CVeDRL: An Efficient Code Verifier via Difficulty-aware Reinforcement Learning

Anonymous ACL submission

Abstract

Code verifiers play a critical role in post-verification for LLM-based code generation, yet existing supervised fine-tuning methods suffer from data scarcity, high failure rates, and poor inference efficiency. While reinforcement learning (RL) offers a promising alternative by optimizing models through execution-driven rewards without labeled supervision, our preliminary results show that naive RL with only functionality rewards fails to generate effective unit tests for difficult branches and samples. We first theoretically analyze showing that branch coverage, sample difficulty, syntactic and functional correctness can be jointly modeled as RL rewards, where optimizing these signals can improve the reliability of unit-test–based verification. Guided by this analysis, we design syntax- and functionality-aware rewards and further propose branch- and sample-difficulty–aware RL using exponential reward shaping and static analysis metrics. With this formulation, CVeDRL achieves state-of-the-art performance with only 0.6B parameters, outperforming GPT-4o by up to 17.55% in pass rate with over 20× faster inference than competitive baselines. [1]

1 Introduction

Recently, large language models (LLMs) have shown impressive capabilities in code generation (Achiam et al., 2023; Dubey et al., 2024; Jaech et al., 2024; Guo et al., 2025). Although LLMs are able to quickly generate code solutions, they struggle to produce correct code in a single attempt. Researchers investigate inference-time scaling methods to alleviate this difficulty, where LLMs first employ repeated sampling to output multiple code results, and then a code verifier selects the final best result (Lightman et al., 2024a; Brown et al., 2024). Consequently, the performance of the code verifier is the key to the success of the code generation task for large language models (Cobbe et al., 2021; Lightman et al., 2024a; Liu et al., 2025).

Code verifiers take the problem description and the candidate code solution as input and leverage LLMs to generate unit tests, which consist of reasonable input and corresponding output pairs. By executing candidate code solutions and their unit tests, the execution outcomes are examined from compilers and interpreters to identify the optimal code solutions among all candidates. In fact, code verifiers are specific LLMs for code generation, as unit tests are special code snippets. As a result, instruction supervised fine-tuning (SFT) can endow LLMs with unit test generation ability and make them qualified code verifiers (Ma et al., 2025). However, there are three challenges in the SFT strategy. First, large-scale high-quality SFT data is unavailable for unit test generation. Although Ma et al. (2025) presents an automatic data pipeline, a lot of incorrect unit tests exist in the final results. Second, the SFT-based model suffers from a high error rate and failure rate on generated unit tests. Besides, SFT-based code verifiers have to sample multiple unit tests to mitigate error and failure issues, causing a serious efficiency bottleneck.

Reinforcement learning (RL) has shown strong potential for LLM post-training by enabling reward-driven exploration (Jaech et al., 2024; Shao et al., 2024; Guo et al., 2025). It offers a promising framework for code verification, as it avoids reliance on large unit-test datasets, naturally incorporates metrics such as pass rate and code coverage as rewards, and improves inference efficiency by better exploiting compact models. However, existing RL formulations for code verification often rely on coarse-grained functionality rewards, which provide limited guidance for generating high-quality and discriminative unit tests. Such rewards fail to explicitly account for branch-level coverage, sam-

[1] Code is available at https://anonymous.4open.science/r/CVeDRL-DF1A/

ple difficulty, and syntactic validity, making it unclear when and how reinforcement learning can reliably improve verifier performance.

To address these limitations, we first derive a confidence bound for unit-test majority voting that characterizes how assertion reliability, branch coverage, and candidate diversity jointly affect verification confidence, providing principled guidance for reward design. Building on this analysis, we propose CVeDRL, an RL-trained code verifier at a compact 0.6B scale. CVeDRL employs a syntax-functionality composite reward with GRPO (Shao et al., 2024), augmented by branch-difficulty-aware exponential reward shaping and sample-difficulty-aware static analysis metrics to handle boundary branches and hard samples respectively.

We perform extensive experiments to verify CVeDRL across three datasets and four policy models. Experimental results demonstrate that CVeDRL-0.6B significantly improves the performance of open-source policy models and also enhances the effectiveness of closed-source models. Additionally, to assess the intrinsic quality of generated unit tests, we directly evaluate various metrics such as error rates, pass rates, and line coverage across three datasets. CVeDRL-0.6B consistently outperforms various baseline models in the task of unit test generation, achieving a considerably higher test-case pass rate on MBPP+ (yielding a 17.55% increase compared to GPT-4o-mini) with marginally higher coverage, and substantiating our theoretical analyses. Furthermore, CVeDRL-0.6B gains more than 20x inference efficiency improvement on token throughput, compared to the traditional SFT model CodeRM.

The contributions of our work are as follows:

- We theoretically analyze the interplay among test case pass rates, branch coverage, and the performance of code verifiers, showing the potential of RL for code verifier training
- We propose CVeDRL, a code verifier trained by RL with syntax and functionality rewards, with a novel branch-sample difficulty-aware mechanism incorporating exponential reward shaping and static analysis metrics.
- Extensive experiments demonstrate that CVeDRL-0.6B achieves sota performances on LLM code verification and unit test generation across six groups of main experiment overall and improves inference efficiency by more than 20x compared with SFT models.

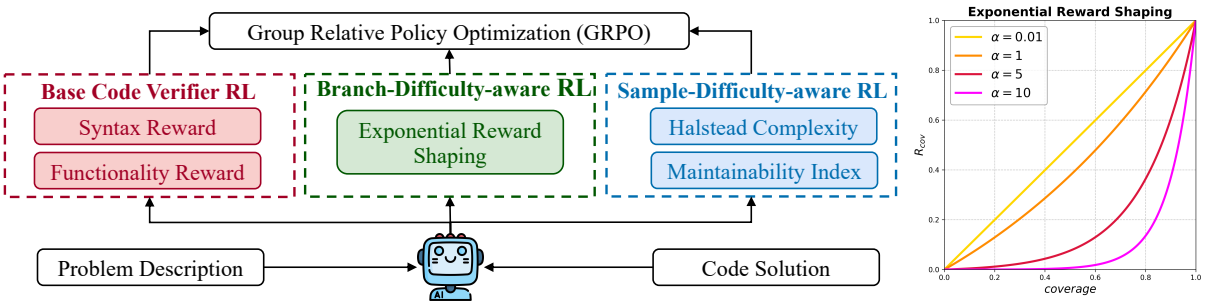
2 Related Work

Enhancement of code generation via unit testing. Enhancing code generation reliability with unit tests can be framed as a two-phase process: improving model training and refining inference. Prior work integrates unit test results into training to improve code accuracy (Liu et al., 2023a; Dou et al., 2024; Wang et al., 2025). At inference, MBR-EXEC (Shi et al., 2022a) applies Bayesian risk decoding to rerank candidates by pass rates, and CodeRM (Ma et al., 2025) leverages a distilled test-generator model for targeted solution selection. Building on these insights, our method combines test accuracy and line-coverage signals within a unified RL reward function, jointly improving test quality and enabling more reliable code selection.

Unit test generation. Traditional approaches employ symbolic analysis (Galeotti et al., 2013) and meta-heuristic search (Harman and Jones, 2001; McMinn, 2004; Harman et al., 2012) to craft tests. LLM-based methods fall into two categories: prompt engineering—decomposing test objectives into sub-questions (Wang et al., 2024), offering static metrics in prompts (Sepidband et al., 2025), or iteratively incorporating execution feedback (Pizzorno and Berger, 2025; Chen et al., 2024; Cheng et al., 2025)—and model fine-tuning (Ma et al., 2025; Eom et al., 2024). In this work, we propose a reinforcement learning-based strategy that integrates dynamic execution feedback with static code analysis to produce high-quality unit tests for code selection.

3 Method

In this section, we present CVeDRL, which has both performance and efficiency advantages at a scale of 0.6B. We begin by introducing the Unit Test Majority-Voting Framework and its associated reliability bound of test case quality. Then we introduce the base code verifier trained by syntax and functionality rewards for GRPO. To handle boundary conditions, we apply an exponential reward shaping mechanism that amplifies rewards for covering rare branches. Additionally, we integrate two static analysis metrics that provide priori code sample complexity assessments, refining the reward function with static insights.



(a) CVeDRL Framework Overview

(b) Exponential Reward Shaping

Figure 1: (a) We design syntax and functionality rewards and employ GRPO to train the base code verifier. Base model struggles to produce effective unit test cases for the difficult branches and samples. Therefore, we propose Branch-Difficulty-aware and Sample-Difficulty-aware reinforcement learning based on exponential reward shaping and static analysis metrics (Halstead Complexity and Maintainability Index). (b) Exponential reward shaping modifies the coverage reward function from a linear format into an exponential format.

