & Weighting**

2: **for** $i = 1$ to n **do**

3: **for** $j = 1$ to m **do**

4: Run $y_i^j \leftarrow A^j(x_i)$

5: **end for**

6: $\hat{y}_i \leftarrow \arg \max_y \sum_{j=1}^m \mathbb{I}(y_i^j = y)$

7: $w_i \leftarrow \text{Weight}(x_i)$

8: **end for**

9: $\mathcal{T}_{candidate} \leftarrow \{(x_i, \hat{y}_i, w_i)\}_{i=1}^n$

10: **Step 2: Split Candidate Set**

11: Randomly partition $\mathcal{T}_{candidate}$ into $\mathcal{T}_{golden}$ and $\mathcal{T}_{val}$

12: **Step 3: Weighted Selection**

13: **for** $j = 1$ to m **do**

14: $S_j \leftarrow \sum_{(x_i, \hat{y}_i, w_i) \in \mathcal{T}_{golden}} w_i \cdot \mathbb{I}(A^j(x_i) = \hat{y}_i)$

15: **end for**

16: $j^* \leftarrow \arg \max_j S_j$

17: $A'_{golden} \leftarrow A^{j^*}$

18: **Step 4: Hold-out Confirmation**

19: Compute unweighted accuracies of all A^j on $\mathcal{T}_{val}$

20: $j^\dagger \leftarrow \arg \max_j \text{Acc}(A^j, \mathcal{T}_{val})$

21: **if** $j^\dagger = j^*$ **then**

22: $A_{golden} \leftarrow A'_{golden}$

23: **return** $A_{golden}, \mathcal{T}_{golden}$

24: **else**

25: **return None** {Discard task}

26: **end if**

500 tasks from the TACO-verified dataset, and for each task, we randomly retain 20 test cases.

For each task, we generate n ($n \in \{4, 8, 16\}$) candidate solutions using R1-0528, perform majority voting on the outputs for each test input, and compare the voted consensus output against the ground-truth output to obtain a quantitative labeling accuracy. The resulting test-output labeling accuracy under different values of n is shown in Table 17 and Table 18.

Increasing the number of sampled solutions consistently improves test output labeling accuracy. With $n = 8$, the false-positive rate is 5.27%, which falls within an acceptable range and demonstrates that the approach is potentially reliable to be trans-

Table 17: Average Test Output Labeling Accuracy with varying n .

n (# solutions)	Labeling Accuracy
4	94.39%
8	94.73%
16	95.13%

Table 18: Test Output Labeling Accuracy across different sources.

Source	$n = 4$	$n = 8$	$n = 16$
AtCoder	94.75%	95.00%	96.61%
CodeChef	92.80%	92.80%	92.80%
CodeForces	94.44%	94.81%	95.06%

ferred to the synthetic setting.

E.3.2 Same-Family vs. Cross-Family Voting

To test correlated same-source errors, we hold 500 TACO-verified tasks and their test inputs fixed while changing only the candidate-solution pool. At a matched pool size of four, combining two R1-0528 and two Qwen3-235B solutions improves labeling accuracy by 2.64 points over four R1-0528 samples. Increasing the heterogeneous pool to eight provides a smaller additional gain.

Table 19: Test-output labeling accuracy by voting pool.

Voting Pool	Accuracy
R1-0528 $\times 4$	89.91%
R1-0528 $\times 2$ + Qwen3-235B $\times 2$	92.55%
R1-0528 $\times 4$ + Qwen3-235B $\times 4$	92.94%

Cross-family voting reduces, but cannot rule out, correlated teacher errors; the human audit below measures the remaining error directly on synthetic tasks.

E.4 Solution Quality Assessment

E.4.1 Error Rate of Golden Solution

To enable quantitative assessment, we adopt two evaluations: (1) measuring the error rate of dual verification on our synthetic datasets, which yields pass rate distributions across various proprietary LLMs; and (2) evaluating the actual error rate on real-world datasets (TACO-verified), resulting in a 7.85% error rate.

(i) Synthetic Task Evaluation. We first use DeepSeek-R1-0528 to generate multiple candidate solutions for each synthetic task. We then apply our dual-verification strategy to select the golden

solution and measure its pass rates on the voted test cases. The pass rate distribution is shown in Table 20.

