CODESCALER: Scaling Code LLM Training and Test-Time Inference via Reward Models

Xiao Zhu[1,2*] **Xinyu Zhou**[1,2*] **Boyu Zhu**[2,4] **Hanxu Hu**[5]
Mingzhe Du[6] **Haotian Zhang**[2] **Huiming Wang**[2†] **Zhijiang Guo**[1,3†]
[1]LARK, HKUST(GZ) [2]Kuaishou Technology [3]HKUST
[4]UCL [5]UZH [6]NUS

Abstract

Reinforcement Learning from Verifiable Rewards (RLVR) has driven recent progress in code large language models by leveraging execution-based feedback from unit tests, but its scalability is fundamentally constrained by the availability and reliability of high-quality test cases. We propose CODESCALER, a reward model designed to scale both reinforcement learning training and test-time inference for code generation. CODESCALER is trained on carefully curated preference data derived from verified code problems and incorporates syntax-aware code extraction and validity-preserving reward shaping to ensure stable and robust optimization. Across four coding benchmarks, CODESCALER consistently outperforms execution-based RL by **+1.55** points on Qwen3-8B-Base and **+4.23** points on Qwen3-14B-Base. By further scaling to 44K problems with additional synthetic data, CODESCALER yields **+14.64** points improvement over the base model without requiring any test cases. At inference time, CODESCALER serves as an effective test-time scaling method, achieving performance comparable to unit test approaches while providing a **10×** reduction in latency. Moreover, CODESCALER surpasses existing reward models on RM-Bench not only in the code domain (**+3.3** points), but also in general and reasoning domains (**+2.7** points on average). Code and model checkpoints are available at `https://github.com/LARK-AI-Lab/CodeScaler` and `https://huggingface.co/collections/LARK-Lab/codescaler`, respectively.

1 Introduction

As the capability boundaries of Large Language Models (LLMs) continue to expand, their applications in software engineering [21, 50, 17, 58] and algorithmic problem solving [3, 34, 10] have grown rapidly. As a highly structured task with executable feedback, code generation has become a central benchmark for evaluating the reasoning ability of LLMs [23, 55].

The success of reinforcement learning in post-training has accelerated the adoption of Reinforcement Learning from Verifiable Rewards (RLVR) for training code LLMs [10, 12, 24]. By executing model-generated programs on unit tests within a sandbox environment, RLVR provides binary feedback signals that effectively improve functional correctness. However, RLVR critically depends on high-quality problems with carefully curated test cases, typically sourced from programming contests [34]. While such data strictly covers problem constraints, it is costly to obtain and limited in scale. Recent work [57, 32] has explored synthesizing problems and test cases to address this limitation, but these approaches are often computationally expensive and difficult to validate.

An appealing alternative is to replace executable rewards with a learned reward model, which assigns dense scalar scores to generated code without requiring execution [30, 61, 26]. However,

*Equal Contribution.

†Corresponding Authors. Contact: `huiming_wang@mymail.sutd.edu.sg`; `zhijiangguo@hkust-gz.edu.cn`.

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

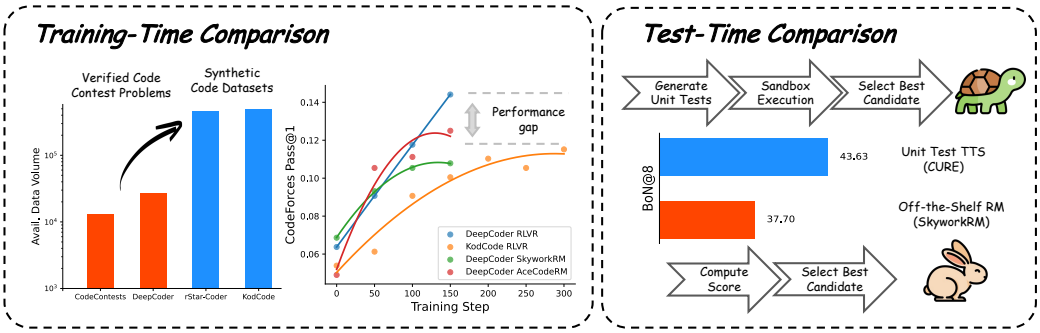


Figure 1: *Left:* Training-Time Comparison shows that despite their larger data scale, synthetic code datasets exhibit a clear performance gap compared to verified code contest problems in RL training. While reward models provide dense supervision, they do not integrate effectively with RL, resulting in weaker performance than RLVR. *Right:* Test-Time Comparison illustrates that Unit Test TTS methods and off-the-shelf reward models demonstrate a clear performance–latency trade-off.

prior attempts to apply reward models to code reinforcement learning have struggled to match the performance of execution-based RLVR, often suffering from training instability and reward hacking [61]. As a result, the potential of code reward models remains largely unexplored.

In this work, we propose CODESCALER, a reward model designed to scale both code LLM training and test-time inference. CODESCALER is trained on high-quality preference data constructed from on-policy RL trajectories, enabling it to reliably distinguish code correctness. We further introduce syntax-aware code extraction and validity-preserving reward shaping to stabilize RM-based RL training. Across four coding benchmarks, CODESCALER consistently outperforms execution-based RL by **+1.55** points on Qwen3-8B-Base and **+4.23** points on Qwen3-14B-Base. By further scaling to 44K problems with additional synthetic data, CODESCALER yields **+14.64** points improvement over the base model without requiring any test cases. At test time, CODESCALER achieves performance comparable to unit test TTS methods while providing a **10**$\times$ speedup, and surpasses prior reward models on RM-Bench across code (**+3.3**), reasoning, and general domains (**+2.7** on average). Our contributions are:

- We construct high-quality preference data from on-policy RL trajectories and train CODESCALER to reliably distinguish code correctness. We further introduce syntax-aware code extraction and validity-preserving reward shaping to stabilize RM-based RL training.
- We demonstrate that CODESCALER serves as an effective reward for both RL training and test-time Best-of-N selection, consistently outperforming RLVR and existing reward models across diverse datasets and model scales.
- We conduct comprehensive ablation studies and analyses that validate each component of CODESCALER and provide insights into reward model design for code.

2 Related Work

Code generation is an inherently verifiable task, and most recent progress relies on RL with execution-based rewards derived from test cases [34, 15, 48, 12, 11]. These test cases are either human-verified or synthetically generated. While verified test cases provide reliable supervision, they are costly and limited in scale. CodeContests [29] contains 13K problems, and TACO [27] provides only 25K training instances. To reduce reliance on execution-based supervision, several works have explored learning reward models from code data [60, 61]. However, most evaluations focus on Best-of-N (BoN) sampling or reward model benchmarks, rather than using reward models as the primary signal for reinforcement learning. Empirically, RM-based RL has not been shown to outperform rule-based RLVR, leaving a clear performance gap between learned rewards and rule-based verification [61]. More related work is included in Appendix E.

3 Motivation

Training-Time Scaling. RLVR leverages high-quality contest problems with curated test suites and achieves strong performance gains. However, such verified datasets are extremely limited. To address this, recent works construct large-scale synthetic datasets (e.g., rStarCoder [32] with 418K problems and KodCode [57] with 447K). However, synthetic problems lack oracle solutions, making it difficult to construct complete and challenging test cases, especially for corner cases. See Appendix Q for a case study. We trained QWEN3-8B-BASE with GRPO on DeepCoder Dataset (24K high-quality verified problems) [34] and KodCode, respectively. Despite using nearly 2× more data for training, the model trained on KodCode fails to outperform that trained on DeepCoder dataset. The DeepCoder-trained model achieves an average performance of 14.41, while the KodCode-trained model reaches only 11.51, resulting in a 2.9 point gap on average (see Figure 1, left).

A natural alternative is to use a pretrained RM to provide dense feedback. However, existing RMs are ineffective in practice. As shown in Figure 1 (left), training with SKYWORK-REWARD-V2-QWEN3-8B [30][3] or ACECODERM-7B [61][4] still underperforms standard RLVR even on high-quality DeepCoder data. In detail, RLVR achieves 14.41 average score, whereas SKYWORKRM and ACECODERM achieve only 10.78 and 12.5, respectively. This suggests that current reward models fail to fully exploit high-quality verified training data when directly applied to RL training.

Test-Time Scaling. Code generation performance can be improved via test-time scaling techniques such as BoN sampling. A common approach is to generate multiple candidate solutions, execute them against synthesized unit tests, and select the candidate with the highest pass rate [51, 25, 6]. While effective, such methods require reliable test generation and sandbox execution, resulting in high latency and computational cost. RM can also perform BoN by assigning scalar scores to candidate solutions, using only a single forward pass per candidate, without requiring code execution.

However, the same limitation of existing RMs also appears at test time. We evaluate `BoN@8` for a fixed policy model on LiveCodeBench. As shown in Figure 1 (right), the unit test TTS method CURE achieves 43.63 `BoN@8` on LiveCodeBench, while SKYWORKRM achieves only 37.7, resulting in a 5.93 point gap despite its computational advantages, which indicates that current RMs lack sufficient discriminative power to reliably rank candidate solutions. Taken together, binary execution feedback does not scale with data quantity, while existing RMs fail to serve as effective substitutes in either reinforcement learning or test-time scaling. These challenges motivate the development of a code-specific reward model that can (1) enable execution-free reinforcement learning at scale, and (2) support efficient and effective test-time scaling without relying on test cases.

4 Method

In this section, we first describe how we train CODESCALER using high-quality code problems, and then explain how CODESCALER is effectively applied to RL training and test-time scaling.

4.1 Reward Model Training

4.1.1 Preliminary

Training Objective. We aim to train a reward model $r_\phi(\cdot)$ that assigns scores to LLM generated code. The model is trained on pairwise preferences with Bradley-Terry loss [5]. Let $\boldsymbol{q}$ denote a code problem, and let $\boldsymbol{c}^+$ and $\boldsymbol{c}^-$ be two candidates code solutions for the same problem, where $\boldsymbol{c}^+$ is preferred over $\boldsymbol{c}^-$. The probability that $\boldsymbol{c}^+$ is preferred over $\boldsymbol{c}^-$ is modeled using the Bradley-Terry loss:

$$P(\boldsymbol{c}^+ \succ \boldsymbol{c}^- | \boldsymbol{q}) = \sigma(r_\phi(\boldsymbol{q}, \boldsymbol{c}^+) - r_\phi(\boldsymbol{q}, \boldsymbol{c}^-)) \tag{1}$$

where $\sigma(\cdot)$ denotes the sigmoid function. Then the reward model is trained by minimizing the negative log-likelihood over the preference dataset $\mathcal{D}$:

$$\mathcal{L}_{\text{RM}} = -\mathbb{E}_{(\boldsymbol{q},\boldsymbol{c}^+,\boldsymbol{c}^-)\sim\mathcal{D}}[\log \sigma(r_\phi(\boldsymbol{q}, \boldsymbol{c}^+) - r_\phi(\boldsymbol{q}, \boldsymbol{c}^-))] \tag{2}$$

[3] https://huggingface.co/Skywork/Skywork-Reward-V2-Qwen3-8B. We denote this as SKYWORKRM throughout the rest of the paper.

