AceCoder: Acing Coder RL via Automated Test-Case Synthesis

Huaye Zeng[†*] , **Dongfu Jiang**[†*] , **Haozhe Wang**[‡], **Ping Nie**[§], **Xiaotong Chen**[♣], **Wenhu Chen**[†]
[†]University of Waterloo, [‡]HKUST, [§]Independent Researcher, [♣]Netmind.AI
{w33zeng,dongfu.jiang,wenhuchen}@uwaterloo.ca

https://tiger-ai-lab.github.io/AceCoder

Abstract

Most progress in recent coder models has been driven by supervised fine-tuning (SFT), while the potential of reinforcement learning (RL) remains largely unexplored, primarily due to the lack of reliable reward data/model in the code domain. In this paper, we address this challenge by leveraging automated large-scale test-case synthesis to enhance code model training. Specifically, we design a pipeline that generates extensive (question, test-cases) pairs from existing code data. Using these test cases, we construct preference pairs based on pass rates over sampled programs to train reward models with Bradley-Terry loss. It shows an average of 10-point improvement for Llama-3.1-8B-Ins and 5-point improvement for Qwen2.5-Coder-7B-Ins through best-of-32 sampling, making the 7B model on par with 236B DeepSeek-V2.5. Furthermore, we conduct reinforcement learning with both reward models and test-case pass rewards, leading to consistent improvements across HumanEval, MBPP, BigCodeBench, and LiveCodeBench (V4). Notably, we follow the R1-style training to start from Qwen2.5-Coder-base directly and show that our RL training can improve model on HumanEval-plus by over 25% and MBPP-plus by 6% for merely 80 optimization steps. We believe our results highlight the huge potential of reinforcement learning in coder models.

1 Introduction

In recent years, code generation models have advanced significantly with compute scaling (Kaplan et al., 2020) and training data quality improvement (Huang et al., 2024; Lozhkov et al., 2024; Guo et al., 2024b). The state-of-the-art coder models, including Code-Llama (Rozière et al., 2023), Qwen2.5-Coder (Hui et al., 2024a), DeepSeek-Coder (Guo et al., 2024a) and so on, have shown unprecedented performance across a wide range of coding tasks like program synthesis (Chen et al., 2021), program repair (Zheng et al., 2024a), optimization (Shypula et al., 2023), test generation (Steenhoek et al., 2023), SQL (Yu et al., 2018), issue fix (Jimenez et al., 2024). These models are all pre-trained and further supervised fine-tuned (SFT) on large-scale coding data from web resources like Common Crawl or Github.

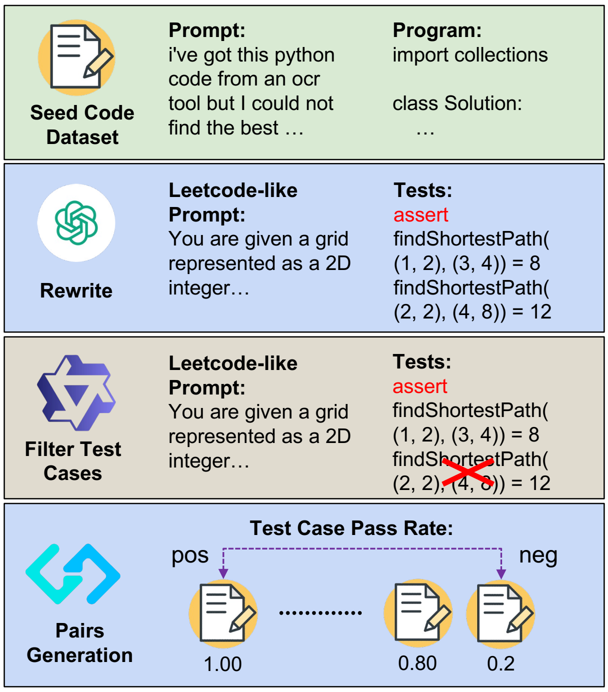

Figure 1: Overall Workflow of our model: We start from the seed code dataset to create well-formatted questions and corresponding test cases. Then we adopt strong models like filter the noisy test cases. Finally, we adopt these test cases to harvest positive and negative program pairs for reward model training and RL.

Though strong performance has been achieved through SFT (Luo et al., 2023; Wei et al., 2024), very few models have explored the potential of reinforcement learning (RL) (Ouyang et al., 2022a), which has proven effective in other domains such as mathematical reasoning like DeepSeek-R1 (Shao et al., 2024). We argue that this absence of RL-

[*]Eqaul Contribution

based training in coder models is primarily due to two key challenges:

(1) Lack of reliable reward signals for code generation. In tasks such as mathematical problem-solving, rewards can be easily derived from rule-based string matches with reference answers (Guo et al., 2025) or large-scale human annotations (Ouyang et al., 2022b). In contrast, evaluating code quality typically requires executing test cases to measure the pass rate, making reward signal design more complex. This also explains why existing reward models like Skywork (Liu et al., 2024a) can hardly generalize to the coding domain (see subsection 4.4).
(2) Scarcity of large-scale coding datasets with reliable test cases. Most existing coding datasets like APPS (Hendrycks et al., 2021; Chen et al., 2021) heavily rely on costly human expert annotations for test cases, which limits their scalability for training purposes.

Therefore, to resolve the above-mentioned issues, we construct ACECODE-89K, the first large-scale verifiable code training dataset. We take a few steps to build the dataset: **(1)** we collect seed coding questions from existing SFT datasets, **(2)** we prompt GPT-4o-mini (Hurst et al., 2024) to rewrite the coding problem in LeetCode style (self-contained with clear problem setup), also 'imagine' around 20 test cases based on its understanding of the problem. The synthesized dataset is in the form of (question, [t_1, t_2, ...]), where t_i is the test case (i.e... 'assert f(input$_i$)=output$_i$'). **(3)** we further adopt Qwen2.5-Coder-32B-Ins (Hui et al., 2024a) to generate programs w.r.t the question. We throw the noisy test cases based on the pass rate of the programs, resulting the final dataset with 89K questions paired with 300K test cases.