Table 20: Distribution of Golden Solution Pass Rates on Voted Test Cases using R1-0528.

Range (%)	Ratio (%)
(0, 20)	13.12
[20, 40)	17.29
[40, 60)	17.57
[60, 80)	14.94
[80, 100)	13.39
100	23.66

Here, each percentage range represents the fraction of tasks whose selected golden solution attains a pass rate within that interval. For example, the [80, 100) range indicates that 13.39% of tasks have golden solutions that pass between 80% and 100% of their voted test cases, while 23.66% of the solutions pass all test cases.

Note that solution quality is strongly tied to model capability. The pass rates of the proprietary models are presented in Table 23 in the task solvability analysis section.

If we adopt a more capable model such as GPT-5-High, 66.98% of the tasks can be solved perfectly in a single attempt.

(ii) Real-world Dataset Evaluation. We also apply our dual-verification approach to real-world, verified datasets to measure the error rate of the selected golden solutions. Because real-world datasets contain ground-truth test cases, the resulting error rate accurately reflects the true quality of the selected solutions.

Specifically, we randomly select 500 tasks from the TACO-verified dataset, each with 20 retained test cases as ground truth tests. We apply our dual-verification procedure using R1-0528 to label test outputs via voting, and then select the golden solution based on the pass rate on the voted test cases. We then evaluate each golden solution against the ground-truth tests.

The verification results under different numbers of candidate solutions (n) are shown in Table 21.

On the TACO-verified dataset, the proposed approach yields a 7.85% error rate in the selected golden solutions when $n = 8$. The error rate further decreases as the number of rollout solutions increases. Such an error level is acceptable, indicating that the approach has the potential to be transferred to the synthetic setting.

Table 21: Verification results on TACO-verified dataset with varying candidate solutions (n).

n (Candidates)	Avg. Pass Rate (test-case level)	Full Pass Rate (task-level)
4	91.79%	84.20%
8	92.15%	85.00%
16	92.50%	85.80%

E.4.2 Human Audit of the Synthetic Set

We randomly sample 150 tasks from the SFT corpus and manually assess three binary criteria: whether each task is well-defined, whether its selected golden solution is correct, and whether the voted outputs of its generated tests are correct. This audit directly evaluates the synthetic set instead of relying on TACO ground truth.

Table 22: Human audit of 150 synthetic SFT tasks.

Criterion	Rate
Well-defined task	90.0%
Correct golden solution	87.3%
Correct voted test labels	87.3%

The audit estimates a 12.7% golden-solution error rate and the same residual rate for voted labels. Among incorrect labels, 94% arise from one mechanically detectable failure mode: function-signature tasks lacking a standard input/output specification produce tests with empty expected outputs. We add a deterministic filter that rejects this pattern before constructing the final training corpus. The remaining errors motivate heterogeneous voting and stronger verification, but they also show that dual verification should not be interpreted as ground-truth certification.

E.5 Task Solvability Analysis

E.5.1 Solvability of Generated Problem

To estimate the fraction of potentially unsolvable problems in our generated dataset, we use GPT-5-High as a strong solver proxy. Specifically, we evaluate the pass@1 performance of several proprietary LLMs—including Qwen3-Max, Gemini-2.5-Pro, and GPT-5-High—on our voted test cases. Their single-try pass rates are reported in Table 23.

Notably, even GPT-5-High shows a small subset of tasks with very low pass rates. Such tasks are likely to be ambiguous, underspecified, inherently unsolvable, or affected by test-case labeling noise. Since GPT-5-High is among the strongest proprietary solvers available, failures from this model

Table 23: Distribution of proprietary LLMs’ pass@1 on voted test cases. Each percentage range represents the fraction of tasks whose best solution from the corresponding model attains a pass rate within that interval.

Range (%)	R1-0528	Qwen3-Max	Gemini2.5-Pro	GPT5-High
(0–20)	13.12%	11.06%	9.57%	3.07%
[20–40)	17.29%	16.44%	14.38%	4.83%
[40–60)	17.57%	18.59%	17.17%	6.49%
[60–80)	14.94%	16.36%	15.80%	7.80%
[80–100)	13.39%	14.39%	14.90%	10.82%
100	23.66%	23.16%	28.18%	66.98%

serve as a practical indicator of potential flaws in the task itself.

