Dynamic Scaling of Unit Tests for Code Reward Modeling

Zeyao Ma[1,3*†], **Xiaokang Zhang**[1,3*], **Jing Zhang**[1,3‡], **Jifan Yu**[2], **Sijia Luo**[1], **Jie Tang**[2]
[1]School of Information, Renmin University of China, [2]Tsinghua University,
[3]Key Laboratory of Data Engineering and Knowledge Engineering, Beijing, China
zeyaoma@gmail.com, {zhang2718, zhang-jing}@ruc.edu.cn
https://code-reward-model.github.io

Abstract

Current large language models (LLMs) often struggle to produce accurate solutions on the first attempt for code generation. Prior research tackles this challenge by generating multiple candidate solutions and validating them with LLM-generated unit tests. The execution results of unit tests serve as reward signals to identify correct solutions. As LLMs always confidently make mistakes, these unit tests are not reliable, thereby diminishing the quality of reward signals. Motivated by the observation that scaling the number of solutions improves LLM performance, we explore the impact of scaling unit tests to enhance reward signal quality. Our pioneer experiment reveals a positive correlation between the number of unit tests and reward signal quality, with greater benefits observed in more challenging problems. Based on these insights, we propose CodeRM-8B, a lightweight yet effective unit test generator that enables efficient and high-quality unit test scaling. Additionally, we implement a dynamic scaling mechanism that adapts the number of unit tests based on problem difficulty, further improving efficiency. Experimental results show that our approach significantly improves performance across various models on three benchmarks (e.g., with gains of 18.43% for Llama3-8B and 3.42% for GPT-4o-mini on HumanEval Plus).

1 Introduction

Code generation aims to automatically produce code solutions that satisfy programming requirements specified in natural language (Svyatkovskiy et al., 2020). Recent advancements in large language models (LLMs) have shown significant progress in this domain (Touvron et al., 2023; Achiam et al., 2023). However, generating correct code on the first attempt remains challenging due to the inherent complexity of reasoning required (Li et al., 2022; Huang et al., 2024b). Beyond developing more powerful LLMs, some research leverages additional test-time computation to generate more code solutions via repeated sampling, while a verifier or reward model reranks and identifies the optimal solution (Inala et al., 2022; Chen et al., 2023). Although repeated sampling enables LLMs to produce correct solutions (Chen et al., 2021; Brown et al., 2024), identifying the correct solution from the multitude of candidates poses a significant challenge (Gao et al., 2023).

Unit tests (i.e., pairs of input and expected output) are frequently used as verifers to identify correct code solutions (Shi et al., 2022; Chen et al., 2023). Specifically, LLMs generate unit tests based on the given instructions and candidate code solutions, which are then executed by a compiler or interpreter. The execution results serve as reward signals to identify correct solutions. However, as LLMs often confidently make mistakes (Huang et al., 2024c), the reliability of these unit tests is not guaranteed, thereby diminishing the quality of the reward signal. Inspired by the performance gains from scaling test-time computation to generate more responses (Snell et al., 2024), we ask: *Can generating more unit tests improve the quality of the reward signal for code solutions?*

To address this question, we conduct a pioneer experiment to investigate the correlation between the number of unit tests and the quality of the code reward signal across different LLMs, code solution quantities, and unit test scales. Our findings reveal:

- Scaling the number of unit tests consistently improves the accuracy of identifying correct solutions across different model sizes and code solution quantities.
- The benefits of scaling unit tests depend on problem difficulty, with more computational

*Equal Contributions.
†Work was done when interned at Zhipu AI.
‡Corresponding Author.

resources yielding greater improvements for challenging problems.

Building on these observations, we develop **CodeRM-8B**, a small yet powerful unit test generator designed to facilitate efficient and high-quality unit test scaling. To support model training, we introduce an automatic data synthetic pipeline that produces high-quality unit tests from existing code instruction-tuning datasets. Leveraging this synthesized data, we perform supervised fine-tuning (SFT) on Llama3.1-8B, resulting in a high-quality unit test generation model. Furthermore, since the benefits of scaling unit tests vary with problem difficulty, we follow Damani et al. (2024) to implement the dynamic unit test scaling on different problems. Specifically, we train a lightweight problem difficulty classifier using the language model probing method (Alain and Bengio, 2017; Kadavath et al., 2022), which extracts implicit information from the LLM's intermediate representation and outputs a scalar problem difficulty. Based on this classifier, we dynamically allocate computation budgets across problems of varying difficulties using a greedy algorithm (Edmonds, 1971).

We conduct extensive experiments to evaluate the effectiveness of CodeRM-8B on three widely used benchmarks and four LLMs with varying parameter scales for solution generation. The results demonstrate that scaling unit tests with CodeRM-8B significantly improves the performance of smaller models (e.g., a performance gain of 18.43% on HumanEval Plus for Llama3-8B). Moreover, CodeRM-8B enhances the performance of significantly larger models or even proprietary models (e.g., a 4.95% gain for Llama3-70B and 3.42% for GPT-4o-mini on HumanEval Plus). We also evaluate the performance of dynamic unit test scaling on two benchmarks. By leveraging a trained problem difficulty classifier and dynamically allocating computation budgets, this approach brings additional performance improvements at a fixed computational cost (e.g., up to approximately 0.5% performance gain on MBPP Plus).

The main contributions of this paper are as follows: 1) A pioneer experiment revealing a positive correlation between the number of unit tests and the quality of the code reward signal, with greater benefits for more challenging problems as unit test scales. 2) A small yet powerful model enabling efficient and high-quality unit test scaling. 3) An implementation of dynamic unit test scaling over problems in different difficulties. 4) Experimental results validating the effectiveness of CodeRM-8B and the implementation of dynamic scaling.

2 Pioneer Experiment

This section explores the correlation between the quantity of LLM-generated unit tests and the quality of the code reward signal. We first present the methodology, including a unit test-based majority voting framework and the setup of the pioneer experiment. Subsequently, we analyze the observations derived from the pioneer results.

2.1 Methodology

The unit test-based majority voting framework follows a standard best-of-N strategy (Cobbe et al., 2021; Lightman et al., 2024). Given a programming question Q, the policy model generates N candidate code solutions:

$$\{C_1, C_2, \ldots, C_N\},$$

where C_i represents the i-th candidate solution. Based on the programming question and the candidate code solution, an auxiliary LLM generates M unit tests:

$$\{T_1, T_2, \ldots, T_M\}$$

Each unit test T_j consists of a set of test cases:

$$T_j = \{(x_{j,1}, y_{j,1}), (x_{j,2}, y_{j,2}), \ldots, (x_{j,K_j}, y_{j,K_j})\},$$

where $x_{j,k}$ is the input for the k-th test case in T_j, and $y_{j,k}$ is the corresponding expected output. K_j represents the number of test cases in T_j. Each candidate solution C_i is executed on the unit tests $\{T_1, T_2, \ldots, T_M\}$. For a given unit test T_j, the execution of C_i produces a binary result:

$$r_{i,j} = \begin{cases} 1, & \text{if } C_i \text{ passes all test cases in } T_j; \\ 0, & \text{otherwise.} \end{cases}$$

The binary results for all unit tests form a reward signal for each candidate solution:

$$\mathbf{R}_i = \{r_{i,1}, r_{i,2}, \ldots, r_{i,M}\}.$$

Finally, we select the optimal candidate solution C_{opt} based on majority voting (Wang et al., 2023). This voting process determines C_{opt} as the solution that passes the maximum number of unit tests:

$$C_{\text{opt}} = \underset{C_i}{\text{argmax}} \sum_{j=1}^{M} r_{i,j}. \quad (1)$$