Based on ACECODE-89K, we trained our reward models: ACECODE-RM-7B and ACECODE-RM-32B. Comprehensive experiments of best-of-N sampling show that ACECODE-RM can significantly boost existing LLM's performance on coding benchmarks. For example, ACECODE-RM-7B can improve the performance of Llama-3.1-8B-Instruct by an average of 8.4 points across the 4 coding benchmarks, i.e. HumanEval (Liu et al., 2023), MBPP (Liu et al., 2023), BigCodeBench (Zhuo et al., 2024) and LiveCodeBench (Jain et al., 2024). Even for the stronger coder model Qwen2.5-Coder-7B-Instruct, our "7B+7B" combination still gets an average of 2.6 improvements. ACECODE-RM-32B is even more powerful, which pushes the former two numbers to 10.7 and 4.7 respectively, showcasing the effectiveness of ACECODE-RM.

Furthermore, we adopt ACECODE-RM-7B and test case pass rate separately to do reinforcement learning with reinforce++ (Hu, 2025) over coder models. Experiments show 2.1 and 0.7 points of average improvement when starting from Qwen2.5-7B-Ins and the Qwen2.5-Coder-7B-Ins respectively, making the latter even more powerful than GPT-4-Turbo on benchmarks like MBPP. Inspired by the recent DeepSeek-R1 (Guo et al., 2025), we also perform RL training directly from the Qwen2.5-Coder-7B-base model and saw a surprising 25% improvement on HumanEval-plus and 6% improvement on MBPP-plus (Liu et al., 2023) with merely 80 optimization steps (48 H100 GPU hours). These improvements are also generalizable to other more difficult benchmarks.

To our knowledge, this is the first work to propose a fully automated pipeline for synthesizing large-scale reliable tests used for the reward model training and reinforcement learning in the coding scenario. We believe our ACECODE-89K will unlock the potential of RL training for code generation models and help the community to further push the boundaries of LLM's coding abilities.

2 Methodology

In this section, we will introduce the overall methodology of ACECODER. We begin with formulations of the problems we are investigating, including reward model training and reinforcement learning for LLMs. We then elaborate on how we synthesize the test cases and construct the ACECODE-89K. Finally, we explain how we perform the reinforcement learning using our ACECODE-RM trained on the ACECODE-89K.

2.1 Problem Formulation

Reward Model Training Let x denote the coding question and $\mathbf{y} = \{y_1, \cdots, y_t\}$ denote the program solution, where y_i represents the i-th token of the program solution and $(\mathbf{x}, \mathbf{y}) \in D$. Assuming θ represents the parameters of the model, then n responses $(\mathbf{y}^1, ..., \mathbf{y}^n)$ will be sampled from the model π_θ given the input $\mathbf{x}$. Let $(s_1, ..., s_n)$ be the target rewards, i.e. the test case pass rates in our scenario, then we define the Bradley-Terry loss (Bradley and Terry, 1952) for every pair of responses $\mathbf{y}^i$ and $\mathbf{y}^j$ with scores of s_i and s_j when we are training a reward model R_ϕ as follows:

$$\mathcal{L}_\phi(\mathbf{x}, s_i, s_j) \\ = \mathbb{1}[s_i > s_j] \log \sigma(R_\phi(\mathbf{x}, \mathbf{y}^i) - R_\phi(\mathbf{x}, \mathbf{y}^j))$$

where $\mathbb{1}[\cdot] = 1$ if the expression inside the brackets is true, otherwise, it's 0. The final loss function for the reward training is:

$$\mathcal{L}(\phi) = -\frac{1}{n(n-1)} \sum_{i=1}^{n} \sum_{j=1}^{n} \mathcal{L}_\phi(\mathbf{x}, s_i, s_j) \quad (1)$$

That means the reward model is trained to assign higher values to preferred responses and lower values to non-preferred ones, maximizing the difference between these ratings.

Best-of-N Sampling After we get the trained reward model R_ϕ, one way to quickly test the performance of the reward model is Best-of-N sampling, which is usually used as a test-time scaling approach. We will simply select the best response according to the predicted value of R_ϕ. That is $\mathbf{y}^* = \arg\max_{\mathbf{y}^i \in \mathbf{y}^1, \ldots, \mathbf{y}^N} R_\phi(\mathbf{x}, \mathbf{y}^i)$.

Reinforcement Learning We can finally conduct reinforcement learning for the original policy model π_θ after we get a well-trained reward model R_ϕ. Proximal Policy Optimization (PPO) is an actor-critic RL algorithm that is widely used for LLM's RL process. Let $\pi_{\theta_{old}}$ be the reference model and π_θ be the current policy model that we are updating frequently during the RL training. We denote $r_t(\theta)$ as the probability ratio of the current policy model over the old policy model on the t-th generated token:

$$r_t(\theta) = \frac{\pi_\theta(y_t | \mathbf{x}, \mathbf{y}_{<t})}{\pi_{\theta_{old}}(y_t | \mathbf{x}, \mathbf{y}_{<t})} \quad (2)$$

Then the PPO algorithms optimize the LLM by the following surrogate objective:

$$\mathcal{L}_{PPO}(\theta) = \\ -\frac{1}{|\mathbf{y}|} \sum_{t=1}^{|\mathbf{y}|} \min\left[r_t(\theta) A_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) A_t\right]$$

where $\mathbf{y} \sim \pi_{\theta_{old}}(\cdot|x)$, and A_t is the advantage computed through the Generalized Advantage Estimation (GAE) (Schulman et al., 2015) via the rewards generated by R_ϕ and the learned value function V_ψ. The PPO training objective will force the policy model π to increase the probability of generating tokens with higher A_t and decrease the probability ratio of generating tokens with lower A_t until the clipped bounds $1+\epsilon$ and $1-\epsilon$ are reached respectively.

