RECOVERING VERIFIABLE PROCESS CREDIT IN REINFORCEMENT LEARNING FOR CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Group-based reinforcement learning with verifiable rewards (RLVR) and its recent advances train a code model by examining the outcomes of sampled rollout programs against a suite of test cases. It thus suffers from the sparse-reward issue: a reward is credited at the level of the whole program. In contrast to this coarse-grained perspective, we view a program as a composition of syntax-tree units and find that even a failing program often contains units that already make tests pass, and that its failure can be traced to a few specific units. Based on the findings, we propose `ASTrace`, an abstract syntax tree (AST) reward attribution method that assigns process credit to program units from code execution. `ASTrace` parses each sampled program into units, namely its functions, methods, and top-level blocks, and credits them through two components: (i) *progress credit*, which, for all-fail rollouts, credits each unit with its marginal gain in passing tests; and (ii) *failure localization*, which, in partial-fail rollouts, penalizes the unit of a failing rollout where a missed test raises its error. We compare `ASTrace` against GRPO and its most recent variants across six code-generation benchmarks and two backbones. `ASTrace` consistently improves pass@1 over GRPO on both common coding benchmarks (*e.g.* 2.2% on HumanEval with Qwen3-4B) and more challenging coding contests (*e.g.* 2.6% on OJBench with Qwen3-4B). Our analysis shows that `ASTrace` turns near-miss programs into solutions and gains the most where failing programs are partially correct. Codes are available at `https://anonymous.4open.science/r/ASTrace-637F/`.

1 INTRODUCTION

Reinforcement learning with verifiable rewards (RLVR) has become a standard recipe for post-training code models, since it uses a suite of test cases to check each sampled program without a learned judge (Zeng et al., 2025; Gehring et al., 2024). Group-relative methods such as GRPO (Shao et al., 2024; DeepSeek-AI, 2025) make the recipe critic-free: each problem is sampled multiple times, each rollout is labeled by whether it passes the full suite, and its advantage measures how much better its label is than the group average. The signal, therefore, exists only where a group disagrees: when every rollout fails, all advantages are zero, and the problem contributes nothing to the gradient. Rollout groups without a reward signal are common in code RL, where over half of binary GRPO groups on competitive programming problems have zero rewards (Wang et al., 2026a).

Current approaches recover the reward from these zero-reward rollouts in two ways. Some give rewards if the rollout passes a fraction of cases (Wang et al., 2026a;b; Li et al., 2026), or judge from a learned outcome reward model (Le et al., 2022; Zeng et al., 2025; Zhu et al., 2026). Others propose advantage calibration, which assigns non-zero advantages to groups in which every rollout fails (Nan et al., 2025; Le et al., 2026). Their signal is defined over the whole program rather than over its units, which carry structural and execution information of their own. Finer-grained credit typically comes from learned process reward models that judge code segments by their text alone (Cui et al., 2025; Yuan et al., 2024; Dai et al., 2024). For example, the program in Figure 1(A) cleans text, counts word frequencies, and composes the two, and it passes 10 of 16 tests yet fails the problem, as do all eight samples drawn for it. A pass-rate reward or advantage calibration pushes all of its tokens in one direction, so neither shows which method makes tests pass.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

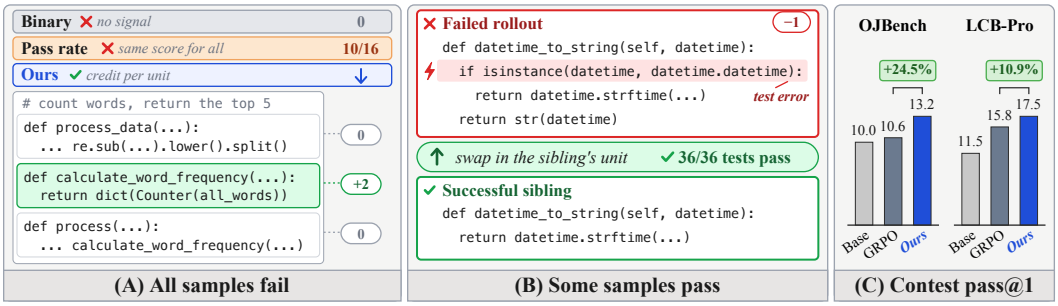


Figure 1: **What `ASTrace` recovers from failing programs.** ClassEval samples from the GRPO checkpoint, code abridged. **(A)** All eight samples fail; a binary or pass-rate reward gives the whole program one value, while `ASTrace` credits the method that makes tests pass. **(B)** A test error points to one method of the failed rollout, which `ASTrace` credits negatively; the passing sibling's method passes all 36 tests. **(C)** Qwen3-4B pass@1 on OJBench and LCB-Pro; `ASTrace` improves on GRPO by 24.5% and 10.9% relative.

