

RewardCode: Training Generalist Code Reward Model via Pairwise Reinforcement Learning

Songtao Huang^{1,2} Shiyang Feng^{2*} Xiaohan He^{1,2}
Xiangchao Yan² Bo Zhang² Lei Bai²

¹Fudan University ²Shanghai Artificial Intelligence Laboratory

Abstract

Reward modeling aligns language models with human preferences, yet coding tasks, requiring strict logic and syntax, pose challenges due to their domain diversity. In this paper, we propose RewardCode, a generalist code reward model trained via a two-stage pipeline, enabling scalable and precise evaluation through Structured Coding Scoring and unit test generation. Specifically, the first stage combines Structured Fine-Tuning and Rejective Fine-Tuning to distill cross-domain summarization and code evaluation capabilities from advanced LLMs. The second stage introduces Pairwise-GRPO, a reinforcement learning method that uses preference pairs to improve solution discrimination, while ensuring the generation of consistent and verifiable unit tests. Further, to train a cross-domain code reward model, we construct CodePair-18K, a dataset of verifiable code preference pairs and executable unit tests. Experiments on multiple benchmarks show that RewardCode outperforms strong baselines in evaluation accuracy, verifying its effectiveness in advancing general-purpose code LLMs.

1 Introduction

Large Language Models (LLMs) (Liu et al., 2024a; Achiam et al., 2023; Yang et al., 2025) have demonstrated superior performance across a wide range of coding tasks, including software engineering coding (Jimenez et al., 2023), machine learning coding (Chan et al., 2024), and programming competitions (Jain et al., 2024).

To further enhance LLMs in coding scenarios, researchers try to explore Reinforcement Learning (RL) for post-training (Wei et al., 2025; Zhang et al., 2024) and Test-time Scaling (TTS) (Zeng et al., 2025a) for inference phase. However, Rule-based rewards for code RL are typically derived from unit test verification results or actual program

*Corresponding author, email: fengshiyang@pjlab.org.cn
Primary Contact: huangsongtao@pjlab.org.cn

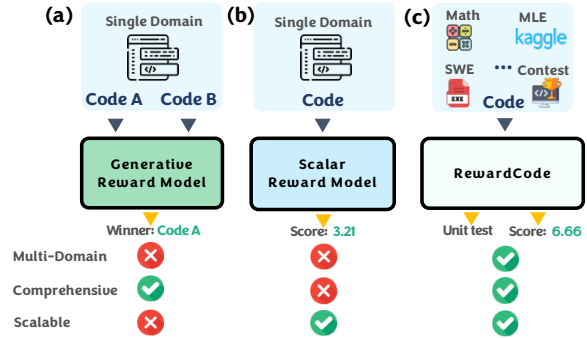


Figure 1: Comparison of code reward modeling paradigms. (a) Generative Reward Models provide comprehensive evaluation but lack scalability. (b) Scalar Reward Models are scalable but often lack comprehensive feedback. (c) RewardCode supports multi-domain evaluation and achieves both scalability and comprehensive verification through execution-based unit test and score generation.

execution results specific to each domain. Constructing such large set of high-quality reward labels in advance is extremely costly. Besides, TTS methods guided by reward-based selection strategies, such as Best-of-N (BoN), also depend on dynamic and accurate reward evaluation. Therefore, a high-quality code reward model is a prerequisite for boosting LLMs in coding tasks.

Furthermore, coding tasks range from algorithmic competitions to complicated Machine Learning Engineering (MLE) and Software Engineering (SWE), where each domain requires distinct expert knowledge. Current reward models fall short in addressing these differences. Existing specialized models, such as AceCodeRM (Zeng et al., 2025b), are capable of evaluating simple programming tasks but struggle to deliver accurate assessments in complicated domains like MLE and SWE. As a result, a unified reward modeling technique in the context of RL and TTS for coding tasks remain under-explored. Motivated by these observations, we raise the following question: *Can we design a unified code reward model that works across*

diverse coding domains?

In this work, we propose RewardCode, a generalist code reward model that integrates Structured Coding Scoring with executable unit test generation. Complementary consensus filtering between the two is used to derive the final evaluation results. Unlike pairwise reward models that rely on repeated comparisons, RewardCode evaluates candidate solutions in a pointwise manner, enabling scalable and fine-grained assessment across large collections of code. Figure 1 illustrates the differences between RewardCode and existing reward models on coding tasks. To support the training of a generalist code reward model, we construct CodePair-18K, a cross-domain dataset of verifiable preference pairs with structured task summaries and executable unit tests, collected from diverse domains including mathematical code translation, programming competitions, data science engineering, and software engineering.

Besides, we design a two-stage training pipeline to train RewardCode. *In the first stage*, we propose Structured Fine-Tuning to transform diverse task descriptions into structured summaries in order to alleviate cross-domain gaps, together with Rejective Fine-Tuning to bootstrap high-quality trajectories from large reasoning LLMs through group rejection sampling. *In the second stage*, a reinforcement learning method, Pairwise-GRPO, is designed, which leverages preference pairs with rule-based rewards to enhance the model’s ability to discriminate between solutions and to generate consistent and verifiable unit tests.

We evaluate RewardCode on different domains, ranging from code snippet to complicated machine learning and software engineering tasks. Experimental results demonstrate that RewardCode significantly outperforms existing reward models in BoN selection scenarios, particularly in complicated agent-based tasks. Moreover, we also find that RewardCode can serve as a reliable reward model for stable GRPO training of the base LLM on code generation tasks, demonstrating its ability to replace complex rule-based reward labeling. Our main contribution can be summarized as follows:

- We introduce **RewardCode**, the first generalist code reward model that provides accurate and scalable evaluation from simple programming tasks to complex real-world scenarios such as MLE and SWE tasks.
- We design **Pairwise-GRPO** to optimize scor-

ing accuracy through relative score comparisons within preference pairs. It also prevents reward hacking by jointly validating unit tests, thereby enhancing the model’s ability to discriminate between solutions and to generate consistent, verifiable tests.

- We present CodePair-18K, a dataset of verifiable code preference pairs coupled with executable unit tests, which aggregates cross-domain tasks, including mathematical translation, algorithmic programming, competitive challenges, data science, and software engineering.

2 Related Work

2.1 Code LLMs and Autonomous Agents

Recent advancements in Large Language Models (LLMs) have catalyzed significant progress in automated code generation. Foundation code LLMs, pretrained on massive code corpora, have established a strong baseline across diverse domains, ranging from software engineering to data science and algorithmic reasoning (Li et al., 2023; Guo et al., 2024; Hui et al., 2024; Team et al., 2025). This progress has necessitated the development of rigorous benchmarks to systematically evaluate model capabilities (Jain et al., 2024; Zhuo et al., 2024; Zhou et al., 2025). Beyond function-level generation, LLM-driven agents have emerged to address complex, open-ended problems (Zhang et al., 2025; Zeng et al., 2025a; Ou et al., 2025; Bai et al., 2026; Feng et al., 2026; Zhang et al., 2026). For instance, SWE-Agent (Yang et al., 2024) introduces an agent-computer interface for repository-level navigation and editing, while AIDE (Jiang et al., 2025) and AutoMLGen (Du et al., 2025) employ systematic search and scoped memory mechanisms to automate software development and data science workflows, respectively. However, the open-ended nature of these complex tasks demands highly flexible and robust reward signals to guide agents effectively, a capability that standard foundation models often lack.

2.2 Reward Model

Reward Models (RMs) have become essential for replacing human evaluators in assessing the quality of LLM outputs. Early reward models often produced scalar scores based on the output’s quality, typically trained with the Bradley-Terry (BT) loss. However, these pointwise scores fail to capture