[4] https://huggingface.co/TIGER-Lab/AceCodeRM-7B. We denote this as ACECODERM throughout the rest of the paper.

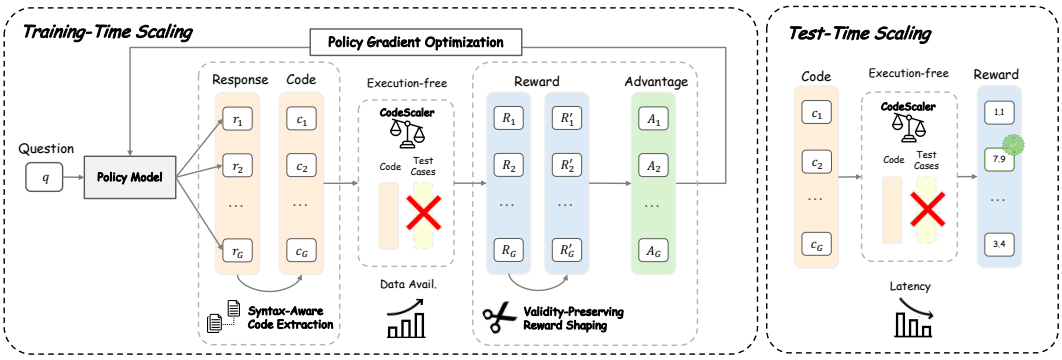


Figure 2: Overall Pipeline of CODESCALER for training-time and test-time scaling, which provides execution-free rewards for policy optimization during RL training, and serves as a lightweight sampler for Best-of-N selection, without relying on test-case execution.

4.1.2 Preference Dataset Curation

Prior work highlights the importance of high-quality preference data, showing that a relatively small but carefully curated dataset can suffice to train strong reward models [59]. As noted in DeepCoder [34], datasets that are overly easy or contain noisy or unverifiable problems can significantly degrade performance. To mitigate this issue, we adopt the DeepCoder dataset as seed data for preference construction, which contains 24K high-quality problems with verified test cases.

Utilizing On-Policy Training Rollouts. Instead of simply prompting multiple open-source LLMs to generate candidate solutions, we collect training data from a policy model trained with RLVR. Specifically, we train QWEN3-8B-BASE on the DeepCoder dataset using GRPO and reuse the resulting on-policy rollouts for reward model construction. As the policy undergoes RL training, its generations naturally span a wide spectrum of solution quality with diversity.

Preference Pair Construction. We treat solutions that pass all test cases as positive samples, while solutions that fail any test case are treated as negative samples. Zeng et al. [61] adopts a threshold-based strategy that labels solutions with a pass rate above 0.8 as correct and enforces a minimum gap between positive and negative samples. In our experiments, we observe that such construction leads to inferior performance in both Best-of-N and RL training (see Appendix I). To improve the robustness of the RM, we augment the preference pairs by adding misaligned pairs $(\boldsymbol{q}, \tilde{\boldsymbol{c}})$ as negative, where $\tilde{\boldsymbol{c}}$ is a solution code generated for a different problem. More training details see Appendix F.

4.2 RL Training with CODESCALER

Most existing code RL pipelines rely on RLVR where reward is provided as binary feedback through direct code execution [12, 34, 24]. In contrast, CODESCALER produces more fine-grained, dense reward signals. However, replacing executable rewards with learned rewards also introduces potential challenges, including training instability and susceptibility to reward hacking, as the reward signal is no longer directly grounded in test-case verification. To mitigate these risks, we adopt several techniques when applying CODESCALER to RL training.

Syntax-Aware Code Extraction. In RLVR, code extraction is straightforward as the execution environment naturally filters flawed or syntactically incorrect code. However, RM-based training lacks this inherent verification, making it sensitive to the quality of the input. RM risks assigning unpredictably high scores to seemingly plausible but syntactically broken code. To improve the reliability , we implement a strict code extraction pipeline to extract codes $\boldsymbol{c}$ from model responses $\boldsymbol{r}$:

1. We extract code only from a single, well-defined code block. Responses containing multiple fragmented code blocks are discarded. Such cases occur primarily during the early stages of RL training, when the policy occasionally attempts to produce partial or incomplete code segments.

2. We perform a static Abstract Syntax Tree (AST) check on extracted code. This ensures that RM only evaluates code that is at least syntactically correct.

Validity-Preserving Reward Shaping. The raw output of a Bradley-Terry reward model, $r_\phi(\cdot) \in \mathbb{R}$, does not guarantee a global minimum for invalid inputs. To enforce a clear penalty for failed

Table 1: Evaluation results (Avg@8) across four code benchmarks, comparing CODESCALER with RLVR on DeepCoder, KodCode, and rStarCoder datasets. **Bold** indicates the best results. ↑ (↓) indicates the improvement (degradation) compared to the RLVR baseline.

MODELS	BENCHMARKS (Avg@8) LiveCodeBench	CodeContests	MBPP	CodeForces	AVG.
DEEPCODER					
QWEN3-8B-BASE	13.75	19.03	61.70	5.35	24.96
w/ RLVR	23.60	32.00	**76.01**	16.43	37.01
w/ CODESCALER *(Ours)*	**24.80**(1.20↑)	**33.94**(1.94↑)	75.90(0.11↓)	**19.61**(3.27↑)	**38.56**(1.55↑)
QWEN3-14B-BASE	21.37	26.25	71.09	8.45	31.79
w/ RLVR	25.94	34.30	75.45	20.79	39.12
w/ CODESCALER *(Ours)*	**27.55**(1.61↑)	**39.33**(5.03↑)	**81.61**(6.16↑)	**24.89**(4.10↑)	**43.35**(4.23↑)
KODCODE					
QWEN3-8B-BASE	13.75	19.03	61.70	5.35	24.96
w/ RLVR	21.40	27.87	73.24	11.64	33.54
w/ CODESCALER *(Ours)*	**23.11**(1.71↑)	**29.34**(1.47↑)	**76.41**(3.17↑)	**12.17**(0.53↑)	**35.26**(1.72↑)
QWEN3-14B-BASE	21.37	26.25	71.09	8.45	31.79
w/ RLVR	26.12	32.32	78.39	14.10	37.73
w/ CODESCALER *(Ours)*	**27.78**(1.66↑)	**32.89**(0.57↑)	**81.90**(3.51↑)	**15.17**(1.07↑)	**39.44**(1.71↑)
RSTARCODER					
QWEN3-8B-BASE	13.75	19.03	61.70	5.35	24.96
w/ RLVR	23.30	29.44	**75.79**	13.19	35.43
w/ CODESCALER *(Ours)*	**24.50**(1.20↑)	**30.75**(1.31↑)	73.07(2.72↓)	**15.95**(2.76↑)	**36.07**(0.64↑)
QWEN3-14B-BASE	21.37	26.25	71.09	8.45	31.79
w/ RLVR	27.24	**33.26**	77.54	15.65	38.42
w/ CODESCALER *(Ours)*	**27.50**(0.26↑)	32.63(0.63↓)	**82.29**(4.75↑)	**19.96**(4.31↑)	**40.60**(2.18↑)

extractions, we define the final reward $R'(\boldsymbol{q}, \boldsymbol{c})$ for a given question $\boldsymbol{q}$ and extracted code $\boldsymbol{c}$ as :

$$R'(\boldsymbol{q}, \boldsymbol{c}) = \begin{cases} f\left(r_\phi(\boldsymbol{q}, \boldsymbol{c})\right) & \text{if extraction succeeds;} \\ 0 & \text{otherwise} \end{cases}$$

where $f(z) = \ln(1 + e^z)$ is a transformation function that maps RM outputs to a strictly positive range, this ensures that $R'(\boldsymbol{q}, \boldsymbol{c}_{invalid}) < R'(\boldsymbol{q}, \boldsymbol{c}_{valid})$, thereby providing a consistent optimization target that discourages the generation of unparseable or flawed code.

4.3 Test-Time Scaling with CODESCALER

We can perform Best-of-N sampling with CODESCALER. Given a code problem $\boldsymbol{q}$ and a list of candidate code solutions $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_n$, we assign scores $r_\phi(\boldsymbol{c}_1), r_\phi(\boldsymbol{c}_2), \cdots, r_\phi(\boldsymbol{c}_n)$ using trained reward model and select the solution with highest score as the final decision.

5 Experiments

We examine the effectiveness of CODESCALER in two complementary settings. We first show that CODESCALER provides high-quality supervision during RL training, enabling stable and scalable optimization beyond verifiable rewards. We then evaluate CODESCALER at inference time, where it serves as an effective ranker for Best-of-N sampling and enables effective and efficient test-time scaling without relying on test-case execution.

5.1 CODESCALER for RL Training

Setup We use GRPO [10] as our RL algorithm and implement it with VeRL[5] [47]. We use QWEN3-8B-BASE and QWEN3-14B-BASE as our base models. Please refer to Appendix G for training configuration. We evaluate on four popular code generation benchmarks: LiveCodeBench [23], CodeContests [29], MBPP [4] and CodeForces [40]. We report Avg@8, the average execution correctness over 8 independent generated solutions per problem, and set temperature to 0.6 during evaluation across all experiments. Please refer to Appendix H for dataset details.

[5]https://github.com/volcengine/verl

Table 2: Comparison of CODESCALER with other reward models on the DeepCoder dataset (Avg@8). **Bold** indicates the best results, Underline denotes the second-best results. ↑ (↓) indicates the improvement (degradation) compared to the RLVR baseline.

MODELS	BENCHMARKS (Avg@8) LiveCodeBench	CodeContests	MBPP	CodeForces	AVG.
		DEEPCODER			
QWEN3-8B-BASE	13.75	19.03	61.70	5.35	24.96
w/ RLVR	23.60	32.00	**76.01**	16.43	37.01
w/ SKYWORKRM	18.50	23.22	67.59	8.00	29.33 (7.68↓)
w/ ACECODERM	22.75	28.34	71.94	9.74	33.19 (3.82↓)
w/ CODESCALER *(Ours)*	**24.80** (1.20↑)	**33.94** (1.94↑)	75.90 (0.11↓)	**19.61** (3.27↑)	**38.56** (1.55↑)
QWEN3-14B-BASE	21.37	26.25	71.09	8.45	31.79
w/ RLVR	25.94	34.30	75.45	20.79	39.12
w/ SKYWORKRM	24.05	31.64	74.26	10.35	35.08 (4.04↓)
w/ ACECODERM	25.98	32.11	81.22	13.35	38.17 (0.95↓)
w/ CODESCALER *(Ours)*	**27.55** (1.61↑)	**39.33** (5.03↑)	**81.61** (6.16↑)	**24.89** (4.10↑)	**43.35** (4.23↑)

5.1.1 High-Quality Supervision as Reward Signal

We evaluate the effectiveness of CODESCALER as a reward signal for RL training across three representative datasets: (1) DeepCoder [34], which consists of 24K high-quality problems with verified test cases, representing an ideal scenario for RLVR; (2) KodCode [57] and (3) rStarCoder [32], both of which are large-scale synthetic datasets where test cases are generated and verified automatically (see Appendix G for details).