We instead view a program as a composition of syntax-tree units, its functions, methods and top-level blocks, and measure by execution how each unit changes the number of passing tests. Measured this way, most failing programs are not wrong as a whole: on ClassEval, 80.5% of programs in all-fail rollouts contain a unit that makes signal tests pass, and in partial-fail rollouts, swapping one localized unit for its counterpart in a passing sibling repairs 58.6% of the failing programs that have such a counterpart. Building on this finding, we propose `ASTrace` (Abstract Syntax Tree Reward Attribution from Code Execution), which derives unit-level credit from syntax-tree analysis and execution and trains the policy on it alongside the binary outcome reward. `ASTrace` treats each failing program as a partial solution and credits the parts that already work.

`ASTrace` splits each sampled program into units along its syntax tree and credits these units whenever a group contains failed rollouts, through two components: (i) *progress credit*: when every rollout fails, `ASTrace` rewards each unit for its own contribution with a single-order Shapley estimate (Shapley, 1953): it adds the units back in one random order and credits each unit with the net change in passing tests at its step, so the credits sum exactly to the program's net gain; (ii) *failure localization*: when some rollouts pass, `ASTrace` takes the tests a failed rollout misses, locates the unit where the first such test fails from its traceback or the last program function it entered, and assigns it raw credit -1, reusing the grading output without extra execution or a reference solution.

We compare `ASTrace` with GRPO and two program-level rewards, a dense verifiable reward and a learned reward model, on Qwen3-4B (Yang et al., 2025) and Qwen2.5-Coder-7B (Hui et al., 2024) and six code generation benchmarks, evaluating with three decoding seeds. `ASTrace` improves mean pass@1 over GRPO on all six benchmarks for both backbones, and the gain extends to the competitive-programming benchmarks OJBench and LCB-Pro, including C++ on LCB-Pro, although training uses only Python. Our main contributions are:

- ⋆ We view failing programs as compositions of syntax-tree units, and formulate process credit as the execution-measured change in passing tests that each unit causes.
- ⋆ We propose `ASTrace`, whose progress credit for all-fail rollouts is a single-permutation Shapley estimate over syntax-tree units, bounded by the rollout's own achievement and delivered on a batch-normalized token channel.
- ⋆ We add failure localization for partial-fail rollouts, which locates a unit from the traceback or call trace of a test that a passing sibling solves, with no extra execution and no reference solution.
- ⋆ `ASTrace` improves mean pass@1 over GRPO on all six benchmarks across backbones, and attains the best pass@1 among trained methods on 11 of 12 benchmark–backbone pairs.
- ⋆ We show that `ASTrace` is precise and selective: swapping its localized unit repairs 58.3% of failing ClassEval programs against 13.5% for another unit, it credits about one rollout in eight, and it trains at 1.16× the GRPO wall-clock.

2 Related Work

Process and segment credit for code RL. A growing line of work assigns credit below the trajectory level for LLM reasoning and code: spans parsed from interpreter feedback (Liu et al., 2023a; Dou et al., 2024), segment and prefix values estimated by extra sampling (Guo et al., 2025; Zhou et al., 2026), token-level values from a critic trained on unit-test outcomes (Le et al., 2022), and learned step

scorers, some induced from outcome labels alone (Dai et al., 2024; Zhang et al., 2026a; 2025; Xia et al., 2026; Cui et al., 2025; Yuan et al., 2024). VPRMs (Pronesti et al., 2026) check reasoning steps with rule-based verifiers, and CodeRL+ (Jiang et al., 2025) adds an auxiliary objective on variable-level execution traces. We share their premise and differ on what the score measures: a unit's credit is the observed change in passing tests when its neighbors are ablated and the program re-executed. The estimator is unbiased for the Shapley value in expectation over permutations; separately, its credits sum exactly along each path, bounding what one rollout can earn. EGCA (Kumar et al., 2026) is the closest, putting the GRPO advantage on the earliest span diverging from a curated reference located by a debugger LLM, whereas our placement takes the reference from a passing sibling and the blamed unit from the failing test's traceback or call trace.