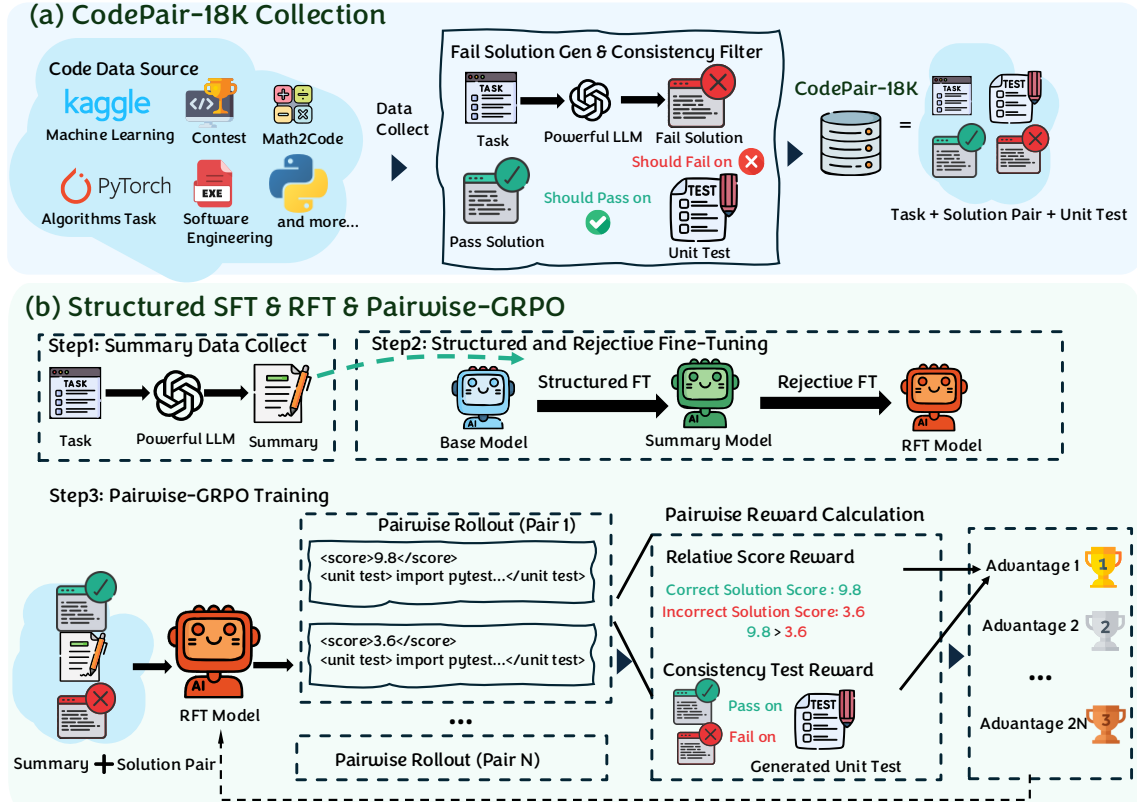


Figure 2: The overall pipeline of **RewardCode**. The training process consists of two stages: it initiates with the supervised fine-tuning of a base model on the CodePair-18K dataset and collected reasoning trajectories. Subsequently, the model is optimized via Pairwise-GRPO to enhance its capability in discriminating code solutions and generating correct unit tests.

the multi-dimensional aspects of code evaluation, such as correctness, efficiency, and test coverage. More recent developments in Generative Reward Models (GRMs) (Mahan et al., 2024; Liu et al., 2025b; Chen et al., 2025c; Guo et al., 2025; Chen et al., 2025a; Zhao et al., 2025; Whitehouse et al., 2025) have explored integrating natural language critiques alongside evaluation signals. DeepSeek-GRM (Liu et al., 2025b) introduces Self-Principled Critique Tuning (SPCT) to guide models in applying their own evaluation principles. Similarly, RM-R1 (Chen et al., 2025c) frames reward modeling as a reasoning task, leveraging high-quality reasoning chains and verifiable rewards to improve evaluation quality. However, in code generation tasks, correctness is often measured by deterministic signals, such as unit tests, and incorporating natural language critiques may introduce inconsistencies or biases. To address this, CodeRM (Ma et al., 2025) has explored the generation of unit tests via LLMs, but it remains limited to relatively simple code tasks. AceCodeRM (Zeng et al., 2025b) constructs code preference pairs to train scalar reward models, yet it is constrained to simpler program-

ming challenges. In contrast, our work is the first to integrate GRMs into a unified evaluation framework for code. We propose a model that jointly supports scalar scoring and unit test generation, enabling accurate and generalizable reward modeling across diverse coding domains.

3 RewardCode

In this work, we develop RewardCode, a generalist code reward model designed to provide judgment across diverse coding domains. Figure 2 presents the overall pipeline. We first construct CodePair-18K, a dataset of 18K validated code preference pairs with corresponding unit tests. RewardCode is trained in two stages. We begin by extracting task descriptions from CodePair-18K and employing a powerful LLM to generate structured summaries for each domain. In addition, we collect high-quality trajectories from large reasoning models through rule-based rejection sampling. These two sources are used to perform supervised fine-tuning the base model, rapidly enhancing its ability to generate structured summaries and to perform general evaluation across different code domains.

Datasets	Code Preference	Verification Code	Complex Task	Executable Unit Test	Correctness Verification	Function Design	Math2Code	Programming Contest	MLE	SWE	# Size
Code-Preference-Pairs	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	54K
Target-DPO-59K	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	59K
CodeDPO-114K	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	114K
AceCodePair-300K	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	300K
CodePair-18K	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	18K

Table 1: Comparison of existing code preference datasets.

Finally, we design Pairwise-GRPO to strengthen the model’s capacity to autonomously discriminate between better and worse solutions when no mutual information is available, and to improve its ability to generate correct and consistent unit tests.

3.1 CodePair-18K Data Collection

Table 1 compares CodePair-18K with existing code preference datasets. Unlike prior datasets that primarily focus on function-level programming tasks, CodePair-18K combines verifiable preference pairs from algorithmic programming, mathematical code generation, machine learning engineering, and repository-level software engineering.

We employ domain-specific construction strategies according to the availability of reference solutions and executable tests. For algorithmic and math-to-code tasks, we introduce localized semantic perturbations into verified reference implementations and retain only perturbations exposed by the original tests. For MLE tasks, we collect candidate implementations from MLE-Dojo trajectories, generate targeted tests with Claude-4-Sonnet, and synthesize buggy counterparts that fail the same tests. For SWE tasks, we derive chosen and rejected responses from the post-fix and pre-fix versions of R2E-Gym commits, respectively, and apply AST- and dependency-based pruning to remove repository code unrelated to the target defect. Finally, every pair is subjected to execution-based consistency filtering, which requires the shared test suite to pass the chosen implementation and fail the rejected implementation. Detailed construction and validation procedures are provided in Appendices A.2 and A.3.

3.2 Structured Coding Scoring and Test Generation

We formulate the general paradigm of code evaluation as follows: given a programming task and its candidate solution, a generalist code reward model should be able to generate both a reliable score and executable unit tests for verification. Based on this

principle, we design RewardCode as a pointwise reward model, avoiding the limited applicability of pairwise RMs and the computational bottleneck in large-scale evaluation.

$$\{r, u\} = \pi_{\theta}(s, y), \quad (1)$$

where s and y represent the coding task and the corresponding solution, respectively, r and u represent the scoring result and the generated unit test. Unlike chat or math reasoning tasks, programming tasks allow verification through carefully designed unit tests, where the execution results directly determine correctness. Therefore, RewardCode adopts Structured Coding Scoring and Test Generation to produce both scores and tests. We design three intuitive principles for code evaluation: Task Completion, Code Correctness, and Efficiency. Task Completion checks whether the solution fulfills the required functionality, Code Correctness assesses the logical soundness of the code, and Efficiency evaluates the quality and resource efficiency of the implementation. RewardCode assigns individual scores for each principle and determines their weights according to the task category. The final score is the weighted sum of the three principle scores. This scoring mechanism actively encourages RewardCode to conduct detailed inspections of the code and strengthens their understanding of candidate solutions. After a thorough understanding, RewardCode generates directly executable unit tests according to the task objectives and code content, thereby validating correctness in practice. All unit tests are formalized in Pytest style, ensuring easy and executable verification. In this way, RewardCode can achieve efficient large-scale code evaluation through score comparison, while executable unit tests provide a reliable ground truth of correctness. During evaluation, we employ a consensus filtering strategy: among the candidates that pass the unit tests, the one with the highest score is selected as the optimal solution.

3.3 Structured and Rejective Fine-Tuning

To equip RewardCode with basic Structured Coding Scoring and Test Generation capabilities, we construct a cold-start dataset for Supervised Fine-Tuning (SFT). This stage incorporates two key strategies: *Structured Fine-Tuning* and *Rejective Fine-Tuning*, which are designed to bridge cross-domain gaps and ensure the generation of reliable evaluations and verifiable tests.

Structured Fine-Tuning. Compared with other tasks, programming tasks contain explicit Structured information such as inputs and outputs. However, task descriptions across different code domains vary significantly and are often task-specific. To enable the model to quickly understand heterogeneous programming tasks, we use GPT-4o to transform diverse task descriptions from CodePair-18K into structured summaries. Each summary includes the task objective, input-output specification, and other necessary details for solving the task. Compared with verbose natural-language descriptions, structured summaries provide RewardCode with concise and informative inputs, enabling it to quickly identify task goals and make more accurate judgments.

Rejective Fine-Tuning. The purpose of Rejective Fine-Tuning is to bootstrap RewardCode’s ability to generate Structured Coding Scoring and Unit Tests in the correct format by leveraging the reasoning capabilities of large models. Starting with CodePair-18K, we sample a subset of task summaries and their corresponding code solution pairs. For each task-solution pair, we use DeepSeek-R1-0528 to generate N responses without revealing correctness labels. To filter low-quality responses, we employ Group Rejective Sampling: for correct solutions, all N responses must score higher than those of incorrect solutions under the same task, and vice versa. This ensures that the model learns to distinguish between correct and incorrect solutions. Additionally, we generate unit tests for each solution and retain only those that pass validation against expected outcomes, ensuring the generated tests are accurate and executable.