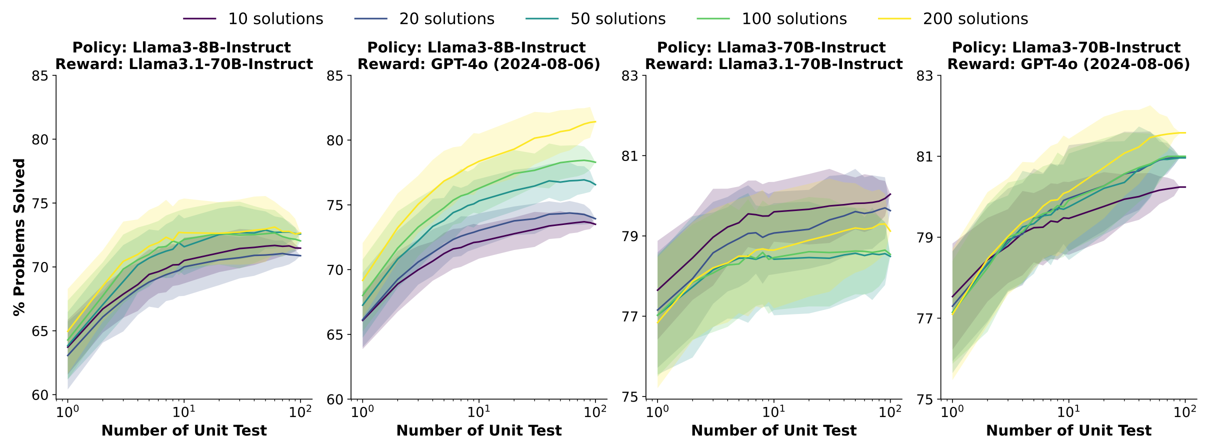

Figure 1: Scaling the quantities of unit tests for majority voting leads to improvements in performance across different policy models and reward models. Policy refers to the model that produces code solutions, while reward denotes the model that generates unit tests.

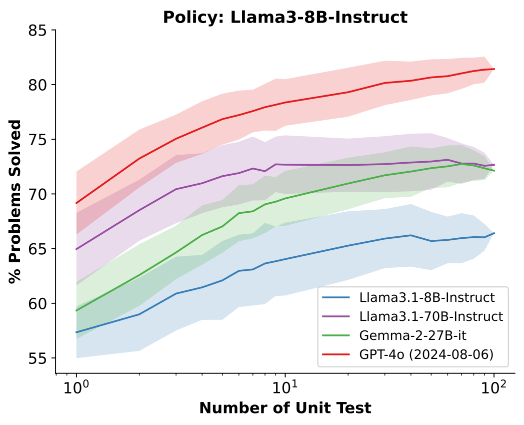

Figure 2: The correlation between the quantities of unit tests and the performance on different unit test generators (reward model) with 200 candidate code solutions.

We utilize HumanEval Plus (Liu et al., 2023), a widely adopted dataset comprising handwritten Python programming questions with comprehensive unit tests. For each question, an LLM (policy model) generates $N = 200$ code solutions, while another LLM (reward model) produces $M = 100$ unit tests for supervision. The optimal solution is selected using Equation (1) and validated against the ground truth unit tests in the dataset. To compute the results for n solutions ($0 \leq n \leq N$) and m unit tests ($0 \leq m \leq M$), we employ the bootstrap resampling method (Efron, 1979), generating 100 bootstrap samples to compute mean values and confidence intervals, as shown in Figure 1 and 2.

2.2 Observations

Scaling the number of unit tests consistently improves the quality of the reward signal. Figure 1 demonstrates that increasing the number of unit tests consistently improves best-of-N performance across different quantities of code solutions, policy models, and reward models. As the number of code solutions increases, performance typically improves with more samples, since a larger sample size enhances the likelihood of generating accurate responses (Brown et al., 2024). However, in the third sub-figure of Figure 1, performance decreases with additional test-time computation. This aligns with observations by Cobbe et al. (2021), where excessive test-time computation can generate adversarial solutions that mislead the verifier. For different policy models, scaling unit tests yields more pronounced performance improvements for weaker models compared to stronger ones. For instance, performance gains of 11% and 5% are observed for Llama3-8B and Llama3-70B, respectively, when employing GPT-4o as the reward model.

Figure 2 compares the reward signals produced by different reward models. Generally, models with more parameters achieve better best-of-N performance. Notably, while Gemma-2-27B-it performs significantly worse than Llama3.1-70B with a single unit test, it achieves comparable performance when scaled to 100 unit tests per question. This may be attributed to smaller models generating responses with greater coverage and diversity, as discussed by Bansal et al. (2024).

Scaling unit tests is more effective for harder problems. We evaluate the effectiveness of scaling unit tests across problems of different difficulty levels. Specifically, we first eliminate problems without a single correct solution. The remaining

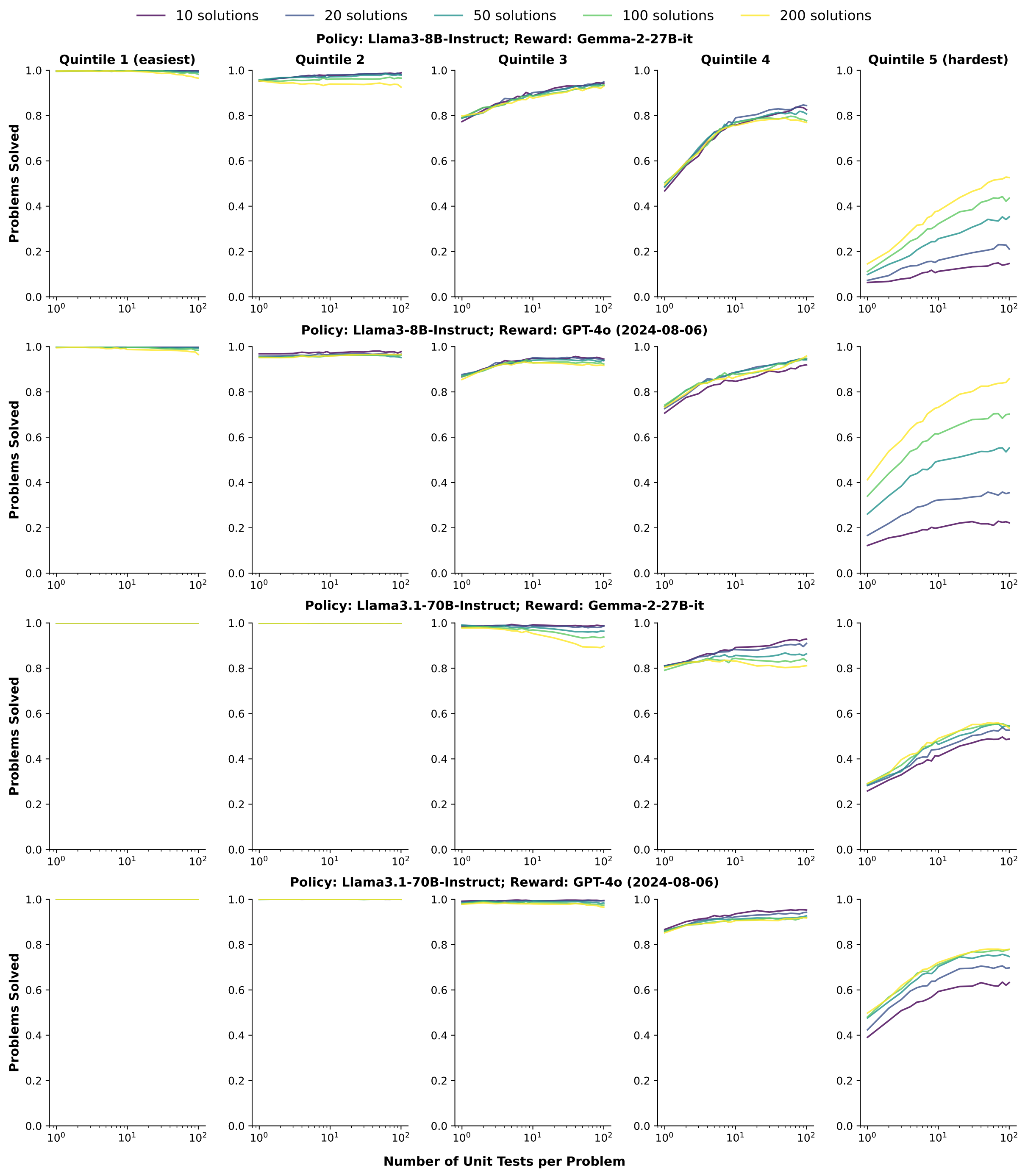

Figure 3: The improvements of best-of-N performance on problems of different difficulties. Quintile 1 (easiest) has the highest pass rate, while Quintile 2 (hardest) has the lowest pass rate. Scaling the quantity of unit tests significantly improves the accuracy on more complex problems.