3.1 Unit Test Majority-Voting Framework and RL Insights

The unit test majority-voting framework adopts the best-of-N decoding strategy, wherein an LLM policy model first generates N candidate programs for a given programming problem Q (Cobbe et al., 2021; Lightman et al., 2024b). Formally, we denote these candidates as $\{s_1, s_2, \ldots, s_N\}$. For functional correctness, an auxiliary LLM produces M unit tests for each (Q, s_i) pair, yielding test suites $\{T_1, T_2, \ldots, T_M\}$, where each test suite $T_j = \{(x_{j,1}, y_{j,1}), (x_{j,2}, y_{j,2}), \ldots, (x_{j,K_j}, y_{j,K_j})\}$ contains K_j input–output pairs. $x_{j,k}$ is the k-th input and $y_{j,k}$ is the expected output. Each candidate C_i is tested against all T_j, with binary outcomes

$$\mathrm{p}_{i,j} = \begin{cases} 1, & \text{if } s_i \text{ passes every case in } T_j, \\ 0, & \text{otherwise.} \end{cases}$$

These results form a reward vector $\mathbf{p}_i = (\mathrm{p}_{i,1}, \ldots, \mathrm{p}_{i,M})$ for each candidate. Finally, under the majority-voting criterion (Wang et al., 2023), we select the candidate that maximizes the total number of passed tests:

$$\mathrm{s}_{\mathrm{opt}} = \arg \max_{i \in \{1,\ldots,N\}} \sum_{j=1}^{M} \mathrm{p}_{i,j} \,.$$

To quantify how test-assertion reliability and branch coverage jointly influence the confidence in the selected program, we derive that the assertion correctness probability p must satisfy the bound

$$p \geq \frac{1 + \sqrt{\frac{2}{M} \ln\left(\frac{1-q}{1-q'} N\right)}}{1 + c} \,,$$

Here, p denotes the *assertion correctness probability*—the probability that a generated assertion is semantically correct, i.e., it accepts valid code and rejects bugs on the exercised branch. q is the prior probability that any individual candidate is functionally correct (before verification), while q' is the target post-selection probability of correctness under majority voting. N is the number of candidate programs, M the number of independent test suites per candidate, and $c \in [0, 1]$ the average branch coverage of the test suites.

Operational interpretation. The bound reveals that increasing the number of test suites M or branch coverage c relaxes the requirement on per-assertion reliability p, since broader execution and repeated voting both strengthen discrimination. Conversely, a larger candidate pool N or a higher target confidence q' demands higher p. The detailed proof is in Appendix A.

Importantly, the quantities appearing in the confidence bound correspond to statistics that can be directly optimized through reinforcement learning. Specifically, assertion correctness p and branch coverage c are reflected in the pass/fail rewards obtained from generated unit tests, while increasing test difficulty and syntactic validity affects the discriminative power of majority voting. Therefore, optimizing these signals via reinforcement learning improves the expected confidence of unit-test–based program selection, providing a theoretical justification for multi-dimensional reward design in code verifier training.

3.2 Base Code Verifier by RL

Previous studies (Xie et al., 2025; Zeng et al., 2025) demonstrate that the design of 'format-answer' reward effectively standardizes the format and ordering of model outputs, which guides the reasoning process of the model. Inspired by these studies, we propose the basic design of reinforcement fine-tuning framework within CVeDRL that dynamic testing feedback via two complementary rewards: Syntax Reward and Functionality Reward. Syntax reward enforces specific AST-derived formatting rules, and Functionality Reward is based on the testing execution results. This approach significantly mitigates the propensity of models to 'hack' evaluations, standardizing generated code format with dynamic test execution signals.

Syntax Reward. Inspired by (Guo et al., 2025), we reexamine the formatting and syntax requirements for the unit-testing task and designed a corresponding syntax-based reward. Specifically, the generated tests must be strictly enclosed within a final Python code block, and traversing the AST of this test code reveals at least one class inheriting from `unittest.TestCase`. Given that W_i is the ith concatenation response of correct code solution s, the format reward is calculated as follows:

$$r_{syn}(W_i|s) = \begin{cases} 1.0, & \text{if syntax is correct,} \\ -1.0, & \text{if syntax is incorrect.} \end{cases}$$

Functionality Reward. Once the syntax is validated, a regex-based extractor retrieves the correctly structured unit-test snippet u_i (i.e., the executable unit-test code extracted from the full model response W_i) from the response of model W_i. To tailor concepts from prior RL-based fine-tuning methods for reducing hallucination rates specifically, we refine these approaches by classifying actual testing execution outcomes to derive reward signals. During unit-test generation, models exhibit two primary types of hallucinations: (1) Errors, where wrong code or invalid inputs cause test execution to crash, i.e. both false positives and false negatives; and (2) Failures, where tests run successfully but incorrectly predict the output from source code s. For tests is passed, we introduce branch coverage rate for the reward. The execution-result reward is computed as follows:

$$r_{func}(u_i, s) = \begin{cases} -2.0, & \text{if } u_i \text{ is error towards } s \\ -1.5, & \text{if } u_i \text{ is failure towards } s \\ +cov(u_i, s), & \text{if } u_i \text{ is passed} \end{cases}$$

In light of GRPO (Shao et al., 2024), for question $q = q(s)$ with corresponding code solution given to LLM, the model samples a group of outputs $\{o_i\}_{i=1}^{G}$ from the old policy $\pi_{\theta_{\text{old}}}$ then optimize the policy with the following objective iteratively:

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

where $\text{clip}(x, \varepsilon) := min(max(x, 1-\varepsilon), 1+\varepsilon)$, π_θ is theaa optimal policy. $a_{o_i} = \frac{r_i - \mu_{r_i}}{\sigma_{r_i}}$ is the advantage term of ith output calculated by:

$$r_i = \begin{cases} r_{syn}(W_i|s), & \text{if syntax is incorrect,} \\ r_{syn}(W_i|s) + r_{func}(u_i, s), & \end{cases}$$

where μ_{r_i} and σ_{r_i} is the mean and standard deviation. We empirically evaluated several different reinforcement learning algorithms and found that the final performance was not sensitive to the choice of RL optimizer.

3.3 Branch-Difficulty-aware RL

The Base Code Verifier employs a linear coverage-based reward, which leads the model to favor generating happy-path test cases and to overlook many boundary conditions. We attribute this behavior to the diminishing reward at high coverage levels, resulting in insufficient exploration incentives. To encourage the model to focus more on boundary conditions and other atypical execution paths, we adopt an empirical yet efficient reward design with coverage awareness:

$$r_{cov}(u_i, s) = (e^{\alpha}-1)^{-1}[exp(\alpha \times cov(u_i, s)) - 1]$$

where $cov(u_i, s) \in [0, 1]$ is the coverage rate of unit-test case with a given code, and $\alpha > 0$ is a tunable hyperparameter controlling the tail-heaviness of the curve. r_{cov} remains nearly linear for small α but stays flat at low coverage and then rises sharply as α increases. By integrating $r_{cov}(u_i, s)$ into our reward signal, we empirically observe that the model devotes additional generation effort to hard-to-reach logic, resulting in more balanced and comprehensive unit-test suites.

Coverage and Assertion Discriminativeness. A natural concern is that high branch coverage does not inherently guarantee strong or discriminative assertions—the agent might maximize coverage by reaching rare branches while pairing them with trivial checks (e.g., asserting `result is not None`). We argue that the exponential shaping mechanism mitigates this risk. Trivial assertions primarily improve coverage on shallow, easily-reached paths, where the exponential reward has already saturated and yields diminishing marginal gain. To obtain substantial reward increments, the agent must produce tests that *genuinely expand branch-level execution* into deeper control flow, which typically requires input construction that exercises specific logic and meaningful output assertions to avoid execution errors or failures. Furthermore, the error and failure penalties in the functionality reward (Section 3.2) impose a complementary constraint: tests with syntactically valid but semantically incorrect assertions trigger failure penalties (-1.5), discouraging weak checks. Empirically, removing BDA causes the largest performance degradation across all policy models in our ablation (Table 3), confirming that exponential coverage shaping is the primary mechanism preventing reward exploitation through trivial assertions.

3.4 Sample-Difficulty-aware RL

A key limitation of relying solely on dynamic feedback, such as pass/fail signals or code coverage metrics, is that these measures only become available **after** the model has already generated and attempted to execute a candidate solution. Consequently, they offer no guidance in distinguishing between "trivial failures" (e.g., minor input mismatches or off-by-one errors) and genuinely challenging code fragments that demand deeper reasoning. To address this gap, we introduce a static difficulty prior that quantifies intrinsic complexity of each solution **before** any execution takes place. Specifically, for all code solution, we compute two complementary static metrics of each code during preprocessing: Halstead Complexity(D_H) and the difficulty of Maintainability(D_M).