However, PPO usually requires training an additional value model V_ψ and thus makes the training inefficient. Recently, there are some other works like Reinforecement++ (Hu, 2025) that eliminate the need for value model but instead compute advantage only using the rewards generated by R_ϕ and the KL-divergence of the tokens after the t-th tokens. This makes the RL process more efficient and has also proved to be more stable.

3 ACECODE-89K

To be able to train a reward model specifically designed for code generation, the first thing is to synthesize reliable test cases for each coding problem and use them as training signals. In this section, we explain the whole procedure of constructing ACECODE-89K step by step. We show the overall statistics in Table 1.

Test Case Synthesis from Seed Dataset We start from existing coding datasets with provided question $\mathbf{x}$ and corresponding program $\mathbf{y}$. Specifically, we combine Magicoder-Evol-Instruct[1], Magicoder-OSS-Instruct-75K[2], and StackPyFunction[3] as our seed dataset. We keep only the questions written in Python that contain either a function or a class, resulting in a total of 124K entries. We find that these datasets contain highly noisy questions that could not be easily evaluated using test cases. Therefore, we feed every question-solution pair $(\mathbf{x}, \mathbf{y})$ into a GPT-4o-mini (Hurst et al., 2024) to propose a refined LeetCode-style question $\mathbf{x_r}$ with highly structured instructions. Meanwhile, we also prompt it to 'imagine' around 20 test cases $(t_1, \ldots, t_m)$ for each refined coding question $\mathbf{x_r}$ based on its understanding of the expected behavior of the desired program. See prompt template used in subsection A.1. Please note that we do not use the program solution $\mathbf{y}$ from the existing datasets at all in our final curated ACECODE-89K. These datasets are purely used as seeds to help LLM formulate well-structured coding problems.

Test Case Filtering These 'imagined' test cases generated from the LLM contain severe hallucina-

[1] ise-uiuc/Magicoder-Evol-Instruct-110K
[2] ise-uiuc/Magicoder-OSS-Instruct-75K
[3] bigcode/stack-dedup-python-fns

Subset	Evol	OSS	Stack Python	Overall
Before Filtering				
# Examples	36,256	37,750	50,000	124,006
# Avg Test Cases	19.33	17.21	18.27	18.27
After Filtering				
# Examples	27,853	26,346	35,223	89,422
# Avg Test Cases	14.77	16.11	15.79	15.56
# Pairs	89,089	91,636	126,784	307,509

Table 1: Dataset statistics of ACECODE-89K before and after test-case filtering.

tions. To filter out those hallucinated test cases, we facilitated a stronger coder model Qwen2.5-Coder-32B-Instruct (Hui et al., 2024a) as a proxy to perform quality control. Specifically, we prompt it for each $\mathbf{x_r}$ to generate a program $\mathbf{y}'$ and then run these programs over the test cases to approximate their quality. We removed all test cases t_i where the generated solution program $\mathbf{y}'$ could not pass. Furthermore, we removed questions with fewer than 5 tests after filtering, as these questions might be overly ambiguous. With the above filtering, we constructed the ACECODE-89K with 89.4K distinct coding questions and 1.39M cleaned test cases, as represented by $(\mathbf{x_r}, (t_1, ..., t_{m_c}))$, where m_c represents the number of test cases after filtering.

Preference Pairs Construction We propose to use the Bradley-Terry model to train the reward model as defined in Equation 1. Therefore, we need to construct (question, [positive program, negative program]) data from ACECODE-89K. Specifically, we sample programs $(\mathbf{y}^1, ..., \mathbf{y}^n)$ from existing models (e.g. Llama-3.1 (Grattafiori et al., 2024)) w.r.t $\mathbf{x}_r$ and utilize the test-case pass rate to distinguish positive and negative programs. Since the pass rate s_i for the sampled program $\mathbf{y}^i$ can be any number between $[0, 1]$, a minor difference in pass rate may not represent that one program is more accurate than another. Therefore, instead of using $\mathbb{1}[s_i > s_j]$ to select the preference pairs, we have thus modified the selection rules to be:

$$\mathbb{1}[s_i > s_j + 0.4, s_i > 0.8, s_j > 0] \qquad (3)$$

This is to ensure the preferred program has at least a 0.8 pass rate to make sure it represents a more correct program. Also, we find many sampled programs with 0 pass rates can be caused by some small syntax errors or some Python packaging missing errors during evaluation, we chose to not include them as the preference pair to make sure our constructed datasets represent only the preference-based on the valid pass rate. We also ensure the sampled programs all come from the backbone of R_ϕ so the reward model is trained in an on-policy way. After that, we train our reward model R_ϕ by fully fine-tuning an instruct coding model. Specifically, We extract the last token's final hidden representations and pass it through a linear model head that generates a single scalar output, which is optimized via the loss function defined in Equation 1.

4 Experiments

4.1 Reward Model Training Setup

We mainly use Qwen2.5-Coder-7B-Instruct [4] as the backbone of the reward model and sample 16 responses from it for each question in ACECODE-89K. Finally, following the rule defined in Equation 3, around 300K preference pairs were created out of 46,618 distinct questions (37.34% of the total questions) that have at least one pair satisfying the condition, and other questions are not used.

