Co-Evolving LLM Coder and Unit Tester via Reinforcement Learning

Yinjie Wang[1*], **Ling Yang**[2*†], **Ye Tian**[3], **Ke Shen**[4], **Mengdi Wang**[2]
[1]University of Chicago, [2]Princeton University, [3]Peking University, [4]ByteDance Seed
Project: https://github.com/Gen-Verse/CURE

Abstract

Mathematical reasoning in large language models has been successfully incentivized through reinforcement learning with verifiable rewards, leading to improved one-shot precision. In this work, we turn our focus to the coding domain. Beyond one-shot precision, we highlight unit test generation as another key factor for enhancing coding ability, since accurate unit tests are essential for enabling self-checking and self-correction during inference. Traditional approaches for fine-tuning LLMs on unit test generation rely heavily on ground-truth code solutions in the training data. 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. Through extensive evaluations, we demonstrate that our CURE models, derived from base models of varying sizes, excel in both code generation and unit test generation. They naturally extend to downstream tasks such as test-time scaling—achieving a 6.2% improvement over the base model—and agentic unit test generation, with a 25.1% improvement. Our CURE-4B model consistently outperforms Qwen3-4B while achieving 64.8% inference efficiency in unit test generation. Notably, we also find that the CURE model can serve as an effective reward model for reinforcement learning on base models, even in the absence of any labeled supervision.

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 [14, 18, 39, 49], as well as through test-time scaling methods guided by reward-based selection strategies [7, 24, 3, 53, 27], 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 [52, 8, 22] and test-time scaling or agentic coding pipelines [28, 5, 16, 35, 21]. Second, generated unit tests can be efficiently reused across all candidate solutions during test-time scaling, avoiding the quadratic complexity

*Equal Contribution. Contact: yangling0818@163.com
†Corresponding Author

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

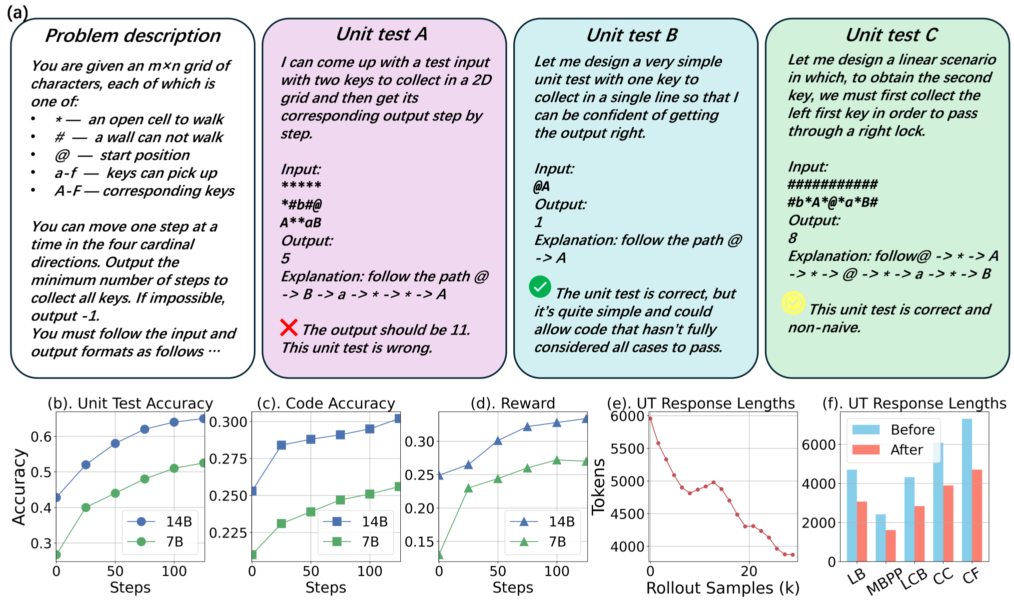


Figure 1: (a). This is an example of a problem description along with three task-derived generated unit tests. The first unit test is incorrect, although it is easily produced due to strong hallucination. The second unit test is correct but naive, allowing some incomplete or unthoughtful code to pass. 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.

inherent in scalar or generative reward models, which require separate reward computations for each candidate [7, 24, 3, 53, 27]. Most importantly, generating a unit test does not necessarily require the model to produce a complete solution or algorithm (see Figure 1(a)), substantially simplifying test construction compared to traditional verification approaches, in which LLMs often struggle to verify and correct self-generated solutions [17]. 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 [11, 30] and machine translation-based approaches [41, 1]. Recent developments show that large language models (LLMs) outperform traditional approaches in unit test generation [38, 50, 36], aided by prompt engineering and agentic techniques [50, 6, 13]. These findings highlight the potential for fine-tuning LLMs to further enhance their unit test generation capabilities [38]. O1-Coder [52] fine-tunes LLMs using unit tests derived from ground-truth code. Inspired by the trade-off between attack rate and accuracy, UTGEN [32] 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 2) 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 1 (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 [48, 42, 14]. 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: CURE-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 (with a 6.2% average gain in accuracy over the base model), and agentic unit test generation tasks (with a 25.1% average gain in accuracy). Our 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.