problems are divided into five equal parts based on the actual pass rate obtained via repeated sampling. Figure 3 presents the results using Llama3-8B and Llama3.1-70B as the policy models, with Gemma-2-27B and GPT-4o as reward models. The results demonstrate that the benefits of scaling unit tests are highly dependent on problem complexity. For more challenging problems, increasing computational resources yields greater performance enhancement. This highlights the potential of dynamically scaling of unit tests based on problem difficulty, representing a viable approach for resource conservation within a fixed computational budget. More fine-grained results are provided in Appendix B.

3 Towards Efficient and High-Quality Unit Test Scaling

In light of the observations in Section 2, we propose CodeRM-8B, a small yet effective unit test generator designed to enable efficient and high-quality unit test scaling. To this end, we introduce a synthetic data pipeline for generating high-quality unit tests and model training. Additionally, as shown in Section 2, scaling unit tests proves to be more effective for harder problems. To further improve efficiency, we implement a dynamic scaling strategy that adapts to problems of varying difficulty. Specifically, following Damani et al. (2024), we train a problem difficulty classifier and employ a greedy algorithm to allocate computational resources dynamically, prioritizing harder problems.

3.1 Unit Test Generator

To train an effective unit test generator, we first construct high-quality training data through a data synthetic pipeline that includes dataset preprocessing and unit test generation, as illustrated in the first two sections in Figure 4. We then apply supervised fine-tuning (SFT) to optimize the generator.

Dataset Preprocessing. We utilize existing high-quality code instruction-tuning datasets as the foundation of our pipeline, including CodeFeedback-Filtered-Instruction[1] (Zheng et al., 2024) and the training set of TACO[2] (Li et al., 2023). CodeFeedback-Filtered-Instruction is a carefully curated dataset of code instruction queries, derived from four prominent open-source code instruction tuning datasets (e.g., Magicoder-OSS-Instruct). TACO focuses on algorithmic code generation, featuring programming challenges from platforms such as LeetCode and Codeforces. To prepare these datasets for unit test generation, we first apply a principle-driven approach to filter out unsuitable questions for unit testing with Llama3.1-70B (e.g., tasks involving randomness), as presented in Appendix F. Subsequently, we restructure the original code solutions into functional formats to facilitate the following unit test generation.

Unit Test Generation. Based on the queries and code solutions in the dataset, we employ Llama3.1-70B to generate diverse unit tests via repeated sampling. Each generated unit test is then executed on the ground truth code solution provided in the dataset. The execution results serve as true/false labels for the unit tests, with correct unit tests enabling the code solution to successfully pass the test. We observe that generating correct unit tests for difficult code instructions requires significant computational resources, often yielding only a small number of valid tests. To address this, we utilize execution feedback from the Python interpreter to repair incorrect unit tests, rather than relying solely on repeated sampling. By leveraging this feedback, Llama3.1-70B can efficiently repair tests, significantly improving the collection process for challenging problems. To further identify high-quality unit tests, we employ a *quality control* process. We adhere to the principle that a high-quality unit test should allow the correct solution to pass while rejecting as many incorrect solutions as possible. To achieve this, we generate incorrect solutions using a less capable model and filter out false positive unit tests (i.e., tests that fail to reject incorrect solutions).

Model Training We utilize supervised fine-tuning (SFT) to train CodeRM-8B based on Llama3.1-8B. For the construction of training data, we use the problem and code solution as the instruction and the high-quality unit test as the answer. An example of the instruction-answer pair is presented in Figure 9.

3.2 Dynamic Unit Test Scaling

Section 2 presents that scaling unit tests is more effective for harder problems. To further enhance the efficiency of unit test scaling, we implement a dynamic scaling strategy that adapts to problems

[1] https://huggingface.co/datasets/m-a-p/CodeFeedback-Filtered-Instruction

[2] https://huggingface.co/datasets/BAAI/TACO

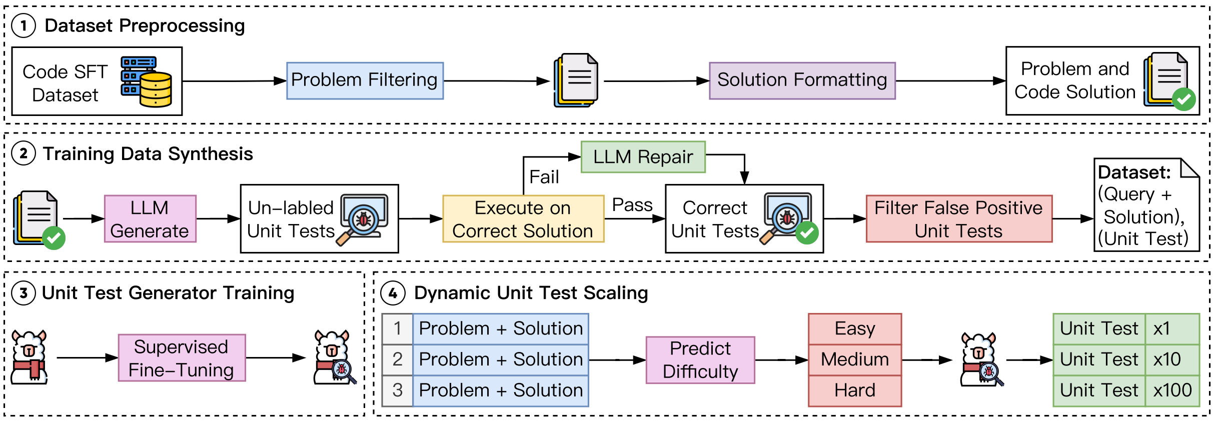

Figure 4: Overview for efficient and high-quality unit test scaling. First, we train a lightweight unit test generator based on high-quality synthetic data. Subsequently, we employ dynamic unit test scaling to further improve efficiency.

of different difficulty, which is shown in the fourth section of Figure 4. Specifically, we follow Damani et al. (2024) to first train a problem difficulty classifier. Subsequently, we use a greedy algorithm to allocate more computation resources on more challenging questions.

Problem Difficulty Estimation. To train the problem difficulty classifier, we first gather data using the preprocessed datasets from the synthetic data pipeline. The policy model generates multiple solutions through repeated sampling, which are then evaluated using previously collected correct unit tests as verifiers. The problem's difficulty is estimated by calculating the mean success probability (i.e., pass rate) of these solutions.

Model Training. We leverage the language model probing method (Alain and Bengio, 2017; Gurnee and Tegmark, 2024) to train a lightweight problem difficulty classifier. The probing method extracts implicit information from the intermediate representation, which is mapped into discrete classes by training a simple classifier (Zhang et al., 2024; Damani et al., 2024). Our problem difficulty classifier is a two-layer feedforward network with a scalar output to indicate the problem difficulties. This classifier generates minimal overhead since its inputs are hidden states already calculated during the decoding process. To train this classifier, we minimize the following cross-entropy loss:

$$\sum_{x_i,\lambda_i}\left[\lambda_i \log\left(\hat{\lambda}(x_i;\theta)\right)+(1-\lambda_i)\log\left(1-\hat{\lambda}(x_i;\theta)\right)\right]$$

where x_i is the i-th query, θ is the parameters of the classifier, λ_i is the actual pass rate, and $\hat{\lambda}(x_i;\theta)$ is the predicted pass rate (i.e., problem difficulty).