We choose Halstead Complexity over embedding-based or semantic complexity measures for three reasons: (1) it is computed directly from the AST with negligible cost, avoiding extra model inference during RL training; (2) it is model-independent, preventing circular bias with the verifier being trained; and (3) it serves only as reward scaling without altering the correctness ordering of the functionality reward, thereby leaving the majority-voting bound unaffected. Halstead Complexity, by quantifying the variety and frequency of operators and operands, serves as a metric for the cognitive load required to parse and is widely utilized for code difficulty analysis (Hariprasad et al., 2017). The formulation is $\hat{D}_H = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$, where η_1 and η_2 denote the counts of distinct operators and operands in the source code, and N_2 is the total number of operand occurrences. This metric captures the cognitive burden imposed by syntactic diversity and operand repetition. To mitigate the influence of extreme outliers, we collect the set of all $\hat{D}$ values across our training corpus and determine the 95 percentile with a clipped min–max normalization $\hat{D}_{95}$. We then perform a clipped min–max normalization to get Halstead difficulty D_H:

$$D_H = min(\hat{D}, \hat{D}_{95})/\hat{D}_{95}.$$

Maintainability Index (MI) (Coleman et al., 1994) further captures the anticipated effort and risk of future modifications, which is a four-metric polynomial equation for measuring how maintainable the code is. Recent work (Zheng et al., 2024) on MI is applied to the study of code sample differentiation benchmark. The formulation is as follows: $\mathrm{MI} = \max\{0, 100 \times (171 - 5.2 \ln V - 0.23G - 16.2 \ln L + 50 \sin(\sqrt{2.4C}))/171\}$, where V denotes the Halstead Volume, which quantifies the size and information content of the code; G is the Cyclomatic Complexity; L represents lines of code and C is the percentage of comment lines. Since higher MI values indicate easier maintenance (lower difficulty), we invert and rescale it into a unified difficulty measure of maintainability:

$$D_M = \max\{0,\ 1 - \frac{MI}{100}\} \in [0, 1].$$

Low-maintainability code often deviates from the idiomatic patterns observed during pretraining, thereby undermining the transferability of learned representations and increasing hallucination rates. To emphasize the synergy between code comprehension and maintenance difficulty, we deviate a geometric mean of two metrics D_H and D_M:

$$D = \sqrt{D_H \times D_M} \in [0, 1],$$

where indicates the co-occurrence of high values in both dimensions, i.e. only code that is both hard

to comprehend and hard to maintain yields a large difficulty.

Motivated by the imperative to distinguish genuinely challenging code fragments from trivial failures prior to execution, we introduce the static difficulty D into our reward. Formally, we define the total augmented execution reward r_{func} with exponential reward shaping in coverage:

$$r_{func}(u_i, s) = \begin{cases} -2.0, & \text{if } u_i \text{ is error towards } s; \\ -1.0 \quad - (1-D), & \text{if } u_i \text{ is failure towards } s; \\ r_{cov}(u_i, s) \cdot (1+D), & \text{if } u_i \text{ is passed,} \end{cases}$$

where LLM receives larger positive feedback when the execution is passed with harder code functions, while failures incur softened penalties. This design ensures that the agent allocates a greater exploration effort and learning capacity to high-complexity regions of the code space, thereby improving overall policy robustness on complex unit-testing tasks with static difficulty awareness.

4 Experiments

4.1 Experimental Setup

Our experiment includes two components: Validation-Coder Performance and CVeDRL Test Quality evaluation. In the validation-coder evaluation, we employ unit-test reward signals to select among candidate code solutions produced by coder models. In the test quality evaluation, we execute all generated unit tests and report metrics for each model's output, providing a direct assessment of test-case quality (Table 2). We select a compact 0.6B parameter model for CVeDRL training to ensure it can serve as a sufficiently fast and efficient code verifier.

Datasets. We utilize four benchmarks to comprehensively evaluate unit-test generation: HUMANEVAL+ (Liu et al., 2023b), MBPP+ (Liu et al., 2023b), LIVECODEBENCH (Jain et al., 2024), and LEETCODE (greengerong, 2023).[2] Following the methodology of CodeRM (Ma et al., 2025), we select 168 function-style problems from LiveCodeBench created between January and September 2024. To assess generation performance in broader contexts, we filter 2,360 LeetCode problems following VALTEST (Taherkhani and Hemmati, 2024) to exclude system-design questions, class-based interfaces, and interactive I/O. We yield 542 problems each defined by a single Python function signature with a return value. During training, we use the CODERM(Ma et al., 2025) dataset, which contains over 50,000 questions selected through a dedicated processing pipeline, and ensures that all of the chosen large model solutions are both appropriate and correct.

Metrics. We report five metrics for test quality evaluation:(1) Pass Rate (PR): the proportion of tests that execute without assertion errors. (2) Failure Rate (FR): the probability that a generated test runs successfully but the model's expected output is incorrect. (3) Error Rate (ER): the probability that the testing doesn't gives any coverage or failure report due to errors. (4) Branch Coverage (BC): the ratio of executed branches or lines covered by the tests. (5) Assertion Number (AN): the number of assertions within the unittest class. For the evaluation of validation, we adopt the pass-of-n metric under a unit-test majority voting framework, where the final solution is selected based on the number of test cases it successfully passes.

Baseline Models. In the evaluation of Validation-Coder Performance, we select four policy models including GPT-3.5, GPT-4o-mini (Achiam et al., 2023), LLaMA3-70B (Dubey et al., 2024), and LLaMA3-8B (Dubey et al., 2024). For reward model inference, the baselines include CodeT (Chen et al., 2023), MBR-E (Shi et al., 2022b),and CodeRM-8B (Ma et al., 2025). We also evaluate LLaMA3-70B as a reward model to compare performance against larger unaligned models, and vanilla method with random selection of code solutions. For the evaluation of test quality, we employ GPT-4o (Achiam et al., 2023), GPT-3.5 (Brown et al., 2020), and LLaMA3-8B as baseline models.

4.2 Main Results

Validation-Coder Performance. Table 1 summarizes verification results across three benchmarks. Following the majority-voting scheme, each model's candidate solutions are verified by running generated test cases. CVeDRL-0.6B significantly improves best-of-n performance over the base Qwen3-0.6B on all benchmarks and yields greater gains than LLaMA-70B in nearly every setting, demonstrating that its generated unit tests more effectively discriminate correct solutions. For example, when selecting among GPT-4o-mini outputs on MBPP+, accuracy rises from 71.32% to

[2]We note that 51/164 HumanEval+ and 161/378 MBPP+ tasks contain only straight-line logic without conditional branches; for these cases we use line coverage as a more stable surrogate. The more complex LiveCodeBench and LeetCode benchmarks provide richer branching structures.

Method	Scale	Policy Model			
		Llama3 8B	Llama3 70B	GPT-3.5	GPT-4o-m
HumanEval+					
Vanilla	–	53.43 $\pm.95$	73.10 $\pm.72$	67.44 $\pm.81$	82.57 $\pm.53$
MBR-E	–	60.18 $\pm.52$	75.47 $\pm.41$	70.53 $\pm.48$	85.31 $\pm.35$
CodeT	–	65.22 $\pm.43$	76.14 $\pm.35$	73.92 $\pm.38$	85.52 $\pm.29$
Llama3.1	70B	71.95 $\pm.21$	78.41 $\pm.18$	**79.88** $\pm.19$	85.48 $\pm.16$
CodeRM	8.0B	72.13 $\pm.18$	78.66 $\pm.16$	78.13 $\pm.17$	86.49 $\pm.14$
Qwen3	0.6B	55.75 ±1.2	73.91 $\pm.68$	69.16 $\pm.74$	83.01 $\pm.51$
Qwen3	32B	70.64 $\pm.22$	78.52 $\pm.17$	78.45 $\pm.19$	86.58 $\pm.13$
CVeDRL	**0.6B**	**72.14** $\pm.04$	**78.72** $\pm.04$	78.96 $\pm.04$	**87.05** $\pm.02$
MBPP+					
Vanilla	–	49.17 $\pm.88$	69.28 $\pm.65$	70.15 $\pm.61$	71.32 $\pm.58$
MBR-E	–	49.98 $\pm.55$	69.75 $\pm.42$	70.49 $\pm.40$	72.12 $\pm.38$
CodeT	–	59.17 $\pm.41$	69.88 $\pm.34$	69.93 $\pm.33$	73.28 $\pm.30$
Llama3.1	70B	65.24 $\pm.20$	71.77 $\pm.18$	75.87 $\pm.16$	75.01 $\pm.17$
CodeRM	8.0B	66.63 $\pm.15$	72.53 $\pm.23$	75.98 $\pm.14$	75.18 $\pm.19$
Qwen3	0.6B	51.01 ±1.1	70.19 $\pm.62$	72.86 $\pm.58$	73.74 $\pm.55$
Qwen3	32B	65.43 $\pm.21$	72.61 $\pm.18$	75.71 $\pm.16$	75.04 $\pm.17$
CVeDRL	**0.6B**	**66.79** $\pm.05$	**73.60** $\pm.03$	**76.21** $\pm.04$	**76.93** $\pm.03$
LiveCodeBench					
Vanilla	–	11.74 ±1.2	25.19 $\pm.85$	20.48 $\pm.92$	34.90 $\pm.78$
MBR-E	–	12.12 $\pm.75$	25.18 $\pm.58$	20.51 $\pm.62$	34.79 $\pm.52$
CodeT	–	12.59 $\pm.62$	25.92 $\pm.48$	20.60 $\pm.51$	35.11 $\pm.45$
Llama3.1	70B	13.33 $\pm.28$	**28.39** $\pm.24$	22.79 $\pm.26$	38.61 $\pm.22$
CodeRM	8.0B	15.24 $\pm.25$	27.81 $\pm.22$	21.81 $\pm.24$	39.18 $\pm.20$
Qwen3	0.6B	12.52 ±1.4	25.47 $\pm.78$	21.91 $\pm.82$	37.18 $\pm.72$
Qwen3	32B	15.17 $\pm.26$	27.95 $\pm.21$	22.86 $\pm.23$	38.51 $\pm.20$
CVeDRL	**0.6B**	**16.75** $\pm.08$	27.96 $\pm.06$	**23.09** $\pm.14$	**39.31** $\pm.85$

Table 1: The result for code verification of CVeDRL and other baselines over three code generation benchmarks. We report mean $\pm$ std over 5 independent runs. The top two performances for each dataset and policy model are marked in **bold** and underlined.