2 Related Work

Unit Test Generation Manually creating unit tests is costly and inefficient [5, 25], motivating the development of automated unit test generation methods, such as software analysis methods [11, 30, 9, 12, 33, 10] and traditional neural machine translation approaches [41, 1]. With the recent advancements in LLMs, prompt-based and agentic methods [50, 6, 13] have demonstrated superior performance, further highlighting the potential of training LLMs for unit test generation. In light of this, methods like O1-Coder [52] and UTGEN [32] 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.

Application of Unit Tests Unit tests have been shown to serve as effective rewards for test-time scaling and agentic coding [28]. 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 [5, 16]. AlphaCodium [35] introduces self-revision by leveraging both public and generated tests to refine solutions. S* [21] further incorporates iterative debugging and pairwise discrimination guided by generated unit tests to enhance final code quality.

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 [34, 26, 4, 29, 46, 44] skip the critic and directly optimize from preferences using closed-form rewards, improving efficiency. Recent efficient Group Relative Policy Optimization (GRPO) [39] scales well with large-scale reinforcement learning [14, 49, 15, 43]. Reinforcement learning applied specifically to coding tasks has also gained traction [8, 22]. 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.

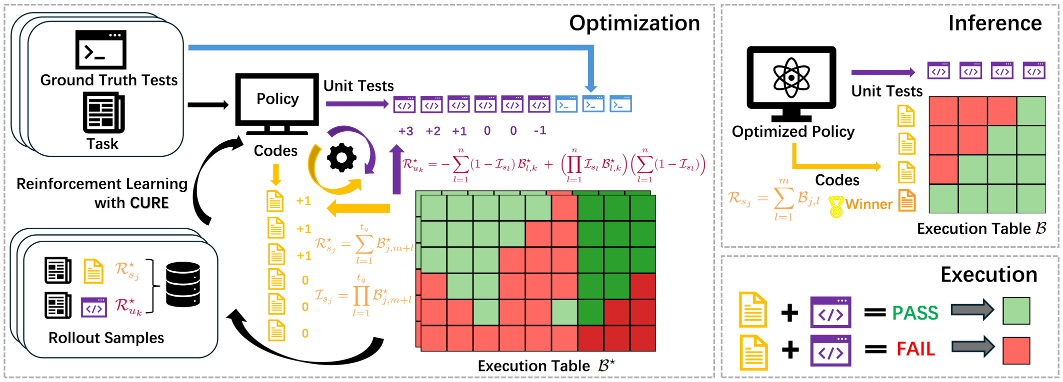


Figure 2: Method Pipeline Overview. In our RL framework, for each task, we generate a batch of unit tests and code solutions, along with some ground-truth unit tests. Using these, we construct an execution table. From this table, we extract rewards for each unit test (Equation 4) and code response (Equation 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 over time.

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 [17], code-generation tasks benefit significantly from the use of unit tests for efficient verification. It has been shown [28] 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:

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

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**:

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

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 $P(\mathcal{R}_{s_{j_1}} > \mathcal{R}_{s_{j_2}}) \to 1$ as $m \to \infty$, 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 $\mathcal{B}^\star \in \{0,1\}^{n \times (m+t_q)}$ 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:

$$\mathcal{R}^\star_{s_j} = \sum_{l=1}^{t_q} \mathcal{B}^\star_{j,m+l}. \tag{3}$$

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 $\mathcal{B}^\star$ by deriving estimators for p_u, p_{01}, and p_{00}. This leads to the following form of the estimated individual-level reward:

$$\mathcal{R}^\star_{u_k} = -\sum_{l=1}^{n}(1 - \mathcal{I}_{s_l})\mathcal{B}^\star_{l,k} + (\prod_{l:\mathcal{I}_{s_l}=1} \mathcal{B}^\star_{l,k})(\sum_{l=1}^{n}(1 - \mathcal{I}_{s_l})), \tag{4}$$

where $\mathcal{I}_{s_j} = \prod_{l=1}^{t_q} \mathcal{B}^\star_{j,m+l}$. The detailed derivation is provided in Appendix A. Intuitively, $\mathcal{R}^\star_{u_k}$ is positive and proportional to the number of incorrect solutions that fail test u_k when u_k correctly passes all accurate solutions. Conversely, $\mathcal{R}^\star_{u_k}$ 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:

$$\mathcal{J}(\theta, \{o_i\}_{i=1}^{G}) = \mathbb{E}_{\substack{q\sim P(Q)\\ o_i\sim\pi_{\theta_{\text{old}}}(\cdot|q)}}\left[\frac{1}{G}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|} C_\epsilon\left(\frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})}, A_{o_i}\right)\right]$$
$$-\mathbb{E}_{\substack{q\sim P(Q)\\ o_i\sim\pi_{\theta_{\text{old}}}(\cdot|q)}}\left[\beta\, \mathrm{D}_{\mathrm{KL}}\big[\pi_\theta \,\|\, \pi_{\text{ref}}\big]\right],$$

where $C_\epsilon(r, A) := \min(rA, \operatorname{clip}(r, \varepsilon)A)$, $\operatorname{clip}(r, \varepsilon) := \min(\max(r,\, 1-\varepsilon), 1+\varepsilon)$, π_θ is the policy to be optimized, π_{old} is the old policy, $\{o_i\}_{i=1}^{G}$ are the rollout responses, and A_{o_i} is the normalized reward corresponding to $\mathcal{R}^\star_{o_i}$. 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 2).

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 [48, 42, 14]. 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 $\widehat{r}_i$ by

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

where $T_l = \operatorname{median}\{l_j \mid r_j > 0\}$, $l_{max} = \max\{l_j \mid r_j > 0\}$. Subsequently, we balance the transformed rewards between positive and negative responses and normalize them, yielding the final transformed reward $r_i^\star$, defined by $r_i^\star = \alpha\widehat{r}_i/\sigma$ if $\widehat{r}_i > 0$, or $r_i^\star = \widehat{r}_i/\sigma$ if $\widehat{r}_i \leq 0$, where $\alpha = \sum_{j:\widehat{r}_j<0}(-\widehat{r}_j)/(\sum_{j:\widehat{r}_j>0}\widehat{r}_j)$, and σ is the standard deviation calculated over the set $\{\alpha\widehat{r}_i \mid \widehat{r}_i > 0\} \cup \{\widehat{r}_i \mid \widehat{r}_i \leq 0\}$. In this way, we aim to preserve the original reward information to some extent, while penalizing overly long responses.

4 Experiments

4.1 Settings

Datasets We select five widely used coding datasets for our comprehensive evaluation: LiveBench [45], MBPP [2], LiveCodeBench [19], CodeContests [23], and CodeForces [31]. 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 [31]; we randomly sample 500 examples from it for evaluation.

Models and Optimization We use Qwen2.5-7B and 14B [47] 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 32 rollouts for unit tests and 32 for code using vLLM [20], with a temperature of 1.0, top-p of 0.95, and top-k of 40. For optimization, we set the learning rate to 1×10^{-6} and the KL coefficient to 0.01. 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.

Table 1: Performance of CURE 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 models. The Coder models here are also instruction-finetuned models.

Model	**LiveBench**			**MBPP**			**LiveCodeBench**			**CodeContests**			**CodeForces**		
	UT	**Code**	**BoN**	**UT**	**Code**	**BoN**	**UT**	**Code**	**BoN**	**UT**	**Code**	**BoN**	**UT**	**Code**	**BoN**
Qwen2.5-14B-Coder	39.0	42.2	53.1	75.1	72.6	84.9	41.6	38.2	47.7	37.3	23.3	32.0	22.1	7.8	13.5
Qwen2.5-14B-Ins	27.8	36.4	51.7	72.8	76.3	83.2	35.7	33.5	45.1	43.8	25.6	33.4	20.7	7.3	12.5
CURE-14B	**55.4**	**45.2**	**57.0**	**85.3**	**78.1**	**85.4**	**63.1**	**39.9**	**48.7**	**64.4**	**30.2**	**38.9**	**53.5**	**10.2**	**19.1**
Qwen2.5-7B-Coder	19.3	35.0	42.9	41.3	68.0	79.6	20.6	29.8	34.8	12.9	22.8	23.8	7.2	6.7	9.1
Qwen2.5-7B-Ins	26.5	31.1	35.9	35.8	66.3	79.4	28.6	26.9	32.6	26.7	21.2	25.8	18.9	5.4	8.9
CURE-7B	**44.2**	**37.3**	**45.5**	**74.5**	**69.4**	**82.3**	**48.7**	**31.6**	**41.8**	**52.5**	**25.7**	**29.7**	**39.4**	**7.7**	**10.8**
Qwen3-4B (Long)	36.8	72.5	78.1	76.5	88.4	90.1	50.9	74.5	80.0	43.6	53.1	58.1	54.1	28.8	38.5
CURE-4B (Long)	**84.6**	**74.6**	**82.0**	**83.3**	**89.5**	**91.2**	**86.8**	**75.9**	**80.6**	**72.2**	**55.4**	**59.9**	**65.8**	**31.3**	**40.2**

Table 2: Application to GPT-series models. We apply CURE-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.