3.4 Pairwise-GRPO

We propose Pairwise-GRPO to leverage large-scale code preference pairs and further enhance RewardCode’s ability in Structured Coding Scoring and Test Generation. Specifically, for the input

code pair dataset $D = (s_i, y_a, y_b)_{i=1}^N$, RewardCode generates scores and unit tests for each solution in a pointwise manner:

$$(\{r_a, u_a\}, \{r_b, u_b\}) = (\pi_\theta(s_i, y_a), \pi_\theta(s_i, y_b)), \quad (2)$$

where s_i, y_a, y_b denote task summary, correct solution and incorrect solution. π_θ represents RewardCode model. $\{r_a, u_a\}, \{r_b, u_b\}$ correspond to the score and unit test generated for the correct and incorrect solutions, respectively. We then compute rewards in a pairwise fashion. First, we define the Relative Score Reward R_{score} , which evaluates whether the relative ranking of r_a and r_b aligns with ground-truth correctness:

$$R_{score} = \begin{cases} 1 & \text{if } r_a > r_b, \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

Intuitively, R_{score} and R_{ut} evaluate solutions at two complementary granularities. The score serves as a **coarse-grained** preference signal that only requires the relative ranking to be consistent, while the unit test provides a **fine-grained** correctness verification signal. To validate unit test quality, we design the Unit Test Reward with a strict requirement: the generated test must pass on the correct solution and fail on the incorrect one, driving the model to generate reliable and discriminative tests. The Unit Test Reward of u_a is represented as:

$$R_{ut}^a = \begin{cases} 0.5 & \text{if } P(y_a, u_a) = 1 \wedge P(y_b, u_a) = 0, \\ 0, & \text{otherwise,} \end{cases} \quad (4)$$

where P denotes the execution result of a unit test. The Group Unit Test Reward is given by $R_{ut} = R_{ut}^a + R_{ut}^b$. The final pairwise reward R is calculated as:

$$R = R_{score} + R_{ut}^a + R_{ut}^b + R_{format}, \quad (5)$$

where R_{format} is Structured format reward. After collecting rollout samples for each code pair and their corresponding rewards, we optimize the policy iteratively with the following objective:

$$\begin{aligned} J(\theta, \{o_i\}_{i=1}^G) = & \mathbb{E}_{\substack{q \sim P(Q) \\ \{o_i\} \sim \pi_{\theta_{old}}(\cdot|q)}} \left[\frac{1}{G} \sum_{i=1}^G \min \left(\frac{\pi_\theta(o_i | q)}{\pi_{\theta_{old}}(o_i | q)} A_{o_i}, \right. \right. \\ & \left. \left. \text{clip} \left(\frac{\pi_\theta(o_i | q)}{\pi_{\theta_{old}}(o_i | q)}, 1 - \varepsilon, 1 + \varepsilon \right) A_{o_i} \right) \right] \\ & - \beta \mathbb{E}_{\substack{q \sim P(Q) \\ \{o_i\} \sim \pi_{\theta_{old}}(\cdot|q)}} \left[\text{D}_{\text{KL}} [\pi_\theta \| \pi_{\text{ref}}] \right], \end{aligned} \quad (6)$$

where $\text{clip}(x, \varepsilon) := \min(\max(x, 1 - \varepsilon), 1 + \varepsilon)$, π_θ is the policy to be optimized, π_{old} is the old policy, $\{o_i\}_{i=1}^G$ are the pairwise rollout, and A_{o_i} is the normalized pairwise reward related to R_{o_i} .

4 Experiment

4.1 Experimental Settings

Evaluation Metric and Benchmark To verify the cross-domain efficacy of different reward models, we evaluate their performance as Best-of-N (BoN) selectors across both complex agent-based benchmarks and single-code benchmarks. For complex agent-based evaluation, we use subsets of MLE-Bench (Chan et al., 2024) and SWE-Bench (Jimenez et al., 2023). Due to computational limitations, we collected 7 executable data science tasks from MLE-Bench and 7 classes of diverse software engineering issues from SWE-Bench-Verified as our complex agent-based benchmarks. Using Qwen3-Coder-30B-A3B-Instruct (Yang et al., 2025) as the backbone, we implement a step-level BoN strategy where the reward model selects the best of 8 candidates at each turn. Specifically, we employ ML-Master (Liu et al., 2025a) for MLE tasks to measure valid submissions within one hour and a Bash-only mini-swe-agent for SWE tasks to measure resolved rates (Yang et al., 2024). For single-code evaluation, we use the code subsets of RewardBench (Lambert et al., 2024) and RM-Bench (Liu et al., 2024b) to measure preference selection accuracy. Additionally, on LiveCodeBench (Jain et al., 2024), we generate 16 candidates using Qwen2.5-Coder-7B-Instruct and report the post-selection success rate to quantify test-time scaling capabilities. The candidate budgets differ because agent-based tasks require expensive multi-turn interactions with an execution environment, whereas single-code generation only requires independent model inference. We therefore use $N = 8$ for agent-based MLE and SWE evaluation and $N = 16$ for single-code evaluation. A controlled sensitivity analysis over different candidate budgets is provided in Appendix A.10.

Model Implementation and Optimization. We use Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as the base model for RewardCode. For Pairwise-GRPO optimization, we set the learning rate to 1×10^{-6} and the KL coefficient to 0.001. RL training is performed for 2 epochs over the full CodePair-18K dataset.

Baselines. For agent-based benchmarks, we compare RewardCode with GPT-4o, RM-R1-Qwen2.5-Instruct-7B, and AceCodeRM-7B. For BoN evaluation on single-code benchmarks, we consider three categories of reward models: (1) scalar reward models, including InternLM2-7B-Reward, Eurus-RM-7B, and RM-Mistral-7B; (2) generative reward models, including JudgeLRM (Chen et al., 2025b), RRM-7B (Guo et al., 2025), and RM-R1-Qwen2.5-Instruct-7B (Chen et al., 2025c); and (3) specialized code reward models, including CodeRM (Ma et al., 2025) and AceCodeRM-7B (Zeng et al., 2025b).

4.2 Results and Analysis

4.2.1 Best-of-N Evaluation

BoN on Agent-based Coding Benchmarks. We evaluate different reward models on Agent-based Coding Benchmarks in more complex scenarios. Compared to LLM-based code benchmarks, agent-based benchmarks involve more complex code implementations and place higher demands on reward models. We find that existing reward models struggle with accurate judgment in complex tasks. Unlike function-level code generation tasks, MLE and SWE tasks require complete software implementations or modifications, demanding accurate judgment of code details and a comprehensive understanding of the task’s macro functionality. We find that RewardCode-7B achieved the best results in improving the completion rate of BoN tasks. This improvement is due to RewardCode benefiting from a large amount of cross-domain preference data during RL training, allowing it to obtain accurate judgment signals at a macro level. Additionally, the unit test generation ability requires RewardCode to understand code details in depth. This complementary generation pattern allows RewardCode to evaluate code accurately from a unified perspective.

BoN on Single Coding Benchmarks. We evaluate the Best-of-N selection performance on LiveCodeBench and the code subsets of RewardBench and RM-Bench. As shown in Table 3, RewardCode-7B consistently outperforms all baseline methods across these diverse benchmarks. We observe that Scalar RMs perform competitively on RewardBench and RM-Bench, likely benefiting from large-scale general preference pretraining. However, they struggle significantly with real-world code evaluation on LiveCodeBench. Conversely, GRMs offer improved selection capabilities over Scalar RMs

Task name	GPT-4o	RM-R1-Qwen2.5-Inst-7B	AceCodeRM-7B	RewardCode-7B
<i>MLE Task</i>				
aptos2019-blindness-detection	0.194	0.444	0.296	0.462
denoising-dirty-documents	0.211	0.071	0.107	0.095
dog-breed-identification	0.300	0.286	0.263	0.300
mlsp-2013-birds	0.326	0.127	0.274	0.268
nomad2018-predict-transparent-conductors	0.487	0.468	0.500	0.581
plant-pathology-2020-fgvc	0.316	0.321	0.288	0.438
random-acts-of-pizza	0.417	0.583	0.600	0.666
<i>SWE Task</i>				
astropy	0.136	0.091	0.091	0.136
django	0.169	0.186	0.156	0.186
pydata	0.182	0.227	0.136	0.364
pylint-dev	0.000	0.000	0.000	0.100
scikit-learn	0.188	0.188	0.250	0.406
sphinx-doc	0.046	0.114	0.046	0.159
sympy	0.160	0.227	0.147	0.173

Table 2: Performance comparison across different models on MLE and SWE tasks.

but incur prohibitive computational costs due to pairwise inputs, with only limited performance gains. Similarly, existing specialized Code RMs show restricted robustness on diverse code distributions, indicating room for improvement in multi-scenario tasks.