LLM	Scale↓	ER%↓	FR%↓	PR%↑	BC%↑	AN↓
HumanEval+						
GPT-4o	–	1.98 $\pm.22$	17.21 $\pm.45$	80.81 $\pm.51$	96.91 $\pm.18$	5.35 $\pm.31$
GPT-3.5	–	3.14 $\pm.35$	26.32 $\pm.62$	70.54 $\pm.71$	96.73 $\pm.21$	4.13 $\pm.28$
LLaMA3.1	8.0B	10.88 $\pm.68$	37.19 $\pm.85$	51.93 $\pm.92$	94.60 $\pm.35$	3.97 $\pm.24$
CodeRM	8.0B	2.44 $\pm.31$	64.73 $\pm.78$	32.83 $\pm.82$	**96.97** $\pm.19$	7.15 $\pm.42$
Qwen3	0.6B	20.70 ±1.5	44.82 ±1.8	34.48 ±1.9	73.19 ±1.4	4.38 $\pm.45$
Qwen3	32B	9.43 $\pm.52$	25.31 $\pm.58$	65.26 $\pm.65$	89.53 $\pm.42$	8.17 $\pm.48$
CVeDRL	**0.6B**	**1.27** $\pm.09$	**12.79** $\pm.18$	**85.94** $\pm.21$	97.53 $\pm.08$	**2.41** $\pm.12$
MBPP+						
GPT-4o	–	3.98 $\pm.28$	29.89 $\pm.52$	66.13 $\pm.58$	96.91 $\pm.20$	6.12 $\pm.35$
GPT-3.5	–	5.14 $\pm.38$	40.15 $\pm.72$	54.71 $\pm.78$	96.65 $\pm.24$	5.97 $\pm.38$
LLaMA3.1	8.0B	15.79 $\pm.75$	47.53 $\pm.92$	36.68 $\pm.98$	95.93 $\pm.38$	4.13 $\pm.28$
CodeRM	8.0B	2.44 $\pm.28$	52.86 $\pm.82$	44.70 $\pm.85$	97.11 $\pm.18$	7.88 $\pm.45$
Qwen3	0.6B	28.18 ±1.6	31.53 ±1.5	40.29 ±1.7	90.12 ±1.2	3.48 $\pm.38$
Qwen3	32B	11.42 $\pm.55$	31.39 $\pm.62$	57.19 $\pm.68$	92.44 $\pm.45$	7.47 $\pm.42$
CVeDRL	**0.6B**	**0.53** $\pm.08$	**15.79** $\pm.21$	**83.68** $\pm.24$	**97.37** $\pm.09$	**3.13** $\pm.11$
LeetCode						
GPT-4o	–	**2.77** $\pm.25$	25.14 $\pm.55$	72.09 $\pm.62$	83.64 $\pm.45$	5.77 $\pm.34$
GPT-3.5	–	3.53 $\pm.32$	37.28 $\pm.75$	59.19 $\pm.82$	76.53 $\pm.58$	5.63 $\pm.35$
LLaMA3.1	8.0B	12.47 $\pm.72$	54.77 $\pm.95$	32.76 ±1.0	70.49 $\pm.75$	3.88 $\pm.25$
CodeRM	8.0B	3.70 $\pm.32$	58.16 $\pm.88$	38.14 $\pm.92$	75.37 $\pm.62$	6.43 $\pm.42$
Qwen3	0.6B	37.31 ±1.8	42.26 ±1.6	20.43 ±1.8	61.47 ±1.5	3.76 $\pm.38$
Qwen3	32B	23.18 $\pm.65$	27.63 $\pm.62$	49.19 $\pm.72$	68.62 $\pm.55$	7.34 $\pm.45$
CVeDRL	**0.6B**	3.49 $\pm.15$	**20.98** $\pm.28$	**75.53** $\pm.32$	**91.61** $\pm.15$	**2.84** $\pm.13$

Table 2: The direct testing results of CVeDRL and other baselines with metrics including Error Rate(ER), Failure Rate(FR), Pass Rate(PR), Branch Coverage(BC) and Assertion Number(AN) across three datasets. We report mean $\pm$ std over 5 independent runs. Top two performances are marked in **bold** and underlined.

76.93% (+5.61%). We report mean$\pm$std over 5 runs; gains consistently exceed variance (e.g., +1.75 on MBPP+ with std$\leq$0.05). Under a fixed coder, selection accuracy is bounded by candidate pool quality—this ceiling effect explains small absolute gains when the coder is already strong.

Test Quality of CVeDRL. Combining coverage reports with static code-analysis tools, we directly executed the generated tests and computed five evaluation metrics, the results of which are summarized in Table 2 for three benchmark datasets. CVeDRL maintains an exceptionally low failure rate and high line coverage, robustly improving test quality. On MBPP+, CVeDRL significantly outperforms every compared model, with the pass rate (PR) exceeding that of the next best model GPT-4o by 16%. Notably, CVeDRL also produces significantly fewer assertions, indicating that our method minimizes redundant test cases while sustaining coverage levels and thus further reduces testing overhead. This behavior arises because generating redundant cases increases assertion failure risk under the functionality reward, so CVeDRL preferentially generates the minimal set of assertions required to satisfy coverage criteria. Moreover, CVeDRL produces the highest-quality test suites on MBPP+ and correspondingly achieves the best verification performance (Table 1), confirming a tight link between coverage gain and solution selection (Appendix A).

4.3 Ablation Study

We ablate CVeDRL (Table 3) to isolate the effects of syntax reward (SYN), branch-difficulty-aware RL (BDA), and sample-difficulty-aware RL (SDA). For comparison of training methodologies, we also evaluate CodeRM-8B and CVeDRL-8B, trained on the same dataset with Llama3.1-8B but with different methods. Under a controlled setting with the same backbone, CVeDRL demonstrates a clear performance advantage over CodeRM. SDA alone

Model	Scale	SYN	BDA	SDA	Mbpp+					Humaneval+				
					ER%↓	FR%↓	PR%↑	BC%↑	AN↓	ER%↓	FR%↓	PR%↑	BC%↑	AN↓
CodeRM	8.0B	–	–	–	2.44	52.86	44.70	97.11	7.88	2.44	52.86	32.83	96.97	7.15
CVeDRL	8.0B	✓	✓	✓	**0.41**	16.31	83.28	**97.53**	4.96	1.48	13.83	85.69	**97.86**	3.79
	0.6B	×	×	×	14.32	34.15	51.53	85.11	3.23	9.86	50.15	41.99	92.41	3.17
	0.6B	✓	×	×	6.17	24.36	69.47	89.79	3.75	4.37	27.50	68.13	94.83	3.64
CVeDRL	0.6B	✓	×	✓	4.86	**15.18**	79.96	92.14	3.47	2.81	13.05	84.14	94.55	3.88
	0.6B	✓	✓	×	1.71	26.87	71.42	96.75	3.08	1.93	18.92	79.15	97.41	2.75
	0.6B	✓	✓	✓	0.53	15.79	**83.68**	97.37	**3.13**	**1.27**	**12.79**	**85.94**	97.53	**2.41**

Table 3: Ablation study on the MBPP+ benchmark to disentangle the contributions of the exponent-shaped exploration for syntax reward("SYN"), branch-difficulty aware RL("BDA") and the static analysis reward for sample-difficulty aware RL("SDA").

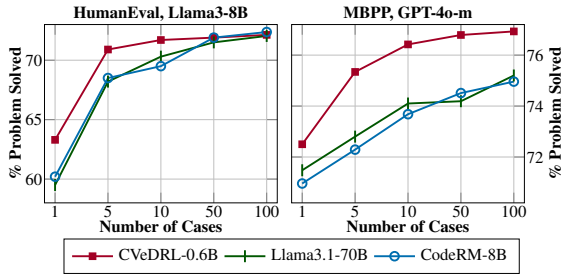


Figure 2: Performance of three unit-test generators at different test-case scales, with LLaMA3-8B on HumanEval+ and GPT-4o-mini on MBPP+ as policy model separately.

LLM	Scale	AMU↓ (GB)	AL↓ (s/iter)	TT↑ (tok/kPar/s)	AEC↓ (W)
CodeRM	8.0B	43.07	1.7823	0.296	293.6
CVeDRL	8.0B	43.13	0.9954	2.384	292.9
Qwen3	4.0B	41.70	4.1951	0.529	294.0
Qwen3	0.6B	40.97	1.3415	6.752	276.4
CVeDRL	**0.6B**	**40.97**	**0.6622**	**7.083**	**246.2**

Table 4: Inference efficiency comparison across models. For each model, we compare four key metrics: Average Memory Utilization (AMU), Average Latency (AL), Tokens Throughput (TT),and Average Energy Consumption (AEC). All results were obtained on the MBPP+ benchmark with 10 inference sampling per question.

substantially improves pass rate (from 69.47% to 79.96% on MBPP+) by steering the agent toward appropriately difficult tasks, while BDA alone significantly boosts code coverage to 96.75% by rewarding exploration of sparsely exercised branches. Combining both yields the highest overall performance (83.68% PR, 97.37% BC), demonstrating their complementary benefits.