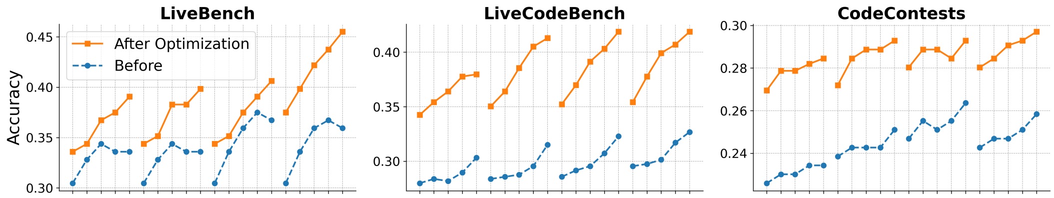
Model	**LB**		**MBPP**		**LCB**		**CC**		**CF**	
	Cost	**Acc**	**Cost**	**Acc**	**Cost**	**Acc**	**Cost**	**Acc**	**Cost**	**Acc**
4o (one-shot)	4.8	48.4	2.7	85.0	5.8	48.7	5.5	41.0	7.1	11.1
4o-mini (one-shot)	0.3	46.3	0.2	80.1	0.4	44.3	0.3	38.8	0.4	12.0
4o-mini (BoN-16)	10.8	55.4	6.7	81.5	12.0	50.7	10.1	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**

Test-time Scaling and Agentic Coding Best-of-N (BoN) is the most straightforward and widely used test-time scaling and agentic coding method [28, 5], 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 the reward defined in Equation 1. We also evaluate our approach under several other agentic coding and test-time scaling pipelines [16, 35, 21]. 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 [35] generates comprehensive unit tests to critique the generated solutions and iteratively refine the code accordingly. S* [21] 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.4.

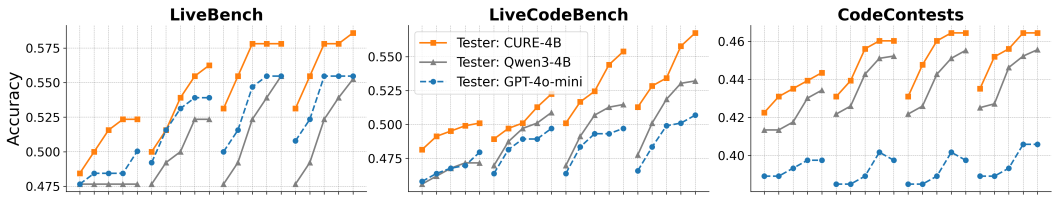
Agentic Unit Test Generation We also evaluate our model's utility in an agentic unit test generation pipeline. Following prior work [50, 6], we first generate unit tests and then iteratively refine them based on their execution results on the corresponding code. See details in Appendix C.5.

4.2 Results

CURE significantly improves the overall coding ability. Specifically, we apply our optimization to derive the CURE-7B and CURE-14B models from the base Qwen2.5-7B-Instruct and Qwen2.5-14B-Instruct models. Figure 1 (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 CURE models surpass their respective base models on average by 24.4% in unit test accuracy, 4.5% in one-shot code generation accuracy, and 5.1% in Best-of-N (BoN) accuracy (using 16 code solutions and 16 unit tests) (Table 1). Notably, CURE also consistently outperforms 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 3 (a)). This indicates that the CURE models not only enhance the overall performance ceiling (when large amounts of code and unit test samples are



(a). Base: Qwen2.5-7B-Instruct



(b). Coder: GPT-4o-mini

Figure 3: The BoN performance improvement across benchmarks. 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. (a). Improvement in BoN performance on open-source models after optimization. The model serves as both coder and unit tester here. (b). BoN improvement with optimized unit tester on GPT-series coders.

generated), but also improve self-check efficiency in low-sample regimes (e.g., when sampling only 1 or 2 candidates).

Robust for long-CoT models and achieve inference efficiency. We also evaluate CURE's optimization on the Long-CoT model, Qwen3-4B, incorporating our response-length-guided reward transformation. The resulting CURE-4B model consistently outperforms Qwen3-4B 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 1 (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 [51]. Long-CoT models have already captured much of the benefit from scaling through CoT reasoning and gain less from BoN compared to standard models.