Our reward model is trained using LlamaFactory (Zheng et al., 2024b). We apply full fine-tuning with DeepSpeed stage 3. We train for 1 epoch using a cosine learning rate schedule, starting at 1e-5 with a warmup ratio of 0.1 to gradually increase the learning rate in the initial training phase. Training batch size is set to 128. We enable bf16 precision to reduce memory overhead without compromising model fidelity. The training takes 24 hours on 8 x A100 GPUs.

4.2 Reinforcement Learning Setup

We perform RL training from three policy models: Qwen2.5-7B-Instruct [5] and Qwen2.5-Coder-7B-Base [6] and Qwen2.5-Coder-7B-Instruct. Two types of reward can be used, i.e. the trained reward model ACECODE-RM-7B and the rule-based reward, i.e. pass rate over the test cases in ACECODE-89K. During training, we set the pass rate to be a binary reward, which is 1.0 when all test cases passed, otherwise 0. This is similar to the verfiable reward used in Tulu3 (Lambert et al., 2024a) and DeepSeek-R1 (Guo et al., 2025). Similar to DeepSeek-R1 (Guo et al., 2025), we also experiment with RL from the base model because SFT may cause the search space of the model to be stuck in the local minimum. Since coding is also

[4] Qwen/Qwen2.5-Coder-7B-Instruct
[5] Qwen/Qwen2.5-7B-Instruct
[6] Qwen/Qwen2.5-Coder-7B

Table 2: AceCode-RM's best-of-n results. We evaluated the model on HumanEval, MBPP, BigCodeBench, and LiveCodeBench. Specifically, -C means completion split and -I means instruct split.

Mehod	# N	HumanEval		MBPP		BigCodeBench-C		BigCodeBench-I		LiveCodeBench	**Average**
		-	Plus	-	Plus	Full	Hard	Full	Hard	V4	
GPT-4o (0806)	1	92.7	87.2	87.6	72.2	58.9	36.5	48.0	25.0	43.6	61.3
DeepSeek-V2.5	1	90.2	83.5	87.6	74.1	53.2	29.1	48.9	27.0	41.8	59.5
DeepSeek-V3	1	91.5	86.6	87.6	73.0	62.2	39.9	50.0	27.7	63.5	64.6
Qwen2.5-Coder-32B	1	92.1	87.2	90.5	77.0	58.0	33.8	49.0	27.7	48.3	62.6
Inference Model = Mistral-7B-Instruct-V0.3											
Greedy	1	36.6	31.1	49.5	41.3	25.9	6.1	20.1	5.4	7.3	24.8
Average	64	37.1	30.8	45.1	38.0	21.7	4.2	17.6	3.0	4.0	22.4
Oracle	64	87.2	78.0	83.9	73.5	68.4	37.8	58.5	31.1	24.3	60.3
AceCodeRM-7B	16	65.9	56.7	59.3	**52.4**	35.1	10.1	29.3	8.8	11.9	36.6
	32	68.3	58.5	59.8	51.6	37.4	8.8	30.7	10.8	14.6	37.8
	64	71.3	61.6	59.8	51.6	39.4	6.8	31.8	9.5	15.4	38.6
Δ (RM-greedy)	-	+34.8	+30.5	+10.3	+11.1	+13.5	+4.1	+11.7	+5.4	+8.1	+13.8
AceCodeRM-32B	16	68.3	61.0	58.7	49.5	37.7	11.5	30.9	10.1	12.9	37.8
	32	72.6	**65.9**	**61.6**	51.6	40.5	9.5	33.9	13.5	16.1	40.6
	64	**75.0**	64.6	60.6	50.0	**42.7**	**15.5**	**35.6**	**13.5**	**17.4**	**41.7**
Δ (RM-greedy)	-	+38.4	+34.8	+12.2	+11.1	+16.8	+9.5	+15.5	+8.1	+10.1	+17.4
Inference Model = Llama-3.1-8B-Instruct											
Greedy	1	68.9	62.2	67.2	54.8	38.5	12.8	31.8	13.5	18.0	40.9
Average	64	61.7	54.9	64.5	54.5	32.8	10.1	26.6	9.0	13.8	36.4
Oracle	64	93.9	90.2	92.1	82.3	80.0	54.7	67.9	48.6	40.8	72.3
AceCodeRM-7B	16	77.4	70.7	**76.5**	**64.3**	45.8	20.3	36.4	12.2	26.1	47.7
	32	79.9	72.6	76.2	62.4	47.6	23.0	37.3	13.5	27.3	48.9
	64	81.7	74.4	74.6	61.9	47.8	**23.6**	38.1	13.5	27.6	49.3
Δ (RM-greedy)	-	+12.8	+12.2	+9.3	+9.5	+9.3	+10.8	+6.2	0.0	+9.6	+8.4
AceCodeRM-32B	16	82.3	74.4	72.8	60.6	49.8	20.3	38.4	13.5	27.5	48.8
	32	81.7	76.2	72.8	60.6	**50.4**	22.3	39.1	13.5	30.3	49.6
	64	**85.4**	**79.3**	72.0	59.0	48.5	19.6	**40.0**	**13.5**	**31.0**	**49.8**
Δ (RM-greedy)	-	+16.5	+17.1	+9.3	+9.5	+11.8	+10.8	+8.2	+0.0	+13.0	+10.7
Inference Model = Qwen2.5-Coder-7B-Instruct											
Greedy	1	**91.5**	86.0	82.8	71.4	49.5	19.6	41.8	20.3	34.2	55.2
Average	64	86.0	80.1	77.9	65.6	45.3	18.6	37.3	16.2	31.8	51.0
Oracle	64	98.2	95.7	97.4	90.7	80.9	62.8	73.5	53.4	57.4	78.9
AceCodeRM-7B	16	90.2	82.9	**88.6**	74.9	53.8	20.9	45.0	21.6	40.1	57.6
	32	90.9	86.0	87.8	74.1	53.4	25.0	43.9	19.6	39.8	57.8
	64	90.9	85.4	87.6	73.8	52.9	24.3	43.5	21.6	40.1	57.8
Δ (RM-greedy)	-	-0.6	0.0	+5.8	+3.4	+4.3	+5.4	+3.2	+1.4	+5.9	+2.6
AceCodeRM-32B	16	90.2	86.6	88.4	74.9	53.9	25.0	45.4	19.6	44.0	58.7
	32	90.2	**86.6**	88.4	**75.4**	**55.4**	**29.7**	45.6	21.6	43.5	**59.6**
	64	89.6	86.0	87.8	75.1	55.0	26.4	**46.1**	**22.3**	**44.5**	59.2
Δ (RM-greedy)	-	-0.6	+0.6	+5.8	+4.0	+6.0	+10.1	+4.3	+2.0	+10.3	+4.7

