

Co-Evolving LLM Coder and Unit Tester via Reinforcement Learning

Yinjie Wang^{1*}, Ling Yang^{2,4*}, Ye Tian³, Ke Shen⁴, Mengdi Wang²

¹University of Chicago ²Princeton University ³Peking University ⁴ByteDance Seed

*Equal Contribution

Abstract

We propose **CURE**, a novel reinforcement learning framework with a dedicated reward design that co-evolves coding and unit test generation capabilities based on their interaction outcomes, without any ground-truth code as supervision. This approach enables flexible and scalable training and allows the unit tester to learn directly from the coder’s mistakes. Our derived **ReasonFlux-Coder** 7B and 14B models improve code generation accuracy by 5.3% and Best of N accuracy by 9.0% after optimization on Qwen2.5-Instruct models, outperforming similarly sized Qwen-Coder, DeepSeek-Coder, and Seed-Coder. They naturally extend to downstream tasks such as test-time scaling and agentic coding—achieving a 8.1% improvement over the base model. For the long-CoT model, our ReasonFlux-Coder-4B consistently outperforms Qwen3-4B while achieving 64.8% inference efficiency in unit test generation. Notably, we also find that our model can serve as an effective reward model for reinforcement learning on base models.

Correspondence: Ling Yang at yangling0818@163.com

Github Page: <https://github.com/Gen-Verse/CURE>

Model Weight: <https://huggingface.co/Gen-Verse/ReasonFlux-Coder>

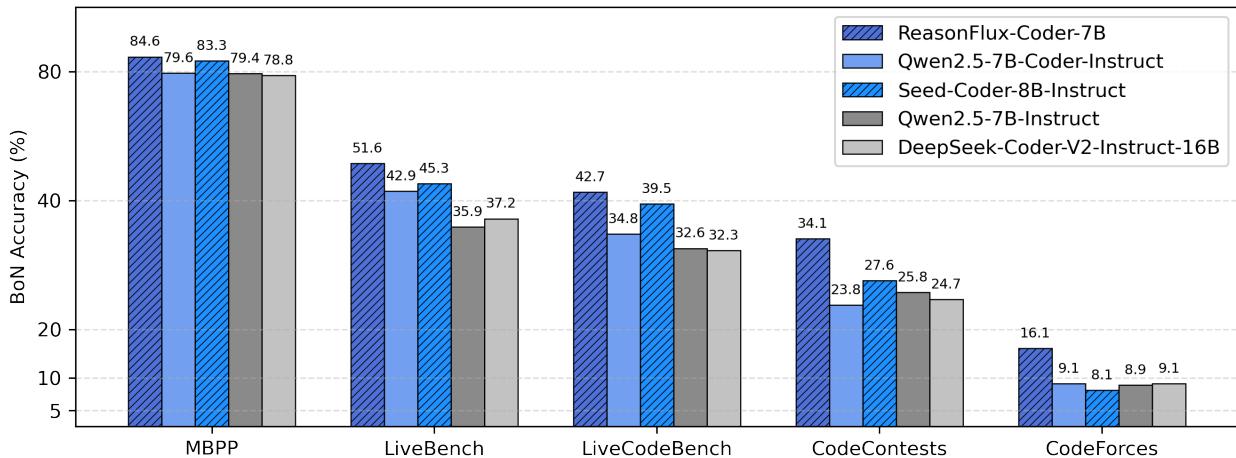


Figure 1 Performance of ReasonFlux-Coder-7B, trained with CURE on only 4.5K coding problems, surpasses models that are specifically fine-tuned on large-scale coding data. We generate 16 candidate solution codes and 16 unit tests, selecting the final solution as the one that passes the most generated unit tests, which is a BoN strategy.

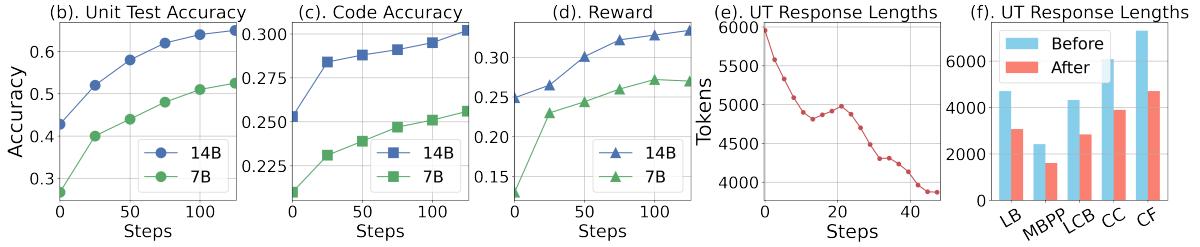
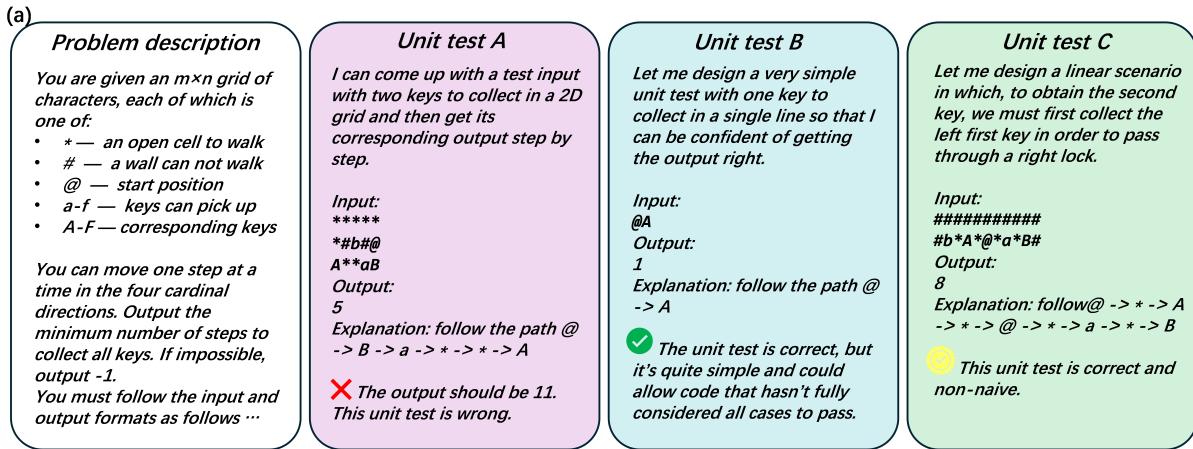


Figure 2 (a). This is an example of a problem description along with three task-derived generated unit tests. The first unit test is incorrect, although it is easily produced due to strong hallucination. The second unit test is correct but naive, allowing some incomplete or unthoughtful code to pass. The final unit test is both correct and non-naive, though generating such a test is much easier than actually solving the full coding problem. (b-d) Co-evolving process: (b) unit test accuracy, (c) code accuracy, and (d) estimated reward versus number of steps. (e-f). The Long CoT unit tester becomes increasingly efficient in reasoning as the response length decreases during optimization.

1 Introduction

Recently, the mathematical reasoning capabilities and precision of large language models (LLMs) have seen substantial improvements through post-training optimization techniques such as reinforcement learning [1–5], as well as through test-time scaling methods guided by reward-based selection strategies [6–10], including Best of N (BoN). In this paper, we focus on enhancing the coding capabilities of LLMs—a domain critical to the advancement of artificial intelligence—through both post-training optimization and test-time scaling approaches.

Beyond scaling the one-shot coding capabilities of LLMs, we identify generating unit tests as a key factor—and a promising entry point—for improving coding performance. Specifically, we focus on task-derived unit tests, which are generated from a given coding task description and are designed to verify the correctness of the corresponding code. We highlight several advantages of using unit tests in this context. First, their direct alignment with code correctness makes unit tests a reliable reward signal, suitable for guiding both reinforcement learning [11–13] and test-time scaling or agentic coding pipelines [14–18]. Second, generated unit tests can be efficiently reused across all candidate solutions during test-time scaling, avoiding the quadratic complexity inherent in scalar or generative reward models, which require separate reward computations for each candidate [6–10]. Most importantly, generating a unit test does not necessarily require the model to produce a complete solution or algorithm (see Figure 2(a)), substantially simplifying test construction compared to traditional verification approaches, in which LLMs often struggle to verify and correct self-generated solutions [19]. Moreover, using generated unit tests at inference time naturally promotes a self-check and self-correction pattern.

Traditional unit test generation techniques include software analysis methods [20, 21] and machine translation-

based approaches [22, 23]. Recent developments show that large language models (LLMs) outperform traditional approaches in unit test generation [24–26], aided by prompt engineering and agentic techniques [25, 27, 28]. These findings highlight the potential for fine-tuning LLMs to further enhance their unit test generation capabilities [24]. O1-Coder [11] fine-tunes LLMs using unit tests derived from ground-truth code. Inspired by the trade-off between attack rate and accuracy, UTGEN [29] further proposes training LLMs with both correct unit tests from ground-truth code and incorrect tests from perturbed code to enhance downstream inference tasks.

However, training unit test generators in these ways requires supervision from ground-truth code solutions, whose collection is both costly and labor-intensive, thereby limiting the scale and diversity of usable training data. If a unit test generator could instead be trained without reliance on ground-truth code, this would substantially improve the flexibility and scalability of the optimization process. To this end, we propose leveraging the code generator to provide supervision for the unit test generator, while simultaneously improving the code generator itself to produce more accurate outputs that guide the generation of correct unit tests.

Motivated by this, we pose the following central research question for scaling LLMs in coding tasks: **Can the unit test generator and code generator coevolve effectively, without access to ground-truth code solutions, to improve LLM coding ability?**

We answer this question affirmatively by introducing **CURE**, a novel reinforcement learning framework (Figure 3) that co-evolves a self-play agent acting as both a code generator and a unit test generator. CURE constructs a pairwise reward matrix based on interactions between generated codes and generated tests, enabling mutual supervision and continuous improvement (Figure 2 (b)-(d)). This setup is well-motivated: during reinforcement learning, the coder naturally produces both correct and incorrect solutions, with the incorrect ones revealing typical failure modes. These, in turn, offer valuable opportunities for the unit test generator to learn to distinguish good code from bad code.

We further demonstrate the utility of the optimized model in two settings. First, and most importantly, it effectively enhances one-shot coding, unit test generation, test-time scaling and agentic coding ability. Second, we find that using the optimized model to generate unit tests, as a reward model for reinforcement learning on the base model, can lead to competitive improvements compared to using ground-truth labeled unit tests. Finally, while long-chain-of-thought (long-CoT) models represent some of the most advanced AI capabilities to date, they suffer from extremely slow inference [1, 30, 31]. To address this, we introduce a response-length-guided transformation on the reward to make the long-CoT unit test generator more efficient in test-time applications.