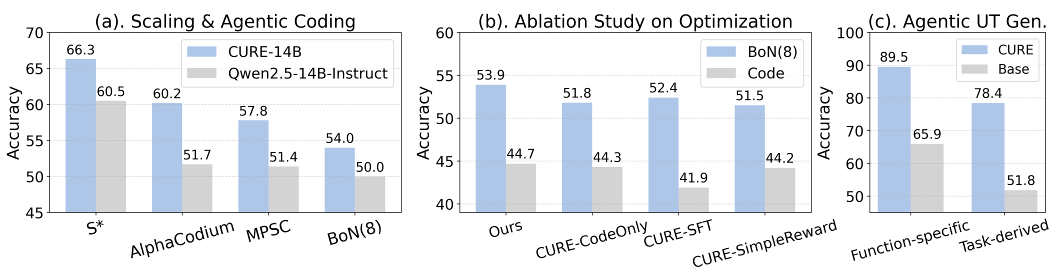


Figure 4: (a). Application of CURE to various test-time scaling and agentic coding methods. We set the number of generated samples to eight in the BoN setting here. (b). Ablation study on optimization strategies and reward design choices, using Qwen2.5-14B-Instruct as the base model. All training runs are conducted with 100 optimization steps. (c). Application of CURE 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.

CURE models help API-inference models become more powerful and cost-efficient. We apply CURE-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 CURE 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

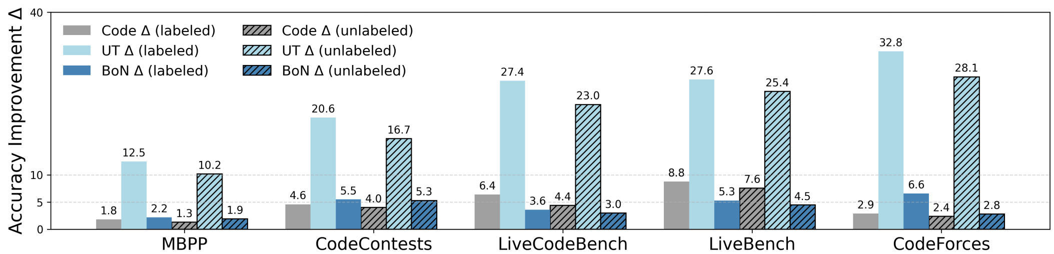


Figure 5: Accuracy improvement of Qwen2.5-14B-Instruct when trained with reinforcement learning using labeled unit tests as rewards versus using CURE-generated unit tests as rewards. Both models are trained for 150 steps. The BoN setting involves generating 16 samples for both code and unit tests.

CURE-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 CURE-4B. As shown in Figure 3(b), CURE-4B consistently outperforms both Qwen3-4B and GPT-4o-mini as a unit tester across different BoN settings. These results demonstrate the effectiveness of using unit tests generated by the CURE model.

Serves as an effective reward model enabling RL without any labeled data. We have already demonstrated the utility of unit tests generated by the CURE model for solution selection. But can the CURE model also serve as a reward model to guide reinforcement learning? We apply CURE-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 5). This demonstrates that CURE can serve as an effective reward model not only for inference-time enhancement but also for guiding optimization during training.

Broad application to test-time scaling and agentic coding methods. In addition to the standard test-time scaling method BoN [28, 5], we also evaluate CURE-14B on several other test-time scaling and agentic methods—MPSC [16], AlphaCodium [35], and S* [21]—achieving an average improvement of 6.2% over the base model Qwen2.5-14B-Instruct (Figure 4(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 CURE 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 4(c)).

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 4(b)). Optimizing only for code generation does not improve the model's ability to produce accurate 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 40.5% and 15.8%, respectively. In contrast, our theoretically derived reward better constrains these values to 30.1% and 10.6%, 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 CURE models achieve significant performance improvements in both code generation and unit test generation tasks. Our long-CoT model CURE-4B consistently outperforms Qwen-4B while achieving significantly higher efficiency in unit test generation. Moreover, CURE proves effective in broader applications, including test-time scaling and agentic coding (6.2% improvement), agentic unit test generation (25.1% improvement), and as a reward model for reinforcement learning.

CURE currently focuses on Python competition-style tasks evaluated via stdin/stdout unit tests. Extending CURE to support functional tests and additional programming languages is essential for greater practical utility. In addition, CURE still depends on ground-truth unit tests during reinforcement learning to achieve high performance. Removing this reliance while preserving comparable performance remains an intriguing direction for future work.

References

[1] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. A3test: Assertion-augmented automated test case generation. *Information and Software Technology*, 176:107565, 2024.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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, 2004, Proceedings*, volume 3440, page 365. Springer, 2005.

[11] 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.

[12] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *ACM SIGSOFT Software Engineering Notes*, 24(6):146–162, 1999.

[13] 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.

[14] 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.

[15] Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model. *arXiv preprint arXiv:2503.24290*, 2025.

[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] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

[36] 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.

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[38] 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.

[39] 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.

[40] Qwen Team. Qwq-32b: Embracing the power of reinforcement learning, March 2025.

[41] 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.