Dynamic Compution Allocation. We allocate more computation budgets on more challenging problems, following the method used by Damani et al. (2024). For a question x with pass rate λ, the reward for allocating b computation budget to this question is given by:

$$q(x,b) = 1-(1-\lambda)^b, \tag{2}$$

which represents the probability of getting at least one correct answer in b attempts. Since for any question x, the reward function $q(x,b)$ is monotonically increasing with respect to b, we adopt a greedy algorithm to allocate computational resources across different questions.

4 Experiments

4.1 Experimental Setup

Datasets and Metrics. We conduct extensive experiments on three widely used benchmarks, including HumanEval Plus (Liu et al., 2023), MBPP Plus (Liu et al., 2023), and LiveCodeBench (Jain et al., 2024). HumanEval Plus and MBPP Plus are derived from HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) with additional test cases. LiveCodeBench is a contamination-free dataset that continuously collects new problems over time. We select the queries from 2024-01 to 2024-09 to avoid the contamination problem with our training data. We focus on the queries with solutions in functional format, ultimately yielding 168 queries. The evaluation metric is Pass@k (Chen et al., 2021), with k set to 1 in our experiments.

Method	Policy Model			
	Llama3-8B	Llama3-70B	GPT-3.5	GPT-4o-m
HumanEval Plus				
Vanilla	53.58	73.74	67.83	82.96
Grading RM	62.20 $_{+8.62}$	75.00 $_{+1.26}$	70.12 $_{+2.29}$	83.50 $_{+0.54}$
MBR-Exec	60.30 $_{+6.72}$	75.80 $_{+2.06}$	70.60 $_{+2.77}$	85.20 $_{+2.24}$
CodeT	65.30 $_{+11.72}$	76.20 $_{+2.46}$	73.89 $_{+6.06}$	85.30 $_{+2.34}$
MPSC	59.72 $_{+6.14}$	75.51 $_{+1.77}$	72.76 $_{+4.93}$	84.82 $_{+1.86}$
Llama3.1-70B	**72.04** $_{+18.46}$	<u>78.54</u> $_{+4.80}$	**79.76** $_{+11.93}$	<u>85.45</u> $_{+2.49}$
CodeRM-8B	<u>72.01</u> $_{+18.43}$	**78.69** $_{+4.95}$	<u>78.01</u> $_{+10.18}$	**86.38** $_{+3.42}$
MBPP Plus				
Vanilla	49.20	69.33	70.53	71.59
Grading RM	48.40 $_{-0.80}$	70.60 $_{+1.27}$	66.67 $_{-3.86}$	69.00 $_{-2.59}$
MBR-Exec	50.00 $_{+0.80}$	69.80 $_{+0.47}$	70.53 $_{+0.00}$	72.30 $_{+0.71}$
CodeT	59.20 $_{+10.00}$	69.90 $_{+0.57}$	69.92 $_{-0.61}$	73.40 $_{+1.81}$
MPSC	53.32 $_{+4.12}$	70.91 $_{+1.58}$	71.59 $_{+1.06}$	73.20 $_{+1.61}$
Llama3.1-70B	<u>65.26</u> $_{+16.06}$	<u>71.85</u> $_{+2.52}$	<u>75.72</u> $_{+5.19}$	<u>74.96</u> $_{+3.37}$
CodeRM-8B	**66.71** $_{+17.51}$	**72.44** $_{+3.11}$	**75.96** $_{+5.43}$	**75.20** $_{+3.61}$
LiveCodeBench				
Vanilla	11.98	25.30	20.55	34.83
Grading RM	13.10 $_{+1.12}$	26.19 $_{+0.89}$	20.83 $_{+0.28}$	36.31 $_{+1.48}$
MBR-Exec	12.04 $_{+0.06}$	25.37 $_{+0.07}$	20.52 $_{-0.03}$	34.83 $_{+0.00}$
CodeT	12.61 $_{+0.63}$	25.89 $_{+0.59}$	20.58 $_{+0.03}$	35.13 $_{+0.30}$
MPSC	11.98 $_{+0.00}$	25.30 $_{+0.00}$	20.55 $_{+0.00}$	34.83 $_{+0.00}$
Llama3.1-70B	<u>13.28</u> $_{+1.30}$	**28.46** $_{+3.16}$	**22.80** $_{+2.25}$	<u>38.60</u> $_{+3.77}$
CodeRM-8B	**15.21** $_{+3.23}$	<u>27.73</u> $_{+2.43}$	<u>21.76</u> $_{+1.21}$	**39.20** $_{+4.37}$

Table 1: The main result of our approach and other baselines over three code generation benchmarks. GPT-4o-m stands for GPT-4o-mini. The improvements are calculated between methods and vanilla. The top two performances for each dataset and policy model are marked in **bold** and <u>underlined</u>.

Baselines and Implementations. We compare several baselines on four policy models, including Llama3-8B (Dubey et al., 2024), Llama3-70B (Dubey et al., 2024), GPT-3.5-turbo[3], and GPT-4o-mini (Achiam et al., 2023). The baselines include the vanilla method that randomly selects a solution, the grading reward model, MBR-Exec (Shi et al., 2022), CodeT (Chen et al., 2023), and MPSC (Huang et al., 2024a). For the grading reward model, we employ ArmoRM-Llama3-8B-v0.1 (Wang et al., 2024) and generate a scalar score for each candidate solution. For other baselines, we utilize Llama3.1-70B to generate test cases, using the same computation budget to conduct fair comparisons. For our method, we employ the majority voting framework in Section 2.1 and utilize Llama3.1-70B and CodeRM-8B as unit test generator. Appendix C presents the detailed experimental setting and the implementation of all baselines.

[3] `https://chat.openai.com/`

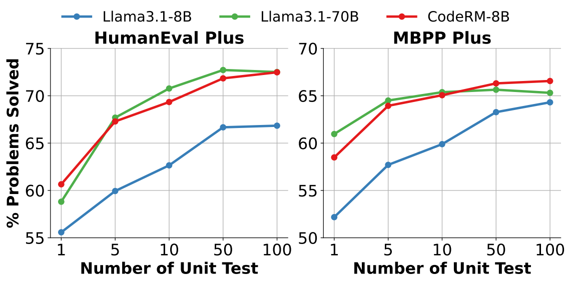

Figure 5: The performance of three different unit test generators (reward model) on different quantities of unit tests, while employing Llama3-8B as the policy model.

4.2 Main Results

Effectiveness of CodeRM-8B. Table 1 summarizes the experimental results on three code generation benchmarks. Under our unit test-based majority voting framework, both Llama3.1-70B and CodeRM-8B significantly improve the performance of four policy models across all benchmarks. Notably, the best-of-N performance of CodeRM-8B is on par with Llama3.1-70B, a model with nearly 8x more parameters. Across all three benchmarks, CodeRM-8B demonstrates consistent and substantial enhancements to the performance of policy models at varying scales and types. For example, on HumanEval Plus, CodeRM-8B achieves a substantial performance improvement of 18.43% for the smaller model Llama3-8B, while also enhancing the performance of larger models and proprietary models, such as Llama3-70B and GPT-4o-mini, by 4.95% and 3.42%, respectively. Figure 5 visualizes the performance of various unit test generators as the number of unit tests scales. The results highlight that CodeRM-8B achieves performance comparable to Llama3.1-70B while substantially surpassing Llama3.1-8B, emphasizing its effectiveness and computational efficiency.

