# CODERM-NT: Reward Model for Code RL without Unit Tests

**Xiao Xia**[1*], **Dan Zhang**[1,2*†], **Tianrui Sun**[3]
[1]Tsinghua University [2]National University of Singapore
[3]Foundational Technologies, Siemens Ltd., China

## Abstract

Providing accurate reward signals for code generated by large language models (LLMs) is a significant challenge in applying reinforcement learning (RL) to code generation. Existing methods rely on unit tests to evaluate code correctness and provide rewards, which are hindered by the difficulty of acquiring and verifying reliable unit tests at scale. In this work, we propose CODERM-NT, a code reward model with no reliance on unit tests. Our method leverages Monte Carlo Tree Search guided by LLMs to generate code snippets and judges execution traces to annotate code with reward signals. We use the rewards to train CODERM-NT that is capable of providing rewards for code during RL. CODERM-NT also facilitates curriculum learning by scoring and sorting training samples based on their difficulty. Experimental results demonstrate that training with CODERM-NT consistently outperforms synthetic unit test-based rewards, yielding superior performance on multiple code generation benchmarks. Additionally, curriculum learning based on CODERM-NT further enhances model performance. Our code and dataset are available at: https://github.com/THUDM/CodeRM-NT.

## 1 Introduction

Reinforcement learning (RL) has become a prominent method for advancing the capabilities of large language models (LLMs) in recent years (Ouyang et al., 2022; Jaech et al., 2024; Guo et al., 2025). A central challenge in this paradigm is designing rewards that credibly reflect the quality of LLM outputs (Le et al., 2022; Wang et al., 2023; Lambert et al., 2024). This issue is critical in code generation, where the output quality depends on whether the generated code correctly implements the target functionality of a coding question. Thus, improving the performance of RL on code generation requires rewards that provide accurate and trustworthy feedback on whether the code meets the intended specification (Dou et al., 2024).

The functional correctness of generated code is typically assessed through unit testing, which involves executing the code with specified inputs and verifying whether the resulting outputs align with the expected outcomes (Le et al., 2022; Shojaee et al., 2023; Liu et al., 2023a; Gehring et al., 2025). Recent LLMs frequently rely on unit tests as the primary source of verifiable rewards for code generation during RL (Guo et al., 2025; Team et al., 2025; Zeng et al., 2025a; Yang et al., 2025; Seed et al., 2025). However, obtaining reliable unit tests for coding questions poses the following challenges:

**(1) Curating unit tests for large datasets is prohibitively expensive.** Unit tests must be faithfully aligned with the target functionality of a programming task, rewarding only correct implementations while penalizing erroneous ones (Dinella et al., 2022). Existing coding datasets like APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), and TACO (Li et al., 2023) derive their unit tests from expert-annotated and validated resources available on coding platforms or by extracting input–output pairs from verified ground-truth solutions. However, manually authoring and validating tests or reference solutions for each question is costly and becomes nearly infeasible when curating new coding datasets (Daka and Fraser, 2014), limiting their scalability.

**(2) Automatically synthesized unit tests often produce unreliable reward signals.** An established approach to automatically generate unit tests for large-scale code datasets leverages LLMs to synthesize inputs for the target code and infer the corresponding outputs. However, the reliability of such tests remains limited, as LLMs must predict code outputs through complex reasoning processes that are prone to logical errors (Yuan et al., 2024;

[*] Equal contribution.
[†] Corresponding author.



*Findings of the Association for Computational Linguistics: ACL 2026*, pages 43316–43333

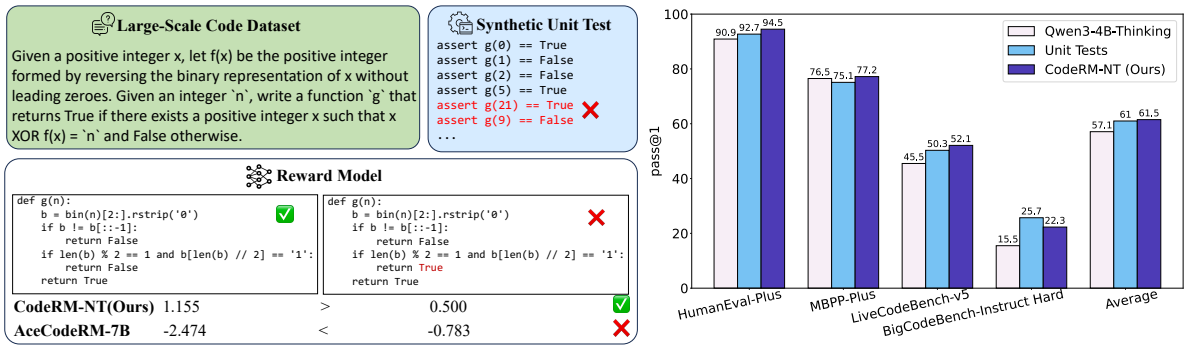


Figure 1: Left: Illustration of rewards for code from synthetic unit tests and reward models. Right: Overall performance of Qwen3-4B-Thinking trained with different rewards on code generation benchmarks. As a reward model built without unit tests, CODERM-NT outperforms synthetic unit tests on most benchmarks and the average result.

Huang et al., 2025), and flawed unit tests may incorrectly validate erroneous code or reject correct code, thereby introducing misleading rewards (Jain et al., 2025a; Ma et al., 2025). To address this issue, previous works (Chen et al., 2023; Xu et al., 2025; Wang et al., 2025) employ self-verification strategies that sample candidate solutions from models and discard unit tests where such solutions fail. Nonetheless, the self-verification procedures still cannot fully eliminate errors and prevent noise in rewards (Zeng et al., 2025b), while introducing substantial computational overhead. For example, KodCode-V1 (Xu et al., 2025) samples up to 10 solutions and unit tests for each of 447K questions.

To overcome the limitations associated with unit-test-based rewards, we propose a novel reward modeling method tailored to function-level Python code generation. This method trains CODERM-NT, a code reward model with no reliance on unit tests to evaluate code quality. To obtain high-quality rewards for code data, we employ Monte Carlo Tree Search (MCTS) guided by LLM-as-a-Judge to annotate code data with rewards without relying on unit tests. This reward-labeled data is then used to train a reward model that estimates the code's functional correctness to provide rewards. We adopt CODERM-NT for RL with GRPO (Shao et al., 2024) on coding tasks and compare the performance to using synthetic unit tests to provide rewards. Comprehensive experiments show that RL with CODERM-NT consistently outperforms the standard unit-test-based approach across benchmarks, as training Qwen3-4B-Thinking (Yang et al., 2025) with CODERM-NT achieves the highest average performance among all our evaluated models. Moreover, unlike unit-test-based approaches that require generating tests for each new training example, our reward model is trained once and can be reused across diverse datasets, reducing the task-specific overhead. Beyond serving as a source of rewards, CODERM-NT can further rerank training samples according to their predicted difficulty and organize data from easier to harder problems, facilitating curriculum learning and delivering additional improvements.

We summarize our contributions as follows:

- We propose CODERM-NT, a novel reward model for code trained with rewards obtained via MCTS, facilitating effective reward learning without dependency on unit tests.
- We show that CODERM-NT can evaluate and rank training samples, thereby enabling a curriculum-based data ordering strategy for RL that enhances model training.
- Experiments demonstrate that RL guided by CODERM-NT leads to improved performance across multiple code generation benchmarks compared to synthetic unit tests, highlighting the efficacy of our proposed approach.

## 2 Preliminary of RL and MCTS

### 2.1 Reinforcement Learning on LLMs

Reinforcement learning (RL) has emerged as a powerful paradigm for aligning LLMs with task-specific objectives. In RL for text generation, an LLM is treated as a policy $\pi_\theta$ that generates a response $\mathbf{y}$ to a question $\mathbf{x}$. A reward signal $\mathcal{R}(\mathbf{x}, \mathbf{y})$ guides the optimization of $\pi_\theta$ through policy–gradient–based algorithms such as PPO (Schulman et al., 2017). Group Relative Policy Optimization (GRPO) (Shao et al., 2024) is a variant of

PPO that estimates advantages within a group of sampled responses. Given a query $q$, a policy $\pi_{\text{old}}$ generates $G$ rollout responses $\{o_i\}_{i=1}^{G}$ with corresponding rewards $\{\mathcal{R}_i\}_{i=1}^{G}$. The advantage of each sample is computed as $A_i = \frac{\mathcal{R}_i - \text{mean}(\{\mathcal{R}_i\}_{i=1}^{G})}{\text{std}(\{\mathcal{R}_i\}_{i=1}^{G})}$, which is then used to optimize $\pi_\theta$ by maximizing:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim\mathcal{D},\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|q)}$$
$$\frac{1}{G}\sum_{i=1}^{G}\left(\min\left(r_t(\theta), \text{clip}\left(r_t(\theta), 1-\varepsilon_{\text{low}}, 1+\varepsilon_{\text{high}}\right)\right)A_i - \beta\mathbb{D}_{\text{KL}}(\pi_\theta\|\pi_{\text{ref}})\right), \tag{1}$$

where the importance sampling ratio $r_t(\theta) = \frac{\pi_\theta(o_i|q)}{\pi_{\theta_{\text{old}}}(o_i|q)}$ and $\mathbb{D}_{\text{KL}}(\pi_\theta\|\pi_{\text{ref}})$ is a KL penalty term, and $\varepsilon_{\text{low}}$ and $\varepsilon_{\text{high}}$ adjust the clip range of $r_t(\theta)$, as suggested by DAPO (Yu et al., 2025).