[42] Libo Wang. Dynamic chain-of-thought: Towards adaptive deep reasoning. *arXiv preprint arXiv:2502.10428*, 2025.

[43] Yinjie Wang, Ling Yang, Bowen Li, Ye Tian, Ke Shen, and Mengdi Wang. Revolutionizing reinforcement learning framework for diffusion large language models. *arXiv preprint arXiv:2509.06949*, 2025.

[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] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

A Proofs of Theoretical Results

Proof. (of Theorem 3.1)

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

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

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 $\mathcal{B}_{j_1 k} = 1$ with probability 1. Conversely, an incorrect solution passes that same correct test with probability p_{01}, so

$$P\big[\mathcal{B}_{j_2 k} = 0\big] = 1 - p_{01}.$$

Hence

$$P\big[X_k = 1\big] = P\big[\mathcal{B}_{j_1 k} = 1,\ \mathcal{B}_{j_2 k} = 0\big] = 1 - p_{01}, \quad P\big[X_k \le 0\big] = p_{01}.$$

Therefore $P\big(X_k > 0\big) = 1 - p_{01}$, 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,

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

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

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

Applying the law of total probability yields the unconditional mass

$$\begin{aligned} P(X_k = 1) &= p_u(1 - p_{01}) + (1 - p_u) \cdot 0 = p_u(1 - p_{01}), \\ P(X_k = -1) &= (1 - p_u)p_{00}, \\ P(X_k = 0) &= 1 - P(X_k = \pm 1). \end{aligned}$$

Denote

$$\mu := E[X_k] = 1 \cdot P(X_k = 1) + (-1) \cdot P(X_k = -1) = p_u(1 - p_{01}) - (1 - p_u)p_{00},$$
$$\sigma_k^2 := \mathrm{Var}(X_k) = E[X_k^2] - \mu^2 = P(X_k = 1) + P(X_k = -1) - \mu^2.$$

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 $\overline{X}_m := \frac{1}{m}\sum_{k=1}^{m} X_k$. Since $E[X_k] = \mu$ and $E[|X_k|] \le 1$, the strong law of large numbers (SLLN) tells us

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

But $D_m/m = \overline{X}_m$, 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 $P(D_m > 0) \to 1$.

- If $\mu < 0$, $\frac{D_m}{m}$ is eventually negative a.s., so $P(D_m > 0) \to 0$.
- If $\mu = 0$, $\frac{D_m}{\sqrt{m}}$ has variance σ_k^2 and remains $O_p(1)$, whence $P(D_m > 0) \to \frac{1}{2}$ 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

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

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

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

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

$$P\Big(\sum_{k=1}^{m} Z_k \le -t\Big) \ \le\ \exp\Big(-\tfrac{t^2}{2mc^2}\Big). \tag{Hoeffding}$$

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

$$\{D_m \le 0\} \ =\ \Big\{\sum_{k=1}^{m} Z_k \le -m\mu\Big\}.$$

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

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

Finally,

$$P(D_m > 0) = 1 - P(D_m \le 0) \ \ge\ 1 - \exp\big(-\tfrac{\mu^2 m}{8}\big),$$

yielding the advertised exponential guarantee.

□

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

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

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

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

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

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

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

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

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

$$\left[(\prod_{l:\mathcal{I}_{s_l}=1} \mathcal{B}^{\star}_{l,k})(\sum_{l=1}^{n}(1 - \mathcal{I}_{s_l})(1 - \mathcal{B}^{\star}_{l,k})) - (1 - \prod_{l:\mathcal{I}_{s_l}=1} \mathcal{B}^{\star}_{l,k})(\sum_{l=1}^{n}(1 - \mathcal{I}_{s_l})\mathcal{B}^{\star}_{l,k}) \right] / \sum_{l=1}^{n}(1 - \mathcal{I}_{s_l})$$

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

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

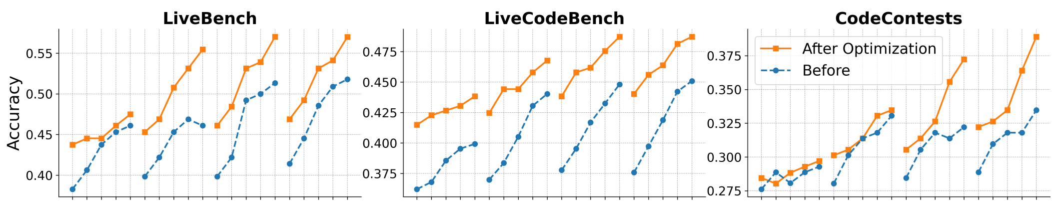
$$-\sum_{l=1}^{n}(1 - \mathcal{I}_{s_l})\mathcal{B}^{\star}_{l,k} + (\prod_{l:\mathcal{I}_{s_l}=1} \mathcal{B}^{\star}_{l,k})(\sum_{l=1}^{n}(1 - \mathcal{I}_{s_l})).$$