Baselines. We compare CODESCALER against RLVR, where rewards are derived from binary execution feedback on test cases. On the DeepCoder dataset, we additionally compare with two existing reward models: SKYWORKRM, a general-purpose reward model [30], and ACECODERM, a code-specific reward model [61].

Results. As shown in Table 1, CODESCALER consistently matches or outperforms RLVR across all four datasets. On DeepCoder, where RLVR benefits from high-quality verified test cases, CODESCALER still achieves an average improvement of 1.55↑ for QWEN3-8B-BASE and even 4.23↑ for QWEN3-14B-BASE, with substantial gains on CodeContests (34.3 → 39.33) and CodeForces (20.79 → 24.89). On synthetic datasets, CODESCALER similarly outperforms RLVR, achieving 1.72↑ on KodCode and 0.64↑ on rStarCoder for the 8B model, and 1.71↑ and 2.18↑ respectively for the 14B model. We attribute these consistent gains to the dense nature of the reward signals provided by CODESCALER, which provides a richer learning gradient, allowing the policy to better capture structural nuances and explore a more diverse solution space.

Furthermore, as shown in Table 2, we compare CODESCALER against other reward models on DeepCoder. Both SKYWORKRM and ACECODERM fail to yield competitive gains during RL training, resulting in substantial degradation compared to RLVR (7.68↓ and 3.82↓ on average for 8B, respectively). In contrast, CODESCALER is the only reward model that surpasses RLVR. We attribute this to two factors: (1) CODESCALER is trained on high-quality preference data derived from verified code problems; and (2) the syntax-aware code extraction and validity-preserving reward shaping introduced in our method effectively stabilize the RL training process. To better understand the contribution of each component, we conduct a series of ablation studies in section 6.

5.1.2 Scaling RL Training without Test Cases

Table 3: Evaluation results (Avg@8) with scaled training on DeepCoder + synthetic data (44K problems). ↑ (↓) indicates the improvement (degradation) compared to the base model.

MODELS	BENCHMARKS (Avg@8) LiveCodeBench	CodeContests	MBPP	CodeForces	AVG.
QWEN3-8B-BASE	13.75	19.03	61.70	5.35	24.96
w/ SKYWORKRM†	17.69	23.48	75.22	7.52	30.98
w/ ACECODERM	18.81	27.56	74.15	9.20	32.43
w/ CODESCALER *(Ours)*	**25.76** (12.01↑)	**33.63** (14.60↑)	**76.81** (15.11↑)	**22.18** (16.83↑)	**39.60** (14.64↑)

† Best checkpoint before training collapse.

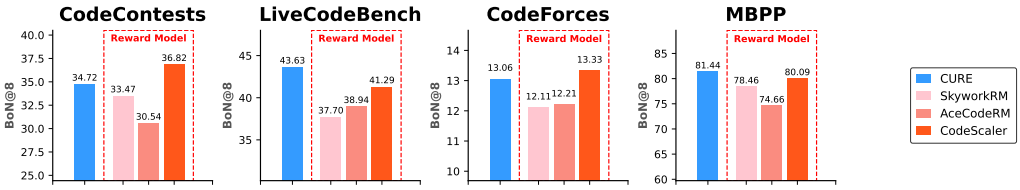


Figure 3: Comparison of Best-of-N (BoN@8) performance across four code generation benchmarks using different test-time scaling methods. CODESCALER consistently outperforms other reward models and achieves performance comparable to CURE.

A fundamental limitation of RLVR is its dependence on test cases, which constrains the scale and diversity of training data. CODESCALER removes this bottleneck entirely, enabling RL training to scale by incorporating large amounts of synthetic problems without execution feedback. Following the pipeline of MathScale [49] and QueST [15], we use TACO as the seed dataset, extract concepts from it, construct a concept graph, and leverage an LLM to generate 20K new problems (see Appendix G for details). We then combine these with the 24K DeepCoder problems to form a 44K mixed training dataset and train QWEN3-8B-BASE from scratch using CODESCALER.

Results. As shown in Table 3, training QWEN3-8B-BASE with CODESCALER on the scaled 44K dataset yields an average improvement of **14.64** points over the base model, with substantial gains across all benchmarks (e.g., LiveCodeBench 13.75 $\rightarrow$ 25.76, CodeForces 5.35 $\rightarrow$ 22.18). This result demonstrates that CODESCALER can effectively leverage synthetic data to achieve continued performance gains through data scaling. We further evaluate SKYWORKRM and ACECODERM as alternative reward models under the same setting. During training, we found that SKYWORKRM suffers from training collapse, with response length exploding and reward scores dropping rapidly, and we report results from the best checkpoint before collapse. ACECODERM avoids collapse but exhibits continuous performance degradation throughout training. In contrast, CODESCALER maintains stable training dynamics and achieves consistent improvements.

5.2 CODESCALER for Test-Time Scaling

5.2.1 Comparison with Other TTS Methods

Baselines. We compare with two types of TTS methods: (1) Unit Test TTS method, represented by CURE [51], which generates unit tests for each problem and selects the candidate solution that passes most of them. (2) Other reward models, where we use SKYWORKRM and ACECODERM as representatives of general-purpose and code-specific RMs.

Evaluation Details. For fair comparisons, we use REASONFLUX-CODER-7B trained with CURE as the policy model, since CURE requires the same model to generate unit tests for BoN sampling. We generate 8 candidate solutions using REASONFLUX-CODER-7B for each problem on four benchmarks. For CURE, we generate 8 unit tests per problem. We report BoN@8 across all benchmarks and set temperature to 0.6 during evaluation. As in Figure 3, CODESCALER substantially outperforms alternative reward models and achieves performance comparable to CURE, attains a favorable trade-off between effectiveness and efficiency. We define latency as the end-to-end wall-clock latency per problem, measured from the moment a prompt is provided to the policy model until the final selection is returned. For unit test TTS methods, the latency can be expressed as:

Table 4: Latency comparison between CURE and CODESCALER.

TIME(S)	CURE	CODESCALER
UNIT TEST GEN.	979.3	N/A
EXECUTION	516.7	N/A
RM COMPUTE	N/A	146.1
AVG(S/QUESTION).	3.20	**0.31**

$$\mathcal{L}_{\text{UNITTEST}} = \underbrace{N \cdot T_{\text{gen}}}_{\text{generating candidates}} + \underbrace{M \cdot T_{\text{test-gen}}}_{\text{generating unit tests}} + \underbrace{N \cdot M \cdot T_{\text{exec}}}_{\text{sandbox execution}}, \tag{3}$$

where N denotes the number of candidate solutions and M denotes the number of generated unit tests.

Table 5: Evaluation results on RM-Bench across diverse reward models. **Bold** indicates the best results. ↑ (↓) indicates the improvement (degradation) compared to the SkyworkRM-8B.

MODELS	RM-BENCH							AVG.
	Chat	Math	Code	Safety	Hard	Normal	Easy	
SKYWORKRM-8B	80.6	75.0	73.6	**96.5**	67.0	85.5	91.8	81.4
ACECODERM-7B	66.7	65.3	66.9	89.9	62.2	74.4	79.9	72.2
ACECODERM-32B	73.7	70.5	72.1	88.0	65.5	78.3	78.3	76.1
CODESCALER-8B *(Ours)*	**83.0**(2.4↑)	**79.9**(4.9↑)	**76.9**(3.3↑)	96.4(0.1↓)	**71.8**(4.8↑)	**87.9**(2.4↑)	**92.5**(0.7↑)	**84.1**(2.7↑)

Reward model TTS assigns scalar scores to candidate solutions without requiring test generation or code execution. The latency of CODESCALER can be expressed as:

$$\mathcal{L}_{\text{CODESCALER}} = \underbrace{N \cdot T_{\text{gen}}}_{\text{generating candidates}} + \underbrace{N \cdot T_{\text{RM}}}_{\text{reward model scoring}}, \tag{4}$$

where T_{RM} denotes the inference time of the reward model for a single candidate.

Since candidate generation is shared across both methods, we omit the $N \cdot T_{\text{gen}}$ term when comparing latency. We evaluate CURE and CODESCALER as representative unit test TTS and reward model TTS methods, respectively. All latency measurements are conducted on the CodeForces (467 questions) using a single NVIDIA A100 GPU with vLLM backbone. We report results under the `BoN@8` setting, where 8 candidate solutions are generated per problem, and for CURE, 8 unit tests are generated and executed for each question. As shown in Table 4, CODESCALER achieves an approximately 10× speedup over CURE. CODESCALER scores candidates using only a single forward pass. This substantially reduces computational overhead and resource consumption, making CODESCALER particularly well suited for real-world deployment.

5.2.2 Cross-Domain Transfer of CODESCALER

To further assess the model, we evaluated CODESCALER on standard reward model benchmarks. Although CODESCALER was not designed as a general purpose reward model, testing it on these benchmarks still provides valuable insights. We used RM-Bench [31] to compare CODESCALER against SKYWORKRM and ACECODERM, as shown in Table 5. Beyond the clear improvement in code tasks (73.6 → 76.9), we found a surprising result: even though CODESCALER was trained only on code, it also improved in general (Chat, 80.6 → 83.0) and reasoning domains (Math, 75.0 → 79.9). Previous research in SFT and RL has shown that including math or code data can boost a model's general abilities [41, 37, 28, 16]. Our results indicate that this also applies to reward model training. Learning to judge code sharpens the model's overall judgment even outside of the code domain.

5.2.3 Multi-Lingual Transfer of CODESCALER

We evaluate CODESCALER on HumanEval-X [62], which spans five programming languages: Python, C++, Java, JavaScript, and Go. As shown in Table 6, CODESCALER consistently matches or outperforms SKYWORKRM across all languages. We attribute this generalization to the nature of the reward signal learned by our model. Rather than relying on language-specific syntax, CODESCALER implicitly models functional correctness, capturing properties such as correct handling of edge cases and logically sound reasoning. These properties are inherently language-agnostic, as a correct algorithmic solution in Python shares the same underlying structure as its counterpart in C++ or Java.

Table 6: Multi-lingual `BoN@8` evaluation on HumanEval-X using QWEN3-8B as the policy model. ↑ (↓) indicates the improvement (degradation) compared to the Pass@1.

