Beyond Binary: Turning Partial Success into Dense Verifiable Rewards for Reinforcement Learning in Code Generation

Longwen Wang[1,3*†], Yirui Liu[1*], Xuan'er Wu[1], Xiaohui Hu[2], Yuankai Fan[1], Kaidong Yu[2], Qizhen Weng[1], Wei Xi[2‡], Xuelong Li[1‡]

[1]Institute of Artificial Intelligence, China Telecom (TeleAI)
[2]Xingchen AGI Lab, China Telecom Artificial Intelligence Technology (Beijing) Co., Ltd
[3]National Key Laboratory of Human-Machine Hybrid Augmented Intelligence, Xi'an Jiaotong University
longwenwang12@gmail.com, xiwei@xjtu.edu.cn, xuelong_li@ieee.org

Abstract

Effective reward design is a central challenge in Reinforcement Learning (RL) for code generation. Mainstream test-suite-level outcome rewards enforce functional correctness but induce sparsity, while external Reward Models (RMs) provide dense supervision at the cost of misalignment and additional overhead. Since code evaluation naturally yields multiple test-case-level outcomes, partial success—passing a subset of test cases—offers an intrinsic, verifiable source of dense supervision. In this paper, we propose **VeRPO** (**V**erifiable **D**ense **R**eward **P**olicy **O**ptimization), an RL framework that systematically turns verifiable partial success into reliable dense rewards. We analyze partial-success rewards using a weighted sum formulation, theoretically identifying a critical cardinality bias that causes policy updates to disproportionately favor gains from easy-test successes over progress on frontier tests. Based on this, VeRPO introduces a dynamic, density-calibrated local reward that explicitly corrects this bias and provides robust dense supervision from partial success. To enhance alignment with end-to-end functional correctness, VeRPO further integrates the local dense reward with global execution outcomes. Extensive experiments across diverse benchmarks and settings demonstrate that VeRPO outperforms outcome-driven and RM-based baselines, achieving up to +8.83 pass@1 gain with negligible time cost (< 0.02%) and zero GPU memory overhead.

1 Introduction

Code generation has emerged as a core capability of Large Language Models (LLMs), empowering LLMs to address competitive programming tasks through single-turn solution synthesis (Roziere et al., 2023; Wang et al., 2021) or iterative multi-turn refinement (Dong et al., 2025; Zheng et al., 2024), where each generated executable solution is evaluated against a suite of problem-specific test cases. Within this domain, Reinforcement Learning (RL) has significantly scaled performance, with its effectiveness hinging on the design of reward signals (Mroueh, 2025; Chen et al., 2025; Lambert et al., 2024).

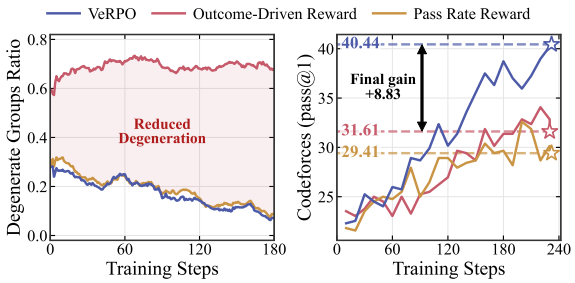


Figure 1: Evaluation of different verifiable reward designs. *Left*: Signal efficiency, where the degenerate group ratio indicates the fraction of zero-advantage groups per batch. *Right*: Pass@1 performance of group-based RL on Codeforces problems using Qwen3-8B.

Current reward designs for code generation predominantly fall into two distinct paradigms, each facing specific limitations. First, mainstream outcome-driven approaches enforce functional correctness[1] with a verifiable binary reward derived from test-suite execution outcomes (e.g., 1 if all unit tests pass and 0 otherwise). However, such strict verification induces severe sparsity: when all sampled solutions receive identical binary outcomes, group-based RL methods such as GRPO (Guo et al., 2025) yield zero relative advantages and thus no gradient signal. Alternatively, recent studies have explored external Reward Models (RMs) to provide dense supervision based on code semantics (Zeng et al., 2025; Ye et al., 2025; Dai et al., 2024), but external learned RMs introduce prohibitive computational overhead and vulnerabil-

*Equal contribution.
†Work done during internship at TeleAI.
‡Corresponding authors

[1]Functional correctness indicates that the generated code produces the expected output for all valid inputs.

ity to reward hacking (Zhang et al., 2025a). This dual-bottleneck scenario raises a central question: *Can we obtain dense rewards for code-generation RL while preserving verifiability and avoiding auxiliary reward models?*

The test-suite structure of programming problems suggests a promising answer: *partial success* (i.e., passing a subset of test cases). Although outcome-driven rewards collapse execution feedback into a single binary signal, each generated solution actually receives multiple test-case-level outcomes[2]. Passing a subset of tests is thus not equivalent to complete failure; rather, it provides verifiable evidence of intermediate functional correctness and offers denser and more informative signal without any auxiliary reward model. Fig. 1 (*Left*) shows that even a naive partial-success reward—the pass rate of the test suite—substantially reduces the fraction of zero-advantage groups over outcome-driven rewards, indicating that partial success effectively densifies supervision.

However, a denser reward is not necessarily better. Although the naive pass-rate reward is both dense and verifiable, it can still degrade final performance compared to binary outcome-driven rewards (Fig. 1 *Right*). This contrast suggests that partial success does not automatically yield an effective reward signal; it must be carefully calibrated.

In this paper, we propose VeRPO, an RL framework that systematically turns verifiable partial success into dense and robust rewards. Rather than proposing another heuristic partial-success reward, we first introduce a general weighted-sum formulation over per-test execution results, which subsumes a broad class of partial-success rewards and provides an analytical lens for examining their optimization behavior. Using this formulation, we reveal a severe **cardinality bias** inherent in rewarding partial success: redundant, easy tests dominate the optimization baseline by their sheer count, causing policy optimization to disproportionately favor gains from easy-test successes rather than progress on frontier tests. Building on this analysis, VeRPO introduces a dynamic, density-calibrated local reward that corrects the cardinality bias and provides informative dense supervision from partial success. To anchor optimization to end-to-end functional correctness, VeRPO further incorporates a global outcome reward derived from complete test-suite execution. Together, the two reward signals enable dense and reliable optimization without auxiliary reward models, yielding strong performance gains of VeRPO (Fig. 1). In diverse code generation benchmarks and settings, extensive experiments demonstrate that VeRPO consistently outperforms outcome-driven and RM-based baselines, achieving up to 8.83 performance gain in pass@1 with negligible time cost ($< 0.02\%$) and zero GPU memory overhead.

Our key contributions are as follows:

- We systematically investigate the mechanism of rewarding partial success, demonstrating its potential for dense supervision while revealing an inherent cardinality bias that misguides policy optimization.
- We propose VeRPO, an RL framework that corrects the cardinality bias in partial-success rewards and enables dense and reliable optimization purely from verifiable execution feedback, without auxiliary reward models.
- Extensive experiments on diverse code-generation benchmarks and settings demonstrate the superior performance of VeRPO over baselines, with zero GPU memory overhead and negligible time cost.

2 Related Work

2.1 RL for Code Generation

RL has emerged as a central paradigm for improving LLM-based code generation (Liu et al., 2023a; Wang et al., 2024; Zhang et al., 2025b). Early methods, including CodeRL (Le et al., 2022) and PPOCoder (Shojaee et al., 2023), employ execution feedback as reward signals under the REINFORCE algorithm (Williams, 1992) and Proximal Policy Optimization framework (PPO; Schulman et al., 2017), respectively, establishing an early foundation for RL in this domain. More recent advances, exemplified by DeepSeek-R1 (Guo et al., 2025), combine scalable RL with chain-of-thought reasoning, boosting interest in RL for code generation. Recent work has further extended RL to multi-turn code generation, where methods such as μCODE (Jain et al., 2025) and RLEF (Gehring et al., 2024) support iterative self-correction for challenging coding problems. In line with the above studies, we focus on competitive code-generation tasks in this paper, where each model response is a complete

[2]Code generation benchmarks typically contain substantial test cases, making partial success naturally dense; see Appendix A for dataset-level statistics.

solution candidate for a standalone programming problem and is evaluated against the same task-level test suite. This setting differs from agentic coding, which often presupposes a code-capable model and focuses on higher-level workflows such as planning, tool use, and environment interaction for software-engineering tasks (Jimenez et al., 2024; Yang et al., 2024).