Performance of Dynamic Scaling. Figure 6 presents the performance of our dynamic unit test scaling implementation, which leverages Equation (2) to allocate computation budgets based on predicted pass rates. We compare it against two baselines: 1) *dynamic allocation with gold pass rates*, which uses actual pass rates instead of predictions, and 2) *equal allocation*, which does not apply dynamic scaling. Results show that our method improves performance under fixed computational budgets, with greater gains on MBPP Plus. This

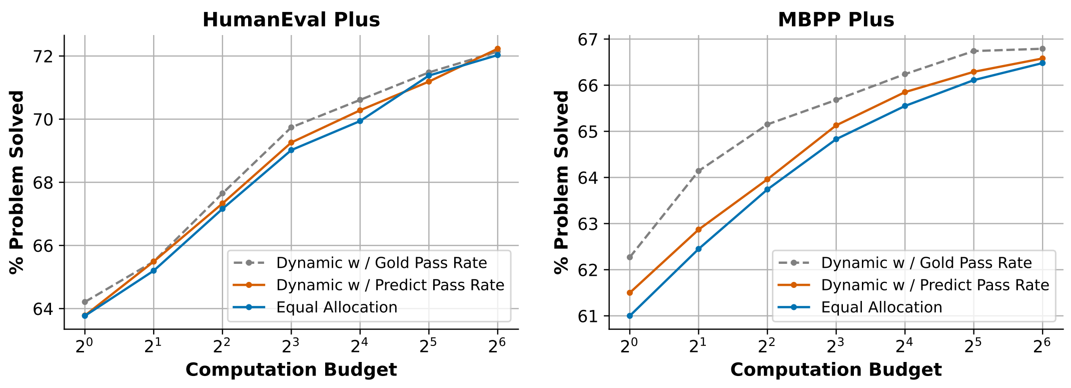

Figure 6: Best-of-N performance comparison under unit test scaling with three computation budget allocation strategies: dynamic allocation with gold pass rate, dynamic allocation with predicted pass rate, and equal allocation.

Model	Acc (↑)	F1 (↑)	FAR (↓)	FRR (↓)
Quality of Individual Unit Tests				
Llama3.1-8B	60.02	44.97	13.66	46.13
Llama3.1-70B	**73.65**	**70.15**	**11.10**	**34.51**
CodeRM-8B (Ours)	69.64	63.63	11.17	38.55
Quality of Multiple Unit Tests				
Llama3.1-8B	74.21	74.35	20.44	30.55
Llama3.1-70B	78.30	78.76	17.19	25.97
CodeRM-8B (Ours)	**80.46**	**81.27**	**16.48**	**22.71**

Table 2: The quality of individual unit tests and the combination of multiple unit tests on HumanEval Plus, utilizing Llama3.1-8B as the policy model. The top two performances are highlighted using **bold** and underlining.

Method	HumanEval+	MBPP+
zero-shot	66.67	63.27
training wo / quality control	69.71 $_{+3.04}$	64.96 $_{+1.69}$
training w / quality control	71.09 $_{+4.42}$	66.31 $_{+3.04}$

Table 3: The effects of synthetic data quality control.

may be due to the higher question difficulty of MBPP Plus or differences in reward hacking probabilities between HumanEval Plus and MBPP Plus, as discussed in Appendix D. Future work could explore more accurate difficulty classifiers or alternative allocation algorithms to further improve performance.

4.3 Quality of Generated Unit Tests

We evaluate the quality of the unit test generated by CodeRM-8B. As each unit test functions as a classifier to determine correct or incorrect solutions, we first utilize accuracy and F1 score as metrics to assess the classification performance of the unit test. We further propose two new metrics to detailed evaluate the possibility of the unit test making incorrect judgments. False Acceptance Rate (FAR) measures the probability that unit tests incorrectly accept invalid solutions. False Rejection Rate (FRR) measures the probability that unit tests incorrectly reject valid solutions. The calculation formulas for these four metrics are introduced in Appendix E.

Table 2 presents the quality of reward signal produced by an individual unit test and the combination of 100 unit tests under our majority voting framework. The results indicate that the trained CodeRM-8B renders more precise assessments of solutions and makes fewer errors than the original Llama3.1-8B. Moreover, we note that while the quality of individual unit tests of CodeRM-8B is inferior to Llama3.1-70B, the quality of multiple unit tests is superior. This phenomenon may suggest that CodeRM-8B produces more diverse unit tests, offering diverse perspectives within our majority voting framework and resulting in a higher quality of code reward signal.

4.4 Ablation Study of Synthetic Data

We conduct ablation studies to investigate the effectiveness of data quality control and instruction-tuning data size in our synthetic data pipeline.

Data Quality. Filtering false positive unit tests is a critical component for increasing training data quality in our synthetic data pipeline. We evaluate the performance of CodeRM-8B against the trained model that lacks this quality control procedure. Table 3 presents that the quality control procedure significantly increased the performance of the trained model, with relative performance gains of approximately 45% and 80% on HumanEval

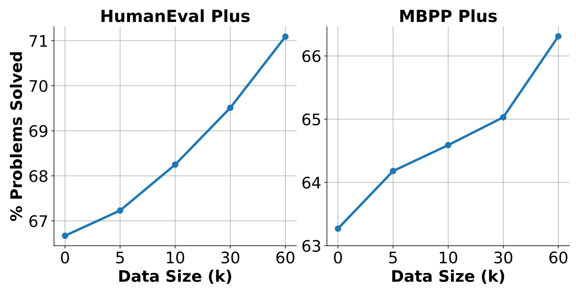

Figure 7: The effects of data size.

Plus and MBPP Plus, respectively, compared to training without quality control.

Data Size. Data size is also a crucial factor in enhancing the model's performance. We evaluate the performance of the model trained on different amounts of training data. Figure 7 presents that as the data size increases, the model's performance consistently improves. This observation demonstrates that collecting more high-quality instruction-tuning data and using our data synthesis pipeline to generate more high-quality unit tests can substantially enhance model performance.

5 Related Work

Reranking and selection for optimal solutions. Current reranking techniques for optimal solution selection can be classified into two categories: execution-based and non-execution-based methods. Execution-based methods leverage test cases as verifiers for selecting optimal code solutions (Li et al., 2022; Shi et al., 2022; Chen et al., 2023; Zhang et al., 2023a; Chen et al., 2024a). For example, Shi et al. (2022) employ execution result-based minimum Bayes risk decoding for solution selection. Chen et al. (2023) evaluate output consistency with generated test cases and concordance with other candidate code solutions. Non-execution-based methods use deep learning-based rerankers for optimal code solution selection (Inala et al., 2022; Zhang et al., 2023b). Inala et al. (2022) trains a neural reranker utilizing the code execution results as labels for classification. Zhang et al. (2023b) utilizes a collaborative system that employs additional reviewer models to assess the likelihood of the instructions based on the generated code solutions. In this paper, we discover the effects of dynamic scaling of unit tests over programming problems.

Unit test generation. The expense of human-created and maintained unit tests prompts the advancement of automatic unit test generation techniques. Traditional methods, such as search-based (Harman and McMinn, 2010; Lukasczyk and Fraser, 2022), constraint-based (Xiao et al., 2013), and probability-based approaches (Pacheco et al., 2007), often achieve acceptable correctness but suffer from limited coverage, poor readability, and are typically restricted to regression or implicit oracles (Barr et al., 2015). Recently, deep learning models, particularly large language models, have gained traction for unit test generation (Alagarsamy et al., 2024; Chen et al., 2024b; Schäfer et al., 2024; Yuan et al., 2024). In this paper, we propose a data synthetic pipeline and train a model for high-quality unit test generation.

6 Conclusion

This paper investigates the impact of scaling unit tests on improving the quality of code reward signals. Pioneer results demonstrate a positive correlation between unit test number and reward signal quality, with more greater benefits observed in challenging problems. To facilitate efficient and high-quality unit test scaling, we train a small yet powerful unit test generator and implement a dynamic scaling strategy. Experimental results demonstrate that our approach significantly boosts the performance of models in various parameter sizes.

Acknowledgments

This work is supported by the National Key Research & Develop Plan (2023YFF0725100) and the National Natural Science Foundation of China (62322214, U23A20299, U24B20144, 62172424, 62276270).