4.4 Sampling Efficiency of CVeDRL

Figure 2 illustrates the performance of three unit-test generators under varying test-suite scales. CVeDRL-0.6B reaches its performance plateau with as few as 10 test cases in both settings, indicating minimal gains when sampling up to 100 cases under majority voting. When using LLaMA3-8B as policy model on HumanEval+, all three reward models perform comparably at scale 100, but CVeDRL-0.6B significantly outperforms baselines at scale 5. Similarly, with GPT-4o-mini on MBPP+, CVeDRL-0.6B surpasses other reward models at every scale. As derived from the confidence bound, fewer majority-voting candidates suffice, accelerating inference (Appendix A).

4.5 Inference Efficiency of CVeDRL

The results in Table 4 reveal clear scale-dependent trade-offs between resource utilization, latency, and throughput. Relative to the SFT-trained CodeRM-8B, CVeDRL-0.6B boosts token throughput from 0.296 to 7.083 tok/kPar/s—an improvement of more than 23×—while halving per-iteration latency. Meanwhile, the 8.0B model does not exhibit a clear performance advantage over the 0.6B model (Table 3). These results indicate that our RL approach is well suited for enabling compact models to perform verification efficiently, maintaining competitive accuracy while significantly reducing computational requirements.

5 Conclusion

We introduce CVeDRL, a unified reinforcement learning framework that leverages static code complexity metrics and test execution results to guide unit-test generation and code verification. We theoretically establish the relationship between test-case pass rates, branch coverage, and verification confidence. By integrating branch-sample difficulty-aware rewards, CVeDRL generates concise test suites that generalize across diverse branches, consistently outperforming strong baselines in both verification accuracy and test quality.

Limitation

Scope of validation. CVeDRL currently targets complete, standalone Python functions. Partial code (e.g., code completion snippets) cannot be directly validated by unit tests without a provisioned execution environment, and the generated tests are effective at filtering incorrect solutions but have limited ability to distinguish among multiple correct implementations. Future work may incorporate code mutation and partial-program execution to address these gaps.

Generalization. The implementation relies on Python-specific tooling (coverage instrumentation, AST parsing, unittest framework); extending to other languages requires corresponding tools. Additionally, while evaluation uses independent reference tests to reduce memorization risk, indirect overlap between training prompts and benchmarks cannot be fully ruled out.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

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*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and Sandhini Agarwal. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Alexander E. I. Brownlee, James Callan, Karine Even-Mendoza, Alina Geiger, Carol Hanna, Justyna Petke, Federica Sarro, and Dominik Sobania. 2023. *Enhancing Genetic Improvement Mutations Using Large Language Models*, page 153–159. Springer Nature Switzerland.

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*.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. *Preprint*, arXiv:2305.04764.

Xiang Cheng, Fan Sang, Yizhuo Zhai, Xiaokuan Zhang, and Taesoo Kim. 2025. Rug: Turbo Llm for Rust Unit Test Generation . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2983–2995, Los Alamitos, CA, USA. IEEE Computer Society.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. 1994. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. 2024. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *Preprint*, arXiv:2402.01391.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing javascript interpreters with coverage-guided reinforcement learning for llm-based mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1656–1668, New York, NY, USA. Association for Computing Machinery.

Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2013. Improving search-based test suite generation with dynamic symbolic execution. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369. IEEE.

greengerong. 2023. leetcode. Hugging Face Datasets.

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

T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. 2017. Software complexity analysis using halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pages 1109–1113.

Mark Harman and Bryan F. Jones. 2001. Search-based software engineering. *Information and Software Technology*, 43(14):833–839.

Mark Harman, Sahar A. Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1).

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

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*.

Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards understanding the effectiveness of large language models on directed test input generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1408–1420, New York, NY, USA. Association for Computing Machinery.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 919–931. IEEE Press.

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

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

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023a. Rltf: Reinforcement learning from unit test feedback. *Preprint*, arXiv:2307.04349.

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

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. 2025. Inference-time scaling for generalist reward modeling. *arXiv preprint arXiv:2504.02495*.

Zeyao Ma, Xiaokang Zhang, Jing Zhang, Jifan Yu, Sijia Luo, and Jie Tang. 2025. Dynamic scaling of unit tests for code reward modeling. *Preprint*, arXiv:2501.01054.

Phil McMinn. 2004. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156.

Juan Altmayer Pizzorno and Emery D. Berger. 2025. Coverup: Effective high coverage test generation for python. *Preprint*, arXiv:2403.16218.

Melika Sepidband, Hamed Taherkhani, Song Wang, and Hadi Hemmati. 2025. Enhancing llm-based code generation with complexity metrics: A feedback-driven approach. *Preprint*, arXiv:2505.23953.

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

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022a. 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.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022b. 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.

Hamed Taherkhani and Hadi Hemmati. 2024. Valtest: Automated validation of language model generated test cases. *Preprint*, arXiv:2411.08254.

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*.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2025. Co-evolving llm coder and unit tester via reinforcement learning. *Preprint*, arXiv:2506.03136.

Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. Hits: High-coverage llm-based unit test generation via method slicing. *Preprint*, arXiv:2408.11324.

Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. 2025. Logic-rl: Unleashing llm reasoning with rule-based reinforcement learning. *Preprint*, arXiv:2502.14768.

Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. 2025. Simplerl-zoo: Investigating and taming zero reinforcement learning for open base models in the wild. *Preprint*, arXiv:2503.18892.

Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond correctness: Benchmarking multi-dimensional code generation for large language models. *Preprint*, arXiv:2407.11470.

A Pass–Coverage Trade-off in Automated Code Verification

A.1 Single-Test Assumptions

In this subsection, we derive a concise relationship between the probability that a candidate program passes a test suite and the average branch coverage of the suite. We begin by introducing the notation for the single–test-case scenario, and then generalize our findings to the multi-test context via the mean coverage.

Let q be the prior probability that a candidate program is *correct* and p is the probability that a single *generated test* returns PASS on correct code, i.e., that the test's own assertion is valid. r_{ij} is the probability that the test suite covers the j-th defective path (branch) of candidate i, where n_i denotes the number of defective paths in candidate i. c means branch coverage, typically approximated by $c = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n_i} r_{ij}$; $0 \le c \le 1$. Then we come up with three assumptions as follows:

Model Assumptions. We employ the following simplifying assumptions:

(a) **Correct code passes all exercised paths.** That is, if the test exercises any path of correct code, it will *pass* with probability p, and with probability $1-p$ it fails due solely to a faulty assertion (a false negative).

(b) **Erroneous code has at least one faulty path.** If the test exercises one of these faulty paths *and* its assertion is valid, the error is revealed (the test fails as intended); otherwise the test yields a *false positive*.

(c) **Suite-level passing criterion.** A program is declared to *pass* the entire test suite and thus be emitted as a candidate if and only if it passes every individual test (the simplified `pass_of_n` rule).

Single-Test Pass Probabilities. Under these assumptions, let C be the event "program is correct". Then for a single test we have

$$P(\text{pass} \mid C) \;=\; p, \qquad P(\text{pass} \mid \bar{C}) \;=\; 1 - p\,r_i,$$

where

$$r_i \;=\; \sum_{j=1}^{n_i} r_{ij}$$

is the total probability that the test covers any one of the defective paths in candidate i. In practice, one often replaces R_i by the average coverage s, yielding the approximation

$$P(\text{pass} \mid \bar{C}) \;\approx\; 1 - p\,c.$$

Posterior Probability of Correctness. Given that a candidate program has *passed* the test, we apply Bayes' theorem to update our belief:

$$\begin{aligned} P(C \mid \text{pass}) &= \frac{P(\text{pass} \mid C)\,P(C)}{P(\text{pass} \mid C)\,P(C) + P(\text{pass} \mid \bar{C})\,P(\bar{C})} \\ &= \frac{q\,p}{q\,p + (1-q)\big(1 - pr_i\big)}. \end{aligned}$$

Threshold for Improved Posterior. We are particularly interested in the condition under which the posterior probability of correctness exceeds the prior:

$$P(C \mid \text{pass}) > q \quad \Longrightarrow \quad p > \frac{1}{1 + r_i}.$$

Substituting the mean coverage approximation $R_i \approx s$ yields the practical threshold

$$p \;>\; \frac{1}{1+c}.$$

In other words, provided that the single-test reliability p exceeds the reciprocal of one plus the average coverage, passing the test suite will *increase* our confidence that the program is indeed correct. This completes the derivation of the basic formula relating pass probability and coverage under our simplified single-test assumptions. Extensions to multiple independent tests follow by replacing p and r_i with their compounded quantities across the suite.

A.2 Majority Voting Framework

In this section, we extend our analysis to a *majority-voting* scheme. Let N be the number of candidate programs (*best-of-N*) and M be the number of independently generated test suites per candidate. K denotes the number of I/O pairs in each suite, i.e. the number of assumptions across suites.