Dense verifiable rewards and their failure under group normalization. A second line keeps the reward verifiable but makes it denser, using partial execution success (Wang et al., 2026a;b; Zeng et al., 2025; Li et al., 2026). VeRPO (Wang et al., 2026a) calibrates per-test reward density, DHRCL (Wang et al., 2026b) composes syntax, execution, pass-rate, and structural signals under a curriculum, AceCoder (Zeng et al., 2025) trains a reward model from synthesized tests, CodeScaler (Zhu et al., 2026) scales training and inference with code reward models, and Skopin & Kotelnikov (2026) shape the unit-test reward with static analysis for small models. Advantage calibration assigns non-zero advantages to all-fail rollouts (Nan et al., 2025; Le et al., 2026), again one value per rollout. Density alone does not settle it: pass-rate rewards do not beat binary rewards in critic-free RL (Li et al., 2026), since they score the level a rollout reached, not the progress it caused; learned scorers are exploitable (Cheng et al., 2025; Tiwari et al., 2026), and Qwen backbones improve even under spurious rewards (Shao et al., 2025). Delivery compounds this: a scalar added to the return is z-scored within the group, so where all samples fail its coefficient cancels and any term constant within the group vanishes (Wang, 2026; Liu et al., 2025). In contrast, `ASTrace` measures a transition by execution, bounds it by the rollout's own achievement, and delivers it on a batch-normalized token channel that group normalization leaves intact.

Execution feedback and behavioral test selection. A third line grounds code RL in execution itself, acting on execution feedback across revision turns (Gehring et al., 2024; Ekbote et al., 2025), rewarding verifiable intermediate execution steps (Tang et al., 2026), synthesizing test cases and filtering them by behavior (Zhang et al., 2026b), or rewarding hardware descriptions with testbench feedback (Wang et al., 2025a). We likewise select signal cases by behavior, but measure progress through counterfactual ablations of a single rollout rather than across revision turns. `ASTrace` thus provides single-shot process credit even when no sibling succeeds.

3 PRELIMINARIES

Reinforcement learning with verifiable rewards for code generation. Given a problem prompt q with a hidden test suite $\mathcal{T}$, a policy π_θ samples a group of K responses $x_1, \dots, x_K \sim \pi_\theta(\cdot \mid q)$. We execute the program extracted from each response against $\mathcal{T}$ and assign the binary reward

$$y_i \;=\; \mathbb{1}[\,\text{the program in } x_i \text{ completes execution and passes all tests in } \mathcal{T}\,], \tag{1}$$

where $\mathbb{1}[\cdot]$ is the indicator; a program that fails one test earns the same as one that fails all.

Group-relative policy optimization. GRPO (Shao et al., 2024; DeepSeek-AI, 2025) replaces the learned value model of PPO with a group baseline: each response's reward is standardized within its group, and all of its tokens share the result,

$$A_{it} \;=\; A_i^{\mathrm{GRPO}} \;=\; \frac{y_i - \bar{y}}{\sigma_y + 10^{-6}}, \tag{2}$$

where $\bar{y}$ and σ_y are the mean and standard deviation of $y_1, \dots, y_K$. For a batch of B responses with prompts q_i, GRPO maximizes the token-normalized clipped surrogate (Yu et al., 2025),

$$\mathcal{J}(\theta) = \frac{1}{\sum_{i,t} m_{it}} \sum_{i=1}^{B} \sum_{t} m_{it} \min\Big(\rho_{it} A_{it},\; \mathrm{clip}\big(\rho_{it}, 1-\epsilon, 1+\epsilon\big) A_{it}\Big), \tag{3}$$

where ρ_{it} is the probability ratio of π_θ to the sampling policy $\pi_{\theta_{\mathrm{old}}}$ at token $x_{i,t}$ given q_i and the prefix $x_{i,<t}$, m_{it} is 1 for generated tokens and 0 otherwise, and we omit the KL penalty. With one gradient step per batch, $\rho_{it} = 1$ at the update and the gradient is proportional to $\sum_{i,t} m_{it} A_{it} \nabla_\theta \log \pi_\theta(x_{i,t} \mid q_i, x_{i,<t})$, so the advantage alone sets the direction and weight of each token's update. We call the