In contrast, RewardCode demonstrates superior robustness by achieving state-of-the-art results across LiveCodeBench, RewardBench, and RM-Bench. Furthermore, RewardCode supports direct evaluation of individual solutions without requiring paired inputs like standard GRMs, which greatly improves evaluation efficiency. Notably, the code subsets in RewardBench and RM-Bench come from various programming languages, while RewardCode and other code RMs are mainly trained on Python data. This result highlights the model’s robust generalization, confirming its ability to transfer evaluation logic to out-of-domain programming languages.

4.2.2 RL Training Evaluation

To evaluate the reliability of RewardCode-7B as a reward provider during RL training, we fine-tune Qwen2.5-7B-Instruct on the KodCode-Light-RL-10K dataset (Xu et al., 2025) for two epochs. We employ Group Relative Policy Optimization with a rollout size of 8 and a learning rate of 5×10^{-7} . We compare RL using reward signals generated by RewardCode-7B with Rule-based RL, which directly uses verified ground-truth unit tests to produce binary execution rewards. Rule-based RL therefore serves as a strong execution-based refer-

ence.

Since RewardCode-7B generates both scalar scores and verification tests, we use the composite reward $r = 0.5r_{\text{score}} + 0.5r_{\text{ut}}$. As shown in Table 4, RewardCode-based RL matches or outperforms Rule-based RL on four of the five benchmarks and remains comparable on LiveCodeBench. This result demonstrates that model-generated reward signals can provide an effective and scalable alternative when task-specific ground-truth tests are unavailable. We further evaluate this behavior across different policy-model families in Appendix A.9.

4.2.3 Ablation Study

We conduct an ablation study on four variants: (1) RewardCode-Zero, optimized solely via Pairwise-GRPO without Structured and Rejective Fine-Tuning; (2) RewardCode-SFT, the version after fine-tuning but without RL optimization; (3) RewardCode-Score, which relies exclusively on scalar score-based judgments; and (4) RewardCode-UT, which utilizes only unit test-based verification.

As shown in Table 5, the full RewardCode outperforms all variants. RewardCode-Zero exhibits significant performance degradation on LiveCodeBench, indicating that the lack of an SFT cold start severely limits the model’s foundational reasoning and verification capabilities. While RewardCode-SFT surpasses the Zero variant by distilling these capabilities from powerful teacher models, it still lags behind the full model. This performance gap underscores the critical role of

Model	RewardBench-Code	RM-Bench-Code	LiveCodeBench-V4
<i>Scalar Reward Models</i>			
Eurus-RM-7B	0.600	0.496	0.369
RM-Mistral-7B	0.940	0.530	0.304
InternLM2-7B-Reward	0.941	0.502	0.344
<i>Generative Reward Models</i>			
JudgeLRM-7B	0.820	0.352	0.363
RM-R1-Qwen2.5-Inst-7B	0.888	0.568	0.362
RRM-7B	0.849	0.528	0.407
<i>Code Reward Models</i>			
AceCodeRM-7B	0.520	0.613	0.404
<i>Our Method</i>			
RewardCode-7B	0.949	0.621	0.418

Table 3: BoN evaluation results on single coding benchmarks.

Model	HumanEval	HumanEval+	MBPP	MBPP+	LiveCodeBench-V4	Avg
Qwen2.5-7B-Instruct	0.817	0.744	0.783	0.672	0.290	0.661
Rule-based RL	0.848	0.762	0.802	0.688	0.300	0.680
RewardCode-7B-based RL	0.854	0.787	0.812	0.688	0.299	0.688

Table 4: Performance comparison between Rule-based RL and RewardCode-7B-based RL methods on code generation benchmarks.

Model	RM-Bench-Code	LiveCodeBench
<i>RewardCode-Variant</i>		
RewardCode-Zero	0.589	0.366
RewardCode-SFT	0.582	0.387
RewardCode-Score	0.603	0.405
RewardCode-UT	0.591	0.347
<i>Our Method</i>		
RewardCode-7B	0.621	0.418

Table 5: Ablation study of different RewardCode variants.

Pairwise-GRPO in fully activating and refining these distilled skills for high-quality evaluation. Finally, the performance decline observed in both RewardCode-Score and RewardCode-UT validates our hybrid evaluation paradigm. This consensus filtering helps RewardCode avoid missing details with score-only evaluations and false positives with unit test-only evaluations, thus maximizing the performance of policy models in TTS.

5 Conclusion

In this paper, we have introduced RewardCode, a generalist code reward model designed to provide precise and scalable code evaluations across diverse coding domains. RewardCode adopts Structured Coding Scoring and test generation, with complementary consensus filtering used to derive the final evaluation results. Besides, we have designed

Pairwise-GRPO to jointly optimize score and unit test generation, which obtains pairwise shared rewards to ensure the correctness of the scoring process and prevent false positives in unit test generation. Experimental results highlight the effectiveness of RewardCode as a generalist evaluator for coding tasks and demonstrate its potential for future applications in more complex code domains.

Limitations

A primary limitation of this work is the linguistic scope of our training data. Although RewardCode shows potential for cross-lingual evaluation, it was trained primarily on the Python-centric CodePair-18K dataset. Its ability to generate executable unit tests and accurately assess other programming languages, such as C++ and Java, may therefore be weaker than in Python. The model may struggle with language-specific syntax and execution environments. Future work will expand CodePair-18K to support more programming languages and improve multilingual code evaluation across practical coding settings. Although CodePair-18K includes math-to-code examples, our current evaluation does not isolate this domain. The associated mathematical reasoning abilities are instead evaluated indirectly through the MLE benchmarks.

Ethics Statement

This work presents RewardCode, a generalist code reward model. The CodePair-18K dataset constructed for this study is derived exclusively from publicly available benchmarks and open-source repositories. We ensure that data collection strictly adheres to original licenses and privacy policies, containing no personally identifiable information. We conduct evaluations under consistent protocols to ensure fair and transparent comparison. Our experiments strictly adhere to the ARR Code of Ethics, particularly regarding transparency and reproducibility. Generative AI tools were used solely for language polishing and did not contribute to the research design, experiments, analysis, or conclusions.

References

- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Anthropic. 2025. [Introducing claude 4](#).
- Lei Bai, Zongsheng Cao, Yang Chen, Zhiyao Cui, Shangheng Du, Yue Fan, Shiyang Feng, Zijie Guo, Haonan He, Liang He, Xiaohan He, Shuyue Hu, Yusong Hu, Songtao Huang, Yichen Jiang, Hao Li, Xin Li, Dahua Lin, Weihao Lin, and 34 others. 2026. [Scaling the horizon, not the parameters: Reaching trillion-parameter performance with a 35b agent](#). *Preprint*, arXiv:2606.30616.
- Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, and 1 others. 2024. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*.
- Bin Chen, Xinzge Gao, Chuanrui Hu, Penghang Yu, Hua Zhang, and Bing-Kun Bao. 2025a. Reasonrm: Enhancing generative reward models through large reasoning models. *arXiv preprint arXiv:2506.16712*.
- Nuo Chen, Zhiyuan Hu, Qingyun Zou, Jiaying Wu, Qian Wang, Bryan Hooi, and Bingsheng He. 2025b. Judgelrm: Large reasoning models as a judge. *arXiv preprint arXiv:2504.00050*.
- Xiusi Chen, Gaotang Li, Ziqi Wang, Bowen Jin, Cheng Qian, Yu Wang, Hongru Wang, Yu Zhang, Denghui Zhang, Tong Zhang, and 1 others. 2025c. Rm-r1: Reward modeling as reasoning. *arXiv preprint arXiv:2505.02387*.
- Shangheng Du, Xiangchao Yan, Dengyang Jiang, Jiakang Yuan, Yusong Hu, Xin Li, Liang He, Bo Zhang, and Lei Bai. 2025. Automlgen: Navigating fine-grained optimization for coding agents. *arXiv preprint arXiv:2510.08511*.
- Shiyang Feng, Runmin Ma, Xiangchao Yan, Yue Fan, Yusong Hu, Songtao Huang, Shuaiyu Zhang, Zongsheng Cao, Tianshuo Peng, Jiakang Yuan, Zijie Guo, Zhijie Zhong, Shangheng Du, Weida Wang, Jinxin Shi, Yuhao Zhou, Xiaohan He, Zhiyin Yu, Fangchen Yu, and 38 others. 2026. [Internagent-1.5: A unified agentic framework for long-horizon autonomous scientific discovery](#). *Preprint*, arXiv:2602.08990.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Jiaxin Guo, Zewen Chi, Li Dong, Qingxiu Dong, Xun Wu, Shaohan Huang, and Furu Wei. 2025. Reward reasoning model. *arXiv preprint arXiv:2505.14674*.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Live-codebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.
- Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2egym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*.
- Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. 2025. [Aide: Ai-driven exploration in the space of code](#).
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

- Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, and 1 others. 2024. Rewardbench: Evaluating reward models for language modeling. *arXiv preprint arXiv:2403.13787*.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.
- 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, and 7 others. 2022. [Competition-level code generation with alphacode](#). *Science*, 378(6624):1092–1097.
- Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.
- Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. 2024b. Rm-bench: Benchmarking reward models of language models with subtlety and style. *arXiv preprint arXiv:2410.16184*.
- Zexi Liu, Yuzhu Cai, Xinyu Zhu, Yujie Zheng, Runkun Chen, Ying Wen, Yanfeng Wang, Siheng Chen, and 1 others. 2025a. Ml-master: Towards ai-for-ai via integration of exploration and reasoning. *arXiv preprint arXiv:2506.16499*.
- Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. 2025b. Inference-time scaling for generalist reward modeling. *arXiv preprint arXiv:2504.02495*.
- Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024. [Mathcoder2: Better math reasoning from continued pretraining on model-translated mathematical code](#). *Preprint*, arXiv:2410.08196.
- Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. 2025. Dynamic scaling of unit tests for code reward modeling. *arXiv preprint arXiv:2501.01054*.
- Dakota Mahan, Duy Van Phung, Rafael Rafailov, Chase Blagden, Nathan Lile, Louis Castricato, Jan-Philipp Fränken, Chelsea Finn, and Alon Albalak. 2024. Generative reward models. *arXiv preprint arXiv:2410.12832*.
- Yixin Ou, Yujie Luo, Jingsheng Zheng, Lanning Wei, Shuofei Qiao, Jintian Zhang, Da Zheng, Huajun Chen, and Ningyu Zhang. 2025. Automind: Adaptive knowledgeable agent for automated data science. *arXiv preprint arXiv:2506.10974*.
- Rushi Qiang, Yuchen Zhuang, Yinghao Li, Dingyue Sagar V K, Rongzhi Zhang, Changhao Li, Ian Shu-Hei Wong, Sherry Yang, Percy Liang, Chao Zhang, and Bo Dai. 2025. [Mle-dojo: Interactive environments for empowering llm agents in machine learning engineering](#). *Preprint*, arXiv:2505.07782.
- NovelSeek Team, Bo Zhang, Shiyang Feng, Xiangchao Yan, Jiakang Yuan, Zhiyin Yu, Xiaohan He, Songtao Huang, Shaowei Hou, Zheng Nie, and 1 others. 2025. Novelseek: When agent becomes the scientist—building closed-loop system from hypothesis to verification. *arXiv preprint arXiv:2505.16938*.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.
- Chenxi Whitehouse, Tianlu Wang, Ping Yu, Xian Li, Jason Weston, Ilia Kulikov, and Swarnadeep Saha. 2025. J1: Incentivizing thinking in llm-as-a-judge via reinforcement learning. *arXiv preprint arXiv:2505.10320*.
- Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. 2025. [Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding](#).
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.
- Guangtao Zeng, Maohao Shen, Delin Chen, Zhenting Qi, Subhro Das, Dan Gutfreund, David Cox, Gregory Wornell, Wei Lu, Zhang-Wei Hong, and 1 others. 2025a. Satori-swe: Evolutionary test-time scaling for sample-efficient software engineering. *arXiv preprint arXiv:2505.23604*.
- Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhui Chen. 2025b. [ACE-CODER: Acing coder RL via automated test-case synthesis](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12023–12040, Vienna, Austria. Association for Computational Linguistics.
- Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhong-Zhi Li, Xiangyuan Xue, Yijiang Li, Yifan Zhou, Yang Chen, Chen Zhang, Yutao Fan, Zihu Wang, Songtao

- Huang, Francisco Piedrahita Velez, Yue Liao, Hongru WANG, and 6 others. 2026. [The landscape of agentic reinforcement learning for LLMs: A survey](#). *Transactions on Machine Learning Research*. Survey Certification.
- Kechi Zhang, Huangzhao Zhang, Ge Li, Jinliang You, Jia Li, Yunfei Zhao, and Zhi Jin. 2025. Sealign: Alignment training for software engineering agent. *arXiv preprint arXiv:2503.18455*.
- Yuxiang Zhang, Shangxi Wu, Yuqi Yang, Jiangming Shu, Jinlin Xiao, Chao Kong, and Jitao Sang. 2024. o1-coder: an o1 replication for coding. *arXiv preprint arXiv:2412.00154*.
- Jian Zhao, Runze Liu, Kaiyan Zhang, Zhimu Zhou, Junqi Gao, Dong Li, Jiafei Lyu, Zhouyi Qian, Biqing Qi, Xiu Li, and 1 others. 2025. Genprm: Scaling test-time compute of process reward models via generative reasoning. *arXiv preprint arXiv:2504.00891*.
- Heng Zhou, Ao Yu, Yuchen Fan, Jianing Shi, Li Kang, Hejia Geng, Yongting Zhang, Yutao Fan, Yuhao Wu, Tiancheng He, Yiran Qin, Lei Bai, and Zhenfei Yin. 2025. [Livesearchbench: An automatically constructed benchmark for retrieval and reasoning over dynamic knowledge](#). *Preprint*, arXiv:2511.01409.
- Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Appendix

A.1 Pairwise-GRPO Training Curve

We visualize the training curves of Pairwise-GRPO in Figure 3 to validate the optimization process. The leftmost plot shows the total mean reward R steadily increasing, indicating stable convergence. The middle plot illustrates the rise in Unit Test Reward (R_{ut}), confirming that the model learns to generate high-quality tests that distinguish between correct and incorrect code. The rightmost plot displays the improvement in Relative Score Reward (R_{score}), showing that the model becomes increasingly accurate in ranking correct solutions higher than incorrect ones. The simultaneous improvement across all sub-rewards demonstrates that Pairwise-GRPO effectively harmonizes the dual goals of precise scoring and verifiable test generation.

A.2 CodePair-18K Data Construction

CodePair-18K is a cross-domain code preference dataset covering algorithmic programming, math-to-code generation, Machine Learning Engineering (MLE), and Software Engineering (SWE). It draws tasks from KodCode (Xu et al., 2025), CodeContest (Li et al., 2022), APPS (Hendrycks et al., 2021), math-to-code benchmarks (Lu et al., 2024), MLE-Dojo (Qiang et al., 2025), and R2E-Gym (Jain et al., 2025). Each example contains a task description, a chosen implementation, a rejected implementation, and an executable test suite. Since the available solutions and tests differ across domains, we adopt a domain-specific construction procedure.

Algorithmic and Math-to-Code Tasks. For algorithmic programming and math-to-code tasks, we begin with verified reference implementations and their associated tests. We use GPT-4o (Hurst et al., 2024) to introduce localized semantic errors into the reference implementations. These errors include incorrect boundary conditions, modified arithmetic operations, and incomplete case handling. The reference implementation is used as the chosen response, while the perturbed implementation serves as a candidate rejected response. We retain the pair only when the original tests pass on the chosen implementation and fail on the perturbed implementation.

MLE Tasks. For MLE tasks, we collect candidate implementations from iterative solution

trajectories produced in the MLE-Dojo environment (Qiang et al., 2025). Trajectory nodes marked as non-buggy are used as candidate chosen responses. Because these intermediate implementations generally do not have executable tests, we use Claude-4-Sonnet (Anthropic, 2025) to generate targeted Pytest suites based on the task specification and the candidate implementation.

Each generated test suite is first executed on the candidate chosen response in its task-specific Docker environment. Candidates that do not pass the generated tests are discarded. We then use o3-mini to construct a rejected counterpart by introducing a localized semantic defect while preserving the original program structure and interfaces. The rejected implementation is evaluated using the same test suite. We retain the pair only when the tests pass on the original implementation and fail on the synthesized counterpart.

SWE Tasks. For SWE tasks, we construct preference pairs from the fail-to-pass repository instances provided by R2E-Gym (Jain et al., 2025). The contents of the modified Python files before the fix are treated as the rejected response, while the corresponding post-fix contents are treated as the chosen response. The regression tests associated with each instance are obtained from its execution records. This procedure preserves realistic repository-level defects and their verified fixes without introducing additional synthetic bugs.

Repository files often contain substantial code unrelated to the target defect. We therefore apply an AST- and dependency-based context reduction procedure. We first compute the difference between the pre-fix and post-fix implementations and map the changed lines to their enclosing functions, classes, methods, and global definitions. We then use reverse dependencies to identify definitions that may be affected by the modified code. Starting from these affected definitions, we collect the forward dependencies required for execution, including referenced functions, classes, global variables, and imports. The retained definitions are finally ordered so that dependencies appear before their consumers. We apply the same procedure to the chosen and rejected implementations, producing compact snippets that preserve the defect-related logic and its required context.