Language	Pass@1	SkyworkRM BoN@8	CODESCALER BoN@8
Python	30.7	34.1	**34.8**(4.1↑)
C++	28.0	32.9	**33.2**(5.2↑)
Java	26.8	32.9	**34.8**(8.0↑)
JavaScript	24.4	25.6	**25.6**(1.2↑)
Go	21.4	24.4	**25.0**(3.6↑)

6 Analysis

6.1 Reward Quality of CODESCALER

To validate that the reward signals produced by CODESCALER faithfully reflect code correctness, we analyze training trajectory records from QWEN3-8B-BASE on DeepCoder, where ground-truth execution labels are available. We evaluate CODESCALER from three complementary perspectives:

Pairwise Ranking Accuracy. Given a randomly sampled pair of one correct and one incorrect solution, CODESCALER assigns a higher score to the correct one 87.9% of the time, corresponding to an AUROC of 0.879. This indicates that CODESCALER possesses strong discriminative ability to distinguish correct solutions from incorrect ones.

Score Separation. The mean RM score for correct solutions is 10.94, compared to 3.77 for incorrect ones, yielding a Cohen's d of 1.71. This large effect size confirms that CODESCALER produces well-separated score distributions, capturing meaningful quality differences rather than relying on marginal distinctions.

Group-Level Selection Accuracy. Within each rollout group, the candidate receiving the highest RM score is correct approximately 80% of the time, achieving a Top-1 selection accuracy of 79.8%. This directly validates the practical utility of CODESCALER for Best-of-N selection, where the top-ranked candidate is chosen as the final output.

These results demonstrate that CODESCALER provides reliable and well-calibrated reward signals, supporting its dual role as both a training reward for RL optimization and a test-time ranker for Best-of-N sampling.

6.2 Training Dynamics

Figure 4 shows the invalid rate and fragment rate for both QWEN3-8B-BASE and QWEN3-14B-BASE trained with CODESCALER on DeepCoder. Both rates initially increase during early exploration but then decrease steadily. The fragment rate converges to near zero ($\leq$0.5% by step 150), and the invalid rate drops from 30–37% to $\sim$1.3% (14B) and $\sim$7% (8B) by step 250, demonstrating that syntax-aware code extraction and validity-preserving reward shaping suppress these failure modes during training.

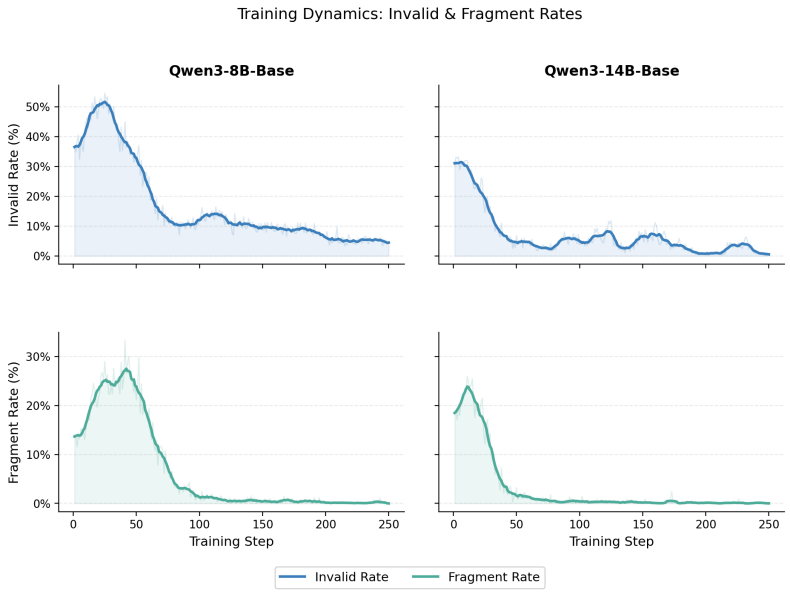


Figure 4: Training dynamics of invalid rate and fragment rate for QWEN3-8B-BASE and QWEN3-14B-BASE trained with CODESCALER on DeepCoder. Both rates decrease steadily after initial exploration.

6.3 Ablation on CODESCALER Components

To enable stable RL training, we make two key contributions. On the training side, we introduce syntax-aware code extraction and validity-preserving reward shaping to stabilize the reward signal. On the data side, we construct high-quality code preference pairs to enhance the reward model's capability. In this section, we conduct ablation to validate the effectiveness of both components. We train QWEN3-8B-BASE on DeepCoder under three settings: (1) the original SKYWORKRM, (2) SKYWORKRM with syntax-aware code extraction and validity-preserving reward shaping, and (3) CODESCALER with syntax-aware code extraction and validity-preserving reward shaping. Results see Figure 5.

We observe a notable improvement when augmenting SKYWORKRM with syntax-aware code extraction and validity-preserving reward shaping. In particular, performance improves by 2.24↑ on LiveCodeBench (18.5 → 20.74), 4.23 ↑ on CodeContests (23.22 → 27.45) and 2.2 ↑ on MBPP (67.59 → 69.79), indicating that the proposed techniques effectively stabilize the reward signal and lead to better RL optimization. Replacing SKYWORKRM with CODESCALER further yields substantial performance gains, with improvements of 4.06↑ on LiveCodeBench (20.74 → 24.80), 6.49 ↑ on CodeContests (27.45 → 33.94), 6.11 ↑ on MBPP (69.79 → 75.90), and 9.61 ↑ on CodeForces (10.00 → 19.61), highlighting the superior reward quality of CODESCALER.

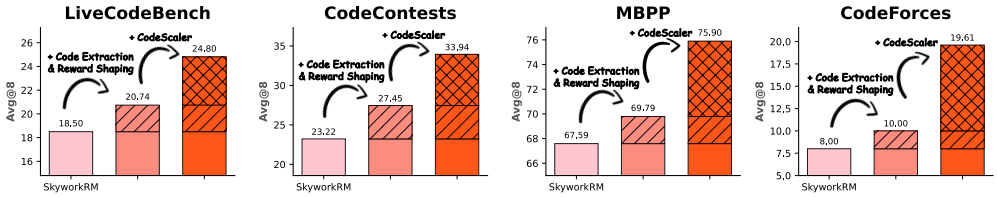


Figure 5: Ablation study on CODESCALER designs. Results compare RL performance using different reward model variants.

7 Conclusion

We introduce CODESCALER, a code reward model designed for both RL training and test-time inference, along with syntax-aware code extraction and validity-preserving reward shaping to stabilize reward signals. Experimental results show that CODESCALER consistently outperforms execution-based rewards during RL training. Scaling to 44K problems with additional synthetic data, CODESCALER yields +14.64 points improvement over the base model without requiring any test cases. At inference time, CODESCALER achieves performance comparable to unit test TTS methods with 10× lower latency. It also surpasses existing reward models on RM-Bench in code and other domains. We hope this work will stimulate broader interest in code reward models.

References

[1] Marah I Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat S. Behl, Lingjiao Chen, Gustavo de Rosa, Suriya Gunasekar, Mojan Javaheripi, Neel Joshi, Piero Kauffmann, Yash Lara, Caio César Teodoro Mendes, Arindam Mitra, Besmira Nushi, Dimitris Papailiopoulos, Olli Saarikivi, Shital Shah, Vaishnavi Shrivastava, Vibhav Vineet, Yue Wu, Safoora Yousefi, and Guoqing Zheng. Phi-4-reasoning technical report. *CoRR*, abs/2504.21318, 2025. doi: 10.48550/ARXIV.2504.21318. URL https://doi.org/10.48550/arXiv.2504.21318.

[2] Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. Opencodeinstruct: A large-scale instruction tuning dataset for code llms, 2025. URL https://arxiv.org/abs/2504.04030.

[3] Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning: Advancing data distillation for competitive coding. *CoRR*, abs/2504.01943, 2025. doi: 10.48550/ARXIV.2504.01943. URL https://doi.org/10.48550/arXiv.2504.01943.

[4] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

[5] Ralph Allan Bradley and Milton E Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

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

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

[8] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2023. URL https://arxiv.org/abs/2211.12588.

[9] Jianbo Dai, Jianqiao Lu, Yunlong Feng, Guangtao Zeng, Rongju Ruan, Ming Cheng, Dong Huang, Haochen Tan, and Zhijiang Guo. Mhpp: Exploring the capabilities and limitations of language models beyond basic code generation. *arXiv preprint arXiv:2405.11430*, 2024.

[10] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948, 2025. doi: 10.48550/ARXIV.2501.12948. URL https://doi.org/10.48550/arXiv.2501.12948.

[11] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *CoRR*, abs/2402.01391, 2024. doi: 10.48550/ARXIV.2402.01391. URL https://doi.org/10.48550/arXiv.2402.01391.

[12] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: grounding code llms in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=PzSG5nKe1q.

[13] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

[14] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

[15] Hanxu Hu, Xingxing Zhang, Jannis Vamvas, Rico Sennrich, and Furu Wei. Quest: Incentivizing llms to generate difficult problems. *CoRR*, abs/2510.17715, 2025. doi: 10.48550/ARXIV.2510.17715. URL https://doi.org/10.48550/arXiv.2510.17715.

[16] Maggie Huan, Yuetai Li, Tuney Zheng, Xiaoyu Xu, Seungone Kim, Minxin Du, Radha Poovendran, Graham Neubig, and Xiang Yue. Does math reasoning improve general LLM capabilities? understanding transferability of LLM reasoning. *CoRR*, abs/2507.00432, 2025. doi: 10.48550/ARXIV.2507.00432. URL https://doi.org/10.48550/arXiv.2507.00432.

[17] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *CoRR*, abs/2312.13010, 2023. doi: 10.48550/ARXIV.2312.13010. URL https://doi.org/10.48550/arXiv.2312.13010.

[18] Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M Zhang. Effilearner: Enhancing efficiency of generated code via self-optimization. *Advances in Neural Information Processing Systems*, 37:84482–84522, 2024.

[19] Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M Zhang. Efficoder: Enhancing code generation in large language models through efficiency-aware fine-tuning. *arXiv preprint arXiv:2410.10209*, 2024.

[20] Hugging Face. Open r1: A fully open reproduction of deepseek-r1, January 2025. URL https://github.com/huggingface/open-r1.

[21] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[22] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020. URL https://arxiv.org/abs/1909.09436.

[23] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

[24] Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng, Tian Wang, Yongding Tao, Rongyu Cao, Binhua Li, Zhi Jin, Wenpin Jiao, Fei Huang, Yongbin Li, and Ge Li. Coderl+: Improving code generation via reinforcement with execution semantics alignment. *CoRR*, abs/2510.18471, 2025. doi: 10.48550/ARXIV.2510.18471. URL https://doi.org/10.48550/arXiv.2510.18471.

[25] Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382*, 2025.

[26] Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. Codeprm: Execution feedback-enhanced process reward model for code generation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 8169–8182. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.findings-acl.428/.