### 2.2 Monte Carlo Tree Search (MCTS)

MCTS is a search algorithm widely used in decision-making tasks like Go (Silver et al., 2016) and reasoning (Zhang et al., 2024). MCTS incrementally builds a tree of possible actions and outcomes, where each node corresponds to a state or partial solution. The search iterates through four stages: 1) *Selection* traverses the tree, choosing the child node with the highest Upper Confidence Bound for Trees (UCT) (Kocsis and Szepesvári, 2006) until a node suitable for expansion is reached, 2) *Expansion* adds new nodes by generating successors of the selected node, 3) *Simulation* evaluates new nodes to estimate their quality, and 4) *Backpropagation* updates the values of ancestor nodes based on simulation outcomes.

## 3 Method

In this section, we describe our approach, which comprises training CODERM-NT for Python code generation tasks and reinforcement learning on code. The overview of our method is in Figure 2.

### 3.1 CODERM-NT

We generate responses and scores for coding questions, forming a dataset to train a model to predict rewards based on questions and responses.

#### 3.1.1 Reward Data Curation

We sample responses to each question using LLMs and obtain rewards using MCTS, building a search tree based on the codes in the responses. The contents of the nodes in the tree correspond to the responses, and the node's valuation is treated as the reward. We follow the MCTS pipeline described in Section 2.2 and provide a detailed description of the key components of MCTS below, with additional details in Appendix A.1.1 and A.2.1.

**Tree Formulation.** Given a coding question $\mathbf{x}$, an LLM $\pi$ generates a response $\mathbf{y} = \pi(\mathbf{x})$ containing a piece of Python code $\mathbf{c}$. The code is split into multiple semantically meaningful snippets $\mathbf{c} = (c_1, c_2, \cdots, c_T)$ as the basic unit of searching. The code is split at the following locations: (1) at the end of a code block indicated by a decrease in indentation, and (2) at print statements inserted during the simulation process that output the execution state of the code. The snippets $(c_1, c_2, \cdots, c_T)$ are represented by nodes $(s_0, s_1, s_2, \cdots, s_T)$ in the tree, where $c_i$ is represented by $s_i$ and the root $s_0$ contains an empty string. $s_i$ is the parent of $s_{i+1}$ for each $0 \le i < T$, and $s_T$ is a leaf node.

For each question, we start by generating a response $\mathbf{y}_0$ containing code $\mathbf{c}_0 = (c_1^0, c_2^0, \cdots, c_{T^0}^0)$ to initialize the tree, which contains nodes $s_0, s_1^0, s_2^0, \cdots, s_{T^0}^0$ forming a path from the root to the leaf, then perform simulation on $s_{T^0}^0$. Each node $s_i$ is assigned a visit count $N(s_i)$ and an estimated $q$-value $Q(s_i)$, both initialized as 0.

**Selection.** We limit a maximum number of $k$ expansions for each node to encourage the exploration of more nodes and select a node that has not reached the maximum number of expansions in each iteration. We prioritize selecting the initial nodes for expansion. After every initial node has been selected once, MCTS starts from the root $s_0$ and traverses the tree until it reaches a node available for expansion or a leaf. We use the UCT criterion as shown in Equation 5. The traversal terminates upon reaching a node that is either eligible for expansion or a leaf node. If the node is not a leaf, expand it; otherwise, perform a simulation with the node.

**Expansion.** For a non-leaf node $s_t$, the expansion creates its descendant nodes by generating subsequent code snippets. From the path $(s_0, \cdots, s_t)$, we extract corresponding code snippets $(c_1, \cdots, c_t)$, which constitute a partially completed code. We employ an LLM to complete this partial code, forming a new response $\mathbf{y}'$, which includes a complete code $\mathbf{c}'$ with $(c_1, \cdots, c_t)$ being its prefix. LLM can refer to previously expanded codes on $s_t$ and their execution results to improve the quality of $\mathbf{c}'$. $\mathbf{c}'$ is transformed into new nodes in the subsequent simulation process.

**Simulation and Backpropagation.** Upon cre-

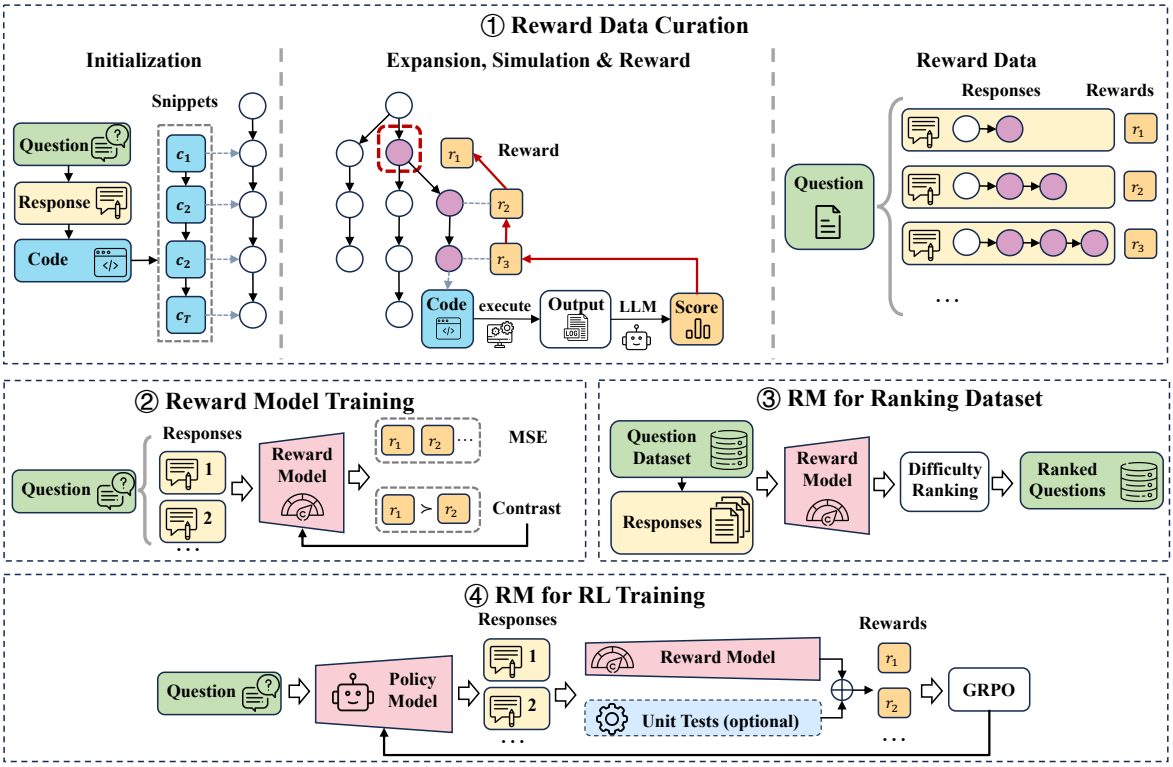


Figure 2: Overview of our method. We use MCTS guided by LLM judges to collect data with rewards, which we use to train CODERM-NT, our reward model. We utilize CODERM-NT to provide rewards for reinforcement learning and rerank training data by difficulty.

ating a new response $\mathbf{y}$ that includes code $\mathbf{c}$, we evaluate $\mathbf{c}$'s functional correctness by analyzing its execution process. Specifically, we record the intermediate states and results of the code's execution process, including key variables, conditional branches, function calls, and runtime errors. The LLM identifies them and inserts print statements accordingly after it generates code during initialization and expansion. The LLM also writes sample calls to the code to maximize execution coverage.

We execute the modified code, and the LLM assigns a score for its output. Based on the question, code, and output, the LLM assesses whether the code correctly addresses the task, appropriately handles edge cases, and conforms to established best practices. The evaluation yields a score normalized within the range of 0 to 1 as the simulated reward $R(s_T)$. We discuss the feasibility of this LLM-as-a-Judge approach in Appendix A.3.1. Detailed prompts for inserting print statements when generating code, writing calling examples, and scoring are provided in Appendix A.4.1.