2.2 Reward Design in Code Generation

Reward design plays a central role in RL for code generation (Sun et al., 2025; Xu, 2025). Mainstream approaches typically rely on outcome-driven rewards computed from test-suite execution results. For example, GRPO and its variants (Yu et al., 2025; Yue et al., 2025; Ahmadian et al., 2024) use binary execution outcomes as verifiable reward signals, directly aligning policy outputs with functional correctness. More recently, external reward models (RMs) have been introduced to provide richer and denser supervision. For instance, AceCoder (Zeng et al., 2025) trains a code-specific RM on large-scale preference pairs for fine-grained quality assessment, while CodePRM (Li et al., 2025) trains an RM on execution traces to provide dense supervision for intermediate reasoning steps. However, both paradigms face inherent limitations: outcome-driven rewards enforce functional correctness but introduce severe sparsity, while RM-based methods offer dense supervision but suffer from misalignment and computational overhead.

Different from these lines of work, we focus on the verifiable partial-success structure present in code execution itself. Rather than relying only on binary terminal outcomes or introducing auxiliary learned reward models, we systematically study how partial success can be translated into reliable reward signals for code-generation RL.

3 Preliminaries

3.1 Code Generation as an MDP

We formulate code generation as a Markov Decision Process (MDP), where a policy π_θ generates up to T code candidates and receives execution feedback after each attempt. Given a problem prompt $x \in \mathcal{X}$, at each turn $t \in \{1, \ldots, T\}$, the policy generates a code candidate y_t conditioned on the current state $s_t = (x, y_1, o_1, \ldots, y_{t-1}, o_{t-1})$, which records the interaction history up to turn t, with $s_1 = x$. After generating y_t, the code is executed against the test suite $\mathcal{U}_x = \{u_1, \ldots, u_{|\mathcal{U}_x|}\}$ and yields the corresponding execution feedback o_t. The process terminates once the generated code passes the test suite $\mathcal{U}_x$ or the maximum turn budget T is reached, yielding a trajectory $\tau = \big((s_1, y_1, o_1), (s_2, y_2, o_2), \ldots, (s_{|\tau|}, y_{|\tau|}, o_{|\tau|})\big)$, where $|\tau|$ denotes the number of executed turns. This formulation naturally unifies both single-turn ($T = 1$) and multi-turn ($T > 1$) code generation.

3.2 Group-Based RL

Group-based RL has become a prominent paradigm for optimizing LLMs. Given a prompt x, a group of N trajectories $\mathcal{G}_x = \{\tau_1, \tau_2, \ldots, \tau_N\}$ is sampled under $\pi_{\theta_{\text{old}}}$. Each trajectory τ_i receives a scalar reward $R(\tau_i)$ indicating its quality (e.g., 1 for correct, 0 for incorrect). Unlike standard actor-critic frameworks such as PPO, which require a separate value network to compute optimization baseline for advantage estimation, group-based RL methods estimate advantages directly from the rewards within the sampled group. Typically, they utilize the group mean reward as the optimization baseline, defining the advantage for each trajectory as:

$$A(\tau_i) = \big(R(\tau_i) - \text{mean}\left(\{R(\tau_j)\}_{j=1}^{N}\right)\big) \;/\; F_{\text{norm}}. \tag{1}$$

where F_{norm} denotes the advantage normalization factor. GRPO defaults to $F_{\text{norm}} = \text{std}$, whereas such normalization introduces task-level difficulty bias (Liu et al., 2025). Unless otherwise specified, we set a constant normalization factor $F_{\text{norm}} = 1$, which yields an unbiased Leave-One-Out estimator (Feng et al., 2025; Kool et al., 2019).

4 Method

This section presents the analysis that motivates VeRPO and the reward design that implements it. Section 4.1 formalizes partial-success rewards and reveals an inherent cardinality bias that misguides policy optimization. Section 4.2 then introduces VeRPO, which corrects this bias and turns partial success into dense, reliable rewards.

4.1 Partial Success and Cardinality Bias

In code generation tasks with multiple test cases, each generated solution is verified against all test cases, yielding a vector of per-test pass/fail outcomes at each turn. The central question is how to turn this partial-success information into an effective reward for RL training. Our formulation is guided by a dual consideration of generality and

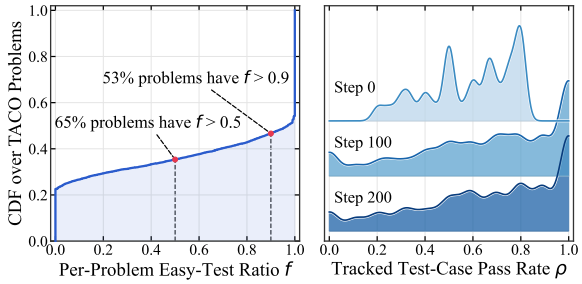


Figure 2: Test-case pass-rate skew on TACO with Qwen3-8B. *Left*: CDF over TACO problems of f, where f denotes the fraction of test cases in $\mathcal{U}_x$ with pass rate $\rho > 0.8$. *Right*: Pass-rate distribution during training for test cases with $\rho \in [0.2, 0.8]$ at Step 0.

simplicity: for generality, test cases differ in hardness, thus their varying contributions to the reward need to be explicitly modeled; for simplicity, we do not model cross-turn credit assignment—since each turn produces a complete code solution evaluated against the full test suite, the execution feedback at each turn directly measures the functional quality of that solution and requires no temporal attribution across turns. Consequently, we quantify this turn-level partial success as a weighted sum over per-test outcomes, yielding a non-binary reward signal across both single- and multi-turn settings:

$$r_{t,i} = \sum_{j=1}^{|\mathcal{U}_x|} w_j \, p_{t,i}^{(j)}, \tag{2}$$

where $p_{t,i}^{(j)} \in \{0, 1\}$ denotes whether the turn-t code in trajectory τ_i passes test case $u_j \in \mathcal{U}_x$, and $w_j \geq 0$ specifies the contribution of test case u_j. Eq. (2) allows flexible modeling of test-level contributions, with the standard pass-rate reward serving as the uniform-weight special case.

Cardinality bias in the optimization baseline. One key aspect of implementing Eq. (2) is how to assign weights properly to each test case. Rather than simply relying on heuristics, we identify a structural skewness in the difficulty distribution of test cases and formally show how it induces an intrinsic bias in the optimization baseline—which we term **cardinality bias**. This in turn gives us a principled foundation for weight design.

The structural skewness arises from the uneven difficulty distribution of test cases. We measure test-case difficulty by empirical pass rate—the fraction of generated code passing a given test. Formally, for problem x with test suite $\mathcal{U}_x$, the empirical pass rate of test case $u_j \in \mathcal{U}_x$ over rollout group $\mathcal{G}_x$ is:

$$\rho_j = \frac{1}{M_x} \sum_{i=1}^{|\mathcal{G}_x|} \sum_{t=1}^{|\tau_i|} p_{t,i}^{(j)}, \quad M_x = \sum_{i=1}^{|\mathcal{G}_x|} |\tau_i|, \tag{3}$$

where M_x is the total number of turns in trajectory group $\mathcal{G}_x$. A test case is easy if ρ_j is close to 1, and in practice, easy test cases constitute the majority. As shown in Fig. 2 (*Left*), the difficulty distribution of test cases is highly skewed before training begins: for 53% of TACO problems with Qwen3-8B, over 90% of their respective test cases are already easy ($\rho_j > 0.8$). Moreover, as RL training proceeds, this skewness is further exacerbated, with previously intermediate-difficulty tests ($\rho_j \in [0.2, 0.8]$) progressively migrating into the high-pass-rate region (Fig. 2, *Right*). Similar results on other datasets are shown in Appendix B.