We summarize our contributions as follows:

1. We propose **CURE**, a novel co-evolving reinforcement learning framework that enables a single model to simultaneously excel at unit test generation and coding, without access to any ground-truth code solutions. The framework employs a theoretically derived and well-motivated reward design for unit test generation. In addition, for long-chain-of-thought models, we introduce a response-length-guided reward transformation to enhance test-time efficiency of the fine-tuned unit test generator. This results in models of different scales: **ReasonFlux-Coder-4B, 7B and 14B**.
2. We conduct extensive evaluations on five benchmarks and demonstrate that CURE effectively enhances the abilities of the model in unit test generation and coding, naturally extends to test-time scaling and agentic coding tasks and agentic unit test generation tasks. Specifically, ReasonFlux-Coder 7B and 14B models improve code generation accuracy by 5.3% and Best of N accuracy by 9.0% after optimization on Qwen2.5-Instruct models, outperforming similarly sized Qwen-Coder, DeepSeek-Coder, and Seed-Coder. Our long-CoT 4B model consistently outperforms Qwen3-4B while achieving 64.8% inference efficiency in unit test generation.
3. Finally, we show that the trained unit test generator can serve as a reward model to fine-tune LLMs via reinforcement learning—improving coding performance without any human-labeled or ground-truth unit test supervision.

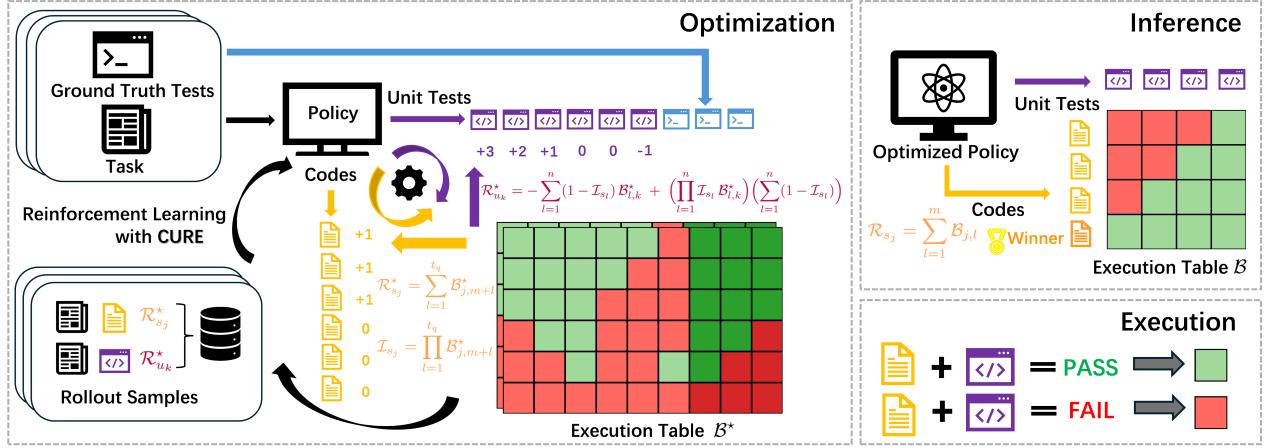


Figure 3 Method Pipeline Overview. In our RL framework, for each task, we generate a batch of unit tests and code solutions, along with some ground-truth unit tests. Using these, we construct an execution table. From this table, we extract rewards for each unit test (Equation 4) and code response (Equation 3). For the long-CoT model, we apply a transformation on the reward to ensure efficiency. Then we optimize both the unit tester and the coder iteratively.

2 Related Work

2.1 Unit Test Generation

Manually creating unit tests is costly and inefficient [15, 32], motivating the development of automated unit test generation methods, such as software analysis methods [20, 21, 33–36] and traditional neural machine translation approaches [22, 23]. With the recent advancements in LLMs, prompt-based and agentic methods [25, 27, 28] have demonstrated superior performance, further highlighting the potential of training LLMs for unit test generation. In light of this, methods like O1-Coder [11] and UTGEN [29] construct datasets using ground-truth code solutions to fine-tune LLMs for better unit test generation. However, relying on ground-truth code solutions in the training data limits both flexibility and scalability.

2.2 Application of Unit Tests

Unit tests have been shown to serve as effective rewards for test-time scaling and agentic coding [14]. A common strategy is to generate multiple code and unit test candidates using the model, then select the best-performing sample based on execution results against the generated unit tests [15, 16]. AlphaCodium [17] introduces self-revision by leveraging both public and generated tests to refine solutions. S* [18] further incorporates iterative debugging and pairwise discrimination guided by generated unit tests to enhance final code quality.

2.3 Reinforcement Learning for LLM Improvement

Proximal Policy Optimization (PPO) [37] uses an actor-critic setup with clipped updates for stability. Direct Preference Optimization (DPO) and its variants [38–44] skip the critic and directly optimize from preferences using closed-form rewards, improving efficiency. Recent efficient Group Relative Policy Optimization (GRPO) [3] scales well with large-scale reinforcement learning [1, 45]. Reinforcement learning applied specifically to coding tasks has also gained traction [12, 13]. We do not aim to compete with existing reinforcement learning algorithms for code generation; in fact, these RL-on-coding methods can be naturally integrated into our co-evolutionary framework by directly applying them to optimize the coding component.

3 Method

In this section, we begin by formulating our final objective and introducing the general concept of *reward precision* (Section 3.1). We then provide a theoretical analysis of reward precision to derive individual-level

rewards for each generated unit test (Section 3.2). Next, we present our novel co-evolving reinforcement learning framework, **CURE**, in Section 3.3. Finally, we introduce a response-length-guided transformation on the reward, designed to improve the efficiency of the unit test generator for long CoT models (Section 3.4).

3.1 Motivation: Using Unit Tests for Inference

Unlike mathematical tasks, which are computationally intensive and challenging to verify accurately [19], code-generation tasks benefit significantly from the use of unit tests for efficient verification. It has been shown [14] that the accuracy of code generation can be enhanced by adopting the following BoN approach: For each task q , the policy LLM generates n candidate solutions s_j , where $1 \leq j \leq n$, and m additional unit tests u_k , where $1 \leq k \leq m$. Executing the n generated solutions against these m unit tests produces a binary evaluation matrix $\mathcal{B} \in \{0, 1\}^{n \times m}$, where each entry indicates whether a given solution passes a specific test. The reward for solution s_j is defined as follows, and is used to select the optimal coding solution:

$$\mathcal{R}_{s_j} = \sum_{l=1}^m \mathcal{B}_{j,l}. \quad (1)$$

Empirically, this reward is typically valid because incorrectly generated unit tests also rarely favor incorrect solutions. However, this assumption can break down when the generated unit tests are of low accuracy, under ambiguous problem formulations, or in binary output tasks. Therefore, we propose our objective for optimizing the unit test generator, **reward precision**:

$$P(\mathcal{R}_{s_{j_1}} > \mathcal{R}_{s_{j_2}} \mid s_{j_1} \text{ is correct}, s_{j_2} \text{ is wrong}). \quad (2)$$

The higher the reward precision, the more accurately the generated unit tests can identify and promote correct solutions. But this is merely an overall objective. To obtain rewards at the individual level for generated unit tests, we conduct the following analysis to derive the reward formulation.

3.2 Analysis on Reward Precision

In this section, we identify the key factors that ensure the validity and accuracy of the reward precision defined in Equation 2. Given that the generated responses are i.i.d., we model the binary evaluation results with the following generative process: First, the correctness of a generated solution, denoted by c_s , and the correctness of a generated unit test, denoted by c_u , are modeled as Bernoulli random variables with success probabilities p_s and p_u , respectively. Conditional on their correctness, the execution outcome is another Bernoulli random variable with success probability $p_{c_s c_u}$. Specifically, we have $p_{10} = 0$ and $p_{11} = 1$, while the parameters p_{00} and p_{01} remain unknown.

In the theorem below, we demonstrate increasing the number of generated unit tests m causes the reward precision to converge to 1, provided that certain conditions involving the parameters p_u , p_{00} , and p_{01} are satisfied. We naturally derive our optimization objective with this theoretical analysis.

Theorem 3.1. *Consider a ground truth unit test u_k , a correct solution s_{j_1} , and an incorrect solution s_{j_2} . The precision based on a single ground truth test is given by*

$$P(\mathcal{B}_{j_1,k} > \mathcal{B}_{j_2,k}) = 1 - P(\text{the incorrect solution } s_{j_2} \text{ passes test } u_k).$$

However, when using the aggregated reward defined in Equation 1, we have $P(\mathcal{R}_{s_{j_1}} > \mathcal{R}_{s_{j_2}}) \rightarrow 1$ as $m \rightarrow \infty$, if and only if $\mu > 0$, where

$$\mu := p_u(1 - p_{01}) - (1 - p_u)p_{00}.$$

Moreover, under this condition, the reward precision satisfies

$$P(\mathcal{R}_{s_{j_1}} > \mathcal{R}_{s_{j_2}}) \gtrsim 1 - e^{-\mu^2 m / 8}.$$

From this theorem, we observe that μ not only guarantees the convergence and validity of the aggregated reward (Equation 2), but also governs the rate at which it converges to 1. Specifically, a larger value of μ implies that fewer unit tests are needed to obtain a reliable reward signal.

Therefore, we use μ as the optimization objective for the unit test generator, estimating the individual value of μ for each unit test from the execution matrix to serve as its reward. Intuitively, optimizing μ corresponds to increasing the accuracy p_u while controlling the error rates p_{01} and p_{00} for the generated unit tests. We now introduce our algorithm to co-evolve the coder and the unit tester.

3.3 Co-evolving Coder and Unit Tester with RL

For each task q in the training set, which is paired with t_q ground truth unit tests, the policy LLM generates n candidate solutions and m additional unit tests u_k , where $1 \leq k \leq m$. Similarly, we obtain a binary evaluation matrix $\mathcal{B}^* \in \{0, 1\}^{n \times (m+t_q)}$ by executing the n generated solutions against these $m + t_q$ unit tests. The last t_q columns correspond to the ground truth unit tests. This evaluation matrix serves as the basis for estimating rewards for both the solution generator and the unit test generator, enabling joint optimization via reinforcement learning.

For solution s_j , where $1 \leq j \leq n$, we assign higher rewards to solutions that pass more ground-truth unit tests, reflecting greater correctness and generalizability. The reward is defined as:

$$\mathcal{R}_{s_j}^* = \sum_{l=1}^{t_q} \mathcal{B}_{j,m+l}^*. \quad (3)$$

For each generated unit test u_k , where $1 \leq k \leq m$, we estimate the reward $\mu = p_u(1 - p_{01}) - (1 - p_u)p_{00}$ from the execution matrix \mathcal{B}^* by deriving estimators for p_u , p_{01} , and p_{00} . This leads to the following form of the estimated individual-level reward:

$$\mathcal{R}_{u_k}^* = - \sum_{l=1}^n (1 - \mathcal{I}_{s_l}) \mathcal{B}_{l,k}^* + \left(\prod_{l=1}^n \mathcal{I}_{s_l} \mathcal{B}_{l,k}^* \right) \left(\sum_{l=1}^n (1 - \mathcal{I}_{s_l}) \right), \quad (4)$$

where $\mathcal{I}_{s_j} = \prod_{l=1}^{t_q} \mathcal{B}_{j,m+l}^*$. The detailed derivation is provided in Appendix A. Intuitively, $\mathcal{R}_{u_k}^*$ is positive and proportional to the number of incorrect solutions that fail test u_k when u_k correctly passes all accurate solutions. Conversely, $\mathcal{R}_{u_k}^*$ is negative and proportional to the number of incorrect solutions that pass test u_k when u_k fails even one correct solution. Here, a correct solution is defined as one passing all ground-truth unit tests, whereas an incorrect solution fails at least one ground-truth test. Therefore, this theoretically derived reward serves as an effective objective, optimizing the accuracy and discriminative power of generated unit tests. Naively using reward functions like “whether the unit test passes all correct codes” incentivize the generation of trivial or overly permissive tests that simply maximize pass rates. This undermines the reliability of the reward signal and diminishes the overall effectiveness of the co-evolution process.