[27] Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. TACO: topics in algorithmic code generation dataset. *CoRR*, abs/2312.14852, 2023. doi: 10.48550/ARXIV.2312.14852. URL https://doi.org/10.48550/arXiv.2312.14852.

[28] Yu Li, Zhuoshi Pan, Honglin Lin, Mengyuan Sun, Conghui He, and Lijun Wu. Can one domain help others? a data-centric study on multi-domain reasoning via reinforcement learning. *arXiv preprint arXiv:2507.17512*, 2025.

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

[30] Chris Yuhao Liu, Liang Zeng, Yuzhen Xiao, Jujie He, Jiacai Liu, Chaojie Wang, Rui Yan, Wei Shen, Fuxiang Zhang, Jiacheng Xu, Yang Liu, and Yahui Zhou. Skywork-reward-v2: Scaling preference data curation via human-ai synergy. *CoRR*, abs/2507.01352, 2025. doi: 10.48550/ARXIV.2507.01352. URL https://doi.org/10.48550/arXiv.2507.01352.

[31] Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. Rm-bench: Benchmarking reward models of language models with subtlety and style. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=QEHrmQPBdd.

[32] Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset. *CoRR*, abs/2505.21297, 2025. doi: 10.48550/ARXIV.2505.21297. URL https://doi.org/10.48550/arXiv.2505.21297.

[33] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024. URL https://arxiv.org/abs/2402.19173.

[34] 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. Deepcoder: A fully open-source 14b coder at o3-mini level, 2025. Notion Blog.

[35] Xianzhen Luo, Jinyang Huang, Wenzhen Zheng, Qingfu Zhu, Mingzheng Xu, Yiheng Xu, YuanTao Fan, Libo Qin, and Wanxiang Che. How many code and test cases are enough? evaluating test cases generation from a binary-matrix perspective. *CoRR*, abs/2510.08720, 2025. doi: 10.48550/ARXIV.2510.08720. URL https://doi.org/10.48550/arXiv.2510.08720.

[36] Yichuan Ma, Yunfan Shao, Peiji Li, Demin Song, Qipeng Guo, Linyang Li, Xipeng Qiu, and Kai Chen. Unitcoder: Scalable iterative code synthesis with unit test guidance, 2025. URL https://arxiv.org/abs/2502.11460.

[37] Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. At which training stage does code data help llms reasoning? *arXiv preprint arXiv:2309.16298*, 2023.

[38] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. URL https://arxiv.org/abs/2203.02155.

[39] Arkil Patel, Siva Reddy, Dzmitry Bahdanau, and Pradeep Dasigi. Evaluating in-context learning of libraries for code generation. In Kevin Duh, Helena Gómez-Adorno, and Steven Bethard, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 2908–2926. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.NAACL-LONG.161. URL https://doi.org/10.18653/v1/2024.naacl-long.161.

[40] Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. Codeforces. https://huggingface.co/datasets/open-r1/codeforces, 2025.

[41] Jackson Petty, Sjoerd van Steenkiste, and Tal Linzen. How does code pretraining affect language model task performance? *Trans. Mach. Learn. Res.*, 2025, 2025. URL https://openreview.net/forum?id=pxxmUKKgel.

[42] Yuhao Qing, Boyu Zhu, Mingzhe Du, Zhijiang Guo, Terry Yue Zhuo, Qianru Zhang, Jie Zhang, Heming Cui, Siu Ming Yiu, Dong Huang, et al. Effibench-x: A multi-language benchmark for measuring efficiency of llm-generated code. *Advances in Neural Information Processing Systems*, 38, 2026.

[43] Musfiqur Rahman, SayedHassan Khatoonabadi, and Emad Shihab. A large-scale class-level benchmark dataset for code generation with llms, 2025. URL https://arxiv.org/abs/2504.15564.

[44] Morgane Rivière, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charline Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur, Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagic, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Welty, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozinska, Dustin Herbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshev, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucinska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju-yeong Ji, Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjösund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, and Lilly McNealus. Gemma 2: Improving open language models at a practical size. *CoRR*, abs/2408.00118, 2024. doi: 10.48550/ARXIV.2408.00118. URL https://doi.org/10.48550/arXiv.2408.00118.

[45] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[47] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 1279–1297. ACM, 2025. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

[48] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning. *Trans. Mach. Learn. Res.*, 2023, 2023. URL https://openreview.net/forum?id=0XBuaxqEcG.

[49] Zhengyang Tang, Xingxing Zhang, Benyou Wang, and Furu Wei. Mathscale: Scaling instruction tuning for mathematical reasoning. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=Kjww7ZN47M.

[50] Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

[51] Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving LLM coder and unit tester via reinforcement learning. *CoRR*, abs/2506.03136, 2025. doi: 10.48550/ARXIV.2506.03136. URL https://doi.org/10.48550/arXiv.2506.03136.

[52] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics, 2021. doi: 10.18653/V1/2021.EMNLP-MAIN.685. URL https://doi.org/10.18653/v1/2021.emnlp-main.685.

[53] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922, 2023. doi: 10.48550/ARXIV.2305.07922. URL https://doi.org/10.48550/arXiv.2305.07922.

[54] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models, 2022. URL https://arxiv.org/abs/2206.07682.

[55] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddartha V. Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. Livebench: A challenging, contamination-limited LLM benchmark. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=sKYHBTAxVa.

[56] Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. Large-scale terminal agentic trajectory generation from dockerized environments, 2026. URL https://arxiv.org/abs/2602.01244.

[57] Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 6980–7008. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-ACL.365. URL https://doi.org/10.18653/v1/2025.findings-acl.365.

[58] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

[59] Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. LIMO: less is more for reasoning. *CoRR*, abs/2502.03387, 2025. doi: 10.48550/ARXIV.2502.03387. URL https://doi.org/10.48550/arXiv.2502.03387.

[60] Huimu Yu, Xing Wu, Weidong Yin, Debing Zhang, and Songlin Hu. Codepmp: Scalable preference model pretraining for large language model reasoning. *CoRR*, abs/2410.02229, 2024. doi: 10.48550/ARXIV.2410.02229. URL https://doi.org/10.48550/arXiv.2410.02229.

[61] Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. ACECODER: acing coder RL via automated test-case synthesis. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 12023–12040. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.acl-long.587/.

[62] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining*, pages 5673–5684, 2023.

[63] Jiayi Zhou, Jiaming Ji, Juntao Dai, Dong Li, and Yaodong Yang. Sequence to sequence reward modeling: Improving rlhf by language feedback, 2025. URL https://arxiv.org/abs/2409.00162.

[64] Terry Yue Zhuo, Xiaolong Jin, Hange Liu, Juyong Jiang, Tianyang Liu, Chen Gong, Bhupesh Bishnoi, Vaisakhi Mishra, Marek Suppa, Noah Ziems, Saiteja Utpala, Ming Xu, Guangyu Song, Kaixin Li, Yuhan Cao, Bo Liu, Zheng Liu, Sabina Abdurakhmanova, Wenhao Yu, Mengzhao Jia, Jihan Yao, Kenneth Hamilton, Kumar Shridhar, Minh Chien Vu, Dingmin Wang, Jiawei Liu, Zijian Wang, Qian Liu, Binyuan Hui, Meg Risdal, Ahsen Khaliq, Atin Sood, Zhenchang Xing, Wasi Uddin Ahmad, John Grundy, David Lo, Banghua Zhu, Xiaoning Du, Torsten Scholak, and Leandro von Werra. Bigcodearena: Unveiling more reliable human preferences in code generation via execution, 2025. URL https://arxiv.org/abs/2510.08697.

A Broader Impact

This work introduces a reward model designed to enhance the code generation capabilities of LLMs. By providing dense scalar feedback and mitigating reward hacking, our approach facilitates the development of more reliable and functional software assistants. These advancements have the potential to democratize programming, increase developer productivity, and accelerate innovation in fields relying on complex algorithmic problem-solving. Furthermore, our findings on test-time scaling offer a computationally efficient alternative to traditional unit-test-based verification, potentially reducing the energy consumption and environmental footprint associated with large-scale inference.

Despite these benefits, the deployment of enhanced code LLMs carries inherent security and safety risks. More capable code generation models could inadvertently introduce subtle security vulnerabilities into software or be misused to automate the creation of malicious exploits. Additionally, while our method reduces the reliance on expensive human-curated data, it may inherit or amplify biases present in the synthetic or on-policy training sets. We encourage practitioners to utilize our provided model documentation and licenses to ensure responsible deployment and to maintain human oversight in critical software development pipelines.

B Limitations

While CODESCALER demonstrates strong empirical performance, it shares the inherent challenges of learned reward models in structured domains like code generation. Unlike execution-based verification that provides binary correctness signals via test cases, CODESCALER functions as a probabilistic estimator of code quality. Consequently, while it excels at identifying high-quality candidates, it does not formally guarantee functional correctness for every output. Future research could explore hybrid frameworks that integrate CODESCALER's semantic understanding with the rigorous verification of lightweight execution engines to achieve even more robust self-evolution.

C LLM Usage Declaration

This manuscript uses LLMs strictly for the purpose of language editing and textual polishing to enhance presentation quality. We declare that the novel ideas, methodological framework, experimental execution, and data analysis are the original work of the authors. All content modified by AI tools has been carefully reviewed and validated by the authors to ensure accuracy.

D Usage of Compute Resources

All RL training and evaluations are conducted on a cluster of 8x NVIDIA A100 80G GPUs. Each RL run required approximately 30 hours, corresponding to roughly 240 A100 GPU hours in total. TTS evaluations and latency measurements are conducted on one single NVIDIA A100 GPU.

E Extended Related Work

LLMs for Code Generation. LLMs are recently studied for code generation based on their outstanding capability [34]. Early milestones established that transformer-based LLMs can synthesize functional programs across a wide range of tasks, sparking extensive follow-up research [7, 1, 44]. Subsequent models and open-source efforts have further improved performance through architectural scaling, curated pretraining corpora, and supervised instruction fine-tuning [61, 53, 52, 33, 45, 13]. Beyond supervised fine-tuning, in-context learning enables LLMs to adapt to novel coding tasks or adhere to strict output formats using demonstrations and prompts [54, 8, 39, 18]. Some works are involved with training models with distilled trajectories from advanced models [3, 20, 56]. More recently, RL has been widely adopted to improve LLM's generation ability further [34, 15], where the test cases evaluation is essential to obtain the reliable reward signals [35, 7, 14].