After domain-specific construction, all examples are converted into a unified representation and processed by the execution-based consistency filter

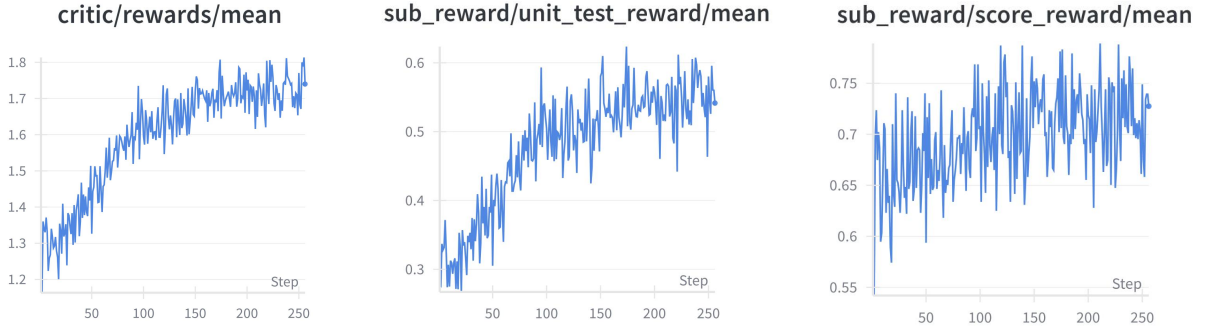


Figure 3: Training reward curves during the Pairwise-GRPO stage.

described in Appendix A.3.

A.3 Execution-Based Consistency Filtering

We apply execution-based consistency filtering to ensure that each preference pair provides a reliable training signal. Given a chosen implementation, a rejected implementation, and their shared test suite, we retain the pair only when the tests pass on the chosen implementation and fail on the rejected implementation. The executions are performed in isolated environments, including restricted sand-boxes for lightweight programming tasks and task-specific Docker environments for MLE tasks.

A rejected implementation is not required to crash or fail to compile; it may execute normally but produce an incorrect output or violate an expected behavior. For LLM-generated MLE tests, the tests are treated as candidate verification artifacts and are subjected to the same dual-execution criterion. Tests that fail on the candidate chosen implementation, pass on both implementations, or cannot be executed reliably are discarded together with the corresponding pair.

A.4 Diversity of Generated Unit Tests

The execution-based consistency filter ensures that a generated unit test can distinguish a correct solution from an incorrect one. However, this criterion alone does not demonstrate whether the model generates diverse testing behaviors across different coding domains. We therefore conduct a systematic analysis of the unit tests generated by RewardCode on MLE, SWE, and algorithmic programming tasks.

Specifically, we drew a stratified sample of 293 tasks and parsed 885 generated test functions using the Python AST. We measure the number of test functions and assertion points for each task, and identify tests that explicitly exercise exception paths. To further analyze semantic diversity, we

cluster the representations of individual test functions using K-Means with $K = 7$. We report the within-domain cluster entropy normalized by the maximum entropy $\log K$, where a higher value indicates that the generated tests cover a broader range of testing patterns.

Metric	MLE	SWE	KodCode
Test functions per task	2.68	2.58	3.77
Assertion points per task	6.07	3.72	4.07
Assertions per test function	2.26	1.44	1.08
Exception-checking tests	1.1%	28.3%	0.3%
Normalized cluster entropy	53.5%	44.4%	39.5%

Table 6: Structural and semantic diversity of the unit tests generated by RewardCode across different coding domains.

As shown in Table 6, the generated tests exhibit distinct domain-dependent characteristics. MLE tasks contain more assertion points per test function, reflecting the need to jointly verify data properties, tensor shapes, preprocessing behavior, and model outputs. SWE tasks show a substantially higher proportion of exception-checking tests, consistent with the importance of error handling and component interactions in repository-level programs. In contrast, tests for algorithmic tasks more frequently instantiate multiple concrete input-output cases. The non-trivial within-domain cluster entropy further indicates that the generated tests do not collapse into a single testing template. These results suggest that the execution-based objective encourages RewardCode to generate discriminative tests while adapting their structure to different coding domains.

A.5 Contamination Audit

To examine whether the evaluation results are affected by task overlap, we conduct a contamination audit between CodePair-18K and the evaluation

sets of LiveCodeBench, RewardBench-Code, and RM-Bench-Code. We normalize each task description by converting it to lowercase, removing punctuation, and collapsing consecutive whitespace. We then apply four complementary overlap detectors: exact normalized matching, contiguous 13-gram matching, 8-gram Jaccard similarity, and ROUGE-L F1 similarity.

Benchmark	#Eval	Exact	13-gram	Jac. ≥ 0.5	Jac. ≥ 0.8	R-L ≥ 0.8
LiveCodeBench	713	0	21 (2.94%)	0	0	0
RewardBench-Code	984	0	6 (0.61%)	0	0	0
RM-Bench-Code	228	0	1 (0.44%)	0	0	0

Table 7: Contamination audit between CodePair-18K and the evaluation benchmarks. Jac. and R-L denote 8-gram Jaccard similarity and ROUGE-L F1, respectively.

Table 7 shows that exact matching, Jaccard similarity, and ROUGE-L identify no high-similarity task pairs. The more sensitive 13-gram detector produces a small number of matches. Manual inspection shows that these matches are caused by common programming-problem expressions, standard input descriptions, or generic numerical examples rather than duplicated task specifications or solutions. We therefore find no evidence of substantial task-level contamination between CodePair-18K and the evaluated benchmarks.

A.6 Pairwise-GRPO vs. Standard GRPO

The primary motivation behind Pairwise-GRPO is to prevent the severe reward hacking that occurs when adapting standard GRPO for reward model training. Since standard GRPO is designed to optimize policy models, applying it directly to evaluate single code solutions allows the model to easily exploit the reward: it can consistently generate artificially high scores or trivial unit tests that always pass to secure rewards without genuine evaluation. Pairwise-GRPO addresses this by expanding the input to paired solutions and redesigning the reward calculation. The model receives a reward only if its generated unit test passes the correct solution while explicitly failing the incorrect one, and if the relative score correctly reflects the preference. Importantly, Pairwise-GRPO does not discard the group-level ranking information present in standard GRPO. Instead of comparing rollouts on a single input, it compares rollouts on pairwise inputs, thereby preserving the native diversity optimization advantages of GRPO.

As shown in Table 8, the standard GRPO variant yields inferior performance on LiveCodeBench

(0.391 vs. 0.418) and exhibits a much larger gap on the complex MLE Task (0.287 vs. 0.401). This empirical contrast confirms that Pairwise-GRPO is essential for preventing reward hacking and training a functional generalist reward model.

Model	LiveCodeBench	MLE Task
Qwen2.5-Coder-7B-Instruct	0.355	0.218
RewardCode-GRPO	0.391	0.287
RewardCode-Pairwise-GRPO	0.418	0.401

Table 8: Comparison between standard GRPO and Pairwise-GRPO for reward model training.

A.7 Data Efficiency

We highlight the data efficiency of RewardCode by comparing it with AceCodeRM-7B, which is trained on 300K code preference pairs via DPO. As shown in Table 9, RewardCode-7B achieves superior performance across all evaluated scenarios using only 18K samples for Pairwise-GRPO training, which constitutes approximately 6% of the data used by AceCodeRM-7B. This demonstrates that our training framework, combining cross-domain data with the Pairwise-GRPO optimization, is highly sample-efficient.

Model	Data Size	LCB	MLE	SWE
AceCodeRM-7B	300K	0.404	0.333	0.118
RewardCode-7B	18K	0.418	0.401	0.218

Table 9: Data efficiency comparison. LCB denotes LiveCodeBench. RewardCode-7B achieves better performance with only 6% of the training data used by AceCodeRM-7B.

A.8 Generalization Across Base Models

To validate the generalizability of the RewardCode training framework, we conduct experiments using the more advanced Qwen3-8B as an alternative base model. As shown in Table 10, both base models exhibit substantial improvements after training with our proposed pipeline. Notably, RewardCode-Qwen3-8B achieves a LiveCodeBench score of 0.460 and an MLE Task score of 0.423, further surpassing the Qwen2.5-Coder-7B-Instruct-based variant. These results confirm that the effectiveness of the RewardCode training framework is not limited to a specific base model and that the CodePair-18K dataset provides high-quality training signals that transfer across different foundation models.

Model	LiveCodeBench	MLE Task
Qwen2.5-Coder-7B-Instruct	0.355	0.218
RewardCode-Qwen2.5-Coder-7B	0.418	0.401
Qwen3-8B	0.383	0.302
RewardCode-Qwen3-8B	0.460	0.423

Table 10: Generalization across different base models. Both base models show significant improvements after training with the RewardCode pipeline.