After collecting the rollout samples for codes and unit tests and their rewards, we optimize the policy with the following objective iteratively:

$$\begin{aligned} \mathcal{J}(\theta, \{o_i\}_{i=1}^G) &= \mathbb{E}_{\substack{q \sim P(Q) \\ \{o_i\} \sim \pi_{\theta_{\text{old}}}(\cdot|q)}} \left[\frac{1}{G} \sum_{i=1}^G \min \left[\frac{\pi_\theta(o_i | q)}{\pi_{\theta_{\text{old}}}(o_i | q)} A_{o_i}, \text{clip} \left(\frac{\pi_\theta(o_i | q)}{\pi_{\theta_{\text{old}}}(o_i | q)}, \varepsilon \right) A_{o_i} \right] \right] \\ &\quad - \mathbb{E}_{\substack{q \sim P(Q) \\ \{o_i\} \sim \pi_{\theta_{\text{old}}}(\cdot|q)}} \left[\beta \text{D}_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}] \right], \end{aligned} \quad (5)$$

where $\text{clip}(x, \varepsilon) := \min(\max(x, 1 - \varepsilon), 1 + \varepsilon)$, π_θ is the policy to be optimized, $\pi_{\theta_{\text{old}}}$ is the old policy, $\{o_i\}_{i=1}^G$ are the rollout responses, and A_{o_i} is the normalized reward corresponding to $\mathcal{R}_{o_i}^*$. Specifically, we iteratively optimize the policy for coding ability with $\mathcal{J}(\theta, \{s_j\}_{j=1}^n)$, and unit test generation ability with $\mathcal{J}(\theta, \{u_k\}_{k=1}^m)$ (see Figure 3).

Algorithm 1 CURE

```

1: Input:
2:   1) A set of coding tasks  $D = \{q_1, q_2, \dots, q_N\}$ .
3:   2) A policy  $\pi_\theta$  parameterized by  $\theta$ .
4:   3) Number of iterations  $M$ .
5:   4) Number of code solutions generated in each step:  $n$ .
6:   5) Number of unit tests generated in each step:  $m$ .
7:   5) Learning rate  $\eta$ , KL coefficient  $\beta$ .
8: Initialize: Policy parameters  $\theta$ .
9: for  $t = 1$  to  $M$  or not converged do
10:   Collect rollout samples:
11:     for each task  $q \in D$  do
12:       Generate  $n$  code solutions,  $s_j$ ,  $1 \leq j \leq n$ , by policy  $\pi_\theta$ .
13:       Generate  $m$  unit tests,  $u_k$ ,  $1 \leq k \leq m$ , by policy  $\pi_\theta$ .
14:       Executing the  $n$  generated solutions against these  $m$  unit tests produces a binary evaluation matrix
         $\mathcal{B}^* \in \{0, 1\}^{n \times m}$ .
15:     end for
16:   Obtain the reward for each code solution:
17:     for each task  $s_j$ ,  $1 \leq j \leq n$  do
18:        $\mathcal{R}_{s_j}^* = \sum_{l=1}^{t_q} \mathcal{B}_{j,m+l}^*$ 
19:     end for
20:   Obtain the reward for each unit test:
21:     for each task  $u_k$ ,  $1 \leq k \leq m$  do
22:        $\mathcal{R}_{u_k}^* = -\sum_{l=1}^n (1 - \mathcal{I}_{s_l}) \mathcal{B}_{l,k}^* + (\prod_{l=1}^n \mathcal{I}_{s_l} \mathcal{B}_{l,k}^*) (\sum_{l=1}^n (1 - \mathcal{I}_{s_l}))$ 
23:       if  $\pi_\theta$  is long-CoT model then
24:          $\mathcal{R}_{u_k}^* = \text{trans}(\mathcal{R}_{u_k}^*, l_k)$  (see Section 3.4)
25:       end if
26:     end for
27:   Optimize the policy  $\pi_\theta$ :
28:     Compute advantages  $A_{s_j} = \text{normalize}(\mathcal{R}_{s_j}^*)$ .
29:     Fine-tune  $\pi_\theta$  to obtain updated parameters  $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{J}(\theta, \{s_j\}_{j=1}^n)$  (see Equation 5).
30:     Compute advantages  $A_{u_k} = \text{normalize}(\mathcal{R}_{u_k}^*)$ .
31:     Fine-tune  $\pi_\theta$  to obtain updated parameters  $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{J}(\theta, \{u_k\}_{k=1}^m)$ .
32:   end for
33: Output: Trained generator  $\pi_\theta$ .

```

3.4 Improve Efficiency of Long-CoT Unit Tester

In addition to experiments conducted on base LLMs, we also perform experiments using the long-CoT model, which currently exemplifies the highest reasoning capabilities of LLMs. However, it is well-documented that these long-CoT models suffer from significantly increased inference times [1, 30, 31]. To enhance efficiency, we propose a general response-length-aware transformation applied to the rewards of unit tests specifically when utilizing long-CoT models.

Formally, for each task q , consider a set of standardized rewards $\{r_i\}_{i=1}^m$ (standardized by subtracting the mean) and the corresponding response lengths $\{l_i\}_{i=1}^m$. Our goal is to assign negative values to overly long responses proportionally to their lengths while ensuring that the transformed rewards maintain a clear separation such that negative original rewards remain negative and positive original rewards remain positive. Specifically, we first transform the rewards to \hat{r}_i by

$$\hat{r}_i = \begin{cases} -l_i + T_l & \text{if } r_i > 0, \\ -l_{\max} + T_l & \text{if } r_i \leq 0, \end{cases}$$

where $T_l = \text{median}\{l_j \mid r_j > 0\}$, $l_{\max} = \max\{l_j \mid r_j > 0\}$. Subsequently, we balance the transformed rewards

Table 1 Performance of ReasonFlux-Coder models and baseline models across five benchmarks. Each entry reports the average accuracy (%) of generated unit tests (UT), the average one-shot code generation accuracy (Code), and the Best-of-N (BoN) accuracy, using 16 generated code solutions and 16 generated unit tests. “Long” refers to the long-CoT base models. The Coder models here are also instruction-finetuned models. The full name of DeepSeek-Coder-V2-16B is DeepSeek-Coder-V2-Lite-Instruct. Numbers in bold indicate the best performance, and those with an underline indicate the second-best.

Model	LiveBench			MBPP			LiveCodeBench			CodeContests			CodeForces		
	UT	Code	BoN	UT	Code	BoN	UT	Code	BoN	UT	Code	BoN	UT	Code	BoN
DeepSeek-Coder-V2-16B	35.4	31.9	37.2	68.7	65.2	78.8	30.0	26.8	32.3	30.4	20.3	24.7	20.5	5.0	9.1
Qwen2.5-14B-Coder-Instruct	<u>39.0</u>	<u>42.2</u>	<u>53.1</u>	<u>75.1</u>	72.6	84.9	<u>41.6</u>	<u>38.2</u>	<u>47.7</u>	37.3	23.3	32.0	<u>22.1</u>	7.8	<u>13.5</u>
Qwen2.5-14B-Instruct	27.8	36.4	51.7	72.8	<u>76.3</u>	83.2	35.7	33.5	45.1	<u>43.8</u>	<u>25.6</u>	<u>33.4</u>	20.7	7.3	12.5
ReasonFlux-Coder-14B	73.3	47.5	60.2	91.6	78.5	88.2	81.4	40.5	50.5	86.0	32.1	44.4	82.3	12.1	25.9
Seed-Coder-8B-Instruct	<u>31.4</u>	<u>35.6</u>	<u>45.3</u>	60.0	64.7	83.3	<u>28.7</u>	31.7	39.5	<u>29.1</u>	18.9	27.6	12.1	<u>7.1</u>	8.1
Qwen2.5-7B-Coder-Instruct	19.3	35.0	42.9	41.3	<u>68.0</u>	79.6	20.6	29.8	34.8	12.9	<u>22.8</u>	23.8	7.2	6.7	<u>9.1</u>
Qwen2.5-7B-Instruct	26.5	31.1	35.9	35.8	66.3	79.4	28.6	26.9	32.6	26.7	21.2	25.8	<u>18.9</u>	5.4	8.9
ReasonFlux-Coder-7B	54.8	37.1	51.6	79.4	70.2	84.6	57.7	<u>31.2</u>	42.7	62.6	25.9	34.1	45.6	8.2	16.1
Qwen3-4B (Long)	36.8	72.5	78.1	76.5	88.4	90.1	50.9	74.5	80.0	43.6	53.0	58.3	54.1	28.8	38.5
ReasonFlux-Coder-4B	84.6	74.6	82.0	83.3	89.5	91.1	86.8	74.9	80.6	72.2	54.6	59.9	65.8	30.9	40.2

between positive and negative responses and normalize them, yielding the final transformed reward r_i^* , defined by $r_i^* = \alpha \hat{r}_i / \sigma$ if $\hat{r}_i > 0$, or $r_i^* = \hat{r}_i / \sigma$ if $\hat{r}_i \leq 0$, where $\alpha = \sum_{j: \hat{r}_j < 0} (-\hat{r}_j) / (\sum_{j: \hat{r}_j > 0} \hat{r}_j)$, and σ is the standard deviation calculated over the set $\{\alpha \hat{r}_i \mid \hat{r}_i > 0\} \cup \{\hat{r}_i \mid \hat{r}_i \leq 0\}$. Finally, after the rewards are assigned, we truncate responses longer than 8K tokens, retaining only the first 8K tokens during training. In this way, we aim to preserve the original reward information to some extent, while penalizing overly long responses.

4 Experiments

4.1 Experimental Settings

4.1.1 Datasets

We select five widely used coding datasets for our comprehensive evaluation: LiveBench [46], MBPP [47], LiveCodeBench [48], CodeContests [49], and CodeForces [50]. Specifically, for CodeContests, we extract tasks with difficulty level ≤ 2 , and randomly split them into a training set of 4.5k examples and an evaluation set of 200 examples. For LiveCodeBench, we utilize version 2, which contains 511 problems. For MBPP, we use its standard test set for evaluation. The CodeForces data used in our experiments has no overlap with CodeContests [50]; we randomly sample 500 examples from it for evaluation.

4.1.2 Models and Optimization

We use Qwen2.5-7B and 14B instrut models [51] as our standard base models, and select Qwen3-4B as the base model for the long-CoT variant. At each sampling step during reinforcement learning, we generate 16 rollouts for unit tests and 16 for code using vLLM [52], with a temperature of 1.0, top- p of 1.0. For optimization, we set the learning rate to 1×10^{-6} and the KL coefficient β to 0.01. We derive the 7B and 14B models by training for 350 steps, and the 4B model by 50 steps. Specifically, for the long-CoT model, we use a lower temperature of 0.8 and apply a response-length-guided transformation to the unit test reward to improve post-training inference efficiency. We train these models using 8 A100 GPUs.