To formally analyze how the uneven difficulty distribution induces a bias, we examine the optimization baseline in group-based RL, which corresponds to the group mean in Eq. (1). Since Eq. (2) defines a turn-level reward signal, we instantiate its group-mean baseline at the same granularity:

$$\bar{r}^{\text{turn}} = \frac{1}{M_x} \sum_{i=1}^{|\mathcal{G}_x|} \sum_{t=1}^{|\tau_i|} r_{t,i}. \tag{4}$$

Next, substituting Eq. (2) into Eq. (4) gives

$$\begin{aligned} \bar{r}^{\text{turn}} &= \frac{1}{M_x} \sum_{i=1}^{|\mathcal{G}_x|} \sum_{t=1}^{|\tau_i|} \sum_{j=1}^{|\mathcal{U}_x|} w_j p_{t,i}^{(j)} \\ &= \sum_{j=1}^{|\mathcal{U}_x|} w_j \underbrace{\left(\frac{1}{M_x} \sum_{i=1}^{|\mathcal{G}_x|} \sum_{t=1}^{|\tau_i|} p_{t,i}^{(j)} \right)}_{\text{empirical pass rate of } j\text{-th test}} \\ &= \sum_{j=1}^{|\mathcal{U}_x|} w_j \rho_j. \end{aligned} \tag{5}$$

Eq. (5) reveals that the optimization baseline equals the weighted sum of per-test empirical pass rates, which can be rewritten in integral form (derivation in Appendix D.1):

$$\bar{r}^{\,\text{turn}} = \int_0^1 \bar{w}(\rho) \, \rho \, N(\rho) \, d\rho. \tag{6}$$

where $\bar{w}(\rho)$ is the average weight over tests with its pass rate equal to ρ, and $N(\rho) = \sum_{j=1}^{|\mathcal{U}_x|} \delta(\rho - \rho_j)$ is the empirical density of test cases at pass rate ρ—precisely the difficulty distribution.

Recall that $N(\rho)$ is heavily skewed toward the high-pass-rate end (Fig. 2). The high-ρ region therefore dominates the baseline contribution in

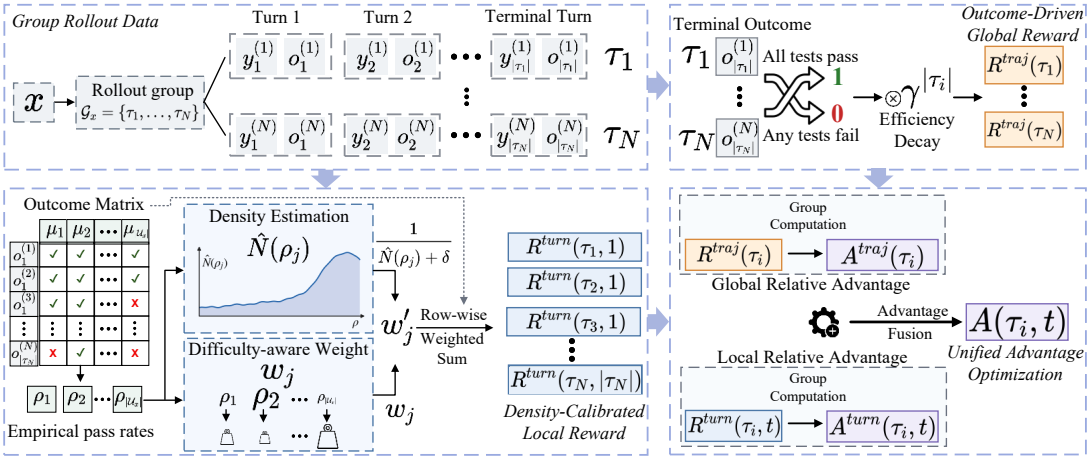


Figure 3: Overview of VeRPO. VeRPO fuses density-calibrated local rewards derived from partial success with outcome-driven global rewards, enabling effective and dense policy optimization for code generation.

Eq. (6) by sheer count, making easy tests disproportionately influential despite the designed weights $\{w_j\}$ (see Appendix D.2 for formal analysis). As a result, a partial-success reward $r_{t,i}$ tends to exceed the inflated baseline primarily by passing more easy tests than average—progress on hard frontier tests contributes far less toward pushing the reward above the baseline. Policy updates therefore preferentially reinforce easy-test gains rather than progress at the model's capability boundary. We term this **cardinality bias**.

4.2 VeRPO

We propose VeRPO, a group-based RL framework that addresses the cardinality bias analyzed in Section 4.1 and retains dense, robust supervision from partial success. VeRPO comprises two complementary components: a *density-calibrated local reward* that corrects cardinality bias to extract informative dense supervision from partial success, and an *outcome-driven global reward* that anchors optimization to holistic functional correctness. The overview of VeRPO is depicted in Fig. 3.

Density-Calibrated Local Reward. Cardinality bias arises because $N(\rho)$ in Eq. (6) amplifies the baseline contribution of dense easy-test regions regardless of the designed weights. VeRPO corrects this by estimating $N(\rho)$ at each ρ_j via a Gaussian kernel density estimator:

$$\hat{N}(\rho_j) = \sum_{j'=1}^{|\mathcal{U}_x|} \exp\left(-(\rho_j - \rho_{j'})^2 / 2\sigma^2\right), \quad (7)$$

where σ is the kernel bandwidth. Dividing each test-case weight by $\hat{N}(\rho_j)$ counteracts the local density factor $N(\rho)$ in Eq. (6), yielding an approximation to the density-free baseline $\int_0^1 \bar{w}(\rho)\,\rho\,d\rho$ and ensuring it is governed by the designed weights rather than by the cardinality of easy tests. Beyond density correction, since ρ_j naturally reflects test difficulty under the current policy, VeRPO further assigns an exponential difficulty-aware weight:

$$w_j = \exp(-\alpha \rho_j), \quad (8)$$

where $\alpha > 0$ controls the degree of emphasis on rarely-passed frontier tests relative to already-mastered ones. Combining density calibration and difficulty-aware weighting, the final adjusted weight for test case u_j is:

$$w'_j = \exp(-\alpha \rho_j) / (\hat{N}(\rho_j) + \delta). \quad (9)$$

where $\delta > 0$ is a small constant for numerical stability. The local partial-success reward for turn t in trajectory τ_i is thus:

$$R^{turn}(\tau_i, t) = \sum_{j=1}^{|\mathcal{U}_x|} w'_j \cdot p_{t,i}^{(j)}, \quad (10)$$

which focuses optimization on the unmastered frontier rather than the redundant majority.

Outcome-Driven Global Reward. While the density-calibrated local reward provides dense supervision, it optimizes over partial execution outcomes and does not directly target whether the complete test suite passes. To anchor optimization to end-to-end functional correctness, VeRPO further incorporates a binary trajectory-level outcome reward $R^{traj} \in \{0, 1\}$, where $R^{traj}(\tau_i) = 1$ if the terminal execution feedback passes the complete test suite, and $R^{traj}(\tau_i) = 0$ otherwise. Since this

reward is defined on the holistic test-suite outcome rather than an aggregation of per-test partial success, it is free from the cardinality bias analyzed in Section 4.1. We further apply an efficiency-aware decay to favor trajectories that reach correct solutions in fewer turns:

$$\tilde{R}^{traj}(\tau_i) = R^{traj}(\tau_i) \cdot \gamma^{|\tau_i|}, \tag{11}$$

where $\gamma \in (0, 1]$ controls the strength of the efficiency preference. Together, the local and global rewards provide complementary supervision: the former densifies the training signal through calibrated partial success, while the latter preserves strict alignment with functional correctness.

Unified Advantage Optimization. To combine the two reward signals in a unified policy update, VeRPO constructs a local and a global advantage, and fuses them into a turn-level training objective. For the local reward, we collect all turn-level partial-success rewards within the rollout group as $G(\mathcal{G}_x) = \{R^{turn}(\tau_i, t) \mid \tau_i \in \mathcal{G}_x,\ 1 \le t \le |\tau_i|\}$, and define the local relative advantage as

$$A^{turn}(\tau_i, t) = R^{turn}(\tau_i, t) - \text{mean}(G(\mathcal{G}_x)). \tag{12}$$

At the trajectory level, we compute a global relative advantage from the decayed outcome reward:

$$A^{traj}(\tau_i) = \tilde{R}^{traj}(\tau_i) - \text{mean}(\{\tilde{R}^{traj}(\tau_j)\}_{j=1}^{N}). \tag{13}$$

We then broadcast the trajectory-level advantage to each turn and combine it with the local advantage:

$$A(\tau_i, t) = A^{traj}(\tau_i) + \beta \cdot A^{turn}(\tau_i, t), \tag{14}$$

where $\beta \ge 0$ controls the contribution of the two advantages. The trajectory-level advantage $A^{traj}(\tau_i)$ anchors the global optimization, ensuring policy updates adhere to end-to-end functional correctness, while the turn-level advantage $A^{turn}(\tau_i, t)$ enriches the learning signal with dense feedback from local partial success. Finally, VeRPO optimizes the policy with a clipped group-based objective using the unified turn-level advantage $A(\tau_i, t)$; the full objective is provided in Appendix D.3.