□

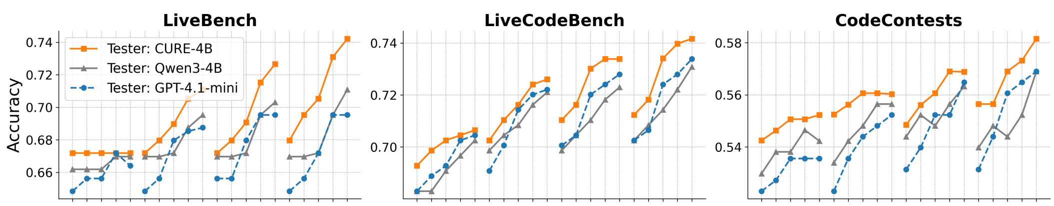
B Additional Experimental Results

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	UT	Code	UT	Code	UT	Code	UT	Code
CURE-14B	0.276 (0.008)	0.088 (0.0042)	0.125 (0.010)	0.018 (0.0025)	0.274 (0.012)	0.064 (0.0029)	0.206 (0.015)	0.046 (0.0031)	0.328 (0.038)	0.029 (0.0013)
CURE-7B	0.177 (0.005)	0.062 (0.0037)	0.387 (0.030)	0.031 (0.0021)	0.201 (0.009)	0.047 (0.0032)	0.258 (0.014)	0.045 (0.0036)	0.205 (0.022)	0.023 (0.0011)
CURE-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)



(a). Base: Qwen2.5-14B-Instruct



(b). Coder: GPT-4.1-mini

Figure 6: BoN performance improvement across benchmarks. 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. (a). Improvement in BoN performance on open-source models after optimization. (b). BoN improvement with optimized unit tester on GPT-series coders.

Table 4: Response length (in tokens) of Qwen3-4B and CURE-4B in unit test generation task, corresponding to Figure 1 (e).

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

C Details of Experiments

C.1 Detailed Algorithm

We have our detailed CURE optimization pipeline as follows. The training data we use is our training split of CodeContests. We set $n = m = 32$, $\eta = 1e-6$ and $\beta = 0.01$.

Algorithm 1 CURE

Input:
1) A set of coding tasks $D = \{q_1, q_2, \dots, q_N\}$.
2) A poliy π_θ parameterized by θ.
3) Number of iterations M.
4) Number of code solutions generated in each step: n.
5) Number of unit tests generated in each step: m.
5) Learning rate η, KL coefficient β.
Initialize: Policy parameters θ.
for $t = 1$ to M or not converged **do**
  Collect rollout samples:
  for each task $q \in D$ **do**
    Generate n code solutions, s_j, $1 \leq j \leq n$, by policy π_θ.
    Generate m unit tests, u_k, $1 \leq k \leq m$, by policy π_θ.
    Executing the n generated solutions against these m unit tests produces a binary evaluation matrix $\mathcal{B} \in \{0,1\}^{n \times m}$.
  end for
  Obtain the reward for code solutions:
  for each task s_j, $1 \leq j \leq n$ **do**

$$\mathcal{R}^\star_{s_j} = \sum_{l=1}^{t_q} \mathcal{B}^\star_{j,m+l}$$

  end for
  Obtain the reward for each unit test:
  for each task u_k, $1 \leq k \leq m$ **do**

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

    if π_θ is long-cot model **then**
      $\mathcal{R}^\star_{u_k} = trans(\mathcal{R}^\star_{u_k}, l_k)$
    end if
  end for
  Optimize the policy π_θ:

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

  Fine-tune π_θ to obtain updated parameters $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{J}(\theta, \{s_j\}_{j=1}^n)$,
  where $A_{s_j} = normalize(\mathcal{R}^\star_{s_j})$.
  Fine-tune π_θ to obtain updated parameters $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{J}(\theta, \{u_k\}_{k=1}^m)$,
  where $A_{u_k} = normalize(\mathcal{R}^\star_{u_k})$.
end for
Output: Trained generator π_θ.