4.1.3 Test-time Scaling and Agentic Coding

Best-of-N (BoN) is the most straightforward and widely used test-time scaling and agentic coding method [14, 15], and serves as a primary metric for evaluating coding performance in our setting. Specifically, the policy generates n candidate code solutions and m unit tests, then selects the best solution based on

Table 2 Application to GPT-series models. We apply ReasonFlux-Coder-4B as a unit tester to scale GPT models serving as coders, achieving improved performance while maintaining cost efficiency. The two entries report the average API cost (Cost, in units of 10^{-3} USD) per task and the overall accuracy (Acc) for each benchmark.

Model	LB		MBPP		LCB		CC		CF	
	Cost	Acc								
4o (one-shot)	4.8	48.4	2.7	85.0	5.8	48.7	5.5	41.0	7.1	11.1
4o-mini (one-shot)	0.3	46.3	0.2	80.1	0.4	44.3	0.3	38.8	0.4	12.0
4o-mini (BoN-16)	10.8	55.4	6.7	81.5	12.0	50.7	10.1	40.6	13.1	13.5
4o-mini- CURE (BoN-16)	4.7	58.6	2.7	86.1	5.6	56.8	5.3	46.4	6.9	21.2
4.1-mini (one-shot)	0.6	65.4	0.3	88.4	0.6	68.1	1.0	51.3	1.5	22.8
4.1-mini (BoN-16)	32.5	69.5	14.7	88.2	31.9	73.4	42.2	56.9	59.5	34.1
4.1-mini- CURE (BoN-16)	9.3	74.2	4.6	89.6	9.6	74.1	15.5	58.1	24.4	35.1

the reward defined in Equation 1. We also evaluate our approach under several other agentic coding and test-time scaling pipelines [16–18]. In particular, MPSC [16] generates multiple code solutions, unit tests, and specifications per task, and selects the best solution by computing a consistency score. AlphaCodium [17] generates comprehensive unit tests to critique the generated solutions and iteratively refine the code accordingly. S* [18] organically combines iterative debugging using public unit tests and generates unit tests for pairwise discrimination, in order to select the most promising solution. See details in Appendix C.3.

4.1.4 Agentic Unit Test Generation

We also evaluate our model’s utility in an agentic unit test generation pipeline. Following prior work [25, 27], we first generate unit tests and then iteratively refine them based on their execution results on the corresponding code. See details in Appendix C.4.

4.2 Results

4.2.1 CURE Significantly Improves the Overall Coding Ability

Specifically, we apply our optimization to derive the ReasonFlux-Coder-7B and ReasonFlux-Coder-14B models from the base Qwen2.5-7B-Instruct and Qwen2.5-14B-Instruct models. Figure 2 (b–d) show the co-evolution process for unit test accuracy, code accuracy, and estimated reward, demonstrating a stable and promising co-evolving pattern. The resulting ReasonFlux-Coder models surpass their respective base models on average by 37.8% in unit test accuracy, 5.3% in one-shot code generation accuracy, and 9.0% in Best-of-N (BoN) accuracy (using 16 code solutions and 16 unit tests) (Table 1). Note that we have minimized the formatting error¹ of the base model by designing appropriate prompts (Appendix C.1), making the formatting error rate of unit tests 9% and 1% for Qwen2.5-7B and 14B Instruct models respectively, and the formatting error rate of one-shot coding as 0.08% and 0.05%, which are all significantly smaller than the improvement we gained with CURE. Notably, our models also consistently outperform the corresponding coding-supervised fine-tuned (SFT) models—Qwen2.5-Coder-Instruct—across all three metrics. Moreover, our results show that the optimization leads to consistent and robust improvements across various BoN settings (Figure 4). This indicates that the ReasonFlux-Coder models not only enhance the overall performance ceiling (when large amounts of code and unit test samples are generated), but also improve self-check efficiency in low-sample regimes (e.g., when sampling only 1 or 2 candidates).

4.2.2 Robust for Long-CoT Models and Achieves Inference Efficiency

We also evaluate CURE’s optimization on the Long-CoT model, Qwen3-4B, incorporating our response-length-guided reward transformation. The resulting ReasonFlux-Coder-4B model consistently outperforms Qwen3-4B

¹Formatting errors refer to model outputs that do not conform to the expected extraction format, resulting in failure to extract the final answer.

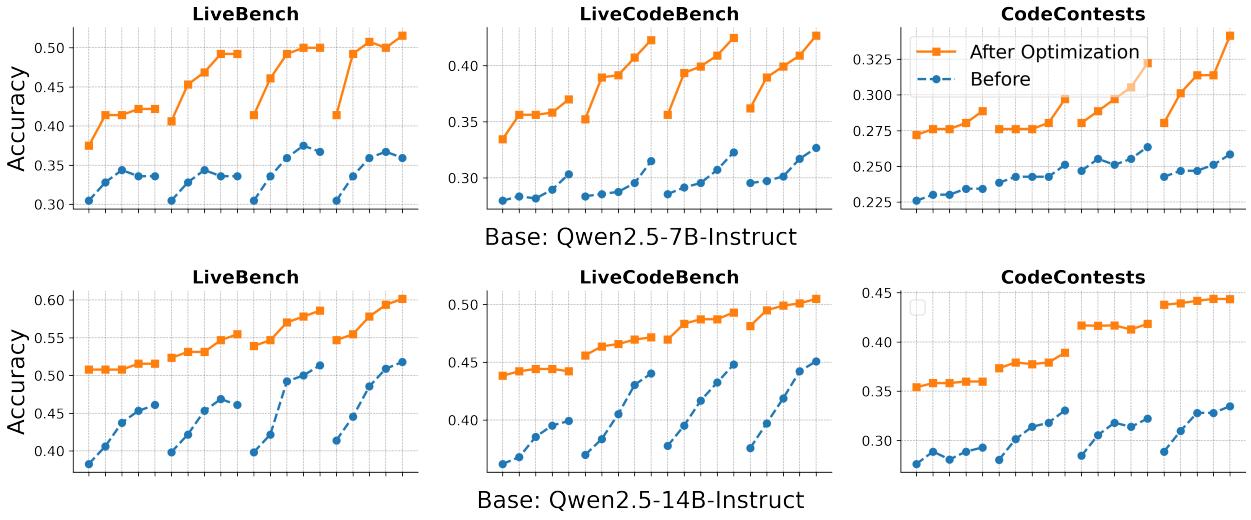


Figure 4 The BoN performance improvement after optimization on base model. Four curves (left to right) show sampling 2, 4, 8, and 16 generated codes; each curve’s five points represent 1, 2, 4, 8, and 16 generated unit tests.

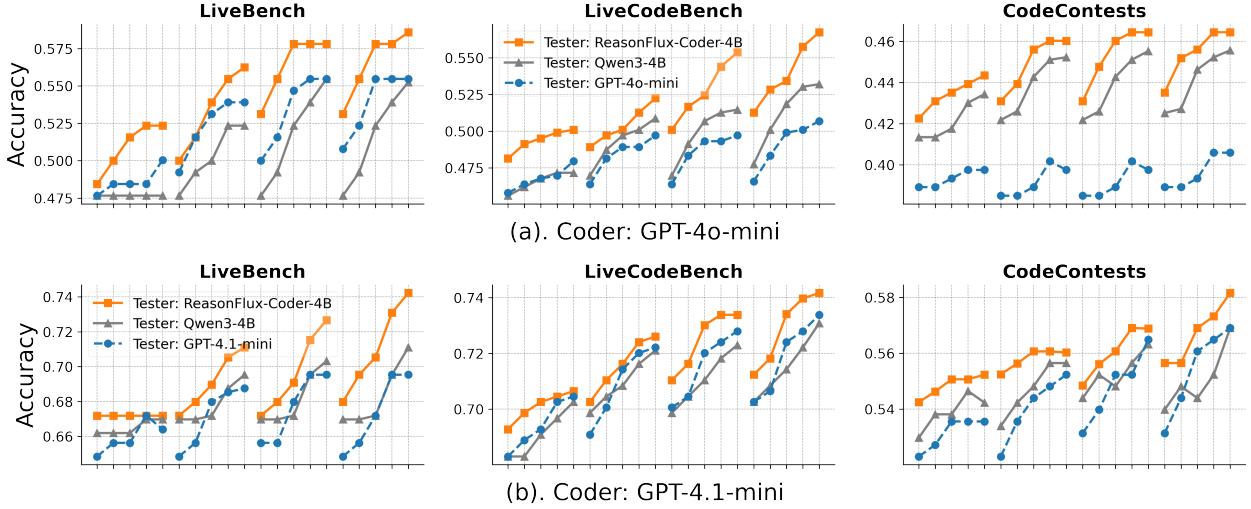


Figure 5 The BoN performance improvement across benchmarks when using ReasonFlux-Coder-4B as unit tester. Four curves (left to right) show sampling 2, 4, 8, and 16 generated codes; each curve’s five points represent 1, 2, 4, 8, and 16 generated unit tests.

in unit test accuracy, code accuracy, and BoN accuracy (Table 1). Notably, the average response length for unit test generation is reduced to 64.8% of its original length (Figure 2 (e-f)), significantly improving inference-time efficiency. We also observe that the accuracy gains for standard base models are more substantial than for long-CoT models, which aligns with the demonstrated findings [53]. Long-CoT models have already captured much of the benefit from scaling through CoT reasoning and gain less from BoN compared to standard models.

4.2.3 Boosts API-inference Models’ Performance and Cost-efficiency

We apply ReasonFlux-Coder-4B as the unit tester and evaluate its effect when paired with GPT-series models as coders, to disentangle the effects of the long-CoT coders’ strong coding ability from the unit test generation ability. We find that ReasonFlux-Coder-4B improves the BoN accuracy of GPT-4o-mini and GPT-4.1-mini by an average of 5.5% and 1.8%, respectively (Table 2). Notably, using GPT-4o-mini as the coder and

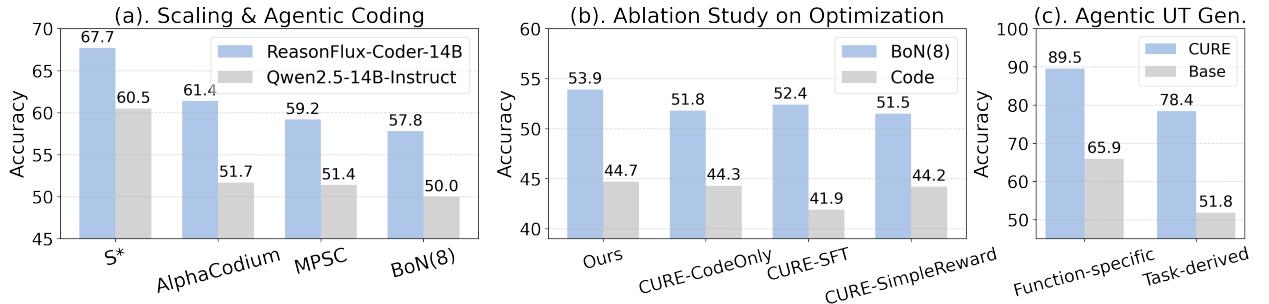


Figure 6 (a). Application of our model to various test-time scaling and agentic coding methods. We set the number of generated samples to eight in the BoN setting here. (b). Ablation study on optimization strategies and reward design choices, using Qwen2.5-14B-Instruct as the base model. All training runs are conducted with 100 optimization steps. (c). Application of our model to different agentic unit test generation tasks. “Function-specific” refers to tasks where the input includes both the problem description and the ground-truth code, whereas “Task-derived” refers to tasks where the input consists solely of the problem description. (a–c) are all evaluated on LiveBench, with Qwen2.5-14B-Instruct used as the base model.