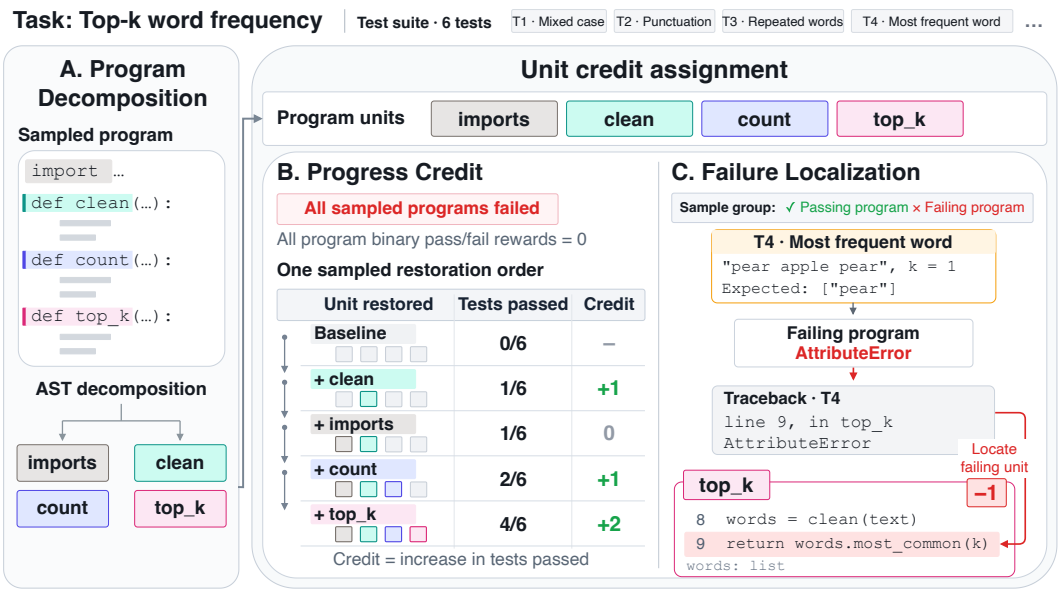


Figure 2: **Overview of `ASTrace` on a word-frequency task. (A)** The program is decomposed into syntax-based units. **(B)** In all-fail rollouts, units receive progress credit from changes in test success as they are restored in a sampled order. **(C)** In partial-fail rollouts, failure traces identify a unit for negative credit.

rollouts of a group *all-pass* if every $y_i = 1$, *all-fail* if every $y_i = 0$, and *partial-fail* otherwise. In all-pass and all-fail rollouts every advantage is zero, whatever the partial correctness of the failing programs.

4 ASTRACE: AST REWARD ATTRIBUTION FROM CODE EXECUTION

With binary rewards, GRPO assigns zero outcome advantages to all-fail rollouts and a uniform advantage across each response's tokens. To address these two limitations, `ASTrace` assigns execution feedback to program units. We measure their contributions to partial correctness in all-fail rollouts and use failure traces to place negative credit in partial-fail rollouts. We then map the unit credits to tokens and add them to the GRPO advantages, as outlined in Figure 2 and Algorithm 1.

4.1 CODE UNIT DECOMPOSITION AND ISOLATION

As illustrated in Figure 2(A), we use the abstract syntax tree (AST) to assign credit to coherent code regions. Within the program extracted from a response, we identify top-level functions, methods of top-level classes, and contiguous blocks of other top-level statements as units. To limit cost, we retain the $k_{\max} = 4$ longest units by character length and record their character spans. We index the k retained units by $\mathcal{U} = \{1, \ldots, k\}$; if parsing fails, we extract no units. We call a program SEMANTIC if it has at least two callable units (functions or methods); progress credit uses only such programs.