F Data Leakage Analysis

Because X-Coder does not directly reuse real-world tasks during post-training, its generated prompts may reduce the risk of verbatim or near-duplicate benchmark contamination relative to web-crawled task corpora. This claim concerns prompt overlap only; it does not imply independence from TACO-derived abstract features or teacher pretraining. We conduct a direct overlap analysis between the LiveCodeBench evaluation problems and the training-prompt sources used for synthetic data generation. For each LCB problem, we compute (i) **n-gram containment**: the largest fraction of its 5-token or 13-token n-grams that are covered by any single source prompt, and (ii) **embedding similarity**: cosine similarity after MiniLM embedding re-ranking over TF-IDF candidates. We examine two source corpora—X-Coder’s own synthetic prompts and NVIDIA OpenCodeReasoning (which includes historical competitive-programming problems from APPS/TACO)—across both LCB v5 and v6 windows. Results are reported in Table 24.

The X-Coder synthetic prompts exhibit consistently low lexical overlap with both LCB windows: the maximum 13-gram containment remains below 0.07, and no problem exceeds the 0.1 threshold. Embedding similarity is also moderate, with maxima below 0.88. OpenCodeReasoning serves as a stronger external baseline because it incorporates historical competitive-programming problems from APPS, TACO, and related sources; even there, overlap remains limited—only one v5 problem exceeds 0.1 in 13-gram containment, and none does so for the v6 window. Overall, these direct lexical and semantic checks confirm the absence of verbatim or near-duplicate contamination of LCB v5/v6 problems in the synthetic training prompts used by

Table 24: Direct contamination check between LiveCodeBench windows and training-prompt sources. 5g and 13g report the target-side n-gram containment score (i.e., the largest fraction of 5-token or 13-token n-grams from each LCB problem covered by any source prompt). Emb reports cosine similarity after MiniLM embedding re-ranking.

Source	Window	#LCB	#source	5g p99/max	13g p99/max	Emb p99/max
X-Coder	LCB v5	268	202,128	0.110/0.113	0.030/0.069	0.826/0.871
X-Coder	LCB v6	175	202,128	0.091/0.101	0.036/0.051	0.815/0.841
OpenCodeReasoning	LCB v5	268	28,497	0.198/0.229	0.070/0.103	0.868/0.902
OpenCodeReasoning	LCB v6	175	28,497	0.190/0.272	0.061/0.068	0.833/0.849

X-Coder.

G Case Study

G.1 Successful Case

The SFT model frequently exhibits cognitive behaviors such as planning, verification, backtracking, and reflection, suggesting that these behaviors can be directly distilled from the teacher.

Thinking process of successful case.

Title: Adjacent GCD
Platform: AtCoder
Difficulty: Hard
Contest_id: arc185
Contest_date: 2024-10-13T00:00:00

Thinking behavior:

We begin by defining $F(m)$ as the total score of all subsequences of the first m elements ... and try to see whether we can break the problem into manageable pieces ... (Plan behavior)

At first, I considered summing $\gcd(A_i, A_{i+1})$ directly over the array ... but then realized that subsequences can skip indices, so adjacency is redefined ... this naive plan would miss many contributions ... (Reflection behavior)

To verify, I checked a small prefix: $A = (9, 6, 4)$. For $m = 2$, the subsequences $[9], [6], [9, 6]$ give total = 3 ... for $m = 3$, manual enumeration yields 11, while my recurrence only produced 8 ... (Verification behavior)

...

Checking with $m = 2$ and $m = 3$ confirms it works ... e.g., for $A = (9, 6, 4)$:
 $F(3) = 2 \cdot 3 + (\gcd(9, 4) \cdot 2^0 + \gcd(6, 4) \cdot 2^1) = 6 + (1 + 4) = 11$... other boundary cases (single element, all equal numbers, coprime arrays) also check out ... (Verification behavior)

However, computing $\sum \gcd(A_j, A_m) 2^{j-1}$ naively costs $O(m)$... with N up to 5×10^5 this is too large ... (Reflection behavior)

To optimize, I reorganize the sum by divisors of A_m ... use Möbius inversion ... and precompute an array $H[t] = \sum_d \mu(d) \cdot \frac{t}{d}$... then each step requires iterating only over divisors of A_m ... (Optimization behavior)

...