ReasonFlux-Coder-4B as the unit tester yields a 7.0% improvement over GPT-4o one-shot performance, while also reducing cost. This demonstrates our model’s strong potential for reducing the cost of API-based pipelines. In contrast, scaling GPT-4o-mini alone results in only a 1.5% gain while incurring nearly twice the API cost compared to using ReasonFlux-Coder-4B. As shown in Figure 5, ReasonFlux-Coder-4B consistently outperforms both Qwen3-4B, GPT-4o-mini and GPT-4.1-mini as a unit tester across different BoN settings. These results demonstrate the effectiveness of using unit tests generated by the our model.

4.2.4 Facilitates Label-free RL

We have already demonstrated the utility of unit tests generated by the ReasonFlux-Coder model for solution selection. But can the model also serve as a reward model to guide reinforcement learning? We apply ReasonFlux-Coder-4B to generate unit tests as supervision for reinforcement learning training on the Qwen2.5-14B-Instruct model. Surprisingly, the resulting performance improvements are comparable to those achieved using ground-truth labeled supervision, across all three metrics: code generation accuracy, unit test accuracy, and BoN accuracy (Figure 7). This demonstrates that ReasonFlux-Coder can serve as an effective reward model not only for inference-time enhancement but also for guiding optimization during training.

4.2.5 Broad Application to Test-time Scaling and Agentic Coding Methods

In addition to the standard test-time scaling method BoN [14, 15], we also evaluate ReasonFlux-Coder-14B on several other test-time scaling and agentic methods—MPSC [16], AlphaCodium [17], and S* [18]—achieving an average improvement of 8.1% over the base model Qwen2.5-14B-Instruct (Figure 6(a)). Beyond code and unit test generation, these pipelines involve iterative refinement and debugging based on execution results, which require comprehensive coding and self-correction capabilities—capabilities our model successfully demonstrates. We further evaluate on agentic unit test generation tasks, which focus on refining unit tests based on execution results from code, and observe an average improvement of 25.1% in unit test accuracy over the base model (Figure 6(c)).

4.2.6 Ablation Study on Optimization Methods and Reward Designs

We conduct ablation studies on two aspects of the optimization process. First, we conduct experiments optimizing only the coder and using supervised fine-tuning (selecting the samples with positive rewards to fine-tune) instead of reinforcement learning. Second, we evaluate a simplified reward design for the unit test: assigning a reward of 1 if all correct codes pass, and 0 otherwise, which is an estimate of p_u . We find that CURE consistently outperforms these alternatives and remains the optimal choice across all ablations (Figure 6(b)). Optimizing only for code generation does not improve the model’s ability to produce accurate

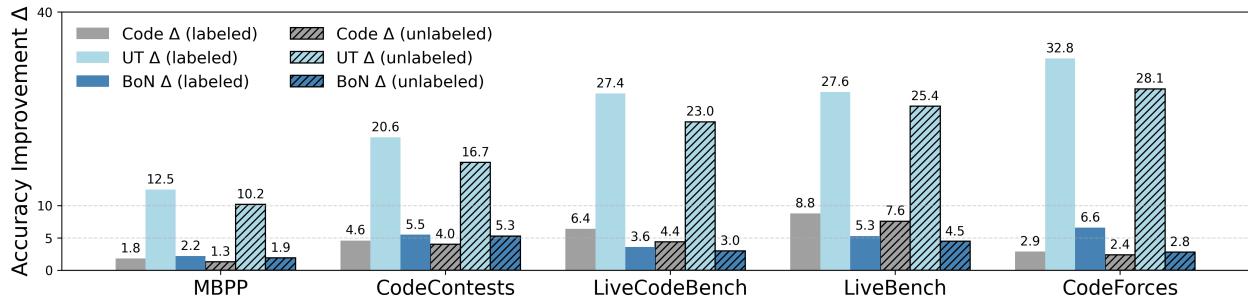


Figure 7 Accuracy improvement of Qwen2.5-14B-Instruct when trained with reinforcement learning using labeled unit tests as rewards versus using ReasonFlux-Coder-4B generated unit tests as rewards. Both models are trained for 150 steps. The BoN setting involves generating 16 samples for both code and unit tests.

unit tests and therefore falls short in self-check-based inference scaling (e.g., BoN). Supervised fine-tuning focuses solely on positive examples, ignoring informative negative samples. Moreover, using a simple reward during optimization leads to poor control over key error probabilities: p_{01} and p_{00} reach 42.2% and 14.7%, respectively. In contrast, our theoretically derived reward better constrains these values to 36.5% and 9.1%, improving the precision of selection and the overall effectiveness of solution ranking.

5 Discussions

In this paper, we propose CURE, a novel optimization framework combined with a theoretically derived reward for the unit tester, that co-evolves models’ coding and unit test generation capabilities without requiring any ground-truth code for supervision, which greatly enhances flexibility and scalability. Through extensive evaluations on five benchmarks, our results demonstrate that ReasonFlux-Coder models achieve significant performance improvements in both code generation and unit test generation tasks. Our long-CoT model ReasonFlux-Coder-4B consistently outperforms Qwen-4B while achieving significantly higher efficiency in unit test generation. Moreover, ReasonFlux-Coders proves effective in broader applications, including test-time scaling and agentic coding (8.1% improvement), agentic unit test generation (25.1% improvement), and as a reward model for reinforcement learning. We also outline future directions. Given the surprising result of using the well-trained model as reward supervision for reinforcement fine-tuning, a promising direction is to scale CURE optimization via self-supervision without any labeled data.

References

- [1] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. [arXiv preprint arXiv:2501.12948](#), 2025.
- [2] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. [arXiv preprint arXiv:2412.16720](#), 2024.
- [3] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. [arXiv preprint arXiv:2402.03300](#), 2024.
- [4] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. [arXiv preprint arXiv:2503.14476](#), 2025.
- [5] Ling Yang, Zhaochen Yu, Bin Cui, and Mengdi Wang. Reasonflux: Hierarchical llm reasoning via scaling thought templates. [arXiv preprint arXiv:2502.06772](#), 2025.
- [6] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. [arXiv preprint arXiv:2110.14168](#), 2021.
- [7] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In [The Twelfth International Conference on Learning Representations](#), 2023.
- [8] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. [arXiv preprint arXiv:2407.21787](#), 2024.
- [9] Jian Zhao, Runze Liu, Kaiyan Zhang, Zhimu Zhou, Junqi Gao, Dong Li, Jiafei Lyu, Zhouyi Qian, Biqing Qi, Xiu Li, et al. Genprm: Scaling test-time compute of process reward models via generative reasoning. [arXiv preprint arXiv:2504.00891](#), 2025.
- [10] Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. Inference-time scaling for generalist reward modeling. [arXiv preprint arXiv:2504.02495](#), 2025.
- [11] Yuxiang Zhang, Shangxi Wu, Yuqi Yang, Jiangming Shu, Jinlin Xiao, Chao Kong, and Jitao Sang. o1-coder: an o1 replication for coding. [arXiv preprint arXiv:2412.00154](#), 2024.
- [12] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, et al. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. [arXiv preprint arXiv:2402.01391](#), 2024.
- [13] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. Acecoder: An effective prompting technique specialized in code generation. [ACM Transactions on Software Engineering and Methodology](#), 33(8):1–26, 2024.
- [14] Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. Dynamic scaling of unit tests for code reward modeling. [arXiv preprint arXiv:2501.01054](#), 2025.
- [15] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. [arXiv preprint arXiv:2207.10397](#), 2022.
- [16] Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. Enhancing large language models in coding through multi-perspective self-consistency. [arXiv preprint arXiv:2309.17272](#), 2023.
- [17] Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. [arXiv preprint arXiv:2401.08500](#), 2024.
- [18] 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.
- [19] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. [arXiv preprint arXiv:2310.01798](#), 2023.

- [20] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.
- [21] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.
- [22] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.
- [23] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. A3test: Assertion-augmented automated test case generation. *Information and Software Technology*, 176:107565, 2024.
- [24] Ye Shang, Quanjun Zhang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. A large-scale empirical study on fine-tuning large language models for unit testing. *arXiv preprint arXiv:2412.16620*, 2024.
- [25] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.
- [26] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2023.
- [27] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576, 2024.
- [28] Siqi Gu, Chunrong Fang, Quanjun Zhang, Fangyuan Tian, and Zhenyu Chen. Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration. *arXiv e-prints*, pages arXiv–2408, 2024.
- [29] Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. Learning to generate unit tests for automated debugging. *arXiv preprint arXiv:2502.01619*, 2025.
- [30] Edward Yeo, Yuxuan Tong, Morry Niu, Graham Neubig, and Xiang Yue. Demystifying long chain-of-thought reasoning in llms. *arXiv preprint arXiv:2502.03373*, 2025.
- [31] Libo Wang. Dynamic chain-of-thought: Towards adaptive deep reasoning. *arXiv preprint arXiv:2502.10428*, 2025.
- [32] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.
- [33] Eduard P Enoiu, Adnan Čaušević, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Paul Pettersson. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer*, 18:335–353, 2016.
- [34] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *ACM SIGSOFT Software Engineering Notes*, 24(6):146–162, 1999.
- [35] Corina S Păsăreanu, Peter C Mehlitz, David H Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26, 2008.
- [36] Unit Tests Using Symbolic Execution. Symstra: A framework for generating object-oriented. In *Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4–8, 2005, Proceedings*, volume 3440, page 365. Springer, 2005.
- [37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [38] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741, 2023.

- [39] Yongshuai Liu, Jiaxin Ding, and Xin Liu. Ipo: Interior-point policy optimization under constraints. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 4940–4947, 2020.
- [40] Shicong Cen, Jincheng Mei, Katayoon Goshvadi, Hanjun Dai, Tong Yang, Sherry Yang, Dale Schuurmans, Yuejie Chi, and Bo Dai. Value-incentivized preference optimization: A unified approach to online and offline rlhf. *arXiv preprint arXiv:2405.19320*, 2024.
- [41] Yu Meng, Mengzhou Xia, and Danqi Chen. Simpo: Simple preference optimization with a reference-free reward. *Advances in Neural Information Processing Systems*, 37:124198–124235, 2024.
- [42] Haoran Xu, Amr Sharaf, Yunmo Chen, Weiting Tan, Lingfeng Shen, Benjamin Van Durme, Kenton Murray, and Young Jin Kim. Contrastive preference optimization: Pushing the boundaries of llm performance in machine translation. *arXiv preprint arXiv:2401.08417*, 2024.
- [43] Souradip Chakraborty, Jiahao Qiu, Hui Yuan, Alec Koppel, Furong Huang, Dinesh Manocha, Amrit Singh Bedi, and Mengdi Wang. Maxmin-rlhf: Alignment with diverse human preferences. *arXiv preprint arXiv:2402.08925*, 2024.
- [44] Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025.
- [45] Ling Yang, Ye Tian, Bowen Li, Xincheng Zhang, Ke Shen, Yunhai Tong, and Mengdi Wang. Mmada: Multimodal large diffusion language models. *arXiv preprint arXiv:2505.15809*, 2025.
- [46] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, et al. Livebench: A challenging, contamination-free llm benchmark. *arXiv preprint arXiv:2406.19314*, 2024.
- [47] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- [48] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.
- [49] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
- [50] 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.
- [51] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.
- [52] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- [53] Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *arXiv preprint arXiv:2504.13837*, 2025.
- [54] Qwen Team. Qwq-32b: Embracing the power of reinforcement learning, March 2025.