A.9 RL Generalization Across Policy Backbones

We further investigate whether the effectiveness of RewardCode as a reward provider depends on a particular policy-model family. In addition to Qwen2.5-7B-Instruct, we apply the same RL pipeline to Mistral-7B-Instruct and Llama-3.2-3B-Instruct. For each policy backbone, we compare the original model, RL using verified execution-based rewards, and RL using reward signals produced by RewardCode.

Across the three policy-model families, RewardCode-based RL consistently improves or matches the corresponding untrained policy. It remains competitive with the execution-based reference on function-level benchmarks and provides particularly strong improvements on LiveCodeBench for the Mistral and Llama backbones. These results demonstrate that the reward signal produced by RewardCode is not restricted to policies from the Qwen model family.

A.10 Sensitivity to the Candidate Budget

Agent-based and single-code evaluations use different candidate budgets because their inference costs differ substantially. Agent-based MLE and SWE tasks require multi-turn interaction with an execution environment, whereas single-code generation only requires independent model inference. We therefore use a smaller candidate budget for agent-based evaluation.

To examine whether the selection performance depends on a particular value of N , we vary the number of candidates on LiveCodeBench and compare RewardCode-based selection with random selection.

As shown in Table 12, RewardCode consistently outperforms random selection across all candidate budgets. Its performance increases up to $N = 16$ and remains stable with larger candidate pools, with diminishing returns beyond this point. These results support $N = 16$ as an effective trade-off between selection performance and inference cost

for single-code evaluation.

A.11 Human Evaluation of Score Interpretability

We conduct a blind human evaluation to examine whether the scores produced by RewardCode are consistent with expert judgments. The evaluation contains 60 preference pairs sampled equally from KodCode, MLE, and SWE. For each pair, domain experts and RewardCode independently select the preferred implementation without access to the execution result. We use execution-based correctness labels as the reference. We report the selection accuracy of RewardCode, the accuracy of the human experts, and the agreement between their judgments.

As shown in Table 13, RewardCode achieves the same overall accuracy as the human experts. Their judgments agree on 80% of the evaluated pairs. The results also reveal different strengths across domains. Human experts perform better on SWE tasks, where repository conventions and interactions between software components are important. RewardCode performs better on the sampled MLE tasks, which often contain numerical or data-processing errors that do not immediately cause execution failures.

The structured scoring outputs show similar domain-specific behavior. For algorithmic tasks, RewardCode focuses on boundary conditions and logical correctness. For SWE tasks, it pays greater attention to repository context, interfaces, and exception handling. For MLE tasks, it frequently examines data leakage, tensor dimensions, label alignment, and preprocessing behavior. In a representative MLE example, both candidate implementations executed without an immediate runtime error. The human expert identified a label-indexing error. RewardCode identified the same defect and additionally detected that continuous scaling was applied to one-hot encoded features, which altered their categorical semantics. This example illustrates how the structured scoring dimensions can expose silent data-processing errors that are difficult to identify from execution behavior alone.

A.12 Task Summary Example

We present a task summary example in SWE task, which provides a raw programming issue description regarding a context initialization error, while the Output demonstrates the expected structured JSON summary.

Policy Model	HumanEval	HumanEval+	MBPP	MBPP+	LiveCodeBench-V4
Qwen2.5-7B-Instruct	0.817	0.744	0.783	0.672	0.290
Qwen2.5-7B-Instruct + Rule-RL	0.848	0.762	0.802	0.688	0.300
Qwen2.5-7B-Instruct + RewardCode-RL	0.854	0.787	0.812	0.688	0.299
Mistral-7B-Instruct	0.372	0.305	0.495	0.410	0.081
Mistral-7B-Instruct + Rule-RL	0.439	0.390	0.563	0.484	0.118
Mistral-7B-Instruct + RewardCode-RL	0.433	0.384	0.519	0.426	0.202
Llama-3.2-3B-Instruct	0.530	0.482	0.624	0.513	0.116
Llama-3.2-3B-Instruct + Rule-RL	0.567	0.512	0.635	0.540	0.126
Llama-3.2-3B-Instruct + RewardCode-RL	0.561	0.512	0.624	0.526	0.198

Table 11: RL performance across different policy-model families. Rule-RL uses verified execution-based rewards, whereas RewardCode-RL uses reward signals generated by RewardCode.

Selection Strategy	$N = 4$	$N = 8$	$N = 16$	$N = 32$	$N = 64$
Random Selection	0.321	0.320	0.335	0.311	0.333
RewardCode Selection	0.363	0.380	0.418	0.404	0.400

Table 12: Sensitivity of LiveCodeBench selection performance to the candidate budget N .

Domain	#Pairs	RewardCode Acc.	Expert Acc.	Agreement
KodCode	20	1.00	1.00	1.00
MLE	20	0.80	0.65	0.65
SWE	20	0.75	0.90	0.75
Overall	60	0.85	0.85	0.80

Table 13: Blind comparison between RewardCode and domain experts. Execution-based correctness labels are used as the reference.

A.13 Structured Coding Scoring and Test Generation Example

We show two Structured Coding Scoring and Test Generation examples in programming competition task. These two examples respectively demonstrate the evaluation scores and unit tests generated by RewardCode for correct and incorrect solutions.

A.14 Prompt

We present the specific prompts used in RewardCode. Below, we list the prompt of Structured Coding Scoring and Test Generation and the prompt of Task Summary, respectively.

Task Summary Example

PROGRAMMING TASK:

[ISSUE]

Title: Context migration fails to remove incompatible contexts, causing initialization errors

Description:

When initializing the
ContextHandler

with a mix of compatible and incompatible contexts, the migration process does not remove the incompatible contexts as expected. Instead, it raises an
IncompatibleContext
error, preventing successful initialization.

Example Code:

```
handler = ContextHandler()
handler.bind(SimpleWidget)
```

```
widget = SimpleWidget()
contexts = [Context(foo=i) for i in (13, 13, 0, 1, 13, 2, 13)]
```

```
def migrate_context(context, _):
    if context.foo == 13:
        raise IncompatibleContext()
```

```
handler.initialize(widget, dict(context_settings=contexts))
# Expected: Incompatible contexts with foo=13 should be removed
# Actual: IncompatibleContext error is raised, and contexts are not removed
Expected Behavior:
During initialization, contexts that are incompatible (e.g., those that cause
IncompatibleContext
to be raised) should be automatically removed, allowing the
ContextHandler
to proceed with only the compatible contexts.
```

Actual Behavior:

The
ContextHandler
does not remove incompatible contexts, resulting in an
IncompatibleContext
error being raised and preventing successful initialization.

[/ISSUE]

Task Summary:

"task_goal": "Ensure that during initialization, the `ContextHandler` removes incompatible contexts to prevent errors and allow successful initialization.",
"input_description": "A list of `Context` objects, each having a `foo` attribute with integer values. A `migrate\_context` function is used to determine compatibility based on the `foo` value.",
"output_description": "The `ContextHandler` should only retain compatible contexts and successfully initialize without raising an `IncompatibleContext` error.",
"key_information": "Contexts with `foo=13` are considered incompatible and should be removed during initialization. The `migrate\_context` function raises an `IncompatibleContext` error for these contexts."

Structured Coding Scoring and Test Generation Example (1)

[The Begin of Code Task]

```
{
    "task_goal": "Design a ticketing system function that calculates total revenue for each seat category from ticket sales data.",
```

```



```

[The Begin of Code Answer]
@file_source: solution.py

```

def calculate_revenue(ticket_sales):
    """
    Calculates the total revenue for each seat category from a list of ticket sales.

    Parameters:
        ticket_sales (list of dict): A list of dictionaries where each dictionary contains
            a "category" (str) and an "amount" (int).

    Returns:
        dict: A dictionary with seat categories as keys and their respective total revenue as
            values.
    """
    revenue = {}

    for ticket in ticket_sales:
        category = ticket["category"]
        amount = ticket["amount"]

        if category in revenue:
            revenue[category] += amount
        else:
            revenue[category] = amount

    return revenue
[The End of Code Answer]

```

Response:
<score>10</score>
<unit_test>
from solution import calculate_revenue

```

def test_empty_input():
    """Test with empty ticket sales list."""
    assert calculate_revenue([]) == {}

def test_single_ticket():
    """Test with one ticket sale."""
    ticket_sales = [{"category": "VIP", "amount": 200}]
    assert calculate_revenue(ticket_sales) == {"VIP": 200}

def test_multiple_same_category():
    """Test aggregation for same category."""