Code Datasets. Research in code generation has relied on a spectrum of datasets ranging from real human-written code corpora to large-scale synthetic data tailored for training and evaluation. Classic benchmarks derived from real software repositories and programming problems (e.g., HumanEval

[7] and MHPP [9] for function synthesis) provide canonical tasks with unit tests grounded in human logic and correctness criteria. In addition to these benchmarks, real code corpora such as GitHub-sourced files and curated collections (e.g., CodeSearchNet [22], EffiBenchX [42]) offer large paired examples of code and documentation for both pretraining and fine-tuning [43]. To address scarcity and improve diversity, recent work has explored synthetic dataset generation using LLMs. Datasets such as OpenCodeInstruct generate millions of instruction–solution pairs with test cases and automatic quality assessments to enable robust supervised fine-tuning [2]. Other synthetic pipelines like UnitCoder leverage model-generated unit tests to guide and validate large synthetic corpora of verifiable programs, demonstrating improvements in both breadth and functional correctness [36]. More recent synthetic benchmarks like KodCode and EffiCoder systematically generate diverse question–solution–test triplets with rigorous verification, enabling both supervised fine-tuning and execution-aware training for modern code LLMs [57, 19].

Reward Models from General to Domain-Specific. Reward models have become a cornerstone of modern alignment for large language models (LLMs), serving as learned proxies that approximate human qualitative judgments and guide reinforcement learning optimization when explicit task rewards are infeasible to define manually. In the general reinforcement learning from human feedback (RLHF) paradigm, a reward model is typically trained on human preference comparisons to assign scalar values to model outputs, thereby capturing human intent and enabling policy models to be optimized via reinforcement methods such as proximal policy optimization (PPO) [38, 46]. More recent research has explored enhancements to reward model training and evaluation, including sequence-to-sequence reward modeling to capture richer language feedback [63]. In the specific domain of code generation, developing effective reward models is particularly challenging due to the need to quantitatively assess functional correctness and implementation quality. Recent work on BigCodeArena highlights the limitations of generic preference-based RMs by collecting extensive human evaluations under execution settings and proposing benchmarks that better reflect real coding preferences and execution outcomes [64].

F RM Training Details

Preference Construction Details. We train QWEN3-8B-BASE on the DeepCoder dataset using GRPO for 250 steps. During training, we set the temperature to 0.6 and top-p to 0.95. For each problem, we rollout 8 candidate solutions and construct preference pairs by treating solutions that pass all test cases as positive and those that fail any test case as negative. We discard rollouts from the first 100 steps, resulting in a total of 52,574 preference pairs.

Data Augmentation. To improve the robustness of our reward model, we augment the preference pairs by adding misaligned pairs $(\boldsymbol{q}, \tilde{\boldsymbol{c}})$, where $\tilde{\boldsymbol{c}}$ is a solution code generated for a randomly different problem. This encourages the reward model to capture semantic alignment between the problem specification and the generated code, to prevent potential hacks that LLMs generate irrelevant solution codes of the corresponding question. In practice, we augment $\sim 30\%$ data samples and combine them with the original curated perference pairs for reward model training.

RM Training Configuration. We use Skywork-Reward-V2-Qwen3-8B as our reward model for training, which achieves state-of-the-art performances on multiple RM benchmarks. We train it on collected preference data for 3 epochs by AdamW, with a learning rate of $1e-6$.

G RL Training Details

G.1 Policy Gradient Optimization

We adopt Group Relative Policy Optimization (GRPO) as our policy gradient optimization method. GRPO estimates advantages using group-normalized rewards. Specifically, for a given question x, the policy generates a group of G responses $\{r_1, r_2, \ldots, r_G\}$. We extract code $\{c_1, c_2, \ldots, c_G\}$ from these responses and assign a transformed reward R'_i to each using the reward model. The GRPO optimization objective is then defined as:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D},\boldsymbol{o}\sim\pi_\theta}\left[\sum_{t=1}^{|\boldsymbol{o}|}\min\left(r_{i,t}(\theta)\cdot A, \tilde{r}_{i,t}(\theta)\cdot A\right)\right] - \beta\cdot\mathcal{D}_{\text{KL}}[\pi_\theta\|\pi_{\text{ref}}] \tag{5}$$

where $\tilde{r}_{i,t}(\theta) = \text{clip}(r_{i,t}(\theta), 1-\varepsilon, 1+\varepsilon)$, and $r_{i,t}(\theta) = \frac{\pi_\theta(\boldsymbol{o}_{i,t}|\boldsymbol{x}, \boldsymbol{o}_{i,<t})}{\pi_{\theta_{\text{old}}}(\boldsymbol{o}_{i,t}|\boldsymbol{x}, \boldsymbol{o}_{i,<t})}$ denotes the importance sampling ratio. $A = \frac{R'_i - \text{mean}(\{R'_1, R'_2, \cdots, R'_G)}{\text{std}(\{R'_1, R'_2, \cdots, R'_G\}}$ means the group-normalized advantage estimate. The KL divergence term prevents the policy from deviating too far from the reference model. Although the KL term has been omitted in some recent approaches, we observe that keeping KL regularization is beneficial for stabilizing reward model based RL training, especially when the quality of synthetic training data is not strictly controlled.

G.2 RL Training Datasets

G.2.1 DeepCoder

We use the DeepCoder training dataset for both reward model training and reinforcement learning. The dataset consists of three sources: (1) verified problems from TACO, (2) verified problems from PrimeIntellect's SYNTHETIC-1 dataset, and (3) LiveCodeBench problems submitted between May 1, 2023 and July 31, 2024.

To ensure high data quality, DeepCoder applies a series of strict filtering criteria. First, each problem is automatically verified using an external official solution, and only problems whose official solutions pass all unit tests are retained. Second, each problem is required to include at least five unit tests. Third, duplicate problems are removed to prevent data leakage. In addition, the dataset is curated to ensure no overlap with the LiveCodeBench test set (08/01/24-02/01/25).

G.2.2 KodCode & rStarCoder

In subsubsection 5.1.1, we use KodCode and rStarCoder as representative examples of synthetic datasets.

KodCode employs a self-verification pipeline to validate automatically generated tests. Specifically, *GPT-4o-0513* is used to generate both candidate solutions and corresponding test cases, and the generated solutions are executed against these tests to verify correctness. In addition, KodCode enforces 100% branch coverage using `pytest-cov`. However, branch coverage serves only as a proxy for test quality and does not explicitly guarantee completeness of the generated test cases. During training, we use dataset from https://huggingface.co/datasets/KodCode/KodCode-V1-SFT-R1 and select only problems labelled with 'online judge' style.

rStarCoder prompts *GPT-4o* to generate two functions for each problem: one for test input generation and another for input validation. It then applies a mutual verification procedure to construct synthetic problems. Specifically, *QWQ-32B* is used to generate 16 long-reasoning candidate solutions per problem, which are executed on at least 50 generated test inputs. If a majority of these candidate solutions produce identical outputs across the entire test set, both the consistent output set and the solutions that generate it are treated as correct. Compared to KodCode, this verification pipeline is more robust. Nevertheless, it still inherits limitations related to test completeness: challenging or adversarial test cases are often unlikely to pass the mutual verification process and are therefore discarded. During training, we use dataset from https://huggingface.co/datasets/microsoft/rStar-Coder and select synthetic_rl_testcase split.

G.2.3 Synthetic Problems

In subsubsection 5.1.2, we follow QueST to generate new code problems, using TACO as the seed dataset. For each problem $\boldsymbol{q} \in \mathbf{Q}_{\text{seed}}$, we prompt *GPT-4o* to extract a set of underlying concepts, including both topics and knowledge points.

Two concepts are considered to form a reasonable combination if they frequently co-occur within the same problem in the seed dataset. We construct a concept graph $\mathcal{G} = (\mathcal{C}, \mathcal{E})$, where nodes represent individual concepts and edge weights encode their co-occurrence strength. The edge weight between two concept nodes u and v is defined as:

$$w(u, v) = \log(\text{freq}(u, v) + \varepsilon),$$

where $\text{freq}(u, v)$ denotes the observed co-occurrence frequency in $\mathbf{Q}_{\text{seed}}$, and ε is a small constant for numerical stability.

Table 7: Ablation study on preference pair construction. Results compare RL training performance using different CODESCALER variant. **Bold** indicates the best results, Underline denotes the second-best results. ↑ (↓) indicates the improvement (degradation) compared to the RLVR (Binary) baseline.

MODELS	BENCHMARKS (Avg@8) LiveCodeBench	CodeContests	MBPP	CodeForces	AVG.
		DEEPCODER			
QWEN3-8B-BASE	13.75	19.03	61.70	5.35	24.96
w/ RLVR (Binary)	23.60	32.00	**76.01**	16.43	37.01
w/ CODESCALER *(Binary)*	**24.80**(1.20↑)	33.94(1.94↑)	75.90(0.11↓)	19.61(3.27↑)	**38.56**(1.55↑)
w/ CODESCALER *(Pass Ratio)*	24.20(0.60↑)	**35.04**(3.04↑)	71.09(4.92↓)	**21.44**(5.01↑)	37.94(0.93↑)

Table 8: Ablation study on preference pair construction. Results compare Best-of-N sampling performance using different CODESCALER variant. **Bold** indicates the best results.

RM VARIANTS	BENCHMARKS (BoN@8) LiveCodeBench	CodeContests	MBPP	CodeForces	AVG.
CODESCALER *(Binary)*	**41.29**	**36.82**	**80.09**	**13.33**	**42.88**
CODESCALER *(Pass Ratio)*	40.84	35.88	79.64	13.28	42.41

Given the constructed graph, we sample concept combinations using a random walk sampling. Specifically, we first uniformly sample an initial topic node, and then perform up to six steps of a random walk on $\mathcal{G}$. At each step, the transition probability from node u to a neighboring node $v \in \mathcal{N}(u)$ is proportional to the corresponding edge weight:

$$P(u, v) = \frac{\exp w(u, v)}{\sum_{v' \in \mathcal{N}(u)} \exp w(u, v')}.$$

Each random walk episode yields a sampled concept set s, which is subsequently used for problem generation. Given the sampled concept set s, we prompt *GPT-4o* to generate 20K new problems for further training.

G.3 RL Training Configuration

During training, we set the temperature to 0.6, batch size to 128, mini-batch size to 64, and learning rate to $1e{-}6$, with a KL loss coefficient $\beta = 0.005$. Unless otherwise specified, models are trained for 250 steps. For each question, we sample 8 rollouts with a maximum response length of $16,384$ tokens. All experiments are conducted on a cluster of 8x NVIDIA A100 80G GPUs.

H Evaluation Details

H.1 Evaluation Datasets

We use four code datasets for evaluation: LiveCodeBench, CodeContests, MBPP and CodeForces. For LiveCodeBench, we select problems from 08/01/24 to 02/01/25 to avoid overlap with the DeepCoder training dataset. For the remaining four datasets, we follow Wang et al. [51]'s process. Specifically, for CodeContests, we extract tasks with difficulty level ≤ 2 and randomly generate an evaluation set of 239 examples. For MBPP, we use the standard test set. For CodeForces, we randomly sample 467 examples for evaluation.

H.2 Sandbox Environment

For evaluations that require code execution, we use a local sandbox implemented as a separate, guard-railed Python subprocess. Our sandbox strictly follows the official LiveCodeBench repository, ensuring consistency and fairness across evaluations.