Appendix

A Proofs of Theoretical Results

Proof. (of Theorem 3.1)

Set-up and intuition. For every test index k ($1 \leq k \leq m$) define

$$X_k := \underbrace{\mathcal{B}_{j_1k}}_{\text{outcome on correct } s_{j_1}} - \underbrace{\mathcal{B}_{j_2k}}_{\text{outcome on wrong } s_{j_2}} \in \{-1, 0, 1\}.$$

Positive X_k means the correct solution beats the wrong one on test k , $X_k = 0$ means they tie, and $X_k = -1$ means the wrong solution wins. The reward difference after m tests is $D_m := \sum_{k=1}^m X_k = \mathcal{R}_{s_{j_1}} - \mathcal{R}_{s_{j_2}}$. Our target event $\{\mathcal{R}_{s_{j_1}} > \mathcal{R}_{s_{j_2}}\}$ coincides with $\{D_m > 0\}$, so we analyse the sign of D_m .

Single ground-truth test. Assume a particular test u_k is correct (i.e. $c_{u_k} = 1$). Because a correct solution always passes a correct test ($p_{11} = 1$) we have $\mathcal{B}_{j_1k} = 1$ with probability 1. Conversely, an incorrect solution passes that same correct test with probability p_{01} , so

$$P[\mathcal{B}_{j_2k} = 0] = 1 - p_{01}.$$

Hence

$$P[X_k = 1] = P[\mathcal{B}_{j_1k} = 1, \mathcal{B}_{j_2k} = 0] = 1 - p_{01}, \quad P[X_k \leq 0] = p_{01}.$$

Therefore $P(X_k > 0) = 1 - p_{01}$, which proves the first statement.

Distribution of X_k . Let $I_k := \mathbf{1}\{c_{u_k} = 1\}$ indicate whether the k -th test is correct. By the data-generation assumption,

$$P(I_k = 1) = p_u, \quad P(I_k = 0) = 1 - p_u.$$

Case $I_k = 1$: we are in the setting of Step 1, so

$$P(X_k = 1 | I_k = 1) = 1 - p_{01}, \quad P(X_k = -1 | I_k = 1) = 0, \quad P(X_k = 0 | I_k = 1) = p_{01}.$$

Case $I_k = 0$: the test itself is wrong. Now a correct solution fails with probability 1 ($p_{10} = 0$), while the incorrect solution can pass spuriously with probability p_{00} . Thus

$$P(X_k = 1 | I_k = 0) = 0, \quad P(X_k = -1 | I_k = 0) = p_{00}, \quad P(X_k = 0 | I_k = 0) = 1 - p_{00}.$$

Applying the law of total probability yields the unconditional mass

$$P(X_k = 1) = p_u(1 - p_{01}) + (1 - p_u) \cdot 0 = p_u(1 - p_{01}),$$

$$P(X_k = -1) = (1 - p_u)p_{00},$$

$$P(X_k = 0) = 1 - P(X_k = \pm 1).$$

Denote

$$\mu := E[X_k] = 1 \cdot P(X_k = 1) + (-1) \cdot P(X_k = -1) = p_u(1 - p_{01}) - (1 - p_u)p_{00},$$

$$\sigma_k^2 := \text{Var}(X_k) = E[X_k^2] - \mu^2 = P(X_k = 1) + P(X_k = -1) - \mu^2.$$

All X_k 's are i.i.d. because the unit tests are generated independently and the solutions themselves are fixed.

Convergence Analysis. Write the empirical mean $\bar{X}_m := \frac{1}{m} \sum_{k=1}^m X_k$. Since $E[X_k] = \mu$ and $E[|X_k|] \leq 1$, the strong law of large numbers (SLLN) tells us

$$\bar{X}_m \xrightarrow{\text{a.s.}} \mu \quad (m \rightarrow \infty).$$

But $D_m/m = \bar{X}_m$, hence

$$\frac{D_m}{m} \xrightarrow{\text{a.s.}} \mu.$$

Consequences.

- If $\mu > 0$, then $\frac{D_m}{m}$ is eventually positive almost surely, so $P(D_m > 0) \rightarrow 1$.
- If $\mu < 0$, $\frac{D_m}{m}$ is eventually negative a.s., so $P(D_m > 0) \rightarrow 0$.
- If $\mu = 0$, $\frac{D_m}{\sqrt{m}}$ has variance σ_k^2 and remains $O_p(1)$, whence $P(D_m > 0) \rightarrow \frac{1}{2}$ by symmetry of the CLT limit distribution.

Explicit tail bound for finite m , assuming $\mu > 0$.

Recall $X_k \in \{-1, 0, 1\}$ and $E[X_k] = \mu > 0$. Define the centred variables

$$Z_k := X_k - \mu \quad (1 \leq k \leq m),$$

so that $E[Z_k] = 0$. Because $-1 \leq X_k \leq 1$, we have $-1 - \mu \leq Z_k \leq 1 - \mu$. Since $\mu \in (0, 1)$, we have

$$|Z_k| \leq 2 \quad \text{almost surely.}$$

Now we apply Hoeffding's additive inequality. Let Z_1, \dots, Z_m be independent, centred random variables satisfying $|Z_k| \leq c$ a.s. for every k . For any $t > 0$,

$$P\left(\sum_{k=1}^m Z_k \leq -t\right) \leq \exp\left(-\frac{t^2}{2mc^2}\right). \quad (\text{Hoeffding})$$

Here $c = 2$, By definition

$$D_m = \sum_{k=1}^m X_k = \sum_{k=1}^m (Z_k + \mu) = m\mu + \sum_{k=1}^m Z_k.$$

Hence

$$\{D_m \leq 0\} = \left\{\sum_{k=1}^m Z_k \leq -m\mu\right\}.$$

Substituting $t = m\mu$ and $c = 2$ into (Hoeffding) gives

$$P(D_m \leq 0) = P\left(\sum_{k=1}^m Z_k \leq -m\mu\right) \leq \exp\left(-\frac{(m\mu)^2}{8m}\right) = \exp\left(-\frac{\mu^2 m}{8}\right).$$

Finally,

$$P(D_m > 0) = 1 - P(D_m \leq 0) \geq 1 - \exp\left(-\frac{\mu^2 m}{8}\right),$$

yielding the advertised exponential guarantee. □

Proposition A.1. *Given the execution table, the individual reward for unit test u_k can be estimated by*

$$\mathcal{R}_{u_k}^* = - \sum_{l=1}^n (1 - \mathcal{I}_{s_l}) \mathcal{B}_{l,k}^* + \left(\prod_{l=1}^n \mathcal{I}_{s_l} \mathcal{B}_{l,k}^* \right) \left(\sum_{l=1}^n (1 - \mathcal{I}_{s_l}) \right).$$

Proof. (of Proposition A.1) We use the following estimation to detect if a code solution s_j is correct or not:

$$\mathcal{I}_{s_j} = \prod_{l=1}^{t_q} \mathcal{B}_{j,m+l}^*.$$

So the accuracy of u_k, \hat{p}_u , can be estimated by

$$\prod_{l=1}^n \mathcal{I}_{s_l} \mathcal{B}_{l,k}^*.$$

Similarly, we can obtain estimator $1 - \hat{p}_{01}$ and \hat{p}_{00} as

$$\sum_{l=1}^n (1 - \mathcal{I}_{s_l})(1 - \mathcal{B}_{l,k}^*) / \sum_{l=1}^n (1 - \mathcal{I}_{s_l}), \quad \sum_{l=1}^n (1 - \mathcal{I}_{s_l})\mathcal{B}_{l,k}^* / \sum_{l=1}^n (1 - \mathcal{I}_{s_l}),$$

respectively. Finally, we derive $\hat{\mu} = \widehat{p_u}(1 - \hat{p}_{01}) - (1 - \hat{p}_u)\hat{p}_{00}$:

$$\begin{aligned} & \left[\left(\prod_{l=1}^n \mathcal{I}_{s_l} \mathcal{B}_{l,k}^* \right) \left(\sum_{l=1}^n (1 - \mathcal{I}_{s_l})(1 - \mathcal{B}_{l,k}^*) \right) - \left(1 - \prod_{l=1}^n \mathcal{I}_{s_l} \mathcal{B}_{l,k}^* \right) \left(\sum_{l=1}^n (1 - \mathcal{I}_{s_l})\mathcal{B}_{l,k}^* \right) \right] / \sum_{l=1}^n (1 - \mathcal{I}_{s_l}) \\ &= \left[- \sum_{l=1}^n (1 - \mathcal{I}_{s_l})\mathcal{B}_{l,k}^* + \left(\prod_{l=1}^n \mathcal{I}_{s_l} \mathcal{B}_{l,k}^* \right) \left(\sum_{l=1}^n (1 - \mathcal{I}_{s_l}) \right) \right] / \sum_{l=1}^n (1 - \mathcal{I}_{s_l}). \end{aligned}$$

Given that $\sum_{l=1}^n (1 - \mathcal{I}_{s_l})$ is constant for different k , we have our final reward for u_k :

$$- \sum_{l=1}^n (1 - \mathcal{I}_{s_l})\mathcal{B}_{l,k}^* + \left(\prod_{l=1}^n \mathcal{I}_{s_l} \mathcal{B}_{l,k}^* \right) \left(\sum_{l=1}^n (1 - \mathcal{I}_{s_l}) \right).$$

□

B Additional Experimental Results

Table 3 This is the error analysis table corresponding to Table 1. Each cell reports the “accuracy improvement over the base model (standard error).” Note that the accuracies of unit test and code are evaluated over 16 independent runs, whereas BoN scaling is computationally intensive, so we report BoN accuracy based on a single run per benchmark.

Model	LiveBench		MBPP		LiveCodeBench		CodeContests		CodeForces	
	UT	Code								
ReasonFlux-Coder-14B	0.455 (0.008)	0.111 (0.0041)	0.188 (0.010)	0.022 (0.0025)	0.457 (0.012)	0.070 (0.0029)	0.422 (0.016)	0.065 (0.0030)	0.616 (0.038)	0.048 (0.0011)
ReasonFlux-Coder-7B	0.283 (0.007)	0.060 (0.0035)	0.436 (0.035)	0.039 (0.0021)	0.311 (0.011)	0.043 (0.0030)	0.359 (0.016)	0.047 (0.0036)	0.267 (0.027)	0.028 (0.0014)
ReasonFlux-Coder-4B	0.478 (0.009)	0.021 (0.0034)	0.068 (0.010)	0.011 (0.0021)	0.359 (0.021)	0.014 (0.0025)	0.286 (0.004)	0.023 (0.0027)	0.117 (0.024)	0.025 (0.0014)

Table 4 Response length (in tokens) of Qwen3-4B and ReasonFlux-Coder-4B in unit test generation task, corresponding to Figure 2 (e).