a highly verifiable task like math, we include the Qwen2.5-Coder-7B-Base in our experiments.

We have trained different policy model backbone with different rewards, resulting in 6 RL models in total. All the RL-tuning are based on `OpenRLHF` (Hu et al., 2024). We adopt the Reinforcement++ (Hu, 2025) algorithm instead of PPO to improve the training efficiency without training the value model. It's also proved to be more stable than PPO and GRPO. We train our model on a subsampled hard version of AceCode-89K, where we keep the 25% of the questions with lower average pass rates and higher variance. This is to ensure the question is hard and also the sampled programs are diverse enough. For the training hyperparameters, we set the rollout batch size to 256, and 8 programs are sampled from per question. The training batch size is 128 with a learning rate of 5e-7. All the models are trained for 1 episode and finished in 6 hours on 8 x H100 GPUs.

4.3 Evaluation Setup

We evaluate our method on three established code-focused benchmarks: EvalPlus (Liu et al., 2023, 2024b), Big Code Bench (Zhuo et al., 2024) and Live Code Bench (Jain et al., 2024). These benchmarks collectively cover a diverse array of coding tasks, enabling us to assess both the correctness and quality of generated code. For Best-of-N sampling experiments, we adopt top-p sampling with

Table 3: ACECODER's Performance after RL tuning using Reinforcement++ algorithm. We start with 3 different initial policy models and 2 kind of reward types, where RM means using our trained ACECODE-RM and $Rule$ means using the binary pass rate. Results show consistent improvement across various benchmarks.

Model	HumanEval		MBPP		BigCodeBench (C)		BigCodeBench (I)		LiveCodeBench	Average
	-	Plus	-	Plus	Full	Hard	Full	Hard	V4	
DeepSeek-V2.5	90.2	83.5	87.6	74.1	53.2	29.1	48.9	27.0	41.8	59.5
Baseline = Qwen2.5-7B-Instruct										
Baseline	81.7	73.2	79.4	67.7	45.6	16.9	38.4	14.2	29.0	49.6
AceCoder$_{RM}$	83.5	**77.4**	**83.1**	**71.2**	**46.8**	**16.9**	39.0	14.9	**30.3**	**51.5**
AceCoder$_{Rule}$	**84.1**	77.4	80.2	68.3	46.8	15.5	**40.2**	**15.5**	30.1	50.9
Δ (RL-baseline)	+2.4	+4.3	+3.7	+3.4	+1.2	0.0	+1.8	+1.4	+1.3	+2.1
Baseline = Qwen2.5-Coder-7B-Base										
Baseline	61.6	53.0	76.9	62.9	45.8	16.2	40.2	14.2	**28.7**	44.4
AceCoder$_{RM}$	83.5	75.6	80.2	67.2	41.9	14.9	36.8	16.2	25.7	49.1
AceCoder$_{Rule}$	**84.1**	**78.0**	**82.3**	**69.3**	**48.6**	**18.2**	**43.2**	**18.2**	28.5	**52.3**
Δ (RL-baseline)	+22.5	+25.0	+5.4	+6.4	+2.8	+2.0	+3.1	+4.1	-0.2	+7.9
Baseline = Qwen2.5-Coder-7B-Instruct										
Baseline	**91.5**	**86.0**	82.8	71.4	49.5	19.6	41.8	**20.3**	34.2	55.2
AceCoder$_{RM}$	89.0	84.1	**86.0**	**72.8**	50.4	18.9	42.0	19.6	**35.0**	55.3
AceCoder$_{Rule}$	90.9	84.8	84.1	71.7	**50.9**	**23.0**	**43.3**	19.6	34.9	**55.9**
Δ (RL-baseline)	-0.6	-1.2	+3.2	+1.3	+1.4	+3.4	+1.5	-0.7	+0.8	+0.7

a temperature of 1.0 to generate multiple candidate solutions per question. We select the response with the highest reward for evaluation. For RL experiments, we use the benchmark's default setting, which is greedy sampling most of the time.

4.4 Main Results

Here we show the reward model evaluation results through Best-of-N.

Best-of-N Results We conduct Best-of-N experiments on 3 inference models, specifically Mistral-Instruct-V0.3-7B(AI, 2023), Llama-3.1-Instruct-8B (Grattafiori et al., 2024), and Qwen2.5-Coder-7B-Insutrct (Hui et al., 2024b; Yang et al., 2024a). We additionally report the average score across all generated samples and also the oracle score (pass@N) for better comparison.