We incorporate the code into the tree. The modified code is denoted as $(c_1, \cdots, c_t, c'_{t+1}, \cdots, c'_{T'})$, where $(c'_{t+1}, \cdots, c'_{T'})$ is the continuation of $(c_1, \cdots, c_t)$. The new code snippets $c'_{t+1}, \cdots, c'_{T'}$ form new nodes $s'_{t+1}, \cdots, s'_{T'}$, where $s'_{t+1}$ becomes a child of $s_t$, connected to the new leaf $s'_{T'}$ via the intermediate nodes $s'_{t+2}, \cdots, s'_{T'-1}$. Finally, a backpropagation step updates $N(s)$ and $Q(s)$ of all nodes along the path from $s_0$ to $s_T$ with $R(s_T)$ as defined in Equation 6. We provide an example of inserting print statements and splitting code into snippets in Appendix A.2.2.

**Reward Data.** After a certain number of MCTS iterations for each question $\mathbf{x}$, we gather reward data from the non-root nodes in the tree. For a node $s_t$, extract $c_1, \cdots, c_t$ from $(s_0, \cdots, s_t)$ to form code $\mathbf{c}_t$ and its corresponding response $\mathbf{y}_t$, with the $q$-value $Q(s_t)$ as reward $r_t$. Each question $\mathbf{x}$, its set of complete or partial responses $\mathbf{y}$, and the reward $r$ for answer $\mathbf{y}$ constitute the reward dataset.

### 3.1.2 CODERM-NT Training

We initialize CODERM-NT, denoted as $\mathcal{R}_\phi$, by appending a linear layer to a pretrained LLM and training it on our curated reward dataset. $\mathcal{R}_\phi$ is first trained with scalar data and MSE loss, followed by pairwise data and contrastive loss.

**Scalar Data with MSE Loss.** The data is orga-

nized as $D_{\text{scalar}} = \{(\mathbf{x}, \mathbf{y}, r)\}$, where $\mathbf{x}$ denotes a question, $\mathbf{y}$ represents a complete or partial response, and $r$ is $\mathbf{y}$'s reward. We minimize the loss:

$$L_{\text{MSE}} = \mathbb{E}_{(\mathbf{x},\mathbf{y},r)\sim D_{\text{scalar}}}(\mathcal{R}_\phi(\mathbf{x}, \mathbf{y}) - r)^2. \quad (2)$$

**Pairwise Data with Contrastive Loss.** The data is denoted as $D_{\text{pair}} = \{(\mathbf{x}, \mathbf{y}_w, \mathbf{y}_l)\}$ where $\mathbf{y}_w$'s reward is higher than $\mathbf{y}_l$. For each $\mathbf{x}$, $\mathbf{y}_w$, and $\mathbf{y}_l$, minimize the loss:

$$L_{\text{contrastive}} = -\mathbb{E}_{(\mathbf{x},\mathbf{y}_w,\mathbf{y}_l)\sim D_{\text{pair}}} \log \sigma(\mathcal{R}_\phi(\mathbf{x}, \mathbf{y}_w) - \mathcal{R}_\phi(\mathbf{x}, \mathbf{y}_l)), \quad (3)$$

where $\sigma$ is the logistic function.

### 3.2 Integrating CODERM-NT into Online RL

We conduct online RL on open-source LLMs using the GRPO algorithm. We study two aspects of using CODERM-NT within the RL framework.

#### 3.2.1 CODERM-NT as Scorer

We use CODERM-NT to assign scores as rewards to LLM's outputs during RL. We consider two sources of rewards: reward model outputs and execution results of unit tests.

**Scoring with Reward Model.** Given a question $\mathbf{x}$ and its response $\mathbf{y}$, the reward model $\mathcal{R}_\phi$ produces $\mathcal{R}_\phi(\mathbf{x}, \mathbf{y})$ as the reward. We only use responses that contain complete code for training.

**Scoring with Unit Tests.** Given a question $\mathbf{x}$, whose response $\mathbf{y}$ contains code $\mathbf{c}$, and $N$ unit tests $\mathbf{T} = \{\mathbf{t}_n\}_{n=1}^N$, execute $\mathbf{c}$ for each test $\mathbf{t}_n$ and obtain the reward as $\mathcal{R}_{\text{test}}(\mathbf{x}, \mathbf{y}) = \mathbf{1}_{\mathbf{c} \text{ passes all } \mathbf{t}_n \in \mathbf{T}}$.

**Scoring by Combining both Sources.** Compute a weighted sum of scores from both sources to balance them. Given the reward model score $\mathcal{R}_\phi(\mathbf{x}, \mathbf{y})$ and the unit test score $\mathcal{R}_{\text{test}}(\mathbf{x}, \mathbf{y})$, the combined reward $\mathcal{R}_{\text{combined}} = \lambda\mathcal{R}_\phi(\mathbf{x}, \mathbf{y}) + (1-\lambda)\mathcal{R}_{\text{test}}(\mathbf{x}, \mathbf{y})$, where $\lambda \in [0, 1]$ is a weight parameter.

#### 3.2.2 CODERM-NT as Ranker

We leverage CODERM-NT to arrange the order of training data based on their difficulty. For each question $\mathbf{x}_i$ from a training dataset $\{\mathbf{x}_i\}_{i=1}^N$, we generate $K$ responses $\{\mathbf{y}_i^j\}_{j=1}^K$ using the policy LLM. Each response is evaluated by the reward model, yielding a set of scores $\{\mathcal{R}_\phi(\mathbf{x}_i, \mathbf{y}_i^j)\}_{j=1}^K$, and the average of these scores $\overline{\mathcal{R}}_\phi(\mathbf{x}_i)$ is taken as the score of question $\mathbf{x}_i$, with lower scores representing higher difficulty for the LLM. We rerank the questions in descending order of their scores:

$$\begin{aligned} &\mathbf{x}_{\pi(1)}, \mathbf{x}_{\pi(2)}, \ldots, \mathbf{x}_{\pi(N)} \\ &\text{where} \quad \overline{\mathcal{R}}_\phi(\mathbf{x}_{\pi(1)}) \geq \cdots \geq \overline{\mathcal{R}}_\phi(\mathbf{x}_{\pi(N)}). \end{aligned} \quad (4)$$

During training, the questions are presented to the LLM in this order, beginning with easier data and gradually introducing more difficult samples.

#### 3.2.3 Online RL Training

We employ GRPO to train the policy $\pi_\theta$ on the training dataset, which may be reordered as described in Section 3.2.2 before training. During training, rollout responses are scored with rewards described in Section 3.2.1, and $\pi_\theta$ is updated to maximize the GRPO objective as in Equation 1.

## 4 Experiments

### 4.1 Reward Model Setup

**Reward Data Curation.** During MCTS, we employ Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) to generate code snippets and assess execution results. For each question, we conduct MCTS for 10 iterations, with up to $k = 4$ expansions per node. We use Magicoder-OSS-Instruct-75K (Wei et al., 2024) as our source of Python coding questions, randomly sample 12,000 questions to perform MCTS. Of these, 6,000 questions are used to construct scalar data comprising 196,098 complete or partial responses and their corresponding rewards, and the remaining 6,000 are used for pairwise data consisting of 71,205 response pairs. The curation process is performed only once to train the reward model. More details are in Appendix A.1.2.

**Reward Model Training.** We use Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as the backbone of our reward model and train it with MSE loss and contrastive loss on the corresponding datasets, training for 2 epochs on each dataset with a batch size of 128. We set the learning rate to 1e-6 for scalar data and 5e-7 for pairwise data.

### 4.2 Reinforcement Learning Setup

**Models.** We perform GRPO training on 5 models: Qwen2.5-Coder-1.5B-Instruct, Qwen2.5-Coder-3B-Instruct, Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), GLM-4-9B-0414 (GLM et al., 2024), and Qwen3-4B-Thinking (Yang et al., 2025).