Benchmark	Qwen3-4B	ReasonFlux-Coder-4B
LiveBench	4711	3067
MBPP	2419	1611
LiveCodeBench	4326	2837
CodeContests	6086	3899
CodeForces	7309	4706

C Details of Experiments

C.1 Prompt Design

This is the prompt for code generation:

Task 1

```
PROMPT = """ <|im_start|>You are a helpful assistant help user solve problems.  
<|im_end|> <|im_start|>User: You need to think first then write python script. Use  
input() to input and print() to output. This is the problem: {{problem}} <|im_-  
end|> <|im_start|>Assistant: """
```

This is the prompt for unit test generation:

Unit test prompt

```
PROMPT = """ <|im_start|>You are a helpful assistant help user generate test  
examples for coding tasks. <|im_end|> <|im_start|>User: Given a coding task,  
instead of providing the final script, your task is to generate a new test example  
(both input, output and explanation). This is the problem: {{problem}} You need  
to provide a new test example. A good test example should be completely accurate  
and conform to the problem's format requirements, while also possessing enough  
discriminative power to distinguish correct code from incorrect code. Before  
providing a test example, you must think carefully and reason step by step to  
derive an input and output you are very confident are correct. For example, start  
by designing an input you can reliably handle, then compute the output step by step.  
If you're unsure about the output, revise or re-design the input to ensure accuracy.  
Finally, after completing these previous thinking and derivation steps, you MUST  
put your final test example in the following format:  
**Test Input:** "input here"  
**Test Output:** "output here"  
**Explanation:** explanation here. <|im_end|> <|im_start|>Assistant: """
```

C.2 Preprocess Data

In our experiments, we adopt the stdio format for inputs and outputs, which is the standard input/output format used in LiveBench [46], LiveCodeBench [48], CodeContests [49], and CodeForces [50]. However, some tasks in LiveBench and LiveCodeBench, as well as all tasks in MBPP [47], originally use a functional input/output format. For consistency and ease of evaluation, we convert these functional formats to stdio. Specifically, the conversion rule is as follows: each variable is placed on a separate line, and lists are flattened into space-separated values on a single line, as illustrated in the following example:

Input and output format example

```
# functional format:  
  
assert work("a", [1, 2, 3]) == 2  
  
# stdio format:  
  
Input:  
  
a  
  
1 2 3
```

Output:

2

For evaluation, we directly use the ground-truth code provided in CodeContests and MBPP. For Codeforces, LiveCode, and LiveCodeBench, we collect code generated by QwQ-32B [54] (using BoN with a maximum of 3 samples) that passes all ground-truth tests to serve as the ground-truth code.

C.3 Test-time Scaling and Agentic Coding

We introduce how we apply MPSC [16], AlphaCodium [17] and S* [18] in our test-time scaling and agentic coding applications.

C.3.1 MPSC

For each task, we generate 8 samples of code, unit tests, and specifications (A specification is a pair of functions—a pre-condition and a post-condition—that define the valid input space and the expected input-output behavior of a program, serving as a formal description of its intended functionality.). We then follow the iterative optimization algorithm to derive the consistency scores, which will be used to identify the optimal code solution.

C.3.2 AlphaCodium

Following their procedure, we generate 8 code solutions per task using reasoning over public tests, along with 8 corresponding unit tests. Each code solution undergoes 2 iterations of refinement based on execution results from the public tests, followed by another 2 iterations based on execution results using the generated unit tests. Specifically, the refinement step asks the model to check the unit tests, code, and execution results, and then decide whether to refine or not.

C.3.3 S*

We generate 8 code solutions and apply 4 iterations of self-debugging using public tests to obtain 8 refined versions. Note that the debugging is based on the execution results of ground-truth unit tests, so we directly ask the model to modify the code if the execution fails. The final solution is selected via their pairwise comparison method, using generated unit tests for evaluation.

C.4 Agentic Unit Test Generation Methods

We first introduce the development of unit test generation methods. Traditional approaches rely on software analysis techniques such as search-based methods (Evosuite) [20], random testing (Randoop) [21], model checking [33, 34], and symbolic execution [35, 36]. To improve scalability, neural machine translation-based methods were introduced [22, 23]. Specifically, AthenaTest [22] employs a BART model, while A3Test [23] uses a PLBART model with post-processing for improved accuracy. With the recent advancements in LLMs, prompt-based agentic methods such as ChatTester [25], ChatUniTest [27], and TestART [28] have demonstrated superior performance, further highlighting the potential of training LLMs for unit test generation. In this paper, we adopt the iterative refinement and generation pipeline used in ChatTester and ChatUniTest.

Detailed Approach for Agentic Unit Test Generation in Our Experiments For the function-specific unit test generation task, where the input includes both the task description and ground-truth code, we prompt the policy model to generate both the code and the corresponding unit test. We then execute them to obtain the test result. If the test passes, the process proceeds to the next iteration; if it fails, we provide the code, unit test, and execution feedback to the policy model, which decides whether to refine the unit test. The maximum number of iterations is set to 3.

C.5 Generated Unit Test Examples

C.5.1 Examples for ReasonFlux-Coder-14B

Task 1

You are given an array of integers `nums` of length `n`.

The **cost** of an array is the value of its first element. For example, the cost of `[1, 2, 3]` is 1, while the cost of `[3, 4, 1]` is 3.

You need to divide `nums` into 3 disjoint contiguous subarrays.

Return the **minimum possible sum** of the cost of these subarrays.

Example 1:

Input:

`1 2 3 12`

Output:

`6`

Explanation: The best possible way to form 3 subarrays is: `[1]`, `[2]`, and `[3, 12]` at a total cost of $1 + 2 + 3 = 6$.

Other possible ways include:

- `[1], [2, 3], [12]` with a cost of $1 + 2 + 12 = 15$
- `[1, 2], [3], [12]` with a cost of $1 + 3 + 12 = 16$

Generated unit test for task 1

1. Start with a simple input and compute the output step by step.

2. Let's take the input as `[2, 3, 4, 1]`.

3. According to the rules:

- The array needs to be divided into three disjoint contiguous subarrays.
- The cost is defined as the first element of each subarray.

Let's think through the possible ways to divide the array `[2, 3, 4, 1]` into three subarrays:

Option 1: `[2], [3], [4, 1]`

Cost: $2 + 3 + 4 = 9$

Option 2: `[2, 3], [4], [1]`

Cost: $2 + 4 + 1 = 7$

Option 3: `[2], [3, 4], [1]`

Cost: $2 + 3 + 1 = 6$

After comparing all options, the minimum cost is from `[2], [3, 4], [1]` yielding a cost of 6.

Test Input:

`2 3 4 1`

Test Output:

`6`

Explanation: The optimal way to divide the array `[2, 3, 4, 1]` is:

- `[2]` with a cost of 2,
- `[3, 4]` with a cost of 3,
- `[1]` with a cost of 1.

Total cost: $2 + 3 + 1 = 6$.

Task 2

You are given a 0-indexed integer array `batteryPercentages` having length `n`, denoting the battery percentages of `n` 0-indexed devices.

Your task is to test each device `i` in order from 0 to `n - 1`, by performing the following test operations:

- If `batteryPercentages[i]` is greater than 0:
 - Increment the count of tested devices.

- Decrease the battery percentage of all devices with indices j in the range $[i + 1, n - 1]$ by 1, ensuring their battery never goes below 0, i.e., $\text{batteryPercentages}[j] = \max(0, \text{batteryPercentages}[j] - 1)$.
- Move to the next device.
- Otherwise, move to the next device without performing any test.

Return an integer denoting the number of devices that will be tested after performing the test operations in order.

Example 1:

Input:

1 1 2 1 3

Output:

3

Explanation:

- At device 0: $\text{batteryPercentages}[0] > 0$, so count = 1. $\text{batteryPercentages}$ becomes [1,0,1,0,2]
- At device 1: $\text{batteryPercentages}[1] == 0$, skip.
- At device 2: $\text{batteryPercentages}[2] > 0$, count = 2. $\text{batteryPercentages}$ becomes [1,0,1,0,1]
- At device 3: $\text{batteryPercentages}[3] == 0$, skip.
- At device 4: $\text{batteryPercentages}[4] > 0$, count = 3.

So the answer is 3.

Generated unit test for task 2

1. Start with a case where devices alternate between having battery and not having battery to ensure we understand the decrement logic correctly.
2. Let's consider the input [1, 0, 2, 0, 3].

Analysis:

- At device 0: $\text{batteryPercentages}[0] = 1 > 0$
 - Increment tested count: 1
 - Decrement indices 1 to 4 $\rightarrow \text{batteryPercentages}$ becomes [1,0,1,0,2]
- At device 1: $\text{batteryPercentages}[1] = 0 \rightarrow$ skip
- At device 2: $\text{batteryPercentages}[2] = 1 > 0$
 - Increment tested count: 2
 - Decrement indices 3 to 4 $\rightarrow \text{batteryPercentages}$ becomes [1,0,1,0,1]
- At device 3: $\text{batteryPercentages}[3] = 0 \rightarrow$ skip
- At device 4: $\text{batteryPercentages}[4] = 1 > 0 \rightarrow$ increment tested count: 3

Test Input:

1 0 2 0 3

Test Output:

3

Explanation:

- Step-by-step execution results in final $\text{batteryPercentages}$: [1,0,1,0,1]
- Devices tested: 0, 2, and 4 \rightarrow total = 3

Task 3

You are given an array `nums` consisting of positive integers.

Return the total frequencies of elements in `nums` such that those elements all have the maximum frequency.

The frequency of an element is the number of occurrences of that element in the array.

Example 1:

Input:

1 2 2 3 1 4

Output:

4

Explanation: The elements 1 and 2 have a frequency of 2, which is the maximum frequency in the array.

So the number of elements in the array with maximum frequency is $2 \text{ (from 1)} + 2 \text{ (from 2)} = 4$.

Generated unit test for task 3

Let's consider an input where we have multiple elements with different frequencies. For instance, an array with several 2s and 3s, and a few 1s and 4s.

Test Input:

1 2 2 3 3 3 4 4 4 4

Step-by-Step Computation:

- Count the frequency of each element:
 - 1: 1 time
 - 2: 2 times
 - 3: 3 times
 - 4: 4 times
- Maximum frequency = 4 (from element 4)
- Only element 4 has this maximum frequency.

Test Output:

4

Explanation: The element 4 appears 4 times, and this is the highest frequency in the array. Therefore, the total number of elements with the maximum frequency is 4.

Task 4

You are given an array of integers `nums` of length `n`.

The **cost** of an array is the value of its first element. For example, the cost of `[1, 2, 3]` is 1, while the cost of `[3, 4, 1]` is 3.

You need to divide `nums` into 3 disjoint contiguous subarrays.

Return the **minimum possible sum** of the cost of these subarrays.

Example 1:**Input:**

1 2 3 12

Output:

6

Explanation:

- The best way to partition is `[1], [2], [3, 12]` with a total cost: $1 + 2 + 3 = 6$.
- Other possibilities:
 - `[1], [2, 3], [12]` → cost = $1 + 2 + 12 = 15$
 - `[1, 2], [3], [12]` → cost = $1 + 3 + 12 = 16$

Generated unit test for task 4

1. Designing the Input:

- Use an array of length 5 to explore multiple partitioning scenarios.
- Mix small and large integers for diversity.