According to Table 2, ACECODE-RM can consistently boost the performance of inference models by a large margin compared to the greedy decoding results. On weaker models like Mistral (AI, 2023) and Llama-3.1 (Zheng et al., 2024b), the overall improvements are greater than 10 points. These improvements can be attributed to our reward model's ability to identify high-quality completions among multiple candidates, thereby reducing the impact of suboptimal sampling on the final output. Notably, these gains become more pronounced on benchmarks where the gap between greedy decoding and oracle performance (i.e., the best possible completion among all samples) is larger. In such cases, the variance among sampled completions is relatively high, providing greater opportunities for the reward model to pinpoint and elevate top-tier responses.

Greedy decoding systematically outperforms the average sampled performance, reflecting the strong code generation capability of these inference models. Consequently, while most reward models achieve best-of-N results above the average, we consider a reward model effective only if it surpasses the performance of greedy decoding.

RL Results We perform RL training over 3 different initial policy models in Table 3 with model-based and rule-based rewards. When starting from Qwen2.5-Instruct-7B, we can see the RL tuning can consistently improve the performance, especially on HumanEval and MBPP. Even for the Plus version with more and harder test cases, the RL-tuned model also has more than 3 points of improvement.

When starting from the Qwen2.5-Coder-Instruct-7B itself, we can still observe improvements, especially when using the rule-based reward. For example, we get more than 3.4 improvement on BigCodeBench-Full-Hard. Using the reward model for RL can also bring 3.2 improvement on MBPP, making it (86.0) only 1.6 points behind compared to DeepSeek-V2.5 (87.6). This highlights the charm of self-improvement given the reward model backbone is the same with the initial policy model.

Another experiment we conduct is to perform

RL training directly from base model Qwen2.5-Coder-7B-base. We show significant improvement, especially through test-case pass rewards on HumanEval, MBPP, and BigCodeBench-I. These results are achieved by only training for 80 steps. We believe further scaling up the training will lead to much larger gains.

Comparison with Other RMs We compare our AceCode-RM with 3 top-ranked RM on the RewardBench, including InternLM2-RM-8B (Cai et al., 2024), Skywork-Llama-3.1-8B, and Skywork-Gemma-27B (Liu et al., 2024a), where results are reported in Table 4. We can see that these general-purpose RM can hardly improve and sometimes decrease the performance through Best-of-N sampling compared to greedy sampling, showcasing the incapability in identifying the correct generated programs. On the other hand, our AceCode-RM surpasses all other publicly released reward models in our evaluation and consistently gets positive gains. These findings further underscore our assumption that previous RM training lacks of reliable signals for codes and prove that our RMs can generate reliable and state-of-the-art reward signals in code generation tasks.

4.5 Ablation Studies

Test Case Quality Matters We also conduct experiments to investigate how filtering the test cases with a proxy model can affect the results. As shown in Table 5, training RM on data after the filtering improve the performance significantly, especially for those hard code questions like MBPP-Plus and BigCodeBench-Hard (C/I). We believe this is because the test case filtering can ensure the remaining ones are consistent with each other and thus point to the same implicit program, which improves the quality of the rewards.

RM Backbone Matters Our results in Table 6 clearly show that the changing the backbone of the reward model from Llama-3.1 to Qwen2.5 can significantly improve the Best-of-16 performance. This is because the Qwen2.5-Coder models have been pre-trained on way more code-related data compared to the Llama-3.1 models, and thus more knowledgeable when tuning it into a reward model.

Does R1-style Tuning Work? Inspired by the recent DeepSeek-R1 (Guo et al., 2025), we also conduct the RL directly from the base model without any SFT. It turns out we get huge improvements when using rule-based rewards. For example, we get 25.0 points of improvements on HumanEval-Plus after training only 6 hours from the Base Model, which is way more efficient that the large-scale SFT. What's more, the AceCoder $_{Rule}$ improve the BigCodeBench-Instruct-Full's performance from 40.2 to 43.2, nearly the same performance with DeepSeek-R1-Distill-Qwen-32B (43.9) which was directly distilled from the DeepSeek-R1 Model. This further consolidates the finding of DeepSeek-Zero. However, we do find that using reward models for RL tuning can lead to worse results. We attribute this to the potential reward hacking during the tuning.

5 Related Works

5.1 LLM for Code Generation

Large language models (LLMs) have demonstrated significant potential in code generation. Due to the unique nature of coding tasks, specialized coding models such as Code Llama (Rozière et al., 2023) and Qwen Coder (Hui et al., 2024b; Yang et al., 2024a) were developed shortly after the emergence of general-purpose LLMs. These models typically undergo a two-phase training process: pre-training and fine-tuning. During pre-training, they are exposed to extensive coding corpora sourced from various internet platforms, including raw text, GitHub repositories, and pull requests. This is followed by supervised fine-tuning, which enhances their instruction-following capabilities. To assess the performance of these models in code generation, several benchmarks have been established, including MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021), EvalPlus (Liu et al., 2023, 2024b), Big Code Bench (Zhuo et al., 2024), and Live Code Bench (Jain et al., 2024). These benchmarks usually include a series of prompts or problems for the LLMs to solve, and they also contain test cases to assess the correctness of the generated code.

5.2 Reward Models

Reward models play a crucial role in aligning LLMs by assigning scalar values to response pairs based on specific evaluation criteria, such as human preference (Ouyang et al., 2022b) and accuracy (Zhang et al., 2025). They are widely used in reinforcement learning with human feedback (RLHF) to refine model behavior and in Best-of-N sampling to enhance test-time performance. However, while general-purpose reward models are ef-

Table 4: AceCode-RM's performance against other open-sourced reward models in terms of Best-of-16 sampling for Llama-3.1-8B-Inst. We can see the top-ranked RM on Reward Bench get little improvements compared to ours.