**Datasets.** For Qwen2.5 models and GLM-4-9B-0414, we use KodCode-V1 (Xu et al., 2025), sampling 9,000 questions. For Qwen3-4B-Thinking,

Table 1: Overall results on coding benchmarks. We report the results of models untrained or trained with unit tests, AceCodeRM-7B, CodeRM-8B, CODERM-NT (ours), or a combination of CODERM-NT and unit tests. KodCode-V1 represents 9,000 questions sampled from KodCode-V1 and OCI 5k represents 5,000 questions sampled from OpenCodeInstruct. We report the average results by taking the mean of all results.

| Data | Reward | | HumanEval | | MBPP | | LCB-v5 | BCB-I-Hard | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| | RM | Unit Test | Base | Plus | Base | Plus | | | |
| Qwen2.5-Coder-1.5B-Instruct | | | 70.1 | 66.5 | 70.2 | 60.3 | **5.8** | 5.4 | 46.4 |
| KodCode-V1 9k | ✗ | ✓ | 73.2 | 67.7 | 70.9 | 61.1 | 5.1 | 6.1 | 47.4 |
| | AceCodeRM-7B | ✗ | 70.7 | 66.5 | 70.6 | 61.1 | 5.2 | **7.4** | 46.9 |
| | Ours | ✗ | 75.0 | 69.5 | 72.0 | 60.8 | 5.5 | **7.4** | 48.4 |
| | Ours | ✓ | **76.2** | **71.3** | **73.3** | **61.6** | 4.9 | 5.4 | **48.8** |
| Qwen2.5-Coder-3B-Instruct | | | 84.8 | 80.5 | 74.6 | 63.5 | 12.8 | 12.8 | 54.8 |
| KodCode-V1 9k | ✗ | ✓ | 86.6 | 82.3 | 74.9 | 64.6 | 13.0 | **15.5** | 56.2 |
| | AceCodeRM-7B | ✗ | 86.0 | 82.3 | 75.7 | 65.9 | 13.2 | 12.8 | 56.0 |
| | Ours | ✗ | **88.4** | 82.3 | 75.9 | **66.1** | **13.6** | 14.2 | **56.8** |
| | Ours | ✓ | 87.2 | **82.9** | **76.7** | **66.1** | 13.5 | 14.2 | **56.8** |
| Qwen2.5-Coder-7B-Instruct | | | 88.4 | 84.8 | 85.4 | 73.0 | **17.8** | 19.6 | 61.5 |
| KodCode-V1 9k | ✗ | ✓ | **90.9** | **87.8** | 85.4 | 73.0 | 17.3 | 18.2 | 62.1 |
| | AceCodeRM-7B | ✗ | 89.0 | 84.8 | 84.4 | 72.8 | 17.1 | 21.0 | 61.5 |
| | Ours | ✗ | 90.2 | 86.0 | **86.8** | **74.6** | 17.5 | 18.2 | **62.2** |
| | Ours | ✓ | 89.6 | 85.4 | 86.0 | 73.5 | 17.1 | **21.6** | **62.2** |
| GLM-4-9B-0414 | | | 82.9 | 79.3 | 79.9 | 67.2 | 14.9 | 15.5 | 56.6 |
| KodCode-V1 9k | ✗ | ✓ | 84.1 | 79.9 | 81.0 | **69.0** | 15.4 | 15.5 | 57.5 |
| | AceCodeRM-7B | ✗ | 82.9 | 79.3 | 81.7 | 68.3 | 12.6 | 16.2 | 56.8 |
| | Ours | ✗ | **87.2** | **81.7** | 79.9 | 67.2 | 15.3 | **18.2** | **58.3** |
| | Ours | ✓ | 83.5 | 79.9 | **82.5** | 68.5 | **16.1** | 16.2 | 57.8 |
| Qwen3-4B-Thinking | | | 97.0 | 90.9 | 91.0 | 76.5 | 45.5 | 15.5 | 69.4 |
| OCI 5k | ✗ | ✓ | **97.6** | 92.7 | 91.0 | 75.1 | 50.3 | **25.7** | 72.1 |
| | AceCodeRM-7B | ✗ | **97.6** | 89.6 | **92.6** | **77.2** | 49.7 | 20.9 | 71.3 |
| | CodeRM-8B | ✗ | **97.6** | 90.2 | 92.1 | 76.7 | 51.5 | 18.9 | 71.2 |
| | Ours | ✗ | **97.6** | **94.5** | **92.6** | **77.2** | **52.1** | 22.3 | **72.7** |
| | Ours | ✓ | 95.7 | 93.3 | 90.7 | 76.5 | 47.9 | 21.6 | 71.0 |

we use OpenCodeInstruct (Ahmad et al., 2025), sampling 5,000 questions. Both datasets contain synthetic unit tests. When reranking datasets, we set the number of responses $K = 8$.

**Training.** During RL, rewards are derived from reward models, unit tests, or their combination. For reward models, we compare CODERM-NT against AceCodeRM-7B (Zeng et al., 2025b), another reward model tailored for code generation. On Qwen3-4B-Thinking, we also compare against CodeRM-8B (Ma et al., 2025). The RL hyperparameters are reported in Appendix A.1.3.

**Evaluation.** We evaluate LLMs on four benchmarks for code generation: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), LiveCodeBench-v5 (Jain et al., 2025b), and BigCodeBench-Instruct Hard (Zhuo et al., 2025). For HumanEval and MBPP, we also include their EvalPlus versions (Liu et al., 2023b). We use pass@1 (Chen et al., 2021) as the evaluation metric.

### 4.3 Main Results

**CODERM-NT as Scorer.** Table 1 summarizes the overall performance of models trained

Table 2: Results of training with reranked data. Results are obtained by training with different rewards and choosing the one with the highest average score. Full results of training with reranked data are provided in Table 11.

| Model | HumanEval | | MBPP | | LCB-v5 | BCB-I-Hard | Avg. |
|---|---|---|---|---|---|---|---|
| | Base | Plus | Base | Plus | | | |
| Qwen2.5-Coder-1.5B-Instruct | 70.1 | 66.5 | 70.2 | 60.3 | 5.8 | 5.4 | 46.4 |
| +KodCode 9k | **76.2** | **71.3** | 73.3 | 61.6 | 4.9 | 5.4 | 48.8 |
| +KodCode 9k (reranked) | 74.4 | 69.5 | **73.5** | **63.0** | **7.8** | **6.1** | **49.1** |
| Qwen2.5-Coder-3B-Instruct | 84.8 | 80.5 | 74.6 | 63.5 | 12.8 | 12.8 | 54.8 |
| +KodCode 9k | **88.4** | **82.3** | 75.9 | 66.1 | **13.6** | 14.2 | 56.8 |
| +KodCode 9k (reranked) | 85.4 | **82.3** | **79.1** | **68.0** | 13.4 | **16.2** | **57.4** |
| Qwen2.5-Coder-7B-Instruct | 88.4 | 84.8 | 85.4 | 73.0 | 17.8 | **19.6** | 61.5 |
| +KodCode 9k | **90.2** | 86.0 | **86.8** | **74.6** | 17.5 | 18.2 | 62.2 |
| +KodCode 9k (reranked) | **90.2** | **87.8** | 86.0 | 73.3 | **18.0** | 18.9 | **62.4** |
| GLM-4-9B-0414 | 82.9 | 79.3 | 79.9 | 67.2 | 14.9 | 15.5 | 56.6 |
| +KodCode 9k | 87.2 | **81.7** | 79.9 | 67.2 | 15.3 | 18.2 | 58.3 |
| +KodCode 9k (reranked) | **87.2** | 81.1 | **80.7** | **68.5** | **16.9** | **19.6** | **59.0** |
| Qwen3-4B-Thinking | 97.0 | 90.9 | 91.0 | 76.5 | 45.5 | 15.5 | 69.4 |
| +OCI 5k | **97.6** | **94.5** | **92.6** | 77.2 | **52.1** | **22.3** | **72.7** |
| +OCI 5k (reranked) | 97.0 | 93.3 | 91.3 | **77.5** | 50.3 | 20.9 | 71.7 |

with different rewards. Training with CODERM-NT consistently yields superior results compared to unit tests and other reward models. For Qwen2.5-Coder-1.5B and 3B, GLM-4-9B-0414, and Qwen3-4B-Thinking, CODERM-NT leads to clear improvements over unit tests in terms of average performance. Notably, training Qwen3-4B-Thinking with CODERM-NT improves the results on LCB-v5 by 6.6% and BigCodeBench-Instruct Hard by 6.8%, achieving an average score of 72.7%, the highest across all evaluated models. On Qwen2.5-Coder-1.5B and 3B and Qwen3-4B-Thinking, CODERM-NT outperforms unit tests on nearly all benchmarks. For Qwen2.5-Coder-7B, the performance of CODERM-NT is comparable to that of unit tests. Furthermore, combining CODERM-NT with unit tests improves performance relative to using unit tests alone, except for Qwen3-4B-Thinking, and achieves the highest average scores for Qwen2.5 models, confirming that CODERM-NT can complement the guidance of unit tests. The detailed results of combining rewards are in Table 10. Overall, these results demonstrate that CODERM-NT provides a reliable alternative to synthetic unit tests in settings where unit tests are unavailable, maintaining competitive performance across different models and datasets.