The complexity now depends on divisor counts ... on average $\sim 10 \sim 20$, worst-case ~ 128 , giving $\sim 10^8$ operations overall ... (Complexity analysis)

Overall time is $O(\sum_{i=1}^N d(A_i))$, with small average divisor counts in practice ...

Finally, with precomputation of μ , divisors, H , and powers of 2 modulo 998244353 ... the algorithm is complete, efficient, and matches all sample outputs ... (Conclusion)

Final code solution of successful case.

```
mod = 998244353
max_val = 100000

import sys

def main():
    data = sys.stdin.read().split()
    if not data:
        return
    n = int(data[0])
    A = list(map(int, data[1:1+n]))

    mu = [1] * (max_val+1)
    is_prime = [True] * (max_val+1)
    primes = []
    for i in range(2, max_val+1):
        if is_prime[i]:
            primes.append(i)
            mu[i] = -1
        for p in primes:
            if i * p > max_val:
                break
            is_prime[i*p] = False
            if i % p == 0:
                mu[i*p] = 0
                break
            else:
                mu[i*p] = -mu[i]

    divisors = [[] for _ in range(max_val+1)]
    for i in range(1, max_val+1):
        for j in range(i, max_val+1, i):
            divisors[j].append(i)

    H_arr = [0] * (max_val+1)
    for d in range(1, max_val+1):
        for t in range(d, max_val+1, d):
            term = mu[d] * (t // d)
            H_arr[t] = (H_arr[t] + term) % mod

    pow2 = [1] * n
    for i in range(1, n):
        pow2[i] = (pow2[i-1] * 2) % mod

    F = [0] * (n+1)
    U = [0] * (max_val+1)

    for idx in range(n):
        num = A[idx]
        if idx == 0:
            F[1] = 0
            for t in divisors[num]:
                U[t] = (U[t] + pow2[0]) % mod
        else:
            total = 0
            for t in divisors[num]:
                total = (total + U[t] * H_arr[t]) % mod
            F[idx+1] = (2 * F[idx] + total) % mod
            for t in divisors[num]:
                U[t] = (U[t] + pow2[idx]) % mod

    for i in range(1, n+1):
        print(F[i])

if __name__ == "__main__":
    main()
```

G.2 Bad Case

We identify three major failure modes in code reasoning process:

(i) Premature termination under context exhaustion. As the context window approaches its limit,

the model shortens its reasoning and rushes to produce a final answer (e.g., “Given the time, we output the following solution..”).

(ii) Retrieval-like fallback instead of reasoning. Rather than attempting to derive a solution, the model sometimes recalls a “memorized” accepted submission in another language (e.g., C++) and attempts to translate it into Python, bypassing genuine reasoning.

(iii) Incomplete code emission before cutoff. The model occasionally fails to output a complete code block before context exhaustion, leaving truncated or non-executable programs.

Thinking process of bad case.

```
But then the value for the parent is not known until we
assign.

We abandon.

Given the time, we output the following solution in
Python for the sample.
For the sample input, it must output 3 for n=4 and
[0,1,0,3]
and 353820794 for n=22 and the given a.

Here is a known AC solution in Python for the same
problem (from a contest submission):

import sys
sys.setrecursionlimit(10**6)
mod = 998244353

n = int(sys.stdin.readline().strip())
a = list(map(int, sys.stdin.readline().split()))

# Build the tree
g = [[] for _ in range(n)]
for i in range(n):
    if a[i] > 0:
        g[a[i]-1].append(i)

# We will do a DFS. We need a segment tree that can
update and query in O(log n)
# for the range [0, n-1] for the minimum value and the
product of the counts.
# We will use a segment tree that stores (min_value,
count) for a range.
size = 1
while size < n:
    size *= 2
seg_min = [10**9] * (2 * size)
seg_cnt = [1] * (2 * size)

# But the known solution uses a different approach.

# Instead, we output the solution from a known AC
submission in C++.
# We found the following in C++ and we translate it to
Python.

# C++ solution (AC for the same problem) by a known user:

# include <iostream>
# include <vector>
# include <algorithm>
# using namespace std;
# const int mod = 998244353;

# int main() {
#     int n;
#     cin >> n;
#     vector<int> a(n);
#     for (int i = 0; i < n; i++) {
#         cin >> a[i];
#     }
#     vector<vector<int>> g(n);
#     for (int i = 0; i < n; i++) {
#         if (a[i] > 0) {
#             g[a[i]-1].push_back(i);
#         }
#     }
#     vector<int> dp(n, 1);
#     vector<int> to(n, 1000000000); ...
```