Per-Suite Pass Probability. When each suite contains K I/O pairs, all assertions must pass for the suite to be considered successful. Hence

$$\alpha_c = p^K \qquad \text{for a correct program,}$$
$$\alpha_w = (1-pc)^K \qquad \text{for an incorrect program.}$$

where $1-p\,c$ is the probability that either the defect path is not covered or the assertion fails to detect the error. We simplify our model in $K=1$.

Binomial Model for Passing Suites. For any fixed candidate, the number of suites that pass follows a binomial distribution:

$$\mathrm{p} \sim \mathrm{Binomial}\big(M, \alpha\big)$$

where

$$\alpha = \begin{cases} \alpha_c, & \text{if the program is correct,} \\ \alpha_w, & \text{if the program is incorrect.} \end{cases}$$

Under majority voting, we select the candidate with

$$\mathrm{s}_{\mathrm{opt}} = \arg\max_{i=1,\dots,N} \mathrm{p}_i.$$

Reliability via Concentration Bounds. To ensure that the probability of selecting a wrong program is at most δ, we require

$$P\big(\mathrm{s}_{\mathrm{opt}} \text{ is correct}\big) \geq 1-\delta.$$

We now show how this follows from a Hoeffding-type concentration inequality applied to the difference in suite-pass counts between the true correct program and any incorrect one.

For each test index $t \in \{1,\dots,M\}$ and each wrong candidate $j \in \{1,\dots,WN\}$ where the wrong proportion of N code solutions is W, define the indicator variables

$$\mathrm{x}_t = \mathbf{1}_{\text{test t passes on the correct solution}},$$
$$\mathrm{y}_{j,t} = \mathbf{1}_{\text{test t passes on wrong solution j}}.$$

By construction these are independent and identically distributed (i.i.d.), with

$$P(\mathrm{x}_t = 1) = \alpha_c, \qquad P(\mathrm{y}_{j,t} = 1) = \alpha_w,$$

where α_c and α_w are the per-suite pass probabilities for correct and incorrect code, respectively.

Let

$$\mathrm{g}_c = \sum_{t=1}^{M} \mathrm{x}_t, \qquad \mathrm{g}_j = \sum_{t=1}^{M} \mathrm{y}_{j,t},$$

$$\mathrm{d}_j = \mathrm{g}_c - \mathrm{g}_j = \sum_{t=1}^{M} \big(\mathrm{x}_t - \mathrm{y}_{j,t}\big) = \sum_{t=1}^{M} \mathrm{z}_{j,t}$$

where each summand $\mathrm{z}_{j,t} = \mathrm{x}_t - \mathrm{y}_{j,t}$ satisfies $\mathrm{z}_{j,t} \in [-1,1]$. The expectation of the gap is

$$\mathbb{E}[\mathrm{d}_j] = \sum_{t=1}^{M} \mathbb{E}[\mathrm{z}_{j,t}] = M\big(\alpha_c - \alpha_w\big) = M\Delta.$$

By the one-sided Hoeffding bound for bounded independent random variables,

$$\begin{aligned} P(\mathrm{d}_j \leq 0) &= P(\mathrm{d}_j - \mathbb{E}[\mathrm{d}_j] \leq -M\Delta) \\ &\leq \exp\Big(-\frac{2(M\Delta)^2}{\sum_{t=1}^{M}(1-(-1))^2}\Big) \\ &= \exp\Big(-\frac{M\Delta^2}{2}\Big) =: \beta. \end{aligned}$$

Hence for each wrong candidate j, the probability that it ties or outperforms the correct program is at most β.

Then we calculate the union bound over all wrong candidates. Let $\mathcal{E}$ be the event that *any* of the WN wrong programs achieves $\mathrm{s}_j \geq \mathrm{s}_c$, which indicates s_{opt} is incorrect. By the union bound,

$$P(\mathcal{E}) \;\leq\; WN\,\beta \;=\; WN\,\exp\Big(-\frac{M\Delta^2}{2}\Big).$$

Imposing the reliability target $P(\mathcal{E}) \leq \delta$ gives

$$WN\,\exp\Big(-\frac{M\Delta^2}{2}\Big) \leq \delta \implies \Delta \geq \sqrt{\frac{2\ln\frac{WN}{\delta}}{M}}$$

Recalling that $\Delta = \alpha_c - \alpha_w$ and $P\big(\mathrm{s}_{\mathrm{opt}} \text{ is correct}\big) \geq 1-\delta$, we obtain the required "safety margin"

$$\alpha_c - \alpha_w \;\geq\; \sqrt{\frac{\ln\frac{WN}{\delta}}{2M}}$$

Operational bound for $K=1$. In the single-case suite model ($K=1$), we have $\alpha_c = p$ and $\alpha_w = 1-p\,c$, hence

$$\Delta = p - (1-p\,c) = p(1+c) - 1.$$

Then we have

$$p(1+c) - 1 \;\geq\; \sqrt{\frac{\ln\frac{WN}{\delta}}{2M}}.$$

Substitution yields the practical requirement

$$p \;\geq\; \frac{1+\sqrt{\frac{2}{M}\ln\big(\frac{WN}{\delta}\big)}}{1+c}.$$

Finally, if we denote by $q' = 1 - \delta$ the overall probability of correct selection with majority voting framework and replace the fraction of wrong solutions W by its expectation $E[W] = 1 - q$, the bound becomes

$$p \geq \frac{1 + \sqrt{\frac{2}{M} \ln\left(\frac{1-q}{1-q'} N\right)}}{1 + c} \qquad (\star)$$

Inference setting with majority-voting framework and confidence bound. As an example, we show how the confidence bound quantifies the trade-off between test quality and the reduction in required test suites M per candidate when selecting among multiple program proposals. The numerical entries in Table 5 (q, q', c, p) is obtained from the experimental tables, where the prior q is the baseline ("Vanilla") correctness rate measured for the policy column, and the posterior target q' is the improved correctness rate reported under CVeDRL (the "CVeDRL" row) from Table 1. The coverage parameter c was taken from the verifier branch coverage statistics (BC) in Table 2 and used as a proxy for average branch coverage (hence $c \approx 0.96$ for HumanEval+ and $c \approx 0.97$ for MBPP+). The per-assertion reliability p was treated as the variable in the sensitivity grid reported in Table 5. The operational bound $(\star)$ indicates that

$$M \approx \frac{2 \ln\left(\frac{1-q}{1-q'} N\right)}{\left((1+c)p - 1\right)^2}.$$

Improving the verifier (as CVeDRL does) reduces the minimal required number of independent suites M through two channels: (i) raising the effective assertion reliability p (signalled by higher PR and lower ER/FR) and (ii) increasing measured coverage c (higher BC). Both actions enlarge the denominator $((1+c)p - 1)^2$ and thus shrink M (the dependence is approximately quadratic in the effective margin). However, the inequality also highlights a countervailing effect: specifying a higher target post-selection confidence q' increases the logarithmic numerator $\ln\left(\frac{1-q}{1-q'} N\right)$, and therefore raises the required M. Fundamentally, we set $N = M = 100$ during inference time. In this case, the number of samples M required for CVeDRL to attain the target correctness rate q' is substantially less than 100. Notably, CVeDRL nearly reaches the desired performance at $M \approx 10$ as illustrated in Figure 2, which clearly surpasses the capability of other code verifiers. As shown in Table 5, CVeDRL achieves a pass rate close to 85%, with the required number of majority-voting candidates being around 20. This indicates that the target performance can be reached when $M \approx 20$, which is more than three times fewer samples than those required by a verifier with a unit test accuracy of $p = 0.70$ for both dataset.

A.3 Conclusion

In summary, our analysis reveals that the interplay between test-assertion reliability p and average branch coverage s fundamentally determines the posterior confidence in a candidate program's correctness. A single test suffices to improve confidence only when $p > 1/(1+c)$. With majority voting framework across M independent suites and N candidates, the pass rate of unit-test generated by LLMs requires a stronger condition $(\star)$. These bounds quantify explicit trade-offs. More test suites or higher coverage can compensate for imperfect assertions, whereas larger candidate pools or stricter error tolerances demand more reliable tests. This provides principled guidelines for designing automated testing pipelines that balance resource expenditure against desired selection reliability. Specifically, the analysis is utilized for choosing the crucial hyperparameter M which is relative to acceleration.

B Additional Discussions

B.1 Impact of Ancillary LLM Components

To investigate how the choice of base model influences the effectiveness of the CVeDRL training method, we include Qwen2.5-Coder-0.5B as a benchmark against Qwen3. Additionally, we perform an ablation study by disabling Qwen3's chain-of-thought capability using the `/no_think` tag. Finally, motivated by prior findings that large LLMs struggle with static code analysis, we augment the prompts with conditional branch information to assess whether this auxiliary context yields further performance gains.

The ablation study in Table 6 indicates that prompting primarily guides the model toward correct solutions with fewer attempts, while chain-of-thought (CoT) encourages broader exploration at the expense of efficiency. In the smaller Qwen2.5 model, prompts reduce the average number of trials but slightly lower overall pass rate, whereas disabling prompts yields higher pass rate with more attempts. Although Qwen2.5 shows reduced pass rate and

Table 5: Estimated minimal number of independent test suites per candidate M (rounded) for $N = 100$ with branch coverage c of CVeDRL in Table 1, varying assertion reliability p. As an illustrative example, we adopt CVeDRL as the verifier and GPT-4o-m as the code generator, which is demonstrated in Table 2.