I Analysis on Pass Ratio Feedback for RM Training

When constructing preference pairs, we adopt a strict binary construction where solutions that pass all test cases are treated as positive samples, while those that fail any test case are treated as negative.

Table 10: Full evaluation results on RM-Bench across diverse reward models. ↑ (↓) indicates the improvement (degradation) compared to the SkyworkRM at same model size.

Models	RM-Bench							Avg.
	Chat	Math	Code	Safety	Hard	Normal	Easy	
AceCodeRM-7B	66.7	65.3	66.9	89.9	62.2	74.4	79.9	72.2
AceCodeRM-32B	73.7	70.5	72.1	88.0	65.5	78.3	78.3	76.1
SkyworkRM-1.7B	69.6	71.4	72.3	92.9	54.5	82.3	92.8	76.6
CodeScaler-1.7B	74.4(4.8↑)	74.7(3.3↑)	73.1(0.8↑)	93.1(0.2↑)	61.5(7.0↑)	83.2(0.9↑)	91.7(1.1↓)	78.8(2.2↑)
SkyworkRM-4B	78.2	73.6	74.4	95.7	64.4	85.0	92.1	80.5
CodeScaler-4B	80.4(2.2↑)	79.0(5.4↑)	76.3(1.9↑)	95.8(0.1↑)	69.2(4.8↑)	86.5(1.5↑)	92.9(0.8↑)	82.9(2.4↑)
SkyworkRM-8B	80.6	75.0	73.6	96.5	67.0	85.5	91.8	81.4
CodeScaler-8B	83.0(2.4↑)	79.9(4.9↑)	76.9(3.3↑)	96.4(0.1↓)	71.8(4.8↑)	87.9(2.4↑)	92.5(0.7↑)	84.1(2.7↑)

In contrast, prior work such as AceCodeRM employs a looser criterion, labeling solutions with a pass ratio greater than 0.8 as positives. This design choice suggests that reward models may potentially learn from partially correct solutions.

To evaluate whether such threshold-based construction is necessary, we train a variant of CodeScaler where preference pairs are constructed based on test case pass ratios rather than binary correctness. Specifically, for each problem, solutions with a pass ratio above 0.7 are treated as positive samples, even if they are not fully correct.

As shown in Table 7 and Table 8, our binary CodeScaler outperforms the pass ratio counterpart in both RL training and Best-of-N sampling. We attribute this degradation to reduced label validity. Unlike binary positives, which correspond to fully verified solutions, high pass-ratio solutions may still contain latent logical errors due to incomplete test coverage. Treating such partially correct solutions as positives introduces noise into preference supervision, weakening the reward model's ability to reliably distinguish code quality. Our results indicate that the reward model can still learn sufficiently fine-grained signals from strictly valid binary supervision.

J CodeScaler at Different Scales

Reward models are often used as lightweight plugins in downstream tasks. Therefore, it is important to provide models of various sizes to meet different resource constraints. We trained different sizes of CodeScaler from 1.7B, 4B, to 8B parameters to meet this need. We evaluated these models in RL training, BoN task and RM-Bench. As shown in Table 9, performance improves consistently as the model size increases. Our smallest model, CodeScaler-1.7B, already performs comparably to AceCodeRM-7B in RL training. Furthermore, CodeScaler-4B surpasses SkyworkRM-8B on RM-Bench.

Table 9: Ablation study on CodeScaler in diverse scales. Evaluation across code benchmarks, BoN sampling and RM-Bench. ↑ (↓) indicates the improvement (degradation) compared to the SkyworkRM-8B.

RM Variants	Benchmarks Avg.	BoN@8 Avg.	RM-Bench Avg.
SkyworkRM-8B	29.33	41.17	81.46
AceCodeRM-7B	33.19(3.86↑)	40.48(0.69↓)	72.20(9.26↓)
CodeScaler-1.7B	35.40(6.07↑)	38.16(3.01↓)	78.83(2.63↓)
CodeScaler-4B	37.20(7.87↑)	39.82(1.35↓)	82.86(1.40↑)
CodeScaler-8B	**38.56**(9.23↑)	**44.15**(3.10↑)	**84.05**(2.59↑)

We further evaluate CodeScaler at different scales on RM-Bench and report the full results in Table 10. Across all model sizes, CodeScaler consistently outperforms its base SkyworkRM across all domains, highlighting the benefits of training on code preference data. Notably, even the smallest model, CodeScaler-1.7B, surpasses AceCodeRM-32B, demonstrating strong capability.

K Generalization Analysis

To evaluate whether CODESCALER generalizes beyond the Qwen model family, we conduct a cross-model-family Best-of-N evaluation.

Cross-Model-Family Transfer. Since CODESCALER serves as a reward model whose objective is to assign fair scores to generated code, it is fundamentally model-agnostic. To empirically verify this, we evaluate several non-Qwen policy models using `BoN@8` on CodeContests. As shown in Table 11, CODESCALER consistently outperforms both SKYWORKRM and ACECODERM across both model families, confirming its cross-family scoring quality.

Table 11: Best-of-N (`BoN@8`) evaluation on CodeContests with non-Qwen policy models. ↑ (↓) indicates the improvement (degradation) compared to the Pass@1.

Policy	Reward Model	Pass@1	BoN@8
DeepSeek-Coder-6.7B-Instruct	SKYWORKRM	18.41	21.76
	ACECODERM		23.85
	CODESCALER		**25.10**$_{(6.69\uparrow)}$
Llama-3.1-8B-Instruct	SKYWORKRM	15.90	19.67
	ACECODERM		20.08
	CODESCALER		**21.76**$_{(5.86\uparrow)}$

L Ablation on CODESCALER Training Data

In reward model training, we use the DeepCoder training dataset, which provides high-quality, verified test cases and thus offers reliable supervision for reward learning. To study the impact of training data quality, we conduct a controlled comparison by collecting RLVR trajectories from QWEN3-8B-BASE trained on KodCode and using them to train an alternative reward model. Since KodCode relies on synthesized test cases, it can be viewed as a lower-quality variant in terms of supervision. We then use these two reward models to train QWEN3-8B-BASE on the rStarCoder dataset. From the Table 12, we observe that CODESCALER trained on DeepCoder consistently outperforms its counterpart. In particular, the model trained with DeepCoder rollouts achieves a 0.71↑ on LiveCodeBench (23.79 → 24.5), 2.56↑ on CodeContests (28.19 → 30.75) and 4.37↑ on CodeForces (11.58 → 15.95). These findings highlight the critical role of high-quality, verified training data in reward model learning and further demonstrate the performance gap between verified code datasets and synthetic datasets.

Table 12: Ablation on CODESCALER training data. Results compare RL training performance using different trajectory sources.

TRAJ.	BENCHMARKS(Avg@8)		
	LiveCodeBench	CodeContests	CodeForces
rSTARCODER			
KODCODE	23.79	28.19	11.58
DEEPCODER	**24.50**$_{(0.71\uparrow)}$	**30.75**$_{(2.56\uparrow)}$	**15.95**$_{(4.37\uparrow)}$

M Training Stability Analysis

We provide two additional diagnostics of training stability, complementing the invalid and fragment rate curves in Figure 4.

Reward Distribution Evolution. Figure 6 illustrates the reward score distribution at training steps 0, 50, 100, 150, and 200 for QWEN3-8B-BASE on DeepCoder. The distribution shifts steadily rightward as training progresses: the mean score rises from 1.73 to 8.82, and the proportion of negative-reward rollouts drops from 32.2% to 2.1%.

Extended Reward Hacking Analysis. To verify that reward hacking does not occur under prolonged training, we extend the QWEN3-8B-BASE experiment on DeepCoder from 250 to 650 steps, evaluat-



Figure 6: Evolution of the RM score distribution at training steps 0, 50, 100, 150, and 200 (QWEN3-8B-BASE on DeepCoder). The distribution shifts rightward steadily with no signs of reward hacking.

ing every 50 steps. As shown in Figure 7, both RM score and pass@1 increase steadily throughout training, providing empirical evidence against reward hacking over extended training horizons.

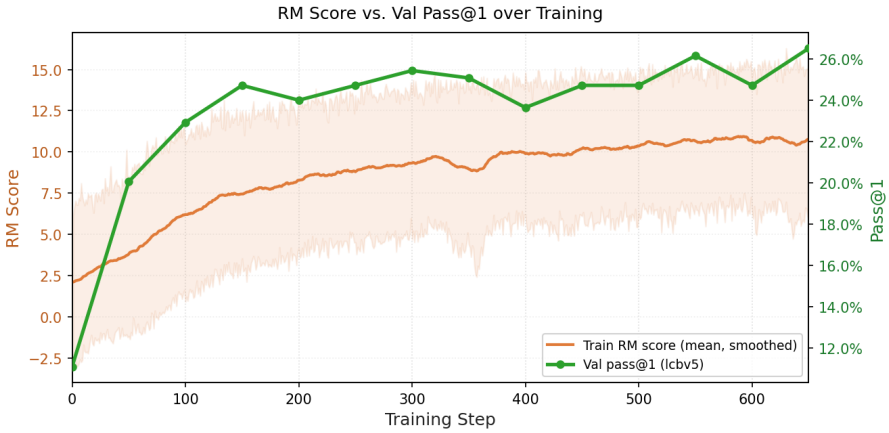


Figure 7: Extended training of QWEN3-8B-BASE on DeepCoder from 250 to 650 steps. Both RM score and pass@1 increase steadily with no divergence, confirming the absence of reward hacking.

N Sensitivity to Misalignment Ratio

To improve the robustness of CODESCALER, we augment the preference pairs by adding misaligned pairs where the code solution is generated for a different problem (see Appendix F). We investigate the sensitivity of CODESCALER to the misalignment augmentation ratio by training three variants with 0%, 30%, and 50% augmentation, and evaluating each on a held-out test set containing both aligned and misaligned samples.

Table 13: Sensitivity analysis of the misalignment augmentation ratio. The 0%→30% jump yields a large improvement in discrimination (AUROC 0.911→0.990), while the 30%→50% gain is marginal, confirming that 30% is a robust and sufficient augmentation ratio.

Misalignment Ratio	Score on Misaligned Code	Margin (pos−neg)	AUROC
0%	-6.87 ± 2.71	15.42 ± 5.69	0.911
30%	-14.85 ± 2.31	23.40 ± 5.37	**0.990**
50%	-17.45 ± 2.20	25.68 ± 6.11	0.990

Without augmentation, CODESCALER assigns only mildly negative scores (-6.87) to misaligned code, with an AUROC of 0.911. With 30% augmentation, the score drops to -14.85 and AUROC rises sharply to 0.990, indicating that the model has internalized a strong notion of problem-code alignment. Increasing the ratio to 50% yields only marginal further improvement, confirming that the alignment signal saturates around 30%.