5 Experiments

5.1 Setup

Datasets. For training, we utilize the subset dataset derived from (Luo et al., 2025), comprising 7.4K verified code problems from TACO (Li et al., 2023). We evaluate performance across four mainstream code generation benchmark suites: 1) HumanEval (Chen et al., 2021) along with its enhanced variant HumanEval-Plus (Liu et al., 2023b); 2) BigCodeBench (Zhuo et al., 2024) reporting results on both its Full and Hard subsets; 3) LiveCodeBench (LCB, V6, 2023.05–2025-04) (Jain et al., 2024); and 4) Codeforces problems derived from CodeElo (Quan et al., 2025).

Baselines. We benchmark VeRPO against the predominant group-based method, GRPO, instantiated with two distinct reward configurations: 1) *vanilla outcome-driven rewards*, representing the standard GRPO configuration that relies on binary execution outcomes indicating overall functional correctness; and 2) *RM-based dense rewards*, which leverages an external RM to assign fine-grained rewards to intermediate turns. Following prior work, we employ AceCodeRM-7B (Zeng et al., 2025) to generate turn-level dense rewards.

Implementation. We use Qwen3-8B as the main RL backbone, with additional backbone-scale results reported in Appendix G. To evaluate performance across varying horizons, we conduct training and evaluation in both single-turn (ST, $T = 1$) and multi-turn (MT, $T = 4$) code generation settings. During training, we utilize a rollout batch size of 32 code problems with 10 responses sampled per problem; maximum response length is set to 16,384 tokens and sampled with temperature 1.0. For evaluation, we set the maximum response length to 16,384 tokens and sampling temperature to 0.6. We report pass@1, computed with the unbiased estimator of (Chen et al., 2021), as the primary functional-correctness metric. See Appendix E for additional training and evaluation details.

5.2 Results and Analysis

5.2.1 Performance of VeRPO

Table 1 reports pass@1 results across six benchmarks under single-turn (ST) and multi-turn (MT) training/evaluation settings. In the ST→ST setting, VeRPO achieves the best performance across all benchmarks, obtaining an average improvement of +1.26 and +2.18 over the outcome-driven GRPO and the RM-based AceCoder, respectively. The superiority underscores the intrinsic efficacy of VeRPO, yielding stronger effectiveness in the standard single-response setting. The performance gain of VeRPO becomes more pronounced in the MT

Method	Train	RM	Reward	HumanEval				BigCodeBench				LCB		Codeforces		Avg	
				-		Plus		Full		Hard		V6		CodeElo			
				ST	MT	ST	MT	ST	MT	ST	MT	ST	MT	ST	MT	ST	MT
Qwen3-8B	-	-	-	91.03	95.05	87.49	90.14	35.85	58.32	16.38	41.69	27.12	28.14	20.77	22.73	46.44	56.01
GRPO	ST	-	0-1	91.99	96.34	88.03	90.32	36.33	57.87	16.87	40.54	28.35	29.28	28.49	30.72	48.34	57.51
	MT	-	0-1	91.46	96.41	87.57	91.23	35.73	60.91	16.55	43.32	28.35	30.50	27.38	31.61	47.84	59.00
AceCoder	ST	7B	Dense	92.04	96.59	88.71	91.46	36.07	58.36	14.86	39.53	27.78	27.57	25.06	26.96	47.42	56.74
	MT	7B	Dense	---------- Training Collapse ----------													
VeRPO	ST	-	Dense	**92.73**	96.49	**89.32**	91.84	**37.42**	58.95	**17.91**	41.55	**29.72**	30.14	**30.50**	33.76	**49.60**	58.79
	MT	-	Dense	91.69	**97.87**	88.05	**93.14**	36.10	**62.52**	17.22	**45.69**	29.07	**33.07**	29.03	**40.44**	48.53	**62.12**

Table 1: Performance comparison on six benchmarks using pass@1. ST/MT denotes single-turn/multi-turn setting for training and evaluation. Dashed entries denote training collapse due to optimization instability. Best/second-best results for each benchmark and evaluation setting are highlighted in **bold**/underlined.

Method	HumanEval	BigCodeBench		LCB	Codeforces
	Plus	Full	Hard	V6	CodeElo
w/o A^{turn}	91.58	61.58	44.25	30.64	34.68
w/o A^{traj}	92.31	62.12	45.18	31.14	37.62
w/ std	93.06	61.87	44.67	31.64	38.05
VeRPO	**93.14**	**62.52**	**45.69**	**33.07**	**40.44**

Table 2: Ablation study on components of VeRPO.

A^{traj}	R^{turn}	HumanEval	BigCodeBench		LCB	Codeforces
		Plus	Full	Hard	V6	CodeElo
GRPO	-	91.23	60.91	43.32	30.50	31.61
-	PS	91.69	59.45	43.49	29.64	29.41
-	Diff	91.46	59.78	41.55	30.28	35.23
-	VeRPO	92.31	62.12	45.18	31.14	37.62
VeRPO	PS	92.14	61.23	44.34	32.00	35.75
VeRPO	Diff	91.31	60.00	43.15	30.57	35.11
VeRPO	VeRPO	**93.14**	**62.52**	**45.69**	**33.07**	**40.44**

Table 3: Ablation study on local partial-success reward designs for constructing the turn-level reward R^{turn}. PS: raw pass-rate rewards; Diff: difficulty-weighted rewards without density normalization.

setting. AceCoder fails to complete stable MT training due to optimization collapse, as detailed in Appendix F.1, suggesting the instability risk of relying on an external black-box reward model for iterative optimization. In contrast, VeRPO is fully grounded in verifiable execution feedback and remains stable during optimization. Compared with execution-based GRPO under MT training and evaluation, VeRPO improves the average pass@1 by 3.12 points, with the largest gain on the challenging Codeforces benchmark (+8.83). This margin indicates that extracting effective dense learning signals from partial success contributes to improved performance on harder problems. Furthermore, VeRPO retains strong cross-setting performance, outperforming the corresponding baselines even under mismatched training and evaluation horizons.

5.2.2 Ablation Study

Effect of Components in VeRPO. We ablate three components of VeRPO in the multi-turn setting: the local turn-level advantage A^{turn}, the global trajectory-level advantage A^{traj}, and the fixed normalization factor $F_{\text{norm}} = 1$. We compare the full model with variants that remove A^{turn}, remove A^{traj}, or replace fixed normalization with standard-deviation normalization. Table 2 shows that removing any component degrades performance. Removing A^{turn} leads to the largest degradation, with consistent drops across all benchmarks and a particularly large decline on Codeforces, from 40.44 to 34.68, which is consistent with the role of A^{turn} in providing dense partial-success supervision beyond sparse outcome feedback. Removing A^{traj} also lowers performance across all benchmarks, indicating that trajectory-level correctness remains important for anchoring optimization to full functional correctness. Finally, replacing the fixed normalization with standard-deviation normalization consistently underperforms VeRPO, but the gap is smaller than those caused by removing either advantage term. This suggests that fixed normalization improves calibration, while the dominant gains come from combining local turn-level and global trajectory-level advantages.