**CODERM-NT as Ranker.** We conduct RL on the reranked data using rewards from CODERM-NT, unit tests, and the combination of the two, and report the result with the highest average score among the three settings for each model. Table 2 presents the results, showing that training with reranked data outperforms original data across most benchmarks for all models and achieves the highest average score across most models. For Qwen3-4B-Thinking, while using reranked data achieves a lower average performance compared to original data, it surpasses or matches the original data on 4 out of 6 benchmarks with the only notable shortfall on BigCodeBench-Instruct Hard. These findings suggest that difficulty-aware ordering with CODERM-NT leads to better performance and generalization across tasks. The full results of training with reranked data are in Table 11.

### 4.4 Intrinsic Study of CODERM-NT

While the previous experiments establish the downstream efficacy of our method in RL, it is crucial to validate the quality of the reward itself. We rigorously evaluate CODERM-NT's accuracy and conduct ablation studies to justify the necessity of our MCTS-driven data construction and training pipeline. We use the hep-python subtest of RewardBench (Lambert et al., 2025), where the models

Table 3: Performance of reward models on the hep-python split of RewardBench. # Coding Qs indicates the number of coding questions in the training data. Partial Resp. indicates whether the training data includes responses that contain partial responses from non-root nodes of MCTS.

| Method | Base Model | # Coding Qs | Partial Resp. | Loss | Acc. |
|---|---|---|---|---|---|
| Skywork-V2-Llama-3.1-8B | Llama3.1-8B | – | ✗ | Contrastive | 0.780 |
| AceCodeRM-7B | Qwen2.5-Coder-7B | 87k | ✗ | Contrastive | 0.957 |
| CODERM-NT (Ours) | Qwen2.5-Coder-7B | 12k | ✗ | MSE | 0.652 |
| | | 12k | ✓ | MSE | 0.909 |
| | | 12k | ✓ | Contrastive | 0.939 |
| | | 12k | ✓ | MSE → Contrastive | **0.963** |

should identify the correct response from two candidates. The results are presented in Table 3.

#### 4.4.1 Accuracy of Rewards

We compare CODERM-NT against two strong baselines: Skywork-V2-Llama-3.1-8B (Liu et al., 2025), the best performing model on RewardBench 2 (Malik et al., 2025), and AceCodeRM-7B (Zeng et al., 2025b), a specialized code reward model. The results in Table 3 reveal that CODERM-NT achieves the highest accuracy among all models, surpassing prior baselines. CODERM-NT outperforms AceCodeRM-7B with only 13.8% of the number of training questions, demonstrating the superior quality of our reward data and the strong suitability of our reward model for coding tasks.

#### 4.4.2 Necessity of MCTS

A central element of our approach is incorporating rewards derived from partial responses via MCTS to train CODERM-NT. To assess the impact of this design, we perform an ablation study by removing partial responses from MCTS and training CODERM-NT solely on complete responses. As reported in Table 3, eliminating partial responses leads to CODERM-NT's accuracy drastically dropping to 65.2%. This decline indicates that modeling code correctness based solely on final outcomes is insufficient and highlights the necessity of MCTS for training a reliable reward model.

#### 4.4.3 Choice of Training Objectives

We analyze the impact of loss functions for training CODERM-NT. MSE loss is useful for stabilizing the rewards within a bounded range, while contrastive loss sharpens its ability to perform fine-grained discrimination between responses. Table 3 reveals that contrastive loss results in higher accuracy than MSE loss, and training with both losses sequentially achieves the highest accuracy. These results suggest that combining the strengths of both objectives yields the most accurate reward model.

## 5 Related Work

Recent LLMs extensively rely on RL for alignment, wherein models are optimized using rewards derived from human feedback (Ouyang et al., 2022), AI preferences (Bai et al., 2022), or rule-based heuristics (Lambert et al., 2024; Guo et al., 2025), typically through algorithms like PPO (Schulman et al., 2017), DPO (Rafailov et al., 2023), and GRPO (Shao et al., 2024). Previous research adapts RL to code generation. For instance, CodeRL (Le et al., 2022) employs an actor-critic approach with execution-based rewards, while PPOCoder (Shojaee et al., 2023) demonstrates the effectiveness of PPO in code generation. Later approaches, including RLTF (Liu et al., 2023a), StepCoder (Dou et al., 2024), and RLEF (Gehring et al., 2025), rely on unit tests for rewards, but presuppose the availability of ground-truth unit tests within the training data. To overcome this limitation, methods such as CodeDPO (Zhang et al., 2025), AceCoder (Zeng et al., 2025b), and CodeRM (Ma et al., 2025) generate synthetic unit tests for reward evaluation. However, ensuring the correctness of automatically generated tests requires additional verification, which remains challenging. In this work, we propose training a reward model that achieves superior performance on datasets lacking ground-truth tests.

## 6 Conclusion

We present CODERM-NT, a reward model for reinforcement learning on code that does not rely on ground-truth unit tests. By using MCTS guided by LLMs, we construct fine-grained reward data for code generation and train a reward model that accurately assesses the functional correctness of code. Experiments show that CODERM-NT outperforms

unit tests across multiple benchmarks, and combining it with unit tests yields complementary improvements. Moreover, using CODERM-NT to rerank data further enhances training. These findings demonstrate that CODERM-NT provides an effective alternative to unit-test-based supervision for code.

## Limitations

**Scope of Language.** This study focuses on Python function-level generation, consistent with the scope of research adopted by many prior benchmarks (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023b; Zhuo et al., 2025) and studies (Le et al., 2022; Dou et al., 2024; Chen et al., 2023). While this study does not empirically evaluate cross-language generalization, the central idea of constructing rewards from execution and intermediate MCTS nodes is not inherently language-specific and could be extended to other languages. The splitting strategy used during MCTS relies on syntactic features, such as indentation, as well as key code identified by the LLM. This strategy can be implemented for other languages, including C++ or JavaScript, using syntax features like AST (Abstract Syntax Tree) representations and structured control flow boundaries. The applicability of our proposed method to other programming languages is left for future work.

**Generalization to Complex Software Engineering Tasks.** Our work primarily addresses function-level code generation problems (e.g., HumanEval, MBPP, LiveCodeBench). The MCTS search space is formulated by splitting code into snippets based on indentation and print statements, which is effective for these types of procedural tasks. However, the extent to which the proposed method generalizes to more complex software engineering scenarios remains an open question. These scenarios often involve multi-file projects, external API interactions, and stateful systems, where conducting searches and capturing execution traces demand careful consideration and design. Expanding the MCTS search and evaluation framework to handle these advanced contexts is a promising area for future research.

## Acknowledgments

This work was supported by Tsinghua University (Department of Computer Science and Technology)-Siemens Ltd., China Joint Research Center for Industrial Intelligence and Internet of Things (JCIIOT).

## References

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. 2025. Opencodeinstruct: A large-scale instruction tuning dataset for code llms. *arXiv preprint arXiv:2504.04030*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, and 1 others. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE.

Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2130–2141.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, and 1 others. 2024. Stepcoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4571–4585.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2025. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, and 1 others. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

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. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and 1 others. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. 2025a. Testgeneval: A real world unit test generation and test completion benchmark. In *The Thirteenth International Conference on Learning Representations*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025b. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, and 1 others. 2024. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, Lester James Validad Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, and 1 others. 2025. Rewardbench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1755–1797.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Chris Yuhao Liu, Liang Zeng, Yuzhen Xiao, Jujie He, Jiacai Liu, Chaojie Wang, Rui Yan, Wei Shen, Fuxiang Zhang, Jiacheng Xu, and 1 others. 2025. Skywork-reward-v2: Scaling preference data curation via human-ai synergy. *arXiv preprint arXiv:2507.01352*.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, QIANG FU, Xiao Han, Yang Wei, and Deheng Ye. 2023a. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. 2025. Dynamic scaling of unit tests for code reward modeling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6917–6935, Vienna, Austria. Association for Computational Linguistics.

Saumya Malik, Valentina Pyatkin, Sander Land, Jacob Morrison, Noah A Smith, Hannaneh Hajishirzi, and Nathan Lambert. 2025. Rewardbench 2: Advancing reward model evaluation. *arXiv preprint arXiv:2506.01937*.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, pages 27730–27744.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741.

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

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, and 1 others. 2025. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and 1 others. 2016. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2023. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *arXiv preprint arXiv:2312.08935*.

Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. 2025. Codecontests+: High-quality test case generation for competitive programming. *arXiv preprint arXiv:2506.05817*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: empowering code generation with oss-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, pages 52632–52657.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. 2025. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 6980–7008, Vienna, Austria. Association for Computational Linguistics.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025a. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025b. ACECODER: Acing coder RL via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12023–12040, Vienna, Austria. Association for Computational Linguistics.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024. Rest-mcts*: Llm self-training via process reward guided tree search. *Advances in Neural Information Processing Systems*, 37:64735–64772.