Dataset	$p = 0.70$	$p = 0.80$	$p = 0.85$
HumanEval+ ($q = 0.8257 \rightarrow q' = 0.8705$, $c = 0.96$)	71	30	21
MBPP+ ($q = 0.7132 \rightarrow q' = 0.7693$, $c = 0.97$)	67	29	20

coverage relative to Qwen3, it requires fewer generated assertions and nonetheless significantly outperforms GPT-4o, highlighting the superiority of the CVeDRL training methodology. The ablation results show that introducing chain-of-thought (CoT) consistently harms overall performance, lowering both pass rate and coverage. Moreover, adding conditional prompts has no appreciable effect on branch coverage but nonetheless reduces the model's success rate. These findings indicate that while CoT and extra prompt constraints aim to guide reasoning, they in fact impede efficiency without delivering coverage benefits.

B.2 Hyperparameter Sensitivity Analysis

Our reward formulation introduces two key hyperparameters: the exponential coverage coefficient α (Section 3.3) and the sample-difficulty scaling factor λ (Section 3.4). We evaluate their sensitivity with 5 independent runs.

Sensitivity of α (BDA). Table 7 reports results on MBPP+ under varying α values. Setting $\alpha > 0$ yields a clear improvement over the linear baseline ($\alpha = 0$). Performance remains competitive across $\alpha \in \{1, 2, 5, 10\}$, with $\alpha = 2$ providing the best overall balance between pass rate and branch coverage. We observe that $\alpha = 10$ introduces slight instability in error rate compared to $\alpha \in \{1, 2, 5\}$, suggesting that overly aggressive exponential shaping may occasionally over-penalize partially-covering tests.

Sensitivity of λ (SDA). In the default formulation, the difficulty term is applied with implicit coefficient $\lambda = 1$. We introduce an explicit scaling λ to measure sensitivity: $r^{\lambda}_{func}(u_i, s) = r_{func}(u_i, s) \cdot (1 + \lambda \cdot D)$ for passed tests, and softened failure penalty $-1.0 - (1 - \lambda \cdot D)$ for failures. Table 8 reports results on HumanEval+.

These results confirm that both BDA and SDA are necessary components, and that performance is robust to the exact hyperparameter values within reasonable ranges.

B.3 Comparison with Frontier Models

To contextualize CVeDRL's performance relative to state-of-the-art proprietary models, we evaluate DeepSeek-V3, Qwen3-Max, GPT-5.2-Codex, and Gemini-3-Pro under identical prompts and execution harness (3 independent experiments). Table 9 reports results on HumanEval+ and MBPP+.

Although frontier models such as GPT-5.2-Codex achieve higher pass rates, CVeDRL-0.6B attains comparable branch coverage ($\sim$97%) with competitive error rates despite a scale gap of over 1000$\times$. Importantly, CVeDRL operates at negligible inference cost, supports large-scale sampling during training, and can be directly embedded into reinforcement-learning pipelines as an execution-based reward module—making it particularly suitable for iterative coder–verifier co-optimization where repeatedly invoking proprietary APIs would be impractical.

B.4 Case Study: Success and Failure Modes

To provide qualitative insight into when and why CVeDRL succeeds or fails, we present representative examples from HumanEval+.

Success Case: Boundary Detection. Consider the function `count_up_to(n)` which returns all primes less than n. The base model (Qwen3-0.6B without RL) generates only happy-path tests such as `assertEqual(count_up_to(10), [2,3,5,7])`, leaving boundary conditions unexplored. CVeDRL generates the following test suite:

```
class TestCountUpTo(unittest.TestCase):
   def test_zero(self):
      self.assertEqual(count_up_to(0), [])
   def test_one(self):
      self.assertEqual(count_up_to(1), [])
   def test_two(self):
      self.assertEqual(count_up_to(2), [])
   def test_prime_boundary(self):
      self.assertEqual(count_up_to(3), [2])
   def test_large(self):
      self.assertEqual(count_up_to(20),
         [2, 3, 5, 7, 11, 13, 17, 19])
```

CVeDRL successfully targets boundary inputs ($n = 0, 1, 2, 3$) that exercise the early-return branches and the primality-check loop boundary.

Table 6: Ablation of Prompt and CoT on CVeDRL Validation-Coder Performance in MBPP+ dataset

Base	Prompt	CoT	ER%	FR%	PR%	BC%	AN
Qwen2.5	✓	\	12.13	16.19	71.68	96.79	**2.09**
Qwen2.5	×	\	9.64	14.93	75.43	96.86	2.53
Qwen3	×	×	**0.53**	15.79	**83.68**	97.37	3.13
Qwen3	×	✓	0.94	18.69	80.37	96.15	3.47
Qwen3	✓	×	2.41	17.61	79.98	**97.21**	3.13
Qwen3	✓	✓	4.74	**15.55**	79.71	96.54	3.64
GPT-4o	\	\	8.71	25.16	66.13	96.91	6.12

α	ER%↓	FR%↓	PR%↑	BC%↑	AN↓
0 (w/o BDA)	4.86	15.18	79.96	92.14	3.47
1	1.16	14.53	84.31	95.41	3.17
2 (selected)	0.53	15.79	83.68	97.37	3.13
5	0.27	16.03	83.70	97.49	3.21
10	0.82	15.44	83.74	97.64	2.94

Table 7: Sensitivity of the exponential coverage coefficient α on MBPP+ (5 runs). $\alpha = 2$ is selected as the default.

λ	ER%↓	FR%↓	PR%↑	BC%↑	AN↓
0 (w/o SDA)	1.93	18.92	79.15	97.41	2.75
1 (selected)	1.27	12.79	85.94	97.53	2.41
5	1.24	12.98	85.76	97.51	2.47

Table 8: Sensitivity of the difficulty scaling λ on HumanEval+ (5 runs). Removing SDA ($\lambda = 0$) degrades PR and increases FR, while $\lambda = 1$ and $\lambda = 5$ yield nearly identical results.

Model	ER%↓	FR%↓	PR%↑	BC%↑	AN↓	Cost
HumanEval+						
DeepSeek-V3	6.58	18.12	75.30	98.17	4.70	$3.91
Qwen3-Max	4.41	15.76	80.19	93.16	1.65	$2.17
CVeDRL-0.6B	1.27	12.79	85.94	97.53	2.41	–
GPT-5.2-Codex	0.61	5.83	93.56	97.39	1.34	$14.81
Gemini-3-Pro	0.61	9.75	90.24	95.31	1.68	$8.79
MBPP+						
DeepSeek-V3	7.41	26.66	65.93	96.10	3.15	$4.52
Qwen3-Max	3.56	20.24	76.20	95.96	1.79	$2.98
CVeDRL-0.6B	0.53	15.79	83.68	97.37	2.84	–
GPT-5.2-Codex	0.26	5.37	94.37	95.95	1.17	$20.27
Gemini-3-Pro	1.85	10.34	87.81	94.79	1.09	$10.31

Table 9: Comparison with frontier models on test quality metrics. Cost denotes per-benchmark API expense.

The exponential coverage reward drives the agent to cover these rare branches rather than generating redundant tests for the happy path.

Failure Case: Complex Algorithmic Logic. For the function `solve(N)` which computes the binary representation of the digit sum of N, CVeDRL generates tests that achieve high branch coverage but with imprecise assertions:

```
class TestSolve(unittest.TestCase):
   def test_zero(self):
      self.assertEqual(solve(0), "0")
   def test_single_digit(self):
      self.assertEqual(solve(5), "101")
   def test_multi_digit(self):
      # Incorrect: solve(123) = bin(1+2+3) = "110"
      # CVeDRL generates: "1001" (wrong assertion)
      self.assertEqual(solve(123), "1001")
```

Here CVeDRL correctly identifies that multi-digit inputs exercise additional branches (digit extraction loop), but the assertion for `solve(123)` is semantically incorrect—it confuses the digit sum (6="110") with another computation. This "correct structure, wrong value" pattern accounts for most of the remaining failure-rate errors. The functionality reward penalizes such cases with -1.5, but when the code solution itself has subtle logic, the model may still hallucinate expected outputs.

Discussion. These cases illustrate that BDA reward shaping effectively drives *structural* exploration of control flow, producing tests that cover boundary branches. The primary remaining weakness is assertion-value hallucination on algorithmically complex tasks, where the model must internally simulate program execution to predict correct outputs. Integrating execution-trace feedback or chain-of-thought reasoning into the generation pipeline may address this limitation in future work.

C Training Configuration

Framework and Algorithm. We fine-tune the Qwen3-0.6B and Qwen2.5-Coder-0.5B checkpoint with the `verl` RL library using *Group Relative Policy Optimization* (GRPO). `verl` is an open-source reinforcement-learning framework designed for post-training fine-tuning of large language models, providing a hybrid single- and multi-controller programming model for scalable PPO and GRPO workflows. It features modular APIs that decou-

ple computation and data dependencies and integrates seamlessly with PyTorch FSDP, Megatron-LM, vLLM, and other LLM infrastructures for efficient, production-ready deployments. GRPO eliminates the separate value network and updates the policy by comparing each sampled trajectory to the within-group reward baseline, thereby reducing both memory footprint and wall-clock cost.