7 Limitations

Implementation of Dynamic Scaling. Our implementation of dynamic unit test scaling is based on the method proposed by Damani et al. (2024), which allocates more computational resources to harder problems. However, while Damani et al. (2024) directly optimizes the allocation of computational resources for the policy model (i.e., models that generate responses), their approach does not directly extend to optimizing the reward model (i.e., models that generate verifiers). Consequently, directly adopting this method may not be entirely appropriate in our context. As shown in Figure 6, the performance improvement is relatively modest when computation budgets are allocated based on the gold pass rate on HumanEval Plus. Future research could explore more effective methods for dynamically scaling the number of unit tests based on varying problem difficulties.

Diversity and Coverage of Unit Test. We observe in Section 2 that Gemma-2-27B-it performs significantly worse than Llama3.1-70B when using a single unit test. However, its performance becomes comparable when scaled to 100 unit tests per question. This observation suggests that Gemma-2-27B-it may generate more diverse and higher-coverage unit tests than Llama3.1-70B through repeated sampling. Future work could explore the diversity and coverage of unit tests as the number of unit tests scales, which may provide insights into training more effective reward models for improved supervision accuracy.

Scaling Law of Code Reward. As shown in Figure 1, the marginal benefits of increasing the number of unit tests vary depending on the current scale of unit tests. The performance improvement trend may plateau once the number of unit tests reaches a sufficiently high threshold. In this paper, we present an empirical study on the impact of unit test scaling. A promising direction for future research is to employ mathematical modeling and analytical tools to systematically characterize the relationship between unit test scaling and best-of-N performance Such an analysis could help identify the saturation point of the performance curve, providing more practical and efficient guidelines for determining the optimal number of unit tests in code generation applications.

Generalization to Software Engineering Task. In this paper, we investigate the relationship between the number of unit tests and best-of-N performance in code generation tasks. Specifically, we follow the settings of previous research (e.g., CodeT, MPSC), which focus on generating short, function-level Python code. Although we include LiveCodeBench, a challenging and contamination-free dataset that continuously collects new problems over time, it may not fully reflect real-world software engineering scenarios. Future work could explore the impact of unit test scaling in broader software engineering tasks, enhancing the practical applicability of our method in real-world coding environments.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: Assertion-augmented automated test case generation. *Inf. Softw. Technol.*, 176(C).

Guillaume Alain and Yoshua Bengio. 2017. Understanding intermediate layers using linear classifier probes. In *ICLR (Workshop)*.

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

Hritik Bansal, Arian Hosseini, Rishabh Agarwal, Vinh Q Tran, and Mehran Kazemi. 2024. Smaller, weaker, yet better: Training llm reasoners via compute-optimal sampling. *arXiv preprint arXiv:2408.16737*.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

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, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Mouxiang Chen, Zhongxin Liu, He Tao, Yusu Hong, David Lo, Xin Xia, and Jianling Sun. 2024a. B4: Towards optimal assessment of plausible code solutions with plausible tests. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1693–1705, New York, NY, USA. Association for Computing Machinery.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024b. Chatunitest: A framework for llm-based test generation.

In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 572–576, New York, NY, USA. Association for Computing Machinery.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. 2024. Learning how hard to think: Input-adaptive allocation of lm computation. *arXiv preprint arXiv:2410.04707*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Jack Edmonds. 1971. Matroids and the greedy algorithm. *Mathematical programming*, 1:127–136.

B. Efron. 1979. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1):1 – 26.

Leo Gao, John Schulman, and Jacob Hilton. 2023. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, pages 10835–10866. PMLR.

Wes Gurnee and Max Tegmark. 2024. Language models represent space and time. In *The Twelfth International Conference on Learning Representations*.

Mark Harman and Phil McMinn. 2010. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247.

Baizhou Huang, Shuai Lu, Xiaojun Wan, and Nan Duan. 2024a. Enhancing large language models in coding through multi-perspective self-consistency. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1429–1450, Bangkok, Thailand. Association for Computational Linguistics.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024b. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2024c. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, et al. 2022. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

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, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 168–172, New York, NY, USA. Association for Computing Machinery.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Benedikt Stroebl, Sayash Kapoor, and Arvind Narayanan. 2024. Inference scaling flaws: The limits of llm resampling with imperfect verifiers. *arXiv preprint arXiv:2411.17501*.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, New York, NY, USA. Association for Computing Machinery.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Haoxiang Wang, Wei Xiong, Tengyang Xie, Han Zhao, and Tong Zhang. 2024. Interpretable preferences via multi-objective reward modeling and mixture-of-experts. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 10582–10592, Miami, Florida, USA. Association for Computational Linguistics.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 246–256.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proc. ACM Softw. Eng.*, 1(FSE).

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023a. ALGO: Synthesizing algorithmic programs with generated oracle verifiers. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023b. Coder reviewer reranking for code generation. In *ICML*, pages 41832–41846.

Xiaokang Zhang, Zijun Yao, Jing Zhang, Kaifeng Yun, Jifan Yu, Juanzi Li, and Jie Tang. 2024. Transferable and efficient non-factual content detection via probe training with offline consistency checking. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12348–12364, Bangkok, Thailand. Association for Computational Linguistics.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12834–12859, Bangkok, Thailand. Association for Computational Linguistics.

A License

We utilize the CodeFeedback-Filtered-Instruction dataset and the training set from TACO as the data sources for generating high-quality unit tests. Both datasets are distributed under the Apache 2.0 license, which permits users to freely use, copy, modify, and distribute the software for both personal and commercial purposes.

The parameters of CodeRM-8B, along with the corresponding training data, will be made publicly available upon acceptance. The training data exclusively consists of synthetic code solutions and unit tests, without any personally identifying information or offensive content. We will release the LLM-generated data and the models fine-tuned on this data under the Apache 2.0 license.

B More Results of Unit Test Scaling

Figure 8 presents a more fine-grained analysis by categorizing questions based on their difficulty. The results further confirm that increasing the number of unit tests generally leads to larger performance improvements on more challenging problems. This trend becomes particularly evident when leveraging more advanced models, such as Llama3.1-70B as the policy model or GPT-4o as the reward model.

C Experiment Settings and Baselines

The experiments employ four policy models of various parameter sizes and types, including Llama3-8B-Instruct, Llama3-70B-Instruct, GPT-3.5-turbo, and GPT-4o-mini. For each policy model, we generate 100 candidate code solutions, following the hyperparameters in Table 4. All models are deployed and inferenced using vLLM (Kwon et al., 2023) and 8 NVIDIA A800 GPUs.

Hyperparameters	Value
Temperature	0.8
Top P	0.95
Frequency Penalty	0
Presence Penalty	0

Table 4: The hyperparameters of LLMs for solution and unit test generation.

For the baselines, we first utilize a grading reward model, which takes a candidate code solution as input and outputs a scalar score representing the quality of the solution. Specifically, we employ ArmoRM-Llama3-8B-v0.1, a powerful reward model that leverages mixture-of-experts (MoE) aggregation across multiple reward objectives. In addition, we include several baselines that use unit tests as verifiers to identify correct code solutions:

- MBR-Exec: This method ranks solutions using minimum Bayes risk (MBR) decoding based on the execution results of LLM-generated test cases. For our experiments, we adopt the hard loss variant of this approach.
- CodeT: This baseline evaluates the consistency of outputs with generated test cases and assesses concordance among candidate code solutions through a dual execution agreement mechanism.
- MPSC: This method evaluates candidate code solutions from three perspectives: solution, specification, and test case. It constructs a 3-partite graph to identify the optimal solution.

To ensure a fair comparison, all baselines leveraging unit tests are provided with the same computational budget of 100 inferences. For MBR-Exec and CodeT, we perform 100 inferences and prompt the LLM to generate 10 test cases for each inference, following the setup of CodeT. For MPSC, under the same computational constraints, we instruct the LLM to generate 50 test cases and 50 specifications.