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2025. CodeDPO: Aligning code models with self generated and verified source code. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15854–15871, Vienna, Austria. Association for Computational Linguistics.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*.

# A Appendix

## A.1 Additional Details

### A.1.1 MCTS

**Selection.** Starting from the root, traverse the tree by choosing the child node $a$ of node $s$ with the

Table 4: Additional hyperparameters for RL. LR stands for learning rate.

| Model | Reward | LR | LR schedule | $\varepsilon_{\text{low}}$ | $\varepsilon_{\text{high}}$ |
|---|---|---|---|---|---|
| Qwen2.5-Coder-1.5B-Instruct | All | 2e-6 | linear | 0.2 | 0.2 |
| Qwen2.5-Coder-3B-Instruct | All | 2e-6 | linear | 0.2 | 0.2 |
| Qwen2.5-Coder-7B-Instruct | RM/Test | 2e-6 | linear | 0.2 | 0.4 |
| Qwen2.5-Coder-7B-Instruct | combine | 2e-6 | linear | 0.2 | 0.28 |
| GLM-4-9B-0414 | All | 1e-6 | cosine | 0.2 | 0.28 |
| Qwen3-4B-Thinking | All | 1e-6 | cosine | 0.2 | 0.28 |

highest Upper Confidence Bound for Trees (UCT) until reaching a node suitable for expansion or a leaf. When traversing to a sufficiently expanded node $s$, the subsequent node for traversal is selected from among its children $a$ by maximizing UCT:

$$UCT(s,a) = Q(a) + \sqrt{\frac{2\ln N(s)}{N(a)}}, \quad (5)$$

where $Q(a)$ denotes the current estimated $q$-value of $a$ and $N(s)$ and $N(a)$ represent the number of visits of $s$ and $a$, respectively.

**Backpropagation.** Following the simulation at leaf $s_T$, a backpropagation step updates all nodes along the path from the root $s_0$ to $s_T$ with the reward $R(s_T)$. For each node $s_t$ along the path, its visit count $N(s_t)$ and $q$-value $Q(s_t)$ are updated:

$$\begin{aligned} N(s_t) &\leftarrow N(s_t) + 1, \\ Q(s_t) &\leftarrow \frac{(N(s_t)-1)\cdot Q(s_t) + R(s_T)}{N(s_T)}. \end{aligned} \quad (6)$$

#### A.1.2 Reward Data Curation

Running MCTS on 12,000 questions produced a total of approximately 238.3 million output tokens. On average, each question requires 19,859 output tokens and 556.8 seconds of wall-time on 2 NVIDIA A100 GPUs. Across the 12,000 questions, the average depth of search trees is 9.28, and the maximum depth reaches 113, demonstrating that the search space for single functions can be diverse. Table 5 provides detailed statistics of the reward datasets. In the pairwise dataset, we collect responses from nodes that are either leaves or have multiple children, so the number of incomplete responses is relatively low.

#### A.1.3 RL Hyperparameters

**Training Batch Size.** For Qwen2.5-Coder-1.5B/3B, we use 4 GPUs, 1 for sampling and 3 for training, and the global batch size is 3 (number of training GPUs) × 8 (batch size per GPU) = 24 questions. For Qwen2.5-Coder-7B, we use 8 GPUs, 1 for sampling and 7 for training, and the global batch size is 7 (number of training GPUs) × 8 (batch size per GPU) = 56 questions. For GLM-4-9B-0414 and Qwen3-4B-Thinking, we use 8 GPUs for colocating training and sampling, and the global batch size is 8 questions.

**Rollout Hyperparameters.** For Qwen2.5 models, 7 responses are sampled per question with a temperature of 1.5. For GLM-4-9B-0414 and Qwen3-4B-Thinking, 8 and 16 responses are sampled per question, respectively, with a temperature of 0.8.

Additional hyperparameters for RL are reported in Table 4.

### A.2 Examples

#### A.2.1 Example of MCTS

Figure 3 shows an example of exploration with MCTS driven by LLM-as-a-Judge.

#### A.2.2 Example of Splitting Code into Snippets

Figure 4 shows an example of inserting print statements into code and splitting code into segments. The code corresponds to a question in Magicoder-OSS-Instruct-75K (Wei et al., 2024).

#### A.2.3 Case Study of Rewards.

We present a case study of rewards assigned by CODERM-NT and prior baseline on responses to a real-world programming contest problem. We consider Codeforces 1058 (Div. 2) Problem C to evaluate the robustness of reward models. For this problem, we construct one correct solution and two incorrect solutions, and then obtain rewards from CODERM-NT and AceCodeRM-7B. Table 6 reports the rewards assigned by CODERM-NT and AceCodeRM-7B to the three candidate solutions. CODERM-NT correctly ranks the correct solution

Table 5: Statistics of the datasets used for training the reward model. #Complete Code indicates the number of responses containing complete code, #Incomplete Code indicates the number of responses containing incomplete code, and #Total indicates the number of samples in the dataset, where a sample represents a question and a response in the scalar dataset and represents a question and a pair of responses in the pairwise dataset.

| Dataset | #Complete Code | #Incomplete Code | #Total |
|---|---|---|---|
| Scalar | 27,207 | 168,893 | 196,098 (responses) |
| Pairwise | 26,740 | 9,176 | 71,205 (response pairs) |

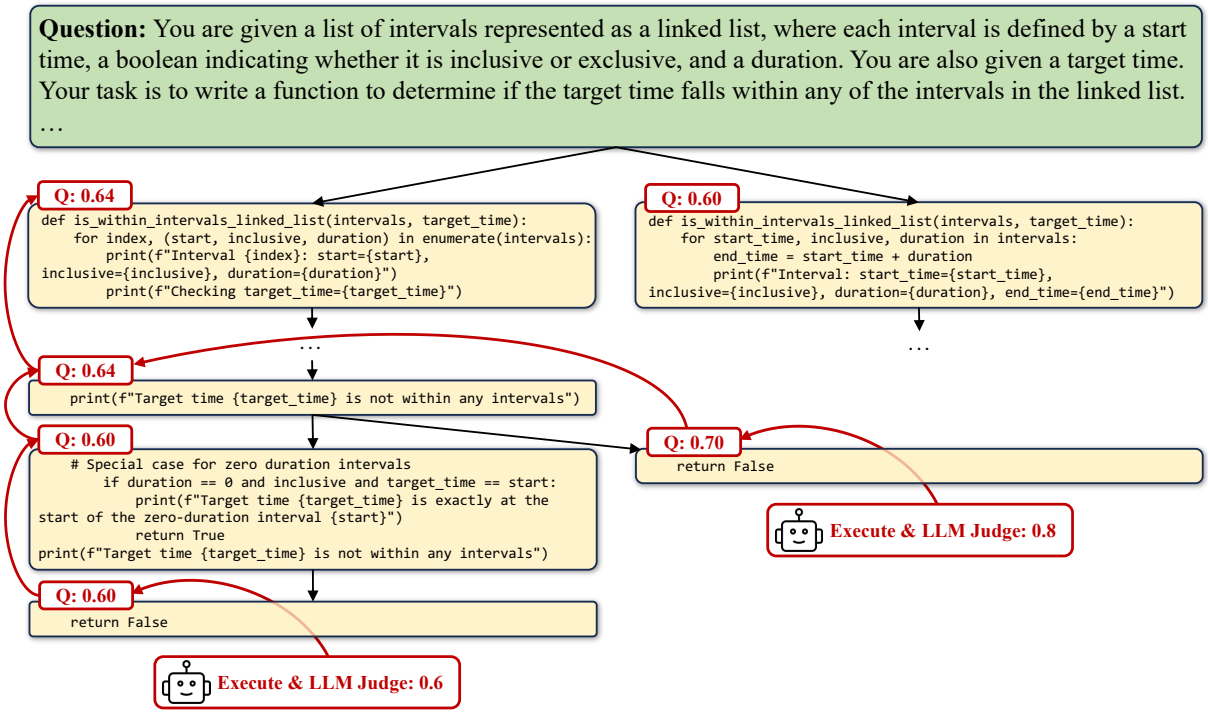


Figure 3: An example of performing MCTS on a coding question.

```
def is_within_intervals_linked_list(intervals, target_time):
    for start, inclusive, duration in intervals:
        print(f"Interval: start={start}, inclusive={inclusive}, duration={duration}")
        print(f"Checking target_time={target_time}")
        if inclusive:
            if start <= target_time <= start + duration:
                print(f"Target time {target_time} is within the interval {start} to {start + duration} (inclusive)")
                return True
            else:
                print(f"Target time {target_time} is not within the interval {start} to {start + duration} (inclusive)")
        else:
            if start < target_time < start + duration:
                print(f"Target time {target_time} is within the interval {start} to {start + duration} (exclusive)")
                return True
            else:
                print(f"Target time {target_time} is not within the interval {start} to {start + duration} (exclusive)")
    print(f"Target time {target_time} is not within any intervals")
    return False
```

: original code
: inserted print statements
: positions for splitting

Figure 4: An example of inserting print statements into code and splitting code into segments. Lines starting with green are the original code, lines starting with pink are the inserted input statements, and the dotted lines are the split locations.

highest and assigns lower rewards to both incorrect solutions, whereas AceCodeRM-7B assigns a higher reward to the first incorrect solution than to the correct solution. This demonstrates that previous reward models can yield misleading rankings, whereas CODERM-NT more faithfully reflects solution quality.