Consider the array `[4, 1, 5, 2, 6]`.

2. Computing the Output: Try all valid 3-way partitions:

- `[4], [1], [5, 2, 6]` → cost = $4 + 1 + 5 = 10$
- `[4], [1, 5], [2, 6]` → cost = $4 + 1 + 2 = 7$

- [4], [1, 5, 2], [6] → cost = 4 + 1 + 6 = 11
- [4, 1], [5], [2, 6] → cost = 4 + 5 + 2 = 11
- [4, 1], [5, 2], [6] → cost = 4 + 5 + 6 = 15
- [4, 1, 5], [2], [6] → cost = 4 + 2 + 6 = 12

Minimum cost = 7 from partition [4], [1, 5], [2, 6].

Test Input:

4 1 5 2 6

Test Output:

7

Explanation: The optimal partition is:

- [4] → cost = 4
- [1, 5] → cost = 1
- [2, 6] → cost = 2

Total cost: 4 + 1 + 2 = 7.

C.5.2 Examples for ReasonFlux-Coder-4B (long-CoT)

Task 5

You are given an array of integers `nums` of length `n`.

The **cost** of an array is the value of its first element. For example, the cost of [1, 2, 3] is 1 while the cost of [3, 4, 1] is 3.

You need to divide `nums` into 3 disjoint contiguous subarrays.

Return the **minimum possible sum** of the cost of these subarrays.

Example 1:

Input:

1 2 3 12

Output:

6

Explanation: The best possible way to form 3 subarrays is: [1], [2], and [3, 12] at a total cost of 1 + 2 + 3 = 6. The other possible ways to form 3 subarrays are:

- [1], [2, 3], [12] → cost = 1 + 2 + 12 = 15
- [1, 2], [3], [12] → cost = 1 + 3 + 12 = 16

Generated unit test for task 5

<think>

Okay, let's see. I need to create a new test case for this problem. The problem is about selecting up to L subjects without overlapping, maximizing happiness. So I need to make sure the input follows the constraints and that the output is correct.

First, I need to make sure the input is valid. Let's start by choosing small values for N, M, L that are within the constraints. Let's say N is 3, M is 3, L is 2. That's manageable.

Now, I need to create courses that don't overlap and can be selected without conflict. Let me think of some courses that are non-overlapping. For example:

Course 1: di=0 (Monday), ai=1, ki=1 (so runs during 1st period). That's 1 period. Then course 2 could be di=0, ai=2, ki=1. Then course 3 could be di=1, ai=1, ki=1. Then they can be selected as they don't overlap. But since L is 2, I need to select 2 courses that give maximum happiness.

Let me think of their ti values. Let's say course 1 has ti=5, course 2 has ti=4, course 3 has ti=3. Then the max would be 5+4=9. But maybe there's a better selection.

Alternatively, if there's a course that overlaps with others. Wait, but in this case, the courses are not overlapping. But maybe in the test case, it's possible to select two courses that don't conflict, but another combination gives higher happiness.

Alternatively, maybe there's a scenario where overlapping courses can't be selected. So I need to design courses that allow different combinations.

Another approach: Let me think of two courses that can be chosen. For instance, course 1 ($d=0$, $a=1$, $k=1$) and course 2 ($d=1$, $a=1$, $k=1$). But course 1 is on Monday, $a=1$, so runs from Monday period 1, and course 2 is Tuesday period 1. So they are not overlapping. They can be both selected if L is 2. So total happiness is sum of their t_i .

But maybe another course has higher t_i . Let me try to make a test case where the best selection is two courses that don't overlap, but perhaps have higher t_i .

Let's design a scenario. For example, $N=3$ (days 0-4?), wait N is the number of periods per day. Wait the problem says that classes are from Monday to Friday (5 days) and for each day, there are N periods (from 1 to N), and courses start at a_i and run k_i frames (consecutive). So for example, if a_i is 1 and k_i is 2, the course runs from 1 to 2 (periods 1 and 2) of that day.

So for the test case, I need to make sure that courses are not overlapping. For example, a course that starts on Monday ($d=0$) at $a=1$, $k=1$ (so runs Monday period 1), and a course that starts on Tuesday ($d=1$) at $a=2$, $k=2$ (so runs Tuesday period 2 and 3) — they don't overlap.

But to make this work, let's create a few courses that can be selected and have different t_i values.

Let me try to create a test case:

$N=3$ (so each day has 3 periods)

$M=3$, $L=2$.

Course1: $d=0$ (Monday), $a=1$, $k=1$, $t=5$. It runs from Monday period 1.

Course2: $d=0$ (Monday), $a=2$, $k=1$, $t=6$. It runs from Monday period 2.

Course3: $d=1$ (Tuesday), $a=1$, $k=1$, $t=4$. It runs on Tuesday period 1.

Now, if L is 2, the best selection is course1 and course3 ($t=5+4=9$) or course2 and course3 ($6+4=10$). But course1 and course2 overlap on Monday, so they cannot be selected together. So the maximum is course2 and course3, which sum to $6+4=10$.

But wait, if course2 and course3 are selected, then course2 is on Monday period 2, course3 is on Tuesday period 1. They don't conflict, so that's allowed.

So in that case, the output would be 10.

But maybe that's the test case. However, I need to make sure that the input is correct.

Let me structure the input as:

$N=3$, $M=3$, $L=2$

Then three courses:

$d1=0$, $a1=1$, $k1=1$, $t1=5$

$d2=0$, $a2=2$, $k2=1$, $t2=6$

$d3=1$, $a3=1$, $k3=1$, $t3=4$

So the input is:

3 3 2

0 1 1 5

0 2 1 6

1 1 1 4

The output would be $6+4=10$?

Yes, that would be the maximum. Because course2 and course3 can be taken.

But what if another combination? For example, course2 ($t=6$) and course3 ($t=4$) is 10.

Alternatively, course2 and course3 is allowed. Then that's the correct answer.

Now, the test input would be:

Test Input:

3 3 2

0 1 1 5

0 2 1 6

1 1 1 4

Then the output is 10. So that's a valid test case.

But I need to check whether the courses don't overlap. Course1 is Monday 1, course2 is Monday 2. So they don't conflict. However, course2 and course1 are on the same day, but different periods, so they can be taken together if L is larger than 1. But in this case, L is 2. But wait, the problem says that each student can take up to L subjects that do not overlap.

But in the above example, course1 and course2 are on the same day, same day, but different periods, but since the course starts from a_i and runs for k_i frames, they are consecutive. So course1 is Monday, period 1. It's for 1 frame. So the course is from Monday period 1. Course2 is Monday period 2, so it runs for 1 frame. So they are on the same day but not overlapping. So they can be selected as two different courses. But in that case, if the $L=2$, then the maximum would be $5+6=11$? Wait, that's a mistake in the previous analysis. Oh, right! Because in this case, course1 and course2 can both be selected if they are on the same day, and not overlapping. So the maximum would be $5+6=11$.

But then that would be a better selection. So the test case needs to have courses that are non-overlapping. So why would the answer be 10? That suggests that I made a mistake in the test case.

Wait, this is a problem. Let me think again.

So if the courses are on the same day, but their periods are non-overlapping, then they can be taken together. For example, course1 is on Monday, period 1, runs for 1 frame. Course2 is on Monday, period 2, runs for 1 frame. So they can be selected together. So the maximum is course1 + course2 ($t=5+6=11$) instead of course2 and course3 ($t=6+4=10$). So this is a problem.

Ah, I think my earlier test case is flawed because the maximum would be $5+6 = 11$, which is higher than the previous thought. That suggests I need to adjust the test case to avoid such cases.

So, what's the correct way to create a test case where the maximum is not selecting overlapping courses on the same day.

Let me re-construct the test case. Let's have courses that cannot be selected together, even if they are on the same day.

For example, course1 is on Monday ($d=0$), $a_i=1$, $k_i=2$. That would cover periods 1 and 2 on Monday. Then course2 is on Monday ($d=0$), $a_i=2$, $k_i=1$. So the course1 starts on $a_i=1$, runs for 2 frames. So it's periods 1 and 2. So course2 starts on period 2 and runs for 1 frame. That would overlap with course1, so they can't be taken together. So the user can't take course1 and course2 together.

So for that scenario, if L is 2, the maximum would be course1 ($t=5$) + course3 ($t=4$) = 9.

Let me try this.

Let's have:

$N=3$ (so periods are 1-3 each day)

$M=3$ courses:

Course1: $d=0$ (Monday), $a_i=1$, $k_i=2$, $t=5$. Runs on Monday 1 and 2.

Course2: $d=0$ (Monday), $a_i=2$, $k_i=1$, $t=6$. So starts on Monday 2. It overlaps with course1.

So cannot be selected with course1.

Course3: $d=1$ (Tuesday), $a_i=1$, $k_i=1$, $t=4$. No overlap with course1. So if L is 2, then the best would be course1 (5) + course3 (4) = 9. Or course2 (6) + course3 (4) = 10, but course2 is on Monday 2, and course3 is on Tuesday. So that's allowed. But course1 and course2 can't be selected together. So the maximum is $6+4=10$. Or course2 and course3.

But that's better than course1 and course3.

So in that case, the correct answer would be 10.

Now, this scenario makes sense. So the input would be:

$N=3$, $M=3$, $L=2$.

Then three courses:

0 1 2 5

0 2 1 6

1 1 1 4

Now, the courses are as follows:

course1: $d=0$ (Monday), starts period 1, runs for 2 frames (periods 1 and 2). So course1 covers Monday periods 1 and 2.

course2: d=0, a=2, k=1: period 2 of Monday, runs 1 frame. So this overlaps with course1.
course3: d=1 (Tuesday), period 1, runs 1 frame.

So, possible selections:

- course2 and course3: $t=6+4=10$.
- course3 and course1: $5+4=9$.
- course2 and course3 is better.

So the correct answer would be 10.

In this case, the test case would be:

Test Input:

```
3 3 2  
0 1 2 5  
0 2 1 6  
1 1 1 4
```

Then the Output is 10.

So this should be a valid test case, and the output is 10. That test case would allow the code to correctly distinguish between overlapping and non-overlapping courses.

I need to verify that each course's ai and ki are valid. For example, for course1, ai is 1, ki is 2. Then $ai+ki-1$ is $1+2-1=2$. Which is ≤ 3 (since N=3). So yes, valid.

For course2, ai=2, ki=1: $ai + ki - 1 = 2+1-1=2$, which is within N=3.

So that's correct.

So this test case should be valid and the output is 10.

Therefore, the final test case is: </think>

Test Input:

```
3 3 2  
0 1 2 5  
0 2 1 6  
1 1 1 4
```

Test Output:

```
10
```

Explanation: We have three courses. The first course runs for two periods (Monday, periods 1 and 2) and provides a happiness of 5. The second course runs for one period (Monday, period 2) and provides a happiness of 6, but overlaps with the first course. The third course runs for one period (Tuesday, period 1) and provides 4. The best combination is the second and third courses, which are non-overlapping. Their total happiness is $6 + 4 = 10$, which is the maximum possible.