C.2 Prompt Design

This is the prompt for code generation:

Code generation prompt

```
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.3 Preprocess Data

In our experiments, we adopt the stdio format for inputs and outputs, which is the standard input/output format used in LiveBench [45], LiveCodeBench [19], CodeContests [23], and CodeForces [31]. However, some tasks in LiveBench and LiveCodeBench, as well as all tasks in MBPP [2], 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 [40] (using BoN with a maximum of 3 samples) that passes all ground-truth tests to serve as the ground-truth code.

C.4 Test-time Scaling and Agentic Coding

We introduce how we apply MPSC [16], AlphaCodium [35] and S* [21] in our test-time scaling and agentic coding applications.

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.

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.

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.5 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) [11], random testing (Randoop) [30], model checking [9, 12], and symbolic execution [33, 10]. To improve scalability, neural machine translation-based methods were introduced [41, 1]. Specifically, AthenaTest [41] employs a BART model, while A3Test [1] uses a PLBART model with post-processing for improved accuracy. With the recent advancements in LLMs, prompt-based agentic methods such as ChatTester [50], ChatUniTest [6], and TestART [13] 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.6 Generated Unit Test Examples

Examples for CURE-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., `batteryPercentages[j] = max(0, 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: batteryPercentages[0] > 0, so count = 1. batteryPercentages becomes [1,0,1,0,2]
- At device 1: batteryPercentages[1] == 0, skip.
- At device 2: batteryPercentages[2] > 0, count = 2. batteryPercentages becomes [1,0,1,0,1]
- At device 3: batteryPercentages[3] == 0, skip.
- At device 4: 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: batteryPercentages[0] = 1 > 0
  - Increment tested count: 1
  - Decrement indices 1 to 4 → batteryPercentages becomes [1,0,1,0,2]
- At device 1: batteryPercentages[1] = 0 → skip
- At device 2: batteryPercentages[2] = 1 > 0
  - Increment tested count: 2
  - Decrement indices 3 to 4 → batteryPercentages becomes [1,0,1,0,1]
- At device 3: batteryPercentages[3] = 0 → skip
- At device 4: batteryPercentages[4] = 1 > 0 → increment tested count: 3

Test Input:

`1 0 2 0 3`

Test Output:

`3`

Explanation:

- Step-by-step execution results in final batteryPercentages: [1,0,1,0,1]
- Devices tested: 0, 2, and 4 → 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 (from 1) + 2 (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 `2`s and `3`s, and a few `1`s and `4`s.
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]` $\rightarrow$ cost = `1 + 2 + 12 = 15`
  - `[1,2], [3], [12]` $\rightarrow$ 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`.

Examples for CURE-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 ti.
But maybe another course has higher ti. Let me try to make a test case where the best selection is two courses that don't overlap, but perhaps have higher ti.
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 ai and run ki frames (consecutive). So for example, if ai is 1 and ki 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, ki=1 (so runs Monday period 1), and a course that starts on Tuesday (d=1) at a=2, ki=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 ti 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 ai and runs for ki 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), ai=1, ki=2. That would cover periods 1 and 2 on Monday. Then course2 is on Monday (d=0), ai=2, ki=1. So the course1 starts on ai=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=1, k=2, t=5. Runs on Monday 1 and 2.
Course2: d=0 (Monday), a=2, k=1, t=6. So starts on Monday 2. It overlaps with course1. So cannot be selected with course1.
Course3: d=1 (Tuesday), a=1, k=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.

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: The abstract and the last paragraph of the introduction directly highlight the contributions.

   Guidelines:

   - The answer NA 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 NA 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: We discuss the limitations of our work at the end of the discussion section.

   Guidelines:

   - The answer NA 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: [Yes]

Justification: The assumption is clearly stated in the main article, and the complete proof is provided in Appendix A.

Guidelines:

- The answer NA 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: Detailed guidelines for reproducing the experimental results are provided in both the experimental setup section of the main paper and Appendix C.

   Guidelines:

   - The answer NA 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: We will provide the data and code used in our experiments.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.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://nips.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, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Experiments details, such like paramter choosing, and data splits are all provided in the experimental setup section of the main paper and Appendix C.

   Guidelines:

   - The answer NA 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: [Yes]

   Justification: We provide the error analysis in Appendix B.

   Guidelines:

   - The answer NA 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: The computation resource used is written in the experiment setting section of main article.

   Guidelines:

   - The answer NA 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 presented in this paper fully complies with the NeurIPS Code of Ethics in all respects.

   Guidelines:

   - The answer NA 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: In the introduction, we state that our work—focusing on improving the coding abilities of LLMs—is an essential step toward advancing AI, which we believe has positive implications for society.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.

- If the authors answer NA 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., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We do not foresee any ethical or safety risks, as both the data and models used in our work are open-sourced and have already been widely adopted without known issues. Furthermore, our model adheres to standard safety practices and poses no additional risks.

    Guidelines:

    - The answer NA 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 models and datasets used in our work have been properly cited in the paper.

    Guidelines:

    - The answer NA 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 derived model is well documented.

    Guidelines:

    - The answer NA 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:[NA]

    Justification: Our paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA 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: [NA]

    Justification: Our paper does not involve crowdsourcing

    Guidelines:

    - The answer NA 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 rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The usage of LLM is written in details in the experiment setting part, as well as the Appendix B.

    Guidelines:

    - The answer NA 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 (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.