Table 6: Reward comparison between CODERM-NT and AceCodeRM-7B on Codeforces 1058 (Div. 2) Problem C.

| Code | Status | CODERM-NT | AceCodeRM-7B |
|---|---|---|---|
| `def g(n):`<br>`  b = bin(n)[2:].rstrip('0')`<br>`  if b != b[::-1]:`<br>`  return False`<br>`  if len(b) % 2 == 1 and b[len(b) // 2] == '1':`<br>`  return False`<br>`  return True` | Correct | 1.155 | -2.474 |
| `def g(n):`<br>`  b = bin(n)[2:].rstrip('0')`<br>`  if b != b[::-1]:`<br>`  return False`<br>`  if len(b) % 2 == 1 and b[len(b) // 2] == '1':`<br>`  return True`<br>`  return True` | Incorrect | 0.500 | -0.783 |
| `def g(n):`<br>`  b = bin(n)[2:]`<br>`  if b != b[::-1]:`<br>`  return False`<br>`  if len(b) % 2 == 1 and b[len(b) // 2] == '1':`<br>`  return False`<br>`  return True` | Incorrect | 0.935 | -2.645 |

## A.3 Additional Results

### A.3.1 Feasibility of LLM-as-a-Judge

During MCTS, we utilize LLM-as-a-Judge to evaluate expanded code and guide the search process as described in Section 3.1.1. For this mechanism to provide meaningful assessment for tree search, it must reliably discriminate between correct and incorrect code. However, because the source dataset, Magicoder-OSS-Instruct-75K, does not provide verifiable ground-truth tests, the accuracy of LLM-as-a-Judge can not be evaluated directly. Instead, we assess the quality of the judge indirectly by running MCTS on the HumanEval and MBPP benchmarks and comparing the guidance provided by the LLM-as-a-Judge with that derived from the public tests available in these benchmarks. HumanEval includes a substantial number of public tests, whereas MBPP offers only a limited set. The MCTS setup follows that described in Section 4.1.

The evaluation results are presented in Table 7. On HumanEval, where public tests are plentiful, the performance of LLM-as-a-Judge is comparable to that of public tests, while on MBPP, with limited public tests, LLM-as-a-Judge outperforms public tests. These results indicate that LLM-as-a-Judge can provide effective assessment, particularly when unit tests are scarce or lacking.

### A.3.2 Sensitivity of MCTS Rewards

A concern regarding MCTS-based reward assignment is whether it can distinguish critical logic from irrelevant code. We investigate this issue by conducting a controlled experiment to assess the sensitivity of MCTS rewards to irrelevant code. Specifically, we randomly sample 200 questions from the reward model's training datasets and rerun MCTS for each question. For each run, we initialize the search tree with the same initial response from the original data curation, insert an irrelevant code snippet, `assert True == True`, at a random valid location, then run MCTS on the modified tree. The initial snippets in the original and modified trees are identical except for the inserted snippet, yielding 1,390 paired snippets and enabling a direct comparison of rewards assigned to the same code segments before and after modification.

The results are presented in Table 8. The average reward assigned to snippets decreases from 0.6837 to 0.6328 after inserting irrelevant code. To evaluate the statistical significance of this effect, we perform a one-sided paired Student's t-test to test whether the rewards after insertion are lower than those before insertion. The test yields a p-value of $5.617 \times 10^{-39}$, demonstrating a significant decrease. This demonstrates that MCTS-derived rewards are sensitive to irrelevant code, even when such code does not affect program behavior or func-

Table 7: The results of MCTS guided by an LLM judge vs public tests, reporting the pass@1 metric.

| Guidance | HumanEval | HumanEval Plus | MBPP | MBPP Plus |
|---|---|---|---|---|
| LLM-as-a-judge | 90.2 | 86.0 | 89.7 | 73.4 |
| Public Tests | 92.1 | 84.8 | 88.7 | 71.7 |

Table 8: Impact of inserting irrelevant code on MCTS-derived rewards. A one-sided paired Student's t-test evaluates whether rewards significantly decrease when irrelevant code is inserted.

| Original Avg. Reward | Modified Avg. Reward | p-value |
|---|---|---|
| 0.6837 | 0.6328 | $5.617 \times 10^{-39}$ |

Table 9: Reward model accuracy on additional benchmarks (BigCodeBench-Instruct-Hard and LiveCodeBench v5).

| Reward Model | BigCodeBench Instruct-Hard | LiveCode-Bench v5 |
|---|---|---|
| AceCodeRM-7B | 0.556 | **0.650** |
| CODERM-NT | **0.571** | **0.650** |

tional correctness, supporting the feasibility of using rewards derived from MCTS to train reward models for code generation.

#### A.3.3 Additional Benchmark Evaluations for CODERM-NT

In addition to evaluating CODERM-NT on RewardBench as described in Section 4.4, we extend our evaluation to harder coding questions from BigCodeBench-Instruct-Hard and LiveCodeBench-v5. For each benchmark, we sample correct and incorrect solutions from past model outputs and ask the reward models to identify the correct one. We select 63 problems from BigCodeBench-Instruct-Hard and 100 problems from LiveCodeBench-v5 and create paired responses, allowing us to evaluate reward model accuracy in a manner consistent with RewardBench's evaluation pipeline.

Table 9 shows the results of evaluating CODERM-NT and AceCodeRM-7B on these additional benchmarks. CODERM-NT outperforms the baseline AceCodeRM-7B on BigCodeBench-Instruct-Hard, and both models achieve equal accuracy on LiveCodeBench-v5. These results complement the findings from RewardBench, showing that our reward model maintains competitive accuracy on harder coding questions.

#### A.3.4 Results of Combining Both Rewards with Different Weights

All results of combining rewards from CODERM-NT and unit tests are presented in Table 10.

#### A.3.5 Full Results of CODERM-NT Reranking Data

All results of using CODERM-NT to rerank data when using rewards from CODERM-NT, unit tests, and their combination are presented in Table 11.

### A.4 Prompts

#### A.4.1 Prompts for MCTS

**Inserting Print Statements When Generating Code**

```
Here is a partially completed code as response:
{code}
You should write the complete response as
follows:
1. Comprehend the user's requirements carefully
 & to the letter.
2. Describe what you plan to do in the code.
3. Provide the complete Python code to run in a
 single code block, adding necessary libraries
and custom functions before the code.
4. Identify all key intermediate variables in
the code and add print statements for them.
You should start the code with the originally
provided code and preserve everything in it,
even if it contains a syntactical error or a
bug that prevents the code from running.
Return the response only. Do not say anything
else. Act like writing the response for the
first time.
```

**Writing Sample Calls**

```
Write up to five calling examples to execute
the code in the response.
Write the calling examples in the way that they
 will be directly attached to the end of the
code when running.
Cover as many input scenarios as possible,
```

```
including edge cases and complicated inputs.
Do not write invalid inputs that are guaranteed
 not to appear.
Do not predict the expected outputs or return
values.
Return the calling examples only. Do not
include the original code.
```