Method & RM	HumanEval		MBPP		BigCodeBench-C		BigCodeBench-I		LiveCodeBench	Average
	-	Plus	-	Plus	Full	Hard	Full	Hard	V4	
Greedy	68.9	62.2	67.2	54.8	38.5	12.8	31.8	**13.5**	18.0	40.9
Average	50.1	42.2	57.9	47.2	22.0	10.6	18.2	12.0	14.9	30.6
InternLM2-RM-8B	57.9	55.5	66.7	54.0	38.7	8.8	29.8	8.8	15.1	37.3
Skywork-Gemma-27B	73.8	67.1	64.3	53.4	40.1	14.9	32.5	12.8	23.6	42.5
Skywork-Llama-3.1-8B	67.7	61.6	69.6	56.9	40.6	10.8	31.8	12.2	18.8	41.1
Δ (max(other RM)-greedy)	+4.9	+4.9	+2.4	+2.1	+2.1	+2.0	+0.6	-0.7	+5.6	+2.7
AceCode-RM-7B	**77.4**	**70.7**	**76.5**	**64.3**	**45.8**	**20.3**	**36.4**	12.2	**26.1**	**47.7**
Δ (RM-greedy)	+8.5	+8.5	+9.3	+9.5	+7.3	+7.4	+4.6	-1.4	+8.1	+6.9

Table 5: Ablation study on test-case filtering. Results are Best-of-16 sampling performance.

Method	HumanEval		MBPP		BigCodeBench-C		BigCodeBench-I		LiveCodeBench	Average
	-	Plus	-	Plus	Full	Hard	Full	Hard	V4	
Inference Model = Llama-3.1-8B-Instruct										
RM w/o Test Case Filter	73.8	65.9	73.3	61.4	44.6	17.6	35.5	9.5	25.1	45.2
RM w/ Test Filter	**77.4**	**70.7**	**76.5**	**64.3**	**45.8**	**20.3**	**36.4**	**12.2**	**26.1**	**47.7**
Δ (w/ Filter - w/o Filter)	+3.7	+4.9	+3.2	+2.9	+1.2	+2.7	+0.9	+2.7	+1.0	+2.6
Inference Model = Qwen2.5-Coder-7B-Instruct										
RM w/o Test Case Filter	**91.5**	**86.0**	86.0	72.2	52.5	**21.6**	43.4	19.6	36.9	56.6
RM w/ Test Filter	90.2	82.9	**88.6**	**74.9**	**53.8**	20.9	**45.0**	**21.6**	**40.1**	**57.6**
Δ (w/ Filter - w/o Filter)	-1.2	-3.0	+2.6	+2.6	+1.3	-0.7	+1.6	+2.0	+3.2	+0.9

Table 6: Comparison of AceCode-RM's performance trained on different base model, where AceCode-RM (Llama) is based on Llama-3.1-Inst-8B and AceCode-RM (Qwen) is based on Qwen-Coder-2.5-7B-Inst. Results are Best-of-16 sampling performance.

Method	HumanEval		MBPP		BigCodeBench-C		BigCodeBench-I		LiveCodeBench	Average
	-	Plus	-	Plus	Full	Hard	Full	Hard	V4	
Inference Model = Llama-3.1-8B-Instruct										
AceCode-RM (LLama)	65.9	59.1	69.6	57.9	42.7	12.8	32.9	**13.5**	19.9	41.6
AceCode-RM (Qwen)	**77.4**	**70.7**	**76.5**	**64.3**	**45.8**	**20.3**	**36.4**	12.2	**26.1**	**47.7**
Δ (Qwen-Llama)	+11.6	+11.6	+6.9	+6.3	+3.1	+7.4	+3.5	-1.4	+6.2	+6.1
Inference Model = Qwen2.5-Coder-7B-Instruct										
AceCode-RM (LLama)	87.8	81.7	82.0	67.7	50.5	**25.0**	39.0	19.6	32.4	54.0
AceCode-RM (Qwen)	**90.2**	**82.9**	**88.6**	**74.9**	**53.8**	20.9	**45.0**	**21.6**	**40.1**	**57.6**
Δ (Qwen-Llama)	+2.4	+1.2	+6.6	+7.1	+3.2	-4.1	+6.0	+2.0	+7.7	+3.6

fective for assessing human preference, they often struggle with specialized domains like mathematics and coding due to the complexity of these tasks. For instance, even top-ranked reward models from Reward Bench (Lambert et al., 2024b), such as Skywork-RM (Liu et al., 2024a), have difficulty providing reliable rewards for these domains. To address this issue, task-specific reward models have been developed, such as Qwen-2.5-Math-PRM (Zhang et al., 2025) for mathematical reasoning. However, coding reward models have remained largely absent due to the lack of reliable training signals—an issue that our proposed AceCode-RM aims to address.

5.3 Reinforcement Learning for LLM

Since the introduction of Reinforcement Learning from Human Feedback (RLHF)(Ouyang et al., 2022b), it has been extensively applied to enhance LLM capabilities in tasks such as conversational interactions and mathematical reasoning(Yang et al., 2024b). Popular reinforcement learning algorithms, including PPO (Schulman et al., 2017), GRPO (Shao et al., 2024), and Reinforcement++(Hu, 2025), have been used to fine-tune models according to reward signals generated either by reward models(Shao et al., 2024) or predefined rule-based rewards (Guo et al., 2025). Despite the fact that coding is inherently a verifiable task, reinforcement learning has seen limited application in code generation due to challenges in defining meaningful and scalable reward signals. The most relevant prior work, CodeRL (Le et al., 2022), leverages the APPS dataset (Hendrycks et al., 2021) to generate rewards based on test case evaluations. However, APPS contains only 5,000 examples, with most problems having just a single test case, making it insufficient for scalable RL-based training. Our work is the first to automate high-quality test case synthesis in code domain.