Ablation on Partial-Success Reward Designs. To rigorously validate the design of the local partial-success reward R^{turn} in VeRPO, we benchmark VeRPO against two execution-based partial-success rewards: (a) *Raw Pass Rate* (PS), which uses the

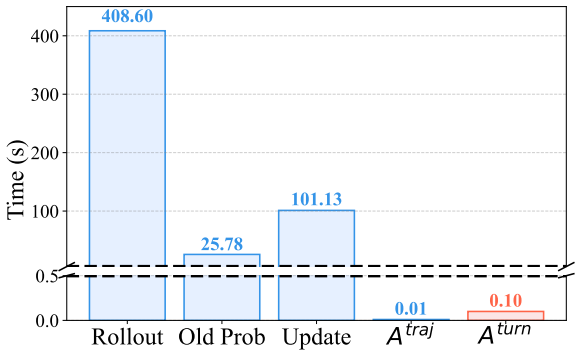


Figure 4: Computational analysis of VeRPO. Components shared with GRPO are highlighted in blue, while VeRPO-specific computations are shown in orange. A broken y-axis scale is employed to enhance the visualization of small values.

fraction of passed unit tests and ignores test difficulty; and (b) *Difficulty-Weighted* (Diff), which applies the difficulty-aware weight in Eq. 8 but removes the kernel density normalization. We evaluate each reward both in isolation and when fused with the global trajectory-level advantage A^{traj}. Table 3 shows that uncalibrated partial-success rewards are not sufficient. When used alone, both PS and Diff degrade performance relative to VeRPO and even underperform the outcome-driven GRPO baseline without turn-level partial-success rewards, especially on harder benchmarks such as BigCodeBench-Hard, LCB, and Codeforces. This indicates that dense partial-success feedback can be harmful if it is not properly calibrated. Adding the global trajectory-level advantage improves these variants, but they still lag behind VeRPO. The remaining gap suggests that difficulty weighting alone does not address the cardinality bias identified in Section 4.1; density calibration is needed to make partial success a reliable learning signal. See Appendix C for rollout-level case studies.

5.2.3 Signal Efficiency and Information Loss

To better understand the performance gains of VeRPO, we analyze how often rollout groups provide non-degenerate optimization signals in the multi-turn setting. We employ the degenerate group ratio: the fraction of rollout groups with identical rewards, which yield zero relative advantages and thus no policy-gradient signal under group-relative optimization. Fig. 1 (*Left*) shows that outcome-driven GRPO maintains a high degenerate group ratio, typically between 60% and 70%, indicating that binary outcome rewards discard substantial information from partial execution outcomes. In contrast, VeRPO keeps this ratio below 25% for most of training and further reduces it as optimization proceeds, showing that partial success can be turned into more frequently usable training signals. Importantly, a lower degenerate group ratio alone does not guarantee better optimization: as discussed in Section 1, naive pass-rate rewards can densify supervision but still degrade final performance. The gains of VeRPO therefore come not merely from making the reward denser, but from calibrating partial-success signals through density-aware weighting while retaining the trajectory-level correctness anchor.

5.2.4 Computational Cost

We assess the computational overhead introduced by VeRPO. As detailed in Section 4, VeRPO follows the same group-based RL pipeline of GRPO, inheriting an identical grouped rollout sampling, computation of old log-probabilities, and clipped actor policy updates. Both pipelines eliminate the need for auxiliary critic or reward models by relying solely on execution feedback to compute rewards and advantages, thus sharing identical GPU memory consumption and LLM rollout overheads.

The only additional computation is the density-calibrated turn-level reward and advantage estimation. Fig. 4 provides a per-iteration time breakdown. Computing A^{turn} adds only 0.10s per iteration, accounting for less than 0.02% of total training time, while rollout and policy updates dominate the cost. These results demonstrate that VeRPO achieves dense and effective supervision with virtually zero marginal computational cost, effectively defying the conventional trade-off between reward granularity and computational efficiency.

6 Conclusion

We introduce VeRPO, an RL framework that turns verifiable partial success into robust dense rewards for code generation. Our systematic analysis revealed an inherent cardinality bias in partial-success rewards: redundant easy tests dominate the optimization baseline by sheer count, causing policy updates to favor easy-test gains over frontier-test progress. Building on this analysis, VeRPO introduces a density-calibrated local reward that corrects the bias and combines it with a global outcome-driven reward that anchors optimization to end-to-end functional correctness. Extensive ex-

periments across various benchmarks and settings demonstrate consistent gains over outcome-driven and RM-based baselines with negligible computational overhead.

References

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to basics: Revisiting reinforce-style optimization for learning from human feedback in llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12248–12267.

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

Yongchao Chen, Yueying Liu, Junwei Zhou, Yilun Hao, Jingquan Wang, Yang Zhang, and Chuchu Fan. 2025. R1-code-interpreter: Training llms to reason with code via supervised and reinforcement learning. *arXiv preprint arXiv:2505.21668*.

Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. 2024. Process supervision-guided policy optimization for code generation. *arXiv preprint arXiv:2410.17621*.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. 2025. Group-in-group policy optimization for llm agent training. In *Advances in Neural Information Processing Systems*, volume 38, pages 46375–46408. Curran Associates, Inc.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. 2024. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638.

Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. 2025. Multi-turn code generation through single-step rewards. In *Forty-second International Conference on Machine Learning*.

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 Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Wouter Kool, Herke van Hoof, and Max Welling. 2019. Buy 4 reinforce samples, get a baseline for free!

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

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

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025. Codeprm: Execution feedback-enhanced process reward model for code generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8169–8182.

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

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

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023a. Rltf: Reinforcement learning from unit test feedback. *arXiv 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. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. 2025. Understanding r1-zero-like training: A critical perspective. In *Conference on Language Modeling (COLM)*.

Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, and 1 others. 2025. Deepcoder: A fully open-source 14b coder at o3-mini level. *Notion Blog*.

Youssef Mroueh. 2025. Reinforcement learning with verifiable rewards: Grpo's effective loss, dynamics, and success amplification. *arXiv preprint arXiv:2503.06639*.

Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. 2025. Codeforces cots. https://huggingface.co/datasets/open-r1/codeforces-cots.

Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, and 1 others. 2025. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Shengjie Sun, Runze Liu, Jiafei Lyu, Jing-Wen Yang, Liangpeng Zhang, and Xiu Li. 2025. A large language model-driven reward design framework via dynamic feedback for reinforcement learning. *Knowledge-Based Systems*, 326:114065.

Junqiao Wang, Zeng Zhang, Yangfan He, Zihao Zhang, Xinyuan Song, Yuyang Song, Tianyu Shi, Yuchen Li, Hengyuan Xu, Kunyu Wu, and 1 others. 2024. Enhancing code llms with reinforcement learning in code generation: A survey. *arXiv preprint arXiv:2412.20367*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. 2025. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. *Preprint*, arXiv:2504.14655.

Xingcheng Xu. 2025. The policy cliff: A theoretical analysis of reward-policy maps in large language models. *arXiv preprint arXiv:2507.20150*.

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

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Yufan Ye, Ting Zhang, Wenbin Jiang, and Hua Huang. 2025. Process-supervised reinforcement learning for code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 14224–14237.

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

Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, and 1 others. 2025. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks. *arXiv preprint arXiv:2504.05118*.

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

Junkai Zhang, Zihao Wang, Lin Gui, Swarnashree Mysore Sathyendra, Jaehwan Jeong, Victor Veitch, Wei Wang, Yunzhong He, Bing Liu, and Lifeng Jin. 2025a. Chasing the tail: Effective rubric-based reward modeling for large language model post-training. *arXiv preprint arXiv:2509.21500*.

Kaiyan Zhang, Yuxin Zuo, Bingxiang He, Youbang Sun, Runze Liu, Che Jiang, Yuchen Fan, Kai Tian, Guoli Jia, Pengfei Li, Yu Fu, Xingtai Lv, Yuchen Zhang, Sihang Zeng, Shang Qu, Haozhan Li, Shijie Wang,

Yuru Wang, Xinwei Long, and 20 others. 2025b. A survey of reinforcement learning for large reasoning models. *Preprint*, arXiv:2509.08827.

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

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Dataset-Level Test-Suite Statistics

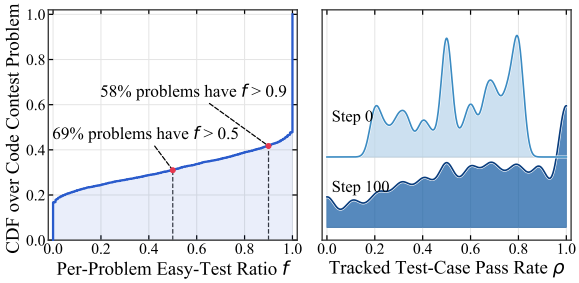
Table 4 reports the number of test cases per problem across representative code-generation benchmarks. These datasets consistently contain multiple test cases per problem, supporting our setting where partial success can be observed directly from test-case-level execution feedback.