O Failure Case Analysis

To understand CODESCALER's failure modes, we conduct a quantitative and qualitative analysis using training trajectory records from QWEN3-8B-BASE on DeepCoder with ground-truth execution labels. We identify two failure types: *False Positives* (FP), where the highest-scored rollout in a group fails execution, and *False Negatives* (FN), where the lowest-scored rollout passes execution. Groups with uniform outcomes are excluded, yielding 347 FP and 166 FN cases.

Comparing surface features across groups reveals two systematic biases:

Length Bias (FP vs. TP). Despite being incorrect, FP code is 28% longer than TP code, contains 37% more algorithmic keywords, and is accompanied by 38% longer chain-of-thought responses. This suggests that CODESCALER partially relies on response length and keyword density as quality proxies.

Brevity Under-Valuation (FN vs. TN). Correct but low-scored FN responses are $2\times$ shorter than incorrect TN responses. FN code also shows higher rates of modular multi-function design (+32%) and recursion (+18%), suggesting that concise, well-structured solutions are systematically undervalued. Figure 8 illustrates a representative case.

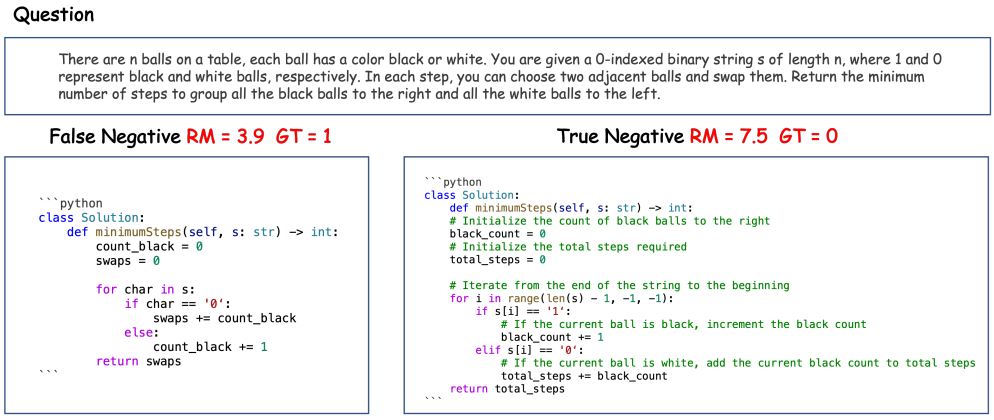


Figure 8: Both solutions implement an O(n) single-pass approach. The FN (correct, 278 chars) sweeps left-to-right, counting for each white ball the number of black balls to its left. The TN solution (incorrect, 661 chars) mirrors the traversal right-to-left, inadvertently solving the opposite goal (blacks left, whites right). Despite being 2.4× shorter and correct, the FN receives RM = 3.969 vs. RM = 7.531 for the wrong solution.

P Prompt Template

Answering Prompt Template

You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program that matches the specification and passes all tests.

{{question}}

Format: Read the inputs from stdin solve the problem and write the answer to stdout (do not directly test on the sample inputs). Enclose your code within delimiters as follows. Ensure that when the python program runs, it reads the inputs, runs the algorithm and writes output to STDOUT.

```python
# YOUR CODE HERE
```

Answer: (use the provided format with backticks)

Concept Extraction Template

Act as a Programming Teacher and analyze the provided question. Start by identifying 1 or 2 general topics that a student is being assessed on. Structure your response as:
"Topics:
1. <Topic 1>
2. <Topic 2>"

Next, highlight 1 to 5 specific knowledge points that the question evaluates. Structure your response as:
"Specific Knowledge Points:
1. <Knowledge Point 1>
2. <Knowledge Point 2>
3. <Knowledge Point 3>
4. <Knowledge Point 4>
5. <Knowledge Point 5>"

The topics and specific knowledge points should be terms that are concise and commonly used in academia or industry.

Provided question:
{{question}}

Analysis:

Problem Generation Prompt Template

Act as a Programming Teacher and create a new question and its solution based on the provided topics and knowledge points. Ensure that the created questions:
1. Adhere to the provided topics.
2. Necessitate the combined use of the associated knowledge points.

Example
Topics:
1. Graph Theory
2. Combinatorics
Knowledge Points:
1. Graph Coloring
2. Connected Components
3. Dynamic Programming
4. Modular Arithmetic
Created Question:
You are given a grid, consisting of 2 rows and n columns. Each cell of this grid should be colored either black or white.
Two cells are considered neighbours if they have a common border and share the same color. Two cells A and B belong to the same component if they are neighbours, or if there is a neighbour of A that belongs to the same component with B.
Let's call some bicoloring beautiful if it has exactly k components.
Count the number of beautiful bicolorings. The number can be big enough, so print the answer modulo 998244353.

—-Input—-
The only line contains two integers n and k ($1 \leq n \leq 1000$, $1 \leq k \leq 2n$) the number of columns in a grid and the number of components required.

—-Output—-
Print a single integer — the number of beautiful bicolorings modulo 998244353.

—-Examples—-
Input
34
Output
12
Input
41
Output
2
Input
12
Output
2

Topics:
1. String Manipulation
Knowledge Points:
1. Understanding and manipulating string data structures
2. Dynamic Programming

Try to create a question for the last one. Structure your response as:
Created Question:
<Question>

Q A Case Study on Flawed Synthesis Test Cases

Taking a synthesis problem from rStarCoder as an example, this task requires computing the number of valid arrangements that contain exactly n successive flowers of the same type. Although the problem description specifies an input range of $3 \leq n \leq 30$, the synthesized test cases only cover values up to $n = 9$. This limitation likely arises because the mutual verification mechanism fails to reach consensus on certain corner or large-input cases, which are consequently discarded during test case construction. As a result, the synthesized test suite is incomplete, and the missing cases are often the most challenging ones that truly distinguish efficient solutions from inefficient ones.

In this setting, it is possible to construct a flawed solution that relies on a naive pure-recursive enumeration strategy without any form of memoization or dynamic programming. The recursive function `count_ways_pure_recursive` explores all possible color assignments at each position, leading to an exponential-time complexity with respect to the flowerbed length $2n - 2$. While this approach is sufficient for small values of n (e.g., $n \leq 9$), it becomes prohibitively expensive as n increases.

Consequently, this solution passes all synthesized test cases and is incorrectly regarded as correct. However, when evaluated on larger inputs within the valid constraint range, it exhibits severe performance degradation and ultimately results in time limit exceeded (TLE) errors.

rStarCoder synthetic_rl_testcase synthetic_50341

Problem Description:
A city council wants to arrange a flowerbed in a park to contain exactly n successive flowers of the same type. There are four types of flowers available for planting.
The flowerbed consists of one line of (2n - 2) spaces. The total number of flowers is greater than the number of spaces available. Additionally, the number of flowers of each type exceeds the number of spaces on the flowerbed. Determine the number of ways to arrange the flowerbed so that it contains exactly n successive flowers of the same type.
Input Format:
An integer n ($3 \leq$ n ≤ 30) representing the number of successive flowers of the same type.
Output Format:
An integer representing the number of ways to arrange the flowerbed in the required manner.
Examples
Input:
3
Output:
24
Input:
4
Output:
132

Inputs:
[3,4,5,6,7,8,9]
Outputs:
[24,132,672,3264,15360,70656,319488]

A flawed solution to synthetic_50341 yet passes all test cases

```
import sys

def solve():
    line = sys.stdin.readline()
    if not line:
        return
    n = int(line.strip())

    total_len = 2 * n - 2
```

```
    def count_ways_pure_recursive(idx, current_streak, found_exactly_n):
        if current_streak > n:
            return 0

        if idx == total_len:
            return 1 if (found_exactly_n or current_streak == n) else 0

        res = 0
        res += count_ways_pure_recursive(idx + 1, current_streak + 1,
            found_exactly_n)

        if current_streak == n:
            res += 3 * count_ways_pure_recursive(idx + 1, 1, True)
        else:
            res += 3 * count_ways_pure_recursive(idx + 1, 1,
                found_exactly_n)

        return res

    final_ans = 4 * count_ways_pure_recursive(1, 1, False)
    print(final_ans)

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

NeurIPS Paper Checklist

1. **Claims**

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

   Answer: [Yes]

   Justification: All claims are supported by experiments in Sections 5–6. Quantitative results match the stated numbers.

   Guidelines:

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

2. **Limitations**

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

   Answer: [Yes]

   Justification: Limitations are discussed in Appendix B.

   Guidelines:

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

3. **Theory assumptions and proofs**

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

   Answer: [N/A]

Justification: This paper is empirical and does not include theoretical results.

Guidelines:

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

4. **Experimental result reproducibility**

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

   Answer: [Yes]

   Justification: Full training and evaluation details are provided in Appendix G, H, and F. All datasets are publicly available.

   Guidelines:

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

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

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

Answer: [Yes]

Justification: Code, preference data, and reward-model weights are public at `https://github.com/LARK-AI-Lab/CodeScaler` and `https://huggingface.co/collections/LARK-Lab/codescaler`.

Guidelines:

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

6. **Experimental setting/details**

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

   Answer: [Yes]

   Justification: All hyperparameters, optimizer, and dataset details are specified in Section 5 and Appendix G, F, H.

   Guidelines:

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

7. **Experiment statistical significance**

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

   Answer: [No]

   Justification: Error bars are not reported due to the high computational cost of RL training.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

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

   Answer: [Yes]

   Justification: All experiments use 8× NVIDIA A100 80G GPUs (Appendix G). Latency is measured on a single A100 (Section 5.2).

   Guidelines:

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

9. **Code of ethics**

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

   Answer: [Yes]

   Justification: The research conforms with the NeurIPS Code of Ethics. All datasets are public and properly cited.

   Guidelines:

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

10. **Broader impacts**

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

    Answer: [Yes]

    Justification: See Appendix A.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

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

    Answer: [N/A]

    Justification: The released artifacts are code reward models and preference data, with intended use and limitations documented in the public repository.

    Guidelines:

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

12. **Licenses for existing assets**

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

    Answer: [Yes]

    Justification: All datasets and models are properly cited. Dataset URLs are provided in the appendix.

    Guidelines:

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

- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

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

    Answer: [Yes]

    Justification: The CodeScaler reward-model checkpoints and preference-pair dataset are available at `https://huggingface.co/collections/LARK-Lab/codescaler`, with code and usage instructions at `https://github.com/LARK-AI-Lab/CodeScaler`.

    Guidelines:

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

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

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

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing or human subjects.

    Guidelines:

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

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

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

    Answer: [N/A]

    Justification: This paper does not involve human subjects.

    Guidelines:

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

- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

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

    Answer: [N/A]

    Justification: The core method development in this research does not involve LLMs as any important, original, or non-standard components.

    Guidelines:

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