Because we follow previous research (Chen et al., 2023; Huang et al., 2024a) that conducts experiments on function-level Python code generation benchmarks, we do not employ symbolic-based or deep learning-based test case generation models designed for other programming languages (Alagarsamy et al., 2024; Chen et al., 2024b), such as Java, as this would introduce domain discrepancies, making direct comparisons methodologically challenging.

D Impact of Reward Hacking

As shown in Figure 6, dynamic scaling has a more significant impact on MBPP Plus compared to HumanEval Plus. In this section, we investigate reward hacking in these two benchmarks and find that its probability may be a key factor contributing to the performance differences between them.

In our experiments, reward hacking refers to a phenomenon where scaling the number of unit

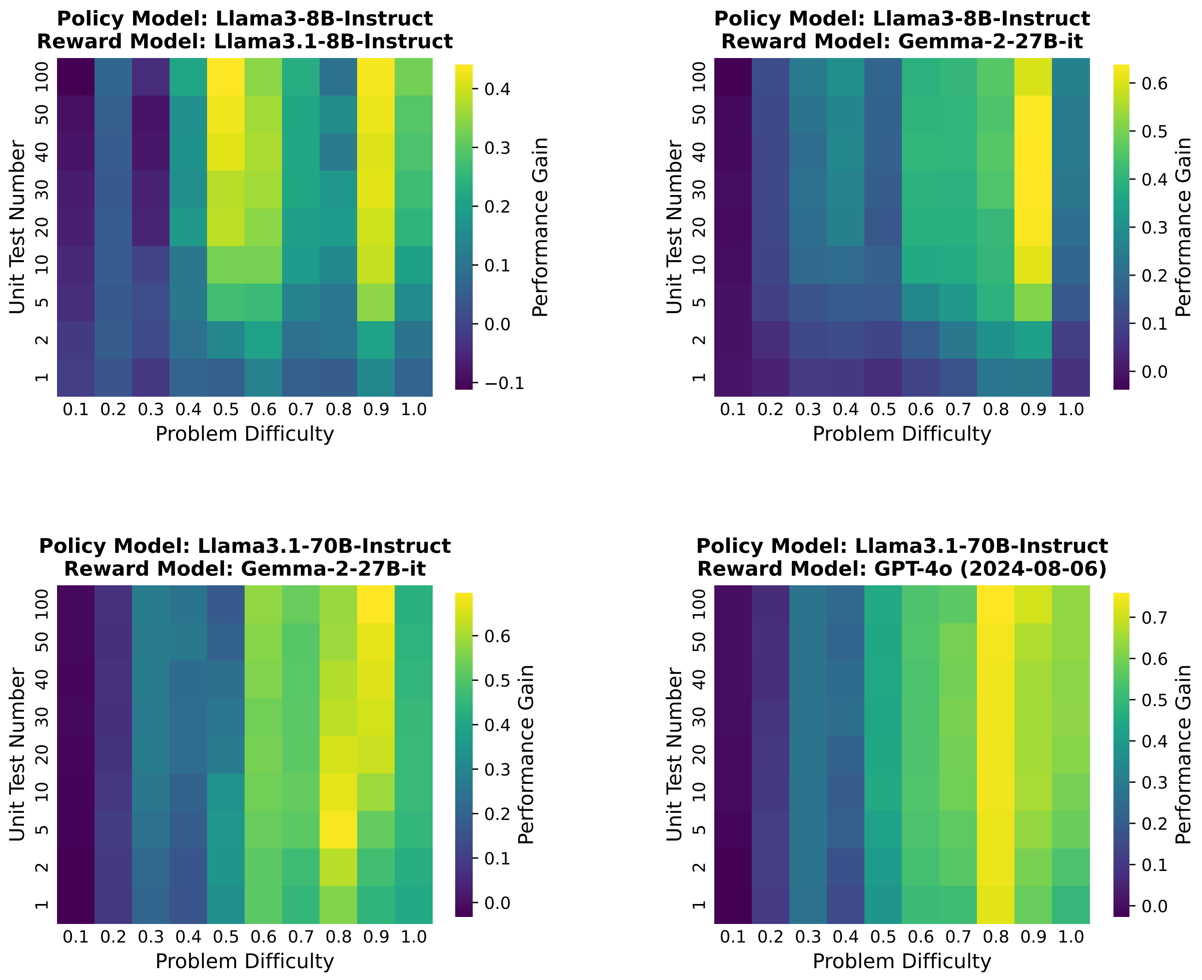

Figure 8: The performance gain of scaling the number of unit tests on problems of different difficulties across various policy model and reward model. Overall, increasing the number of unit tests yields greater performance improvements on more challenging problems, particularly when employing Llama3.1-70B as the policy model.

tests eventually leads to a decline in performance. Unlike Damani et al. (2024), who focus on dynamically allocating computation for policy models, scaling the computation of the reward model can lead to overoptimization, where increasing the number of unit tests reduces the best-of-N performance (Gao et al., 2023).

To investigate this phenomenon, we calculated the percentage of problems affected by reward hacking in both benchmarks. Specifically, we not only computed the overall percentage of reward hacking across the entire benchmark but also analyzed how its occurrence varies across problems of different difficulty levels. Following the setup described in Section 2.2, we split each benchmark into five quintiles based on problem difficulty and present the results in Table 5.

The results reveal that HumanEval Plus has a higher overall proportion of problems (9.76%) experiencing performance degradation during unit test scaling compared to MBPP Plus (7.41%). Furthermore, we observe that reward hacking occurs more frequently in harder problems (Quintile 5) within HumanEval Plus. This observation directly affects our implementation of dynamic scaling, which allocates resources based on problem difficulty. This disparity may partially explain why dynamic scaling yields similar performance to equal allocation on HumanEval Plus but demonstrates improvements on MBPP Plus.

However, reward hacking might not be the sole reason for the observed differences. The behavior of scaling verifier computations remains an underexplored area (Stroebl et al., 2024). As discussed in the Limitation Section, future research could investigate more effective and robust methods for dynamically scaling unit tests, which may further improve performance across a wider range

Benchmark	Quintile 1 (easiest)	Quintile 2	Quintile 3	Quintile 4	Quintile 5 (hardest)	Overall
HumanEval+	0.00%	0.00%	7.41%	4.81%	37.04%	9.76%
MBPP+	0.00%	5.17%	6.90%	12.28%	24.56%	7.41%

Table 5: The probability of reward hacking problems across different difficulty levels and overall in HumanEval Plus and MBPP Plus.

of benchmarks.

E Metrics for Assessing Unit Test Quality

We introduce the details for computing the four metrics for evaluating the quality of unit tests in Section 4.3. To evaluate the quality of the generated unit tests, we define four metrics: *Accuracy*, *F1 Score*, *False Acceptance Rate (FAR)*, and *False Rejection Rate (FRR)*. These metrics are computed in the same manner for both individual unit tests and multiple unit tests under the majority voting framework. However, the interpretation of *True Positives (TP)*, *True Negatives (TN)*, *False Positives (FP)*, and *False Negatives (FN)* slightly differs between these two settings.

The four metrics are formally defined as follows. **Accuracy** measures the proportion of correct predictions and is given by:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3)$$

Precision quantifies the proportion of predicted positives that are actually correct:

$$\text{Precision} = \frac{TP}{TP + FP} \quad (4)$$

Recall (also known as True Positive Rate) measures the proportion of actual positives that are correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN} \quad (5)$$

F1 Score is the harmonic mean of precision and recall:

$$\text{F1 Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (6)$$

False Acceptance Rate (FAR) measures the probability of a wrong solution being incorrectly accepted:

$$\text{FAR} = \frac{FP}{FP + TP} \quad (7)$$

False Rejection Rate (FRR) measures the probability of a correct solution being incorrectly rejected:

$$\text{FRR} = \frac{FN}{FN + TN} \quad (8)$$

The definitions of *True Positives (TP)*, *True Negatives (TN)*, *False Positives (FP)*, and *False Negatives (FN)* are as follows. *True Positives (TP)* denote the number of correct solutions that are classified as correct, while *True Negatives (TN)* refer to the number of incorrect solutions that are classified as incorrect. *False Positives (FP)* represent the number of incorrect solutions that are classified as correct, and *False Negatives (FN)* are the number of correct solutions that are classified as incorrect.