Dataset	Total Examples	Mean Tests	Median Tests
CodeContests	13,328	95.91	101
LeetCodeDataset	2,641	100.10	99
codeforces-cots	9,556	51.07	43
TACO	25,443	53.36	10
PrimeIntellect	16,252	89.67	101

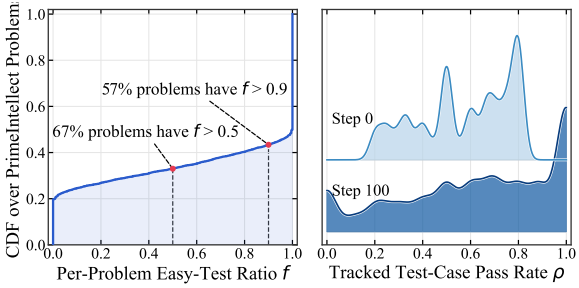
Table 4: Dataset-level test-suite statistics for five representative code-generation datasets: deepmind/code_contests (Li et al., 2022), newfacade/LeetCodeDataset (Xia et al., 2025), open-r1/codeforces-cots (Penedo et al., 2025), BAAI/TACO (Li et al., 2023) and the PrimeIntellect subset from agentica-org/DeepCoder-Preview-Dataset (Luo et al., 2025).

B Additional Evidence of Test-Case Difficulty Skew

To show that test-case difficulty skew is a general phenomenon in code generation rather than an artifact of a particular benchmark, we extend the same analysis to two additional competitive code-generation datasets: CODECONTESTS (deepmind/code_contests (Li et al., 2022)) and PRIMEINTELLECT (for the latter, we randomly sample 10,000 examples from the PrimeIntellect subset in agentica-org/DeepCoder-Preview-Dataset (Luo et al., 2025) with seed 0). Following the analysis protocol of Fig. 2, we define the per-problem easy-test ratio f as the fraction of test cases in problem



(a) CODECONTESTS.



(b) PRIMEINTELLECT.

Figure 5: Additional dataset-level evidence of test-case difficulty skew. For each dataset, the left panel plots the CDF of the per-problem easy-test ratio (f), illustrating the abundance of initially easy test cases. The right panel tracks the pass-rate distribution shifts of initially intermediate test cases ($\rho \in [0.2, 0.8]$) across training steps.

x whose empirical pass rate satisfies $\rho > 0.8$. We also track the pass-rate distribution of test cases that are initially in the intermediate regime, i.e., $\rho \in [0.2, 0.8]$ at Step 0.

Fig. 5 shows that the same pattern persists across datasets. Specifically, a large fraction of problems contain many high-pass-rate test cases before training: on CODECONTESTS, 69% of problems have $f > 0.5$ and 58% have $f > 0.9$; similarly, on PRIMEINTELLECT, 67% of problems have $f > 0.5$ and 57% have $f > 0.9$. Furthermore, initially intermediate test cases tend to migrate toward higher pass-rate regions as training proceeds. These consistency results provide additional evidence that the partial-success signal in code generation is structurally skewed toward already-easy test cases, rather than being uniformly distributed over tests of different learning value.

C Case Study: Partial-Success Biases

This section presents two rollout-level case studies comparing test-suite-level outcome rewards, difficulty-weighted partial-success rewards (Diff), and VeRPO. Following Section 4.2, Diff uses

the uncalibrated difficulty-aware weight $w_j = \exp(-\alpha\rho_j)$, while VeRPO uses the density-calibrated weight $w'_j = \exp(-\alpha\rho_j)/(\hat{N}(\rho_j) + \delta)$. We set $\alpha = 2$ for both Diff and VeRPO, and use the adaptive KDE bandwidth $\sigma = \mathrm{std}(\{\rho_j\}_{j=1}^{|\mathcal{U}_x|})/2$. The outcome-driven reward is 1 if all tests pass and 0 otherwise. Candidate A and Candidate B are two representative members of the same group.

C.1 Correcting Easy-Test Cardinality Bias

Consider a rollout group where the test suite contains many already easy tests and only a few frontier tests. Candidate A passes many easy tests but none of the frontier tests, whereas Candidate B passes fewer easy tests but solves several frontier tests. Since both candidates still fail the complete test suite, outcome-driven rewards cannot distinguish them. Table 5 shows a concrete toy instance with 100 easy tests with empirical pass rate $\rho = 0.95$ and 10 frontier tests with $\rho = 0.20$. Candidate A passes 90 easy tests and 0 frontier tests, while Candidate B passes 50 easy tests and 5 frontier tests.

Set	#	ρ	A pass	B pass	w_j (Diff)	w'_j (VeRPO)
Easy	100	0.95	90	50	0.150	0.0015
Frontier	10	0.20	0	5	0.670	0.067

Reward	$R(A)$	$R(B)$
Outcome-driven	0.00	0.00
Diff	13.50	10.85
VeRPO	0.14	0.41

Table 5: Case 1 setup and resulting rewards. Weights are test-level and shared across candidates.

This example illustrates the cardinality bias analyzed in Section 4.1. Although Diff assigns smaller per-test weights to easy tests, their large quantity still makes Candidate A receive a higher reward than Candidate B. VeRPO instead normalizes test contributions by the local empirical density $\hat{N}(\rho_j)$, reducing the aggregate dominance of easy tests and correctly assigning a higher reward to the candidate that makes frontier progress.

C.2 Amplifying Sparse Frontier Progress

The second case studies a subtler failure mode. Candidate A and Candidate B pass the same large set of easy tests, but Candidate B additionally passes a frontier test. Both candidates still fail the complete suite, so outcome-driven rewards again provide no distinction. Although Diff increases Candidate B's raw reward, the group-mean baseline can still be dominated by the easy-test mass, yielding a weak or even negative relative advantage.

Table 6 shows an illustrative group with 100 easy tests of empirical pass rate $\rho = 0.95$ and one frontier test of empirical pass rate $\rho = 0.20$. Candidate A and Candidate B both pass 90 easy tests, while Candidate B additionally passes the frontier test.

Set	#	ρ	A pass	B pass	w_j (Diff)	w'_j (VeRPO)
Easy	100	0.95	90	90	0.150	0.0015
Frontier	1	0.20	0	1	0.670	0.670

Reward	$R(A)$	$R(B)$	$\bar{R}$	$\mathrm{Adv}(A)$	$\mathrm{Adv}(B)$
Outcome-driven	0.00	0.00	0.00	0.00	0.00
Diff	13.50	14.17	14.38	-0.88	-0.21
VeRPO	0.14	0.81	0.28	-0.14	0.53

Table 6: Case 2 setup and resulting advantages. The group mean $\bar{R}$ is computed over the full group. Weights are test-level and shared across candidates.

This case shows that correct reward ranking alone is insufficient for group-based RL; what matters is the relative advantage after subtracting the group-mean baseline. Under Diff, the easy-test mass inflates $\bar{r}^{\mathrm{turn}}$. Candidate B thus still receives a negative advantage despite passing the frontier test. VeRPO reduces the common easy-test background through density calibration, turning the same frontier progress into a positive and substantially stronger advantage.

D Detailed Proofs

D.1 Derivation of the Integral Form in Eq. (6)

In this section, we provide the rigorous derivation to transform the discrete summation $\bar{r}^{\mathrm{turn}} = \sum_{j=1}^{|\mathcal{U}_x|} w_j \rho_j$ into the integral form.