To isolate the contribution of code units, we construct program variants in which selected units remain active and the other retained units are disabled. Let $S \subseteq \mathcal{U}$ denote the active subset, called a coalition. For retained callables outside S, we preserve their interfaces by keeping their signatures and replacing their bodies with `raise NotImplementedError`. We remove retained blocks outside S and leave units outside $\mathcal{U}$ unchanged in every coalition, so every coalition is a program in which the unretained code runs as written and the retained callables outside S fail at their first call. For a test $\tau \in \mathcal{T}$, let $V_{i,\tau}$ denote the original program's test outcome for response x_i. For a fixed response, we write $V_\tau(S)$ for the corresponding outcome of its coalition variant. In both cases, outcomes are pass for success, fail for an assertion failure, and error for another error.

4.2 PROGRESS CREDIT IN ALL-FAIL ROLLOUTS

All-fail rollouts get no outcome signal, yet their programs can still contain code that passes individual tests; we credit the contributing units with these successes, as illustrated in Figure 2(B).

We measure partial correctness on selected tests $C \subseteq \mathcal{T}$, called signal cases, whose outcomes vary most across the group. For a fixed response, we score a coalition S by the signal cases it passes:

$$v(S) = \sum_{\tau \in C} \mathbb{1}[V_\tau(S) = \text{pass}], \qquad S \subseteq \mathcal{U}. \tag{4}$$

A restored unit $j \notin S$ then contributes $v(S \cup \{j\}) - v(S)$, the net change in passing tests.

A helper's contribution can depend on whether its caller is active. To account for such dependencies, we estimate each unit's Shapley value, its average contribution over all restoration orders (Shapley, 1953). We sample one order $\pi = (j_1, \ldots, j_k)$ uniformly and restore the units one at a time, from $S_0 = \emptyset$ to $S_\ell = \{j_1, \ldots, j_\ell\}$, crediting each unit with its marginal gain:

$$c_{j_\ell} = v(S_\ell) - v(S_{\ell-1}), \qquad \ell = 1, \ldots, k. \tag{5}$$

Sampling the order uniformly makes c_j an unbiased estimate of the Shapley value ϕ_j, $\mathbb{E}_\pi[c_j] = \phi_j$ (Castro et al., 2009; Štrumbelj & Kononenko, 2010). For each order, we also distribute the net score change exactly, $\sum_{j \in \mathcal{U}} c_j = v(\mathcal{U}) - v(\emptyset)$, avoiding repeated credit for the same improvement when units are jointly necessary. Both properties refer to the game of Eq. (4), whose players are the k retained units with the unretained code held fixed; ϕ_j measures the effect of restoring unit j's body under the ablation of §4.1. We describe test selection and execution budgets in Appendix E.

4.3 Failure Localization in Partial-Fail Rollouts

In partial-fail rollouts, the tests that distinguish passing from failing programs are a verifiable source of feedback, and we use their execution traces to locate units for negative credit, as illustrated in Figure 2(C). For a failing response x_i, the divergent set D_i holds its unsuccessful signal cases:

$$D_i = \{\tau \in C : V_{i,\tau} \neq \text{pass}\}. \tag{6}$$

A sibling in the group passes the full suite, so every test in D_i is solvable by the current policy.

We inspect tests in D_i in suite order and use the first recorded location that maps to a code unit. We take this location from the innermost traceback frame in the generated program. If an assertion in the test leaves no such frame, we use the last program function entered, as recorded by a call tracer. We assign raw credit -1 to the selected unit, using the recorded location as a proxy for the source of failure. The value is fixed because the trace identifies a location without measuring its contribution magnitude; §6 tests the location by swapping the unit for the passing sibling's. We reuse the terminal run's tracebacks and call events, so localization requires no additional executions. If no location maps to a unit, the response retains its original GRPO advantage.

4.4 Token-Level Credit Integration

We add unit credit after GRPO normalization so it can provide a signal when the outcome advantage is zero. To keep each unit's total raw credit independent of its token length, we divide it equally among the tokens covered by its character span (Appendix E). We sum contributions from overlapping spans and denote the resulting raw credit for token t in response i by $\tilde{c}_{it}$.