**Scoring the Response**

```
You are an expert at evaluating the quality of
code.
As an impartial evaluator, please assess the
correctness of a code generation assistant's
response to a user's problem using the 5-point
scoring system described below. The code will
go through debugging with print statements
showing the running process. Please grade the
code based on the satisfaction of each
criterion:
1: It means the code is relevant and provides
some information related to the problem, even
if it is incomplete, can not be run, or
contains irrelevant content.
2: It means the code can sometimes run and
produce outputs, but contains bugs or logical
errors so that it does not make the correct
outputs to the problem.
3: It means the code can solve the simple cases
 of the problem, but failes in difficult,
complicated, or edge cases, misses necessary
error handling, or contains any security
vulnerabilities.
4: It means the code solves the problem
directly and comprehensively, having correct
logic and covering every possible input case.
Cases guaranteed not to appear do not need to
be covered.
5: It means the code is optimized for time and
space complexity, reflects the best practices
specific to the algorithm, language, or
framework used, and perfectly meets the user's
specific requirements.

#Problem Begins#
{problem}
#Problem Ends#

The response contains the following code:
{code}

By executing the code, you get the following
output:
#Output Begins#
{exec_result}
#Output Ends#

After examining the user's instruction and the
response:
1. Provide an analysis of the response,
carefully monitoring the execution of the code
for any errors or exceptions that may arise and
 paying close attention to the output produced
by the execution. Use the format: "Analysis: <
analysis>"
2. Justify your score by checking whether the
code meets each criterion using the format: "
Justify: <check criteria>".
3. Conclude with the score using the format: "
Score: <total points>"
```

### A.4.2 Prompt Template for RL

Some coding questions in the training data require generating code containing specified function names that are not given directly in the questions, instead only appearing in the unit tests. For these questions, we use the following template to guide the model to generate code with the correct function names:

```
Can you solve the following problem with Python
 code? The function in this code should be
named `{test_entry_point}`. Write the function
only, do not include any test code.
{question}
```

Table 10: Full results of combining the rewards from CODERM-NT and unit tests. The "Reward" column indicates the ratio between $\mathcal{R}_\phi(\mathbf{x}, \mathbf{y})$ and $\mathcal{R}_{\text{test}}(\mathbf{x}, \mathbf{y})$ when calculating $\mathcal{R}_{\text{combined}}$.

| Data | Reward | HumanEval | | MBPP | | LCB-v5 | BCB-I Hard | Avg. |
|---|---|---|---|---|---|---|---|---|
| | | Base | Plus | Base | Plus | | | |
| Qwen2.5-Coder-1.5B-Instruct | | 70.1 | 66.5 | 70.2 | 60.3 | 5.8 | 5.4 | 46.4 |
| | 1 RM:2 test | 75.0 | 68.9 | 72.5 | 61.6 | 6.1 | 6.8 | 48.5 |
| | 1 RM:6 test | 76.2 | 71.3 | 73.3 | 61.6 | 4.9 | 5.4 | **48.8** |
| Ranked | 1 RM:1 test | 74.4 | 68.9 | 73.8 | 63.2 | 6.7 | 5.4 | 48.7 |
| | 1 RM:2 test | 76.8 | 71.3 | 72.0 | 60.8 | 5.0 | 6.8 | **48.8** |
| | 1 RM:6 test | 75.0 | 68.9 | 69.8 | 59.8 | 5.1 | 7.4 | 47.7 |
| Qwen2.5-Coder-3B-Instruct | | 84.8 | 80.5 | 74.6 | 63.5 | 12.8 | 12.8 | 54.8 |
| | 1 RM:2 test | 85.4 | 79.9 | 75.4 | 64.8 | 12.8 | 15.5 | 55.6 |
| | 1 RM:6 test | 87.2 | 82.9 | 76.7 | 66.1 | 13.5 | 14.2 | **56.8** |
| Ranked | 1 RM:1 test | 86.6 | 81.1 | 75.9 | 65.3 | 12.7 | 15.5 | 56.2 |
| | 1 RM:2 test | 85.4 | 82.3 | 79.1 | 68.0 | 13.4 | 16.2 | **57.4** |
| | 1 RM:6 test | 84.1 | 80.5 | 77.0 | 65.9 | 13.6 | 14.2 | 55.9 |
| Qwen2.5-Coder-7B-Instruct | | 88.4 | 84.8 | 85.4 | 73.0 | 17.8 | 19.6 | 61.5 |
| | 1 RM:1 test | 89.6 | 85.4 | 86.0 | 73.3 | 17.3 | 20.3 | 62.0 |
| | 1 RM:2 test | 86.6 | 82.9 | 87.0 | 74.9 | 15.7 | 23.0 | 61.7 |
| | 1 RM:6 test | 89.6 | 85.4 | 86.0 | 73.5 | 17.1 | 21.6 | **62.2** |
| Ranked | 1 RM:1 test | 90.2 | 87.8 | 86.0 | 73.3 | 18.0 | 18.9 | **62.4** |
| | 1 RM:2 test | 89.6 | 87.2 | 84.9 | 72.8 | 15.8 | 20.3 | 61.8 |
| | 1 RM:6 test | 89.0 | 86.6 | 85.4 | 73.5 | 17.8 | 18.2 | 61.8 |
| GLM-4-9B-0414 | | 82.9 | 79.3 | 79.9 | 67.2 | 14.9 | 15.5 | 56.6 |
| | 1 RM:2 test | 83.5 | 79.9 | 82.5 | 68.5 | 16.1 | 16.2 | **57.8** |
| | 1 RM:6 test | 84.1 | 79.9 | 79.9 | 67.5 | 16.1 | 18.9 | 57.7 |
| Ranked | 1 RM:2 test | 86.6 | 81.7 | 80.2 | 68.5 | 15.5 | 18.2 | 58.5 |
| | 1 RM:6 test | 87.2 | 81.1 | 80.7 | 68.5 | 16.9 | 19.6 | **59.0** |
| Qwen3-4B-Thinking | | 97.0 | 90.9 | 91.0 | 76.5 | 45.5 | 15.5 | 69.4 |
| | 1 RM:2 test | 95.7 | 93.3 | 90.7 | 76.5 | 47.9 | 21.6 | 71.0 |
| Ranked | 1 RM:2 test | 97.0 | 93.3 | 91.3 | 77.5 | 50.3 | 20.9 | **71.7** |

Table 11: Full results of **CODERM-NT used as Ranker**.

| Data | Reward | | HumanEval | | MBPP | | LCB-v5 | BCB-I-Hard | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| | RM | Unit Test | Base | Plus | Base | Plus | | | |
| Qwen2.5-Coder-1.5B-Instruct | | | 70.1 | 66.5 | 70.2 | 60.3 | 5.8 | 5.4 | 46.4 |
| KodCode 9k | ✗ | ✓ | 73.8 | 68.9 | 71.7 | 61.4 | 4.2 | **6.8** | 47.8 |
| | Ours | ✗ | 74.4 | 69.5 | **73.5** | **63.0** | **7.8** | 6.1 | **49.1** |
| | Ours | ✓ | **76.8** | **71.3** | 72.0 | 60.8 | 5.0 | **6.8** | 48.8 |
| Qwen2.5-Coder-3B-Instruct | | | 84.8 | 80.5 | 74.6 | 63.5 | 12.8 | 12.8 | 54.8 |
| KodCode 9k | ✗ | ✓ | 84.8 | 80.5 | 74.6 | 63.8 | 12.8 | 14.9 | 55.2 |
| | Ours | ✗ | **87.2** | **83.5** | 77.0 | 66.4 | **14.5** | 12.8 | 56.9 |
| | Ours | ✓ | 85.4 | 82.3 | **79.1** | **68.0** | 13.4 | **16.2** | **57.4** |
| Qwen2.5-Coder-7B-Instruct | | | 88.4 | 84.8 | 85.4 | 73.0 | 17.8 | 19.6 | 61.5 |
| KodCode 9k | ✗ | ✓ | 89.6 | 86.6 | 85.7 | **73.5** | 17.5 | **20.3** | 62.2 |
| | Ours | ✗ | **90.9** | 87.2 | 85.7 | 73.3 | 17.0 | 18.9 | 62.2 |
| | Ours | ✓ | 90.2 | **87.8** | **86.0** | 73.3 | **18.0** | 18.9 | **62.4** |
| GLM-4-9B-0414 | | | 82.9 | 79.3 | 79.9 | 67.2 | 14.9 | 15.5 | 56.6 |
| KodCode 9k | ✗ | ✓ | 85.4 | 79.9 | 80.7 | 67.7 | 14.8 | 16.9 | 57.6 |
| | Ours | ✗ | **87.2** | **82.3** | **81.2** | **68.8** | 16.1 | 16.9 | 58.8 |
| | Ours | ✓ | **87.2** | 81.1 | 80.7 | 68.5 | **16.9** | **19.6** | **59.0** |
| Qwen3-4B-Thinking | | | **97.0** | 90.9 | 91.0 | 76.5 | 45.5 | 15.5 | 69.4 |
| OCI 5k | ✗ | ✓ | 95.1 | 92.1 | **91.5** | 76.5 | **51.5** | 21.0 | 71.3 |
| | Ours | ✗ | 95.7 | 92.7 | **91.5** | 76.7 | 47.9 | **21.6** | 71.0 |
| | Ours | ✓ | **97.0** | **93.3** | 91.3 | **77.5** | 50.3 | 20.9 | **71.7** |