Since multiple test cases may share the same empirical pass rate ρ but possess different weights w_j, we first group the summation by the unique values of ρ. Let $\mathcal{V}_\rho \subset [0, 1]$ be the set of all unique empirical pass rates in $\mathcal{U}_x$. For a given pass rate $\rho \in \mathcal{V}_\rho$, we define its corresponding index subset as:

$$\mathcal{I}(\rho) = \{j \in \{1, \ldots, |\mathcal{U}_x|\} \mid \rho_j = \rho\}. \quad (15)$$

Let $N_\rho = |\mathcal{I}(\rho)|$ be the number of test cases with pass rate ρ. We define the conditional average weight for the subset $\mathcal{I}(\rho)$ as:

$$\bar{w}(\rho) = \frac{1}{N_\rho} \sum_{j \in \mathcal{I}(\rho)} w_j. \quad (16)$$

Thus, the sum of weights for a specific pass rate ρ is equivalent to $N_\rho \bar{w}(\rho)$. We can rewrite Eq. (5) by summing over the unique values in $\mathcal{V}_\rho$:

$$\sum_{j=1}^{|\mathcal{U}_x|} w_j \rho_j = \sum_{\rho \in \mathcal{V}_\rho} \left(\sum_{j \in \mathcal{I}(\rho)} w_j \right) \rho = \sum_{\rho \in \mathcal{V}_\rho} \bar{w}(\rho) N_\rho \rho. \tag{17}$$

To generalize this into an integral over the continuous interval $[0, 1]$, we define the unnormalized empirical test density using the Dirac delta function $\delta(\cdot)$:

$$N(\rho) = \sum_{j=1}^{|\mathcal{U}_x|} \delta(\rho - \rho_j). \tag{18}$$

Notice that integrating $N(\rho)$ over an infinitesimally small interval containing $\rho^* \in \mathcal{V}_\rho$ yields exactly the count N_{ρ^*}. Using the sifting property of the Dirac delta function, we evaluate the integral:

$$\int_0^1 \bar{w}(\rho)\, \rho\, N(\rho)\, d\rho = \int_0^1 \bar{w}(\rho)\, \rho \sum_{j=1}^{|\mathcal{U}_x|} \delta(\rho - \rho_j)\, d\rho. \tag{19}$$

Interchanging the integral and the summation gives:

$$\sum_{j=1}^{|\mathcal{U}_x|} \int_0^1 \bar{w}(\rho)\, \rho\, \delta(\rho - \rho_j)\, d\rho = \sum_{j=1}^{|\mathcal{U}_x|} \bar{w}(\rho_j) \rho_j. \tag{20}$$

Grouping this final summation back into the unique subsets $\mathcal{V}_\rho$, we obtain:

$$\begin{aligned} \sum_{\rho^* \in \mathcal{V}_\rho} \sum_{j \in \mathcal{I}(\rho^*)} \bar{w}(\rho^*) \rho^* &= \sum_{\rho^* \in \mathcal{V}_\rho} N_{\rho^*} \bar{w}(\rho^*) \rho^* \\ &= \sum_{\rho^* \in \mathcal{V}_\rho} \sum_{j \in \mathcal{I}(\rho^*)} w_j \rho^* \\ &= \sum_{j=1}^{|\mathcal{U}_x|} w_j \rho_j. \end{aligned}$$

This completes the derivation, proving that $\bar{r}^{\,\text{turn}} = \int_0^1 \bar{w}(\rho)\, \rho\, N(\rho)\, d\rho$ is mathematically equivalent to Eq. (5).

D.2 Formal Analysis of the Cardinality Effect

Building upon the integral form derived in Eq. (6), we partition the test cases by a mastery threshold $\epsilon \in (0, 1)$. Specifically, the discrete sets for the conquered tests $\mathcal{U}_\mathcal{C} = \{u_j \in \mathcal{U}_x : \epsilon \le \rho_j \le 1\}$ and the frontier tests $\mathcal{U}_\mathcal{F} = \{u_j \in \mathcal{U}_x : 0 \le \rho_j < \epsilon\}$ correspond to the continuous integration domains $[\epsilon, 1]$ and $[0, \epsilon)$ over the empirical density $N(\rho)$, respectively.

Let the cardinalities of the two sets be defined via the density function as:

$$N_\mathcal{C} = \int_\epsilon^1 N(\rho)\, d\rho, \quad N_\mathcal{F} = \int_0^\epsilon N(\rho)\, d\rho.$$

Accordingly, their respective contributions to the group-mean baseline can be expressed as:

$$\begin{aligned} R_\mathcal{C} &= \int_\epsilon^1 \bar{w}(\rho)\, \rho\, N(\rho)\, d\rho, \\ R_\mathcal{F} &= \int_0^\epsilon \bar{w}(\rho)\, \rho\, N(\rho)\, d\rho. \end{aligned}$$

Theorem D.1 (Contribution-Gap Bound over Test Subsets)**.** *Assume the test weights satisfy* $0 < W_{\min} \le w_j \le W_{\max} < \infty$ *for all* $j \in \{1, \ldots, |\mathcal{U}_x|\}$*, which implies* $W_{\min} \le \bar{w}(\rho) \le W_{\max}$ *for any* ρ *where* $N(\rho) > 0$*. Let* $\Delta R = R_\mathcal{C} - R_\mathcal{F}$ *denote the contribution gap between the conquered test set and the frontier test set. Then,*

$$\epsilon\big(W_{\min} N_\mathcal{C} - W_{\max} N_\mathcal{F}\big) \le \Delta R \le W_{\max} N_\mathcal{C}. \tag{21}$$

Proof. We first establish the lower bound for ΔR by bounding $R_\mathcal{C}$ and $R_\mathcal{F}$ separately. For $R_\mathcal{C}$, the integration domain is $[\epsilon, 1]$. Since $\rho \ge \epsilon$, $\bar{w}(\rho) \ge W_{\min}$ over this domain, and $N(\rho) \ge 0$, we have:

$$\begin{aligned} R_\mathcal{C} &= \int_\epsilon^1 \bar{w}(\rho)\, \rho\, N(\rho)\, d\rho \\ &\ge \int_\epsilon^1 W_{\min}\, \epsilon\, N(\rho)\, d\rho \\ &= \epsilon W_{\min} \int_\epsilon^1 N(\rho)\, d\rho = \epsilon W_{\min} N_\mathcal{C}. \end{aligned} \tag{22}$$

For $R_\mathcal{F}$, the integration domain is $[0, \epsilon)$. Here, $\rho < \epsilon$ and $\bar{w}(\rho) \le W_{\max}$. Thus:

$$\begin{aligned} R_\mathcal{F} &= \int_0^\epsilon \bar{w}(\rho)\, \rho\, N(\rho)\, d\rho \\ &\le \int_0^\epsilon W_{\max}\, \epsilon\, N(\rho)\, d\rho \\ &= \epsilon W_{\max} \int_0^\epsilon N(\rho)\, d\rho = \epsilon W_{\max} N_\mathcal{F}. \end{aligned} \tag{23}$$

Subtracting Eq. (23) from Eq. (22) yields the lower bound:

$$R_\mathcal{C} - R_\mathcal{F} \ge \epsilon\big(W_{\min} N_\mathcal{C} - W_{\max} N_\mathcal{F}\big).$$

Next, we establish the upper bound. For $R_{\mathcal{C}}$, since $\rho \leq 1$ on $[\epsilon, 1]$ and $\bar{w}(\rho) \leq W_{\max}$, we have:

$$\begin{aligned} R_{\mathcal{C}} &= \int_{\epsilon}^{1} \bar{w}(\rho)\, \rho\, N(\rho)\, d\rho \\ &\leq \int_{\epsilon}^{1} W_{\max} \cdot 1 \cdot N(\rho)\, d\rho \\ &= W_{\max} \int_{\epsilon}^{1} N(\rho)\, d\rho = W_{\max} N_{\mathcal{C}}. \end{aligned} \tag{24}$$

Furthermore, since $\bar{w}(\rho) \geq W_{\min} > 0$, $\rho \geq 0$, and $N(\rho) \geq 0$ over the domain $[0, \epsilon)$, the integral for the frontier set is non-negative:

$$R_{\mathcal{F}} = \int_{0}^{\epsilon} \bar{w}(\rho)\, \rho\, N(\rho)\, d\rho \geq 0. \tag{25}$$

Therefore, we conclude:

$$R_{\mathcal{C}} - R_{\mathcal{F}} \leq W_{\max} N_{\mathcal{C}} - 0 = W_{\max} N_{\mathcal{C}}.$$

This proves the upper bound of Eq. (21) and completes the proof. □

Theorem D.1 shows that the contribution gap is explicitly controlled by the cardinalities $N_{\mathcal{C}}$ and $N_{\mathcal{F}}$. In the skewed regime observed in practice, where conquered tests are far more numerous than frontier tests, i.e., $N_{\mathcal{C}} \gg N_{\mathcal{F}}$, the contribution gap is governed primarily by the size of the conquered partition. This formalizes the cardinality effect: even when per-test weights are introduced, the optimization baseline can be dominated by the sheer number of already-conquered tests unless this density effect is explicitly calibrated.