6 Conclusion

We are first work to automate large-scale test-case synthesis and adopt them to train coder language models. Without relying on the most advanced model, our data collection pipeline can still produce very high-quality verifiable code data, which empowers the training of reward model and coder model through reinforcement learning. Though our work demonstrates huge improvement in Best-of-N experiments, the improvement on RL training is less prominent. We believe future work should further our reward model training to improve its robustness to further the results.

References

Mistral AI. 2023. Mistral-7b-instruct-v0.3. https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3.

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*.

Ralph Allan Bradley and Milton E. Terry. 1952. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39:324.

Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, Xiaoyi Dong, Haodong Duan, Qi Fan, Zhaoye Fei, Yang Gao, Jiaye Ge, Chenya Gu, Yuzhe Gu, Tao Gui, Aijia Guo, Qipeng Guo, Conghui He, Yingfan Hu, Ting Huang, Tao Jiang, Penglong Jiao, Zhenjiang Jin, Zhikai Lei, Jiaxing Li, Jingwen Li, Linyang Li, Shuaibin Li, Wei Li, Yining Li, Hongwei Liu, Jiangning Liu, Jiawei Hong, Kaiwen Liu, Kuikun Liu, Xiaoran Liu, Chengqi Lv, Haijun Lv, Kai Lv, Li Ma, Runyuan Ma, Zerun Ma, Wenchang Ning, Linke Ouyang, Jiantao Qiu, Yuan Qu, Fukai Shang, Yunfan Shao, Demin Song, Zifan Song, Zhihao Sui, Peng Sun, Yu Sun, Huanze Tang, Bin Wang, Guoteng Wang, Jiaqi Wang, Jiayu Wang, Rui Wang, Yudong Wang, Ziyi Wang, Xingjian Wei, Qizhen Weng, Fan Wu, Yingtong Xiong, Chao Xu, Ruiliang Xu, Hang Yan, Yirong Yan, Xiaogui Yang, Haochen Ye, Huaiyuan Ying, Jia Yu, Jing Yu, Yuhang Zang, Chuyu Zhang, Li Zhang, Pan Zhang, Peng Zhang, Ruijie Zhang, Shuo Zhang, Songyang Zhang, Wenjian Zhang, Wenwei Zhang, Xingcheng Zhang, Xinyue Zhang, Hui Zhao, Qian Zhao, Xiaomeng Zhao, Fengzhe Zhou, Zaida Zhou, Jingming Zhuo, Yicheng Zou, Xipeng Qiu, Yu Qiao, and Dahua Lin. 2024. Internlm2 technical report. *Preprint*, arXiv:2403.17297.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, and etc. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

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

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024a. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *ArXiv*, abs/2401.14196.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024b. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. In *Thirty-fifth Conference on Neural Information*

Processing Systems Datasets and Benchmarks Track (Round 2).

Jian Hu. 2025. Reinforce++: A simple and efficient approach for aligning large language models. *arXiv preprint arXiv:2501.03262*.

Jian Hu, Xibin Wu, Weixun Wang, Dehao Zhang, Yu Cao, OpenLLMAI Team, Netease Fuxi, AI Lab, and Alibaba Group. 2024. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *ArXiv*, abs/2405.11143.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Shanghaoran Quan, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024a. Qwen2.5-coder technical report. *ArXiv*, abs/2409.12186.

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

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. 2024a. T\" ulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*.

Nathan Lambert, Valentina Pyatkin, Jacob Daniel Morrison, Lester James Validad Miranda, Bill Yuchen Lin, Khyathi Raghavi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hanna Hajishirzi. 2024b. Rewardbench: Evaluating reward models for language modeling. *ArXiv*, abs/2403.13787.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *ArXiv*, abs/2207.01780.

Chris Yuhao Liu, Liang Zeng, Jiacai Liu, Rui Yan, Jujie He, Chaojie Wang, Shuicheng Yan, Yang Liu, and Yahui Zhou. 2024a. Skywork-reward: Bag of tricks for reward modeling in llms. *arXiv preprint arXiv:2410.18451*.

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*.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. 2024b. Evaluating language models for efficient code generation. In *First Conference on Language Modeling*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. 2022a. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022b. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cristian Cantón Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal

Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and P. Abbeel. 2015. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Jun-Mei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, abs/2402.03300.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Reinforcement learning from automatic feedback for high-quality unit test generation. *arXiv preprint arXiv:2310.02368*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024a. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. 2024b. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2025. The lessons of developing process reward models in mathematical reasoning. *arXiv preprint arXiv:2501.07301*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024a. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024b. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Appendix

A.1 Prompt Template

Table 7: Prompt Used for Converting Seed Code Dataset into LeetCode-style Questions and Test Cases

system:
You are an AI assistant that helps people with python coding tasks.

user:
You are the latest and best bot aimed at transforming some code snippet into a leetcode style question. You will be provided with a prompt for writing code, along with a reference program that answers the question. Please complete the following for me:
1. Come up with a leetcode style question which consists of a well-defined problem. The generated question should meet the following criteria:
   a. The question is clear and understandable, with enough details to describe what the input and output are.
   b. The question should be solvable by only implementing 1 function instead of multiple functions or a class. Therefore, please avoid questions which require complicated pipelines.
   c. The question itself should not require any access to external resource or database.
   d. Feel free to use part of the original question if necessary. Moreover, please do not ask for runtime and space complexity analysis or any test cases in your response.
2. Based on the modified question that you generated in part 1, you need to create around 20 test cases for this modified question. Each test case should be independent assert clauses. The parameters and expected output of each test case should all be constants, **without accessing any external resources**.

Here is the original question:
{instruction}

Here is the reference program that answers the question:
“‘python
{program}
“‘

Now give your modified question and generated test cases in the following json format:
{"question": ..., "tests":["assert ...", "assert ..."]}.