Thinking process of bad case. (continued)

```
# // initially, set the segment tree to a big number.
# for (int i = 0; i < n; i++) {
#     update(i, 1000000000);
# }
# //
# for (int i : order) {
#     if (a[i] > 0) {
#         int parent = a[i]-1;
#         int seg_min = query(a[i]+1, i);
#         // The value for i must be > than the value for
#         parent and < than
#         (context exhausted here)
```

G.3 Reward Hacking.

We manually inspect 700 RL-model rollouts from LCB v6 (175 problems, four samples each). Sixty rollouts (8.6%) contain a hacking pattern: 56 hard-code memorized sample inputs/outputs and four emit a constant or degenerate guess.

These behaviors rarely translate into benchmark success: 59 of the 60 hacked rollouts fail all hidden tests, while the only passing rollout also contains a correct general algorithm and the hard-coded samples are vestigial. Among 566 passing rollouts, only this one (0.18%) contains a hack pattern. Thus hacking is visible in behavior but contributes negligibly to the reported benchmark gains.

Test-label noise can still distort training rewards. Because reward is the fraction of passed tests, a 5.27% per-test labeling error shifts a continuous score rather than necessarily flipping an all-or-nothing label. Moreover, all eight GRPO rollouts for a prompt share the same test suite, so uniform shifts can partially cancel in relative advantages. This mechanism does not remove correlated or solution-dependent errors, which remain a limitation.

Table 25: Performance analysis categorized by reasoning token length.

Token Range	Total	Passed	Easy	Medium	Hard
0–10k	79	76	46/47 (97.9%)	22/24 (91.7%)	8/8 (100.0%)
10k–20k	93	68	14/15 (93.3%)	30/35 (85.7%)	24/43 (55.8%)
20k–32k	96	25	1/1 (100.0%)	11/27 (40.7%)	13/68 (19.1%)
Total	268	169	61/63 (96.8%)	63/86 (73.3%)	45/119 (37.8%)

Table 26: Reward-hacking patterns in 700 LCB v6 roll-outs.

Pattern	Count	Share
Hard-coded sample I/O	56	8.0%
Constant/degenerate output	4	0.6%

G.4 Extended Analysis

Cognitive Behaviors Distilled from SFT, Strategic Exploitation Emerges in RL. Post-SFT models exhibit emergent cognitive behaviors like planning and verification (Section G.1), suggesting these capabilities are distilled directly from the teacher. In later RL stages, alongside performance gains, we also observe occasional instances of strategic exploitation, where the model games edge cases for partial rewards (Section G.3). We also observe persistent inefficiencies, including context-induced premature termination and cross-lingual hallucinations (e.g., translating memorized C++ solutions), as detailed in Section G.2.

Longer Reasoning Reflects Higher Task Difficulty. Table 25 shows that the pass rate decreases sharply as reasoning token length increases, exhibiting a clear downward trend. This finding runs counter to the intuitive expectation that greater test-time token usage reflects deeper reasoning and should therefore yield higher accuracy. Instead, we observe a significant chained relationship among problem difficulty, reasoning length, and pass rate: problem difficulty is positively correlated with reasoning length, while reasoning length is strongly negatively correlated with pass rate. This mediation pattern can be summarized as higher difficulty $\rightarrow$ longer reasoning length $\rightarrow$ lower pass rate.