D.3 Full Optimization Objective of VeRPO

Given the unified turn-level advantage $A(\tau_i, t)$ defined in Section 4, VeRPO optimizes the policy with a clipped group-based objective.

$$\begin{aligned} \mathcal{J}_{\text{VeRPO}}(\theta) = \mathbb{E}_{\substack{x \sim \mathcal{X} \\ \{\tau_i\}_{i=1}^{N} \sim \pi_{\theta_{\text{old}}}}} \Bigg[\frac{1}{\sum_{i=1}^{N} |\tau_i|} \sum_{i=1}^{N} \sum_{t=1}^{|\tau_i|} \\ \min\Bigg(\psi_\theta(y_t^{(i)}) A(\tau_i, t), \text{clip}\Big(\psi_\theta(y_t^{(i)}), \\ 1 - \epsilon_{\text{low}}, 1 + \epsilon_{\text{high}} \Big) A(\tau_i, t) \Bigg) \Bigg]. \end{aligned} \tag{26}$$

where $\psi_\theta(y_t^{(i)}) = \frac{\pi_\theta(y_t^{(i)} | s_t^{(i)})}{\pi_{\theta_{\text{old}}}(y_t^{(i)} | s_t^{(i)})}$ is the importance sampling ratio, and ϵ_{low} and ϵ_{high} are clipping thresholds.

E Training and Evaluation Details

All experiments in this paper are implemented on top of the veRL framework (Sheng et al., 2025) and conducted on 8× NVIDIA H800 GPUs. For fair comparison, all methods use the same training and evaluation configuration unless otherwise specified. Source code will be made publicly available upon acceptance.

Training Configuration. We utilize a rollout batch size of 32 code problems, with 10 responses sampled per problem. We set the maximum response length to 16384, with sampling parameters temperature = 1.0, top-p = 1.0 and top-k = -1.0. The policy learning rate is fixed at 1×10^{-6}. We set clipping parameters ϵ_{low} = 0.2 and ϵ_{high} = 0.28. All RL methods omit the KL divergence penalty to foster exploration. For VeRPO, the difficulty sensitivity coefficient α is set to 2.0, the decay factor γ is set to 0.95, and the advantage balancing coefficient β is set to 1.0 without additional tuning.

Evaluation Configuration. Following (Yang et al., 2025), we set sampling parameters to temperature 0.6, top-p 0.95, and top-k 20. The maximum response length is set to 16384. To rigorously evaluate functional correctness, we adopt the unbiased estimator for pass@k proposed by Chen et al., 2021, which calculates the expected probability that at least one of the top-k generated code solutions passes the unit tests, defined as:

$$\text{pass@}k := \mathbb{E}_{\text{problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right],$$

where n denotes the number of samples generated per problem, and c is the number of samples that pass all unit tests. In our experiments, we generate $n = 8$ candidate solutions for each problem to robustly estimate the pass@1 metric.

F Training Analysis

F.1 Training Stability Analysis

In Section 5.2.1, we observed that the RM-based baseline, AceCoder (MT), suffers from training collapse in multi-turn settings. Here, we provide a detailed visualization and analysis of this phenomenon.

The evolution of rollout performance is visualized in Figure 6, which tracks the online average pass rate measured directly on the on-policy rollout batches. While all methods show comparable

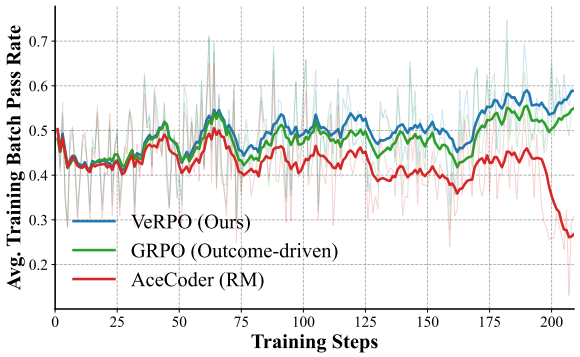


Figure 6: The average ground-truth pass rate computed on the online training rollout batches.

performance in early stages, the RM-based AceCoder fails to exhibit a distinct upward trend and succumbs to catastrophic collapse after approximately 190 training steps. In contrast, VeRPO and the outcome-driven GRPO, which are both fully grounded in verifiable execution feedback, maintain robust stability and improvement. This phenomenon highlights a practical vulnerability of relying on fixed reward models for code generation optimization. Since it is impractical for learned RMs to exhaustively cover the infinite semantic space of complex programs, they inevitably yield misaligned signals for out-of-distribution responses, thereby destabilizing the optimization process. Conversely, signals derived from verifiable execution feedback are inherently immune to such distribution shifts, further underscoring the necessity of VeRPO's design to generate dense learning signals fully grounded in execution feedback.

F.2 Training Dynamics

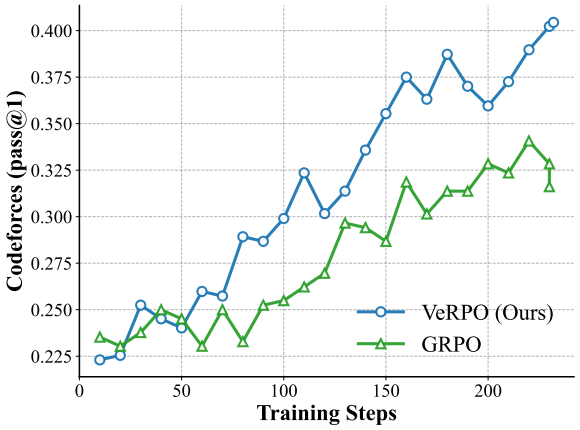


Figure 7: Training dynamics on Codeforces (pass@1).

We analyze model evolution during training by tracking the performance on Codeforces from CodeElo (Quan et al., 2025) using the unbiased pass@1 metric. As illustrated in Fig. 7, while both VeRPO (Ours) and GRPO exhibit comparable performance in the initial training phase (first 50 steps), VeRPO demonstrates significantly superior sample efficiency and convergence stability as training progresses. Starting from approximately Step 80, VeRPO establishes a distinct performance lead, maintaining a robust upward trend without the plateauing observed in the GRPO baseline. Crucially, VeRPO achieves a peak pass@1 of ∼40.0% at the end of training, surpassing GRPO (∼34.0%) by a substantial margin. This divergence highlights that VeRPO facilitates more effective policy optimization, enabling the model to explore and stabilize on higher-quality solutions given the same computational budget.

Method	HumanEval	BigCodeBench	LCB	Codeforces	Avg
	Plus	Full	V6	CodeElo	
Qwen3-4B	83.91	49.76	20.46	25.57	44.93
GRPO	90.32	59.30	30.05	29.64	52.33
VeRPO	**92.84**	**61.76**	**36.34**	**32.50**	**55.86**

Table 7: Additional main results with Qwen3-4B. All values are pass@1 scores.

Metric	Statistics
# Total Examples	7436
# Test Cases Mean ± Std. Dev.	104.08 ± 70.42
# Test Cases Range (Min – Max)	6 – 1440
# Sparse Examples (≤ 10 Tests)	804 (10.81%)

Table 8: Dataset statistics of TACO subset derived from (Luo et al., 2025). "Sparse Examples" denotes problems with limited unit tests (≤ 10 inputs).

G Additional Main Results

To further examine the robustness of VeRPO under a smaller backbone, we additionally conduct the main experiment with Qwen3-4B. As shown in Table 7, VeRPO consistently outperforms GRPO across all evaluated benchmarks, yielding an average absolute improvement of +3.53 points in pass@1. These results provide additional evidence that the benefit of VeRPO is not specific to the Qwen3-8B backbone used in the main experiments.

H Dataset Details

We utilize a filtered subset of the TACO dataset derived from (Luo et al., 2025). The dataset comprises 7436 algorithmic problems, each associated with a varying number of unit tests. The statistical distribution of these test cases is summarized in Table 8.