```

```

ticket_sales = [
    {"category": "Economy", "amount": 100},
    {"category": "Economy", "amount": 50}
]
assert calculate_revenue(ticket_sales) == {"Economy": 150}

def test_mixed_categories():
    """Test different categories in random order."""
    ticket_sales = [
        {"category": "Premium", "amount": 150},
        {"category": "VIP", "amount": 300},
        {"category": "Economy", "amount": 80},
        {"category": "Premium", "amount": 50}
    ]
    result = calculate_revenue(ticket_sales)
    assert result == {"Premium": 200, "VIP": 300, "Economy": 80}

def test_case_sensitivity():
    """Test handling of case-sensitive categories."""
    ticket_sales = [
        {"category": "vip", "amount": 100},
        {"category": "VIP", "amount": 200}
    ]
    assert calculate_revenue(ticket_sales) == {"vip": 100, "VIP": 200}

def test_large_amounts():
    """Test with large numerical values."""
    ticket_sales = [
        {"category": "Premium", "amount": 10**6},
        {"category": "Premium", "amount": 2*10**6}
    ]
    assert calculate_revenue(ticket_sales) == {"Premium": 3000000}
</unit_test>

```

Structured Coding Scoring and Test Generation Example (2)

```

[The Begin of Code Task]
{
  "task_goal": "Design a ticketing system function that calculates total revenue for each seat category from ticket sales data.",
  "input_description": "A list of dictionaries, where each dictionary contains keys 'category' (a non-empty string) and 'amount' (a non-negative integer). The list represents individual ticket sales.",
  "output_description": "A dictionary with seat categories as keys and their total revenue as values. Categories with no sales are not included in the output.",
  "key_information": {
    "seat_categories": [
      "VIP",
      "Premium",
      "Economy"
    ],
    "input_constraints": "1 to 1000 ticket sales",
    "requirements": [
      "Function must handle different orderings of input list.",
      "Function must process large numbers efficiently.",
      "Category names are case sensitive."
    ],
    "function_signature": "def calculate_revenue(ticket_sales):"
  }
}
[The End of Code Task]

[The Begin of Code Answer]
# @file_source: solution.py

def calculate_revenue(ticket_sales):
    """

```

Calculates the total revenue for each seat category from a list of ticket sales.

Parameters:

ticket_sales (list of dict): A list of dictionaries where each dictionary contains a "category" (str) and an "amount" (int).

Returns:

dict: A dictionary with seat categories as keys and their respective total revenue as values.

"""

```
revenue = {}
```

```
for ticket in ticket_sales:
    category = ticket["category"]
    amount = ticket["amount"]
```

```
    if category in revenue:
        revenue[category] += amount
    else:
        revenue[category] = amount
```

```
    return revenue
```

[The End of Code Answer]

Response:

<score>3.6</score>

<unit_test>

```
from solution import calculate_revenue
```

```
def test_single_ticket():
    tickets = [{"category": "Premium", "amount": 150}]
    result = calculate_revenue(tickets)
    assert result == {"Premium": 150}
```

```
def test_multiple_tickets_same_category():
    tickets = [
        {"category": "VIP", "amount": 200},
        {"category": "VIP", "amount": 300},
        {"category": "VIP", "amount": 100}
    ]
    result = calculate_revenue(tickets)
    assert result == {"VIP": 600}
```

```
def test_mixed_categories():
    tickets = [
        {"category": "Economy", "amount": 50},
        {"category": "VIP", "amount": 250},
        {"category": "Economy", "amount": 75}
    ]
    result = calculate_revenue(tickets)
    assert result == {"Economy": 125, "VIP": 250}
```

```
def test_zero_amount_tickets():
    tickets = [
        {"category": "Premium", "amount": 0},
        {"category": "Premium", "amount": 100}
    ]
    result = calculate_revenue(tickets)
    assert result == {"Premium": 100}
```

```
def test_case_sensitivity():
    tickets = [
        {"category": "vip", "amount": 50},
        {"category": "VIP", "amount": 150}
    ]
    result = calculate_revenue(tickets)
    assert result == {"vip": 50, "VIP": 150}
```

```
def test_large_quantities():
    tickets = [{"category": "Economy", "amount": 1000}] * 100
    result = calculate_revenue(tickets)
    assert result == {"Economy": 100000}
</unit_test>
```

Prompt of Task Summary

You are an expert engineer and programming specialist with extensive experience in analyzing
→ and solving various programming problems. Code tasks may include various tasks such as
→ software engineering, data science engineering, code competitions, etc. Your task is to
→ analyze programming tasks and extract key implementation information in a concise,
→ structured format.

When given a programming task description, you should:

1. ****Identify Task Goal****: Identify what exactly needs to be implemented.
2. ****Define Input/Output Specifications****: Clearly specify expected inputs and outputs.
3. ****Identify Key Information****: Note important limitations and requirements. Completely copy
→ some information that is crucial to code implementation, such as the fields of the dataset.
4. ****Filter Irrelevant Information****: Focus only on information crucial for implementation.

Required fields (keep even if empty):

- task_goal: Clear description of the ultimate goal of the given task.
- input_description: Describe input data (type, format, fields, structure, shape, constraints).
- output_description: Describe expected output (type, format, fields, structure, constraints).

Additionally, include any other relevant fields specific to this task that would be crucial for
→ implementation (e.g., constraints, algorithms, edge cases, performance requirements, etc.).

Your analysis should be concise, precise, and contain only the essential information needed to
→ successfully implement the solution. Avoid unnecessary details while ensuring all critical
→ requirements are captured. Do NOT invent facts. Output exactly ONE JSON object.

Prompt of Structured Coding Scoring and Test Generation

Please act as an expert dedicated to analyzing and judging code outputs in diverse programming scenarios.

For a given **Code Task** and its corresponding **Code Answer**, evaluate the Code Answer according to the steps below, and output your answer with structured tags.

Required Steps

1. **Score**:

- Determine how critical each of the four Basic Principles (Task Completion, Code Correctness, Efficiency) is for this task, and assign a percentage weight to each (total = 100%).
- For each principle, restate its objective in the context of the Code Task, then perform a detailed, principle-guided review of the code.
- From your analysis, assign an integer score (1-10) per principle. For each, show its weighted contribution (`score × weight`), then sum these to yield an overall score out of 10.

2. **Unit Test Generation**:

- The goal of Unit Test is to validate the provided solution code. For programming or algorithmic tasks, ensure the tests verify that the program produces the expected correct outputs for given inputs. For engineering-oriented tasks, provide a quick end-to-end validation of the pipeline while avoiding unnecessary or overly complex tests.
- Output a complete Python file using `pytest` that can be run directly. Write multiple `pytest` `def test_*` functions, each testing one critical functionality. Focus on verifying that functionality works as intended, including logical correctness, implementation correctness, or edge conditions.
- The Solution Code may consist of one or multiple files. Each file path will be indicated at the beginning with an `@file_source` tag. You should `import` needed functions/modules from these paths, or redefine small logic snippets when necessary.
- For data-driven tasks, use the **same data-loading method** as in the solution; then slice a small subset. Run with `epochs=1` (or the minimum allowed), **small batch size**. Do not add strict numeric checks. Keep all checks **lightweight** and avoid brittle thresholds.
- For tasks that do not rely on a given dataset, manually create data as needed. Ensure all tests execute quickly.

Basic Principles

1. **Task Completion**

- **9-10**: The code implements all core functionalities and directly achieves the final goal.
- **6-8**: Most functionalities are implemented, with a few subtasks not covered.
- **3-5**: Covers core functionalities but misses several important requirements.
- **1-2**: The core objective is not met and the solution significantly diverges from the task intent.

2. **Code Correctness**

- **9-10**: Algorithm logic is rigorous, with no runtime or boundary errors; robust handling of exceptional cases.
- **6-8**: Core algorithm is correct, with only minor defects in extreme or rare boundary cases; basic error handling is present.
- **3-5**: Several logic or runtime errors exist; error handling is missing or inadequate.
- **1-2**: Major functionality is severely broken and does not run correctly; no error recovery strategies are implemented.

3. **Efficiency**

- **9-10**: The algorithm has low complexity with efficient execution on the critical path; excellent resource (time/memory) utilization.
- **6-8**: Performs well in most scenarios, with a few optimizable bottlenecks.
- **3-5**: Contains obvious redundancies or inefficient implementations; inadequate performance on medium-sized inputs.
- **1-2**: The algorithm is bloated with excessive complexity; performs poorly on typical inputs.

Unit Test Example

```
```python
from solution import add

def test_add_positive_numbers():
 assert add(2, 3) == 5
```

```
def test_add_with_zero():
 assert add(0, 5) == 5
 assert add(5, 0) == 5
```

```
def test_add_negative_numbers():
 assert add(-1, -1) == -2
```

The final response should strictly follow this format:

<score> ... </score> - contains only the final weighted score, which should be a simple number.  
<unit\_test> ... </unit\_test> - the generated pytest-style unit tests without code block tags.

[The Begin of Code Task]\n{task\_summary}\n[The End of Code Task]\n\n[The Begin of Code Answer]\n{solution}\n[The End of Code Answer]\n