Dataset. All prompts are taken from the publicly-available CodeRM-UnitTest corpus, which provides 17,600 training items and 59,600 held-out test items. The CodeRM-UnitTest dataset is a curated collection of over 77 000 synthetic Python unit tests, derived from CodeFeedback-Filtered-Instruction and TACO, and provided in Parquet format for training test-guided code-reward models. It serves as the primary training and evaluation corpus for lightweight unit-test generator models like CodeRM-8B, enabling rigorous performance benchmarks under real execution feedback. Because every roll-out requires real execution of unit tests, we randomly subsample 3,000 test cases for validation to keep evaluation under two hours per checkpoint. Experiments ran on 2 × NVIDIA A100 40 GB GPUs (FP16) with total training time of roughly 48 hours. Table 10 summarises the hyper-parameters that most influence optimization and compute. All other settings follow the official `verl` GRPO recipe.

Parameter	Value
Learning rate	1×10^{-6}
Global prompt batch size	32
Rollouts per prompt	2
Max prompt / response length	6 150 / 2 048 tokens
Mini-batch / micro-batch size	16 / 8
KL loss coefficient	0.001 (low-variance)
Entropy coefficient	0 (disabled)
GRPO clip ratio	0.2 (default)
Total epochs	1 000
Gradient checkpointing	enabled
FSDP offload	disabled

Table 10: Salient hyper-parameters used in GRPO fine-tuning.

D Experimental Details

Policy models. We evaluate four instruction-tuned policy models of different capacity and provider type: *Llama3-8B-Instruct*, *Llama3-70B-Instruct*, *GPT-3.5-turbo*, and *GPT-4o-mini*. Each model produces at most 100 candidate code solutions per prompt. Decoding and verification are run on 4 × NVIDIA A100-40 GB GPUs.

Baselines. We consider three verification-oriented baselines that exploit unit-test feedback to discriminate among candidate programs.

- Vanilla : the top-1 sample of the policy model without any reranking.
- MBR-E : minimum-Bayes-risk decoding that ranks candidates by the empirical risk computed from the execution outcomes of LLM-generated test cases (we use the "hard-loss" variant).
- CodeT : dual-execution agreement that measures both (i) consistency between candidate programs and their generated tests and (ii) cross-candidate concordance.

Besides the three test-driven methods discussed above, we measure performance against four capacity-oriented baselines: an 8B reward model CodeRM-8B for score-based reranking, a strong 70B code LLM Llama3-70B-Instruct, the supervised-fine-tuned backbone Qwen3-0.6B (Base), and our model CVeDRL-0.6B.

- CodeRM-8B : an 8B-parameter reward model that assigns a scalar quality score to each candidate; we select the highest-scoring solution.
- Llama3.1-70B : a strong open-source coder whose single best sample is taken as a standalone baseline.
- Qwen3-0.6B (Base) : the supervised-fine-tuned version of our backbone model, without reinforcement learning.
- CVeDRL-0.6B : our method, trained with GRPO on weighted data (see Appendix C).

Datasets. To obtain a representative view of unit-test generation, we aggregate four publicly available benchmarks: HumanEval+ and MBPP+, LiveCodeBench from Jan to Sep 2024, and the algorithm-oriented subset of LeetCode. Together, they cover both synthetic interview-style problems and organically authored, in-the-wild code, furnishing a diverse test bed for large-language-model (LLM) evaluation.

- LivecodeBench : Following the CodeRM protocol, we retain the 168 function-style tasks released between January – September 2024, because these newer problems have empirically proved the most challenging for current LLMs.

- LeetCode : Starting from 2,360 publicly accessible problems, we apply the VALTEST filtering rules to discard system-design, class-interface, and interactive I/O questions. The remaining 542 tasks each expose one Python function signature with a deterministic return value, enabling uniform test-harness construction.

For every benchmark we normalise signatures, strip extraneous boilerplate, and compile tests into a unified execution harness so that success rate, failure rate, branch coverage, and Pass@N can be measured consistently across datasets.

Experimental Rationale. HUMANEVAL+ and MBPP+ are retained in both the CVeDRL Test Quality and Validation-Coder Performance studies in main result because their moderate task counts, authoritative reference solutions, and fine-grained coverage tooling permit reliable intrinsic scoring while keeping the compute requirements of iterative Pass@N evaluation tractable. The filtered LEETCODE subset is confined to the Test Quality analysis. Its larger problem set and costly edge-case generators provide valuable stress-testing for success, error, and coverage metrics, but render exhaustive validation-coder search computationally infeasible. Conversely, LIVECODEBENCH appears only in the Validation-Coder study. Its developer-written tests supply a strong external oracle for synthesis evaluation, yet their heterogeneity prevents fair aggregation with coverage-based quality metrics.
Due to the constraints of the double-blind review policy, the model weights are not publicly released at this stage.

D.1 Training Pipeline Illustration

To concretize the reward computation described in Section 3, we present a representative training instance drawn from the CodeRM-UnitTest corpus. The target function `count_carry_operations(a, b)` accepts two numeric strings and returns the number of carry operations incurred during right-to-left digit-wise addition. This example exhibits a conditional branch within a loop, making it suitable for demonstrating both branch-difficulty and sample-difficulty reward mechanisms.

Stage 1: Prompt Formulation. Each training instance is presented to the verifier model as a structured prompt comprising the natural-language problem specification and the corresponding reference solution:

```
Below is a question and its 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
Subodhians are taught to add multi-digit numbers
     from
right-to-left one digit at a time. Many find the
"carry" operation - in which 1 is carried from one
digit position to be added to the next - to be a
significant challenge. Your job is to count the
     number
of carry operations for each addition problem.

### code solution
def count_carry_operations(a: str, b: str) -> int:
   max_len = max(len(a), len(b))
   a = a.zfill(max_len)
   b = b.zfill(max_len)
   carry_count = 0
   carry = 0
   for i in range(max_len - 1, -1, -1):
      digit_sum = int(a[i]) + int(b[i]) + carry
      if digit_sum >= 10:
         carry = 1
         carry_count += 1
      else:
         carry = 0
   return carry_count
```

Stage 2: Policy Rollout. The verifier policy π_θ generates a unit-test class conditioned on the prompt. The following output constitutes one sampled trajectory o_i:

```
import unittest

class TestCountCarryOperations(unittest.TestCase):
   def test_no_carry(self):
      self.assertEqual(
         count_carry_operations("123", "456"), 0)
   def test_all_carry(self):
      self.assertEqual(
         count_carry_operations("555", "555"), 3)
   def test_partial_carry(self):
      self.assertEqual(
         count_carry_operations("123", "594"), 1)

if __name__ == '__main__':
   unittest.main()
```

The three test methods are designed to exercise distinct execution paths: `test_no_carry` traverses only the `else` branch, `test_all_carry` activates the `if` branch at every iteration, and `test_partial_carry` induces mixed branch execution.

Stage 3: Sandboxed Execution and Coverage Analysis. The extracted unit-test snippet

u_i is executed against the reference solution s within an isolated environment instrumented with `coverage.py`. The execution report is as follows:

```
Name Stmts Miss Cover Missing
----------------------------------------
src.py 13 0 100%
----------------------------------------
TOTAL 13 0 100%
3 passed in 0.02s
```

The execution harness yields the following metadata: $cov(u_i, s) = 1.0$ (full statement and branch coverage), no assertion failures, no runtime errors, and $|\text{assertions}| = 3$.

Stage 4: Multi-dimensional Reward Computation. Given the execution outcomes, the composite reward is computed according to the formulation in Sections 3.2–3.4:

(i) **Syntax reward.** AST parsing confirms a well-formed code block containing a class inheriting from `unittest.TestCase`; thus $r_{syn}(W_i|s) = +1.0$.

(ii) **Coverage with exponential shaping.** Given $cov(u_i, s) = 1.0$ and $\alpha = 2$:

$$r_{cov} = \frac{e^{\alpha \cdot cov} - 1}{e^{\alpha} - 1} = \frac{e^2 - 1}{e^2 - 1} = 1.0.$$

(iii) **Static difficulty prior.** $D_H = 0.42$, $D_M = 0.38$, yielding $D = \sqrt{D_H \cdot D_M} = 0.40$.

(iv) **Functionality reward.** Since all tests pass:

$$r_{func} = r_{cov} \cdot (1 + D) = 1.0 \times 1.40 = 1.40.$$

(v) **Total reward.** $r_i = r_{syn} + r_{func} = 2.40$.

Contrastive Reward Analysis. To illustrate the discriminative capacity of the reward function, we contrast the above outcome with two failure scenarios on the same sample:

- **Assertion failure** (incorrect expected value):

$$r_i = r_{syn} + \big(-1.0 - (1 - D)\big) = 1.0 - 1.60 = -0.60.$$

  The term $(1-D)$ attenuates the penalty for harder samples, as assertion errors on complex code are more tolerable during early training.

- **Execution error** (runtime exception):

$$r_i = r_{syn} + (-2.0) = -1.0,$$

  imposing the strongest penalty regardless of sample difficulty.

This contrastive structure establishes a clear reward hierarchy—*correct pass* > *assertion failure* > *execution error*—that guides the policy toward generating syntactically valid, semantically accurate, and branch-comprehensive unit tests throughout training.