The distinction between a single unit test and majority voting lies in the classification process. For a single unit test, the classification is directly determined by whether the unit test accepts or rejects a solution. For multiple unit tests under majority voting, a solution is classified as *correct* if it is accepted by the largest number of unit tests among all candidate solutions, while the remaining solutions are classified as *incorrect*. These definitions ensure the metrics are consistently applicable across both single and multiple unit test scenarios.

F Prompt for Data Synthetic Pipeline

Prompt for Filtering Unsuitable Questions for Unit Testing

Below is a programming question and a Python code solution. You need to determine whether this question is challenging to evaluate using traditional unit tests. Apply the following criteria to identify questions that are hard to evaluate using unit tests:

1. Functions involving randomness or probability:
1) Random number generators; 2) Shuffling algorithms; 3) Probability-based functions

2. Time-dependent functions:
1) Functions that get the current time; 2) Timer functions

3. Functions relying on external resources:
1) Network request functions; 2) File system operations; 3) Database queries

4. Concurrency and multithreading functions:
1) Thread synchronization functions; 2) Concurrent operation functions

5. Hardware-related functions:
1) Device driver functions; 2) Hardware sensor reading functions

6. User interface related functions:
1) Graphics rendering functions; 2) User input processing functions

7. Functions with side effects:
1) Functions modifying global state; 2) Logging functions

8. Cryptography-related functions:
1) Functions generating encryption keys; 2) Certain encryption algorithm implementations

9. Machine learning and adaptive algorithm functions:
1) Model training functions; 2) Neural network backpropagation algorithms; 3) Self-tuning algorithms

10. Complex mathematical or simulation functions:
1) High-precision floating-point calculations; 2) Physical simulations (e.g., fluid dynamics, particle collisions); 3) Complex optimization algorithms

### programming question
{question}

### code solution
‘‘‘python
{code}
‘‘‘

Let’s think step by step: If the question and answer meet the above criteria, please answer YES; otherwise, answer NO. Please first give the reason for your judgments, followed by your decision. Your decision should be in the last line of the reply, which ONLY contains one word: YES or NO.

Prompt for Unit Test Repairation

I currently have an incorrect unit test code, where some of the output does not match the correct answer. After running the unit test on the correct code answer, I obtained the execution result of the unit test. Please modify the original unit test based on the execution result to make the output correct.

### Unit test with error output
{unit_test}
### Execution result with the ground truth
{exec_result}

You should output the complete modified unit test code in markdown format. Do NOT add any additional comments to the original unit test except for modifying the output.

Prompt for Code Entrance Generation

Below is a code solution and a corresponding unit test. Please locate the function in the code solution that the unit test is testing, and output the function name, return values, and input parameters.

### Code Solution
{code}
### Execution result with the ground truth
{unit_test}

Please output the function name, return values, and input parameters in the following format WITHOUT any other words:
Function name:
Input parameters:
Return values:
Function declaration:

Prompt for Generating Solutions for TACO Dataset

Below is a programming question and an answer format. You need to use Python to answer this question following the provided function format.

### Programming Question
{question}
### Answer Format
{answer_format}

Please follow the answer format to output your answer in markdown format. You need to return the values required by the question instead of printing them. Attention: You ONLY need to output code answer in function format WITHOUT any other words.

Prompt for Genearting Unit Tests for Reward Model

Below is a question and it's corresponding code answer. Please write test cases to check the correctness of the code answer. You need to use the unittest library in Python and create a test class for testing.

### question
{question}
### code solution
{code}

Please add detailed comments to the test cases you write. You do not need to test the function's ability to throw exceptions.

Prompt for Generating Code Solutions for Policy Model

Please provide a self-contained Python script that solves the following problem in a markdown code block:
```Python
{prompt}
```

Prompt for Reorganizing Questions in CodeFeedBack-Filtered Dataset

Below is a programming question and its corresponding answer in Python code. Please reorganize the programming question to make it precise and clear. The reorganized question should only contain the required function name, information, and restriction of the question and exclude any irrelevant information and irrelevant code snippets. ONLY output the reorganized question in one or few paragraphs without headline, title, subtitle, etc.

### programming question
{query}
### answer in Python code
{answer}

Prompt for Reorganizing Code Solutions in CodeFeedBack-Filtered Dataset

Below is a programming question and its corresponding answer in Python code. Please reorganize the answer to include ONLY the required function in the code snippets. The reorganized answer should exclude any test case in the original code snippets and ensure the function name (entance) is consistent with the name in the question.

### programming question
{query}
### answer in Python code
{answer}

Instruction:

Below is a question and it's corresponding code answer. Please write test cases to check the correctness of the code answer. You need to use the unittest library in Python and create a test class for testing.

```
### question
Create a function called `expected_strangle_return` that takes as input the change in implied volatility (`delta_iv`), the realized volatility (`rv`), the vega of the options (`vega`), and the premium paid for the strangle (`premium`). Return the estimated expected return of a strangle. The function should assume a symmetric strangle with the same number of contracts for puts and calls and consider the difference between the implied volatility of puts and calls due to skew. The function should not actually calculate the vega, delta_iv, or rv, but rather use the provided values as inputs.

### code solution
def expected_strangle_return(delta_iv, rv, vega, premium):
    """
    Estimate the expected return of a strangle.

    Args:
    delta_iv (float): The change in implied volatility.
    rv (float): The realized volatility.
    vega (float): The vega of the options.
    premium (float): The premium paid for the strangle.

    Returns:
    float: The estimated expected return of a strangle.
    """
    return vega * (delta_iv – rv) – premium
```

Please add detailed comments to the test cases you write. You do not need to test the function's ability to throw exceptions.

Response:

```
import unittest

class TestExpectedStrangleReturn(unittest.TestCase):
    def test_positive_return(self):
        """
        Test a scenario where the expected return should be positive.
        This happens when the vega * (delta_iv – rv) is greater than the premium.
        """
        delta_iv = 0.10  # Increase in implied volatility
        rv = 0.05        # Realized volatility
        vega = 100       # Vega of the options
        premium = 50     # Premium paid for the strangle

        # Expected return is calculated as vega * (delta_iv – rv) – premium
        expected_return = vega * (delta_iv – rv) – premium
        self.assertAlmostEqual(expected_strangle_return(delta_iv, rv, vega, premium), expected_return)

    def test_negative_return(self):
        """
        Test a scenario where the expected return should be negative.
        This happens when the vega * (delta_iv – rv) is less than the premium.
        """
        delta_iv = 0.05  # Increase in implied volatility
        rv = 0.10        # Realized volatility
        vega = 100       # Vega of the options
        premium = 60     # Premium paid for the strangle

        # Expected return is calculated as vega * (delta_iv – rv) – premium
        expected_return = vega * (delta_iv – rv) – premium
        self.assertAlmostEqual(expected_strangle_return(delta_iv, rv, vega, premium), expected_return)

    def test_zero_return(self):
        """
        Test a scenario where the expected return should be zero.
        This happens when the vega * (delta_iv – rv) equals the premium.
        """
        delta_iv = 0.07  # Increase in implied volatility
        rv = 0.03        # Realized volatility
        vega = 100       # Vega of the options
        premium = 40     # Premium paid for the strangle

        # Expected return is calculated as vega * (delta_iv – rv) – premium
        expected_return = vega * (delta_iv – rv) – premium
        self.assertAlmostEqual(expected_strangle_return(delta_iv, rv, vega, premium), expected_return)
```

Figure 9: An example of the training data for the unit test generator.