We standardize nonzero token credits across the batch and clip them to limit individual adjustments. Let $Z = \{(i,t) : m_{it}\tilde{c}_{it} \neq 0\}$ be the credited positions, using the generated-token mask m_{it} from §3. With μ_Z and σ_Z the mean and standard deviation of raw credits over Z, the delivered advantage is

$$A_{it} \;=\; A_i^{\text{GRPO}} \;+\; \lambda\, m_{it}\, \mathbb{1}[\tilde{c}_{it} \neq 0]\, \text{clip}\left(\frac{\tilde{c}_{it} - \mu_Z}{\sigma_Z + 10^{-6}}, -\kappa, \kappa\right), \qquad \lambda = 0.1, \quad \kappa = 1, \tag{7}$$

where λ scales the added credit, $\text{clip}(z, -\kappa, \kappa) = \max(-\kappa, \min(z, \kappa))$ bounds the adjustment to magnitude $\lambda\kappa = 0.1$, and 10^{-6} stabilizes the denominator. Uncredited tokens keep $A_{it} = A_i^{\text{GRPO}}$, and no signal is added when Z is empty or all raw credits are equal. The raw credits of Eq. (5) and §4.3 are the attribution, and the efficiency identity $\sum_j c_j = v(\mathcal{U}) - v(\emptyset)$ holds for them. The delivered term is an optimization signal that ranks each credited token against the batch: the token is pushed up when its raw credit exceeds μ_Z and down otherwise, so the sign of an adjustment can differ from that of its raw credit. For localization, the ranking depends on unit length: each token of a blamed unit with N_j tokens has $\tilde{c}_{it} = -1/N_j$, so if only blamed units are credited, $\mu_Z = -1/\bar{N}$ for their mean length $\bar{N}$, and the term pushes down blamed units shorter than $\bar{N}$ and pushes up longer ones. The term also cannot overturn the outcome signal: the smallest nonzero outcome advantage in a group of K=16, reached when a single rollout passes or fails, exceeds 0.2 in magnitude, so an added

Algorithm 1 One training step of `ASTrace`.

Require: groups of K responses with outcomes $V_{i,\tau}$ and traces from the terminal run; λ, κ, budget
$A_i^{\mathrm{GRPO}} \leftarrow$ Eq. (2) for every response; raw credits $\tilde{c}_{it} \leftarrow 0$
for each group, in rotation order **do**
  $C \leftarrow$ the $|C| \le 6$ tests whose outcomes vary most across the group
  if every response fails and budget remains **then** {progress credit, §4.2}
    pick one SEMANTIC response; take its $k \le k_{\max}$ units $\mathcal{U}$; sample an order π
    evaluate $v(S_\ell)$ for $0 \le \ell < k$ by execution and read $v(S_k) = v(\mathcal{U})$ from the terminal run
    $c_{j_\ell} \leftarrow v(S_\ell) - v(S_{\ell-1})$ on the span of unit j_ℓ, for $\ell = 1, \ldots, k$ Eq. (5)
  else if some responses pass **then** {failure localization, §4.3}
    for each failing response with $D_i \neq \emptyset$ (Eq. 6) **do**
      locate the unit from the first test in D_i whose trace maps to a unit; credit -1 on its span
    end for
  end if
end for
split each span's credit evenly over its tokens to obtain $\tilde{c}_{it}$; standardize, clip and add to A_i^{GRPO} Eq. (7)
return A_{it}

term of at most 0.1 never reverses its sign. In partial-fail rollouts, the term therefore reweights tokens within a failing rollout whose advantage stays negative, and in all-fail rollouts, where $A_i^{\mathrm{GRPO}} = 0$, it is the only signal.

5 EXPERIMENTS

5.1 SETUP

Models and training. We train `Qwen3-4B-Instruct-2507` (Yang et al., 2025) and `Qwen2.5-Coder-7B-Instruct` (Hui et al., 2024) with veRL (Sheng et al., 2024) in two stages of 200 steps: the first on 16,000 KodCode-V1 problems (Xu et al., 2025), the second on a mixture of the same size that adds competitive-programming problems (Appendix B). Each step samples 32 problems with K=16 rollouts at temperature 1.0, top-p 1.0 and a 16,384-token limit, and takes one gradient step at learning rate 10^{-6} with no KL term, entropy bonus or length penalty. `ASTrace` builds on GRPO without group filtering, uses λ=0.1 and κ=1, credits at most one rollout per all-fail group, and runs at most 64 extra executions per step (Appendix A).

Baselines. We compare with three baselines, each trained from the same instruct checkpoint for 400 steps and evaluated under the same protocol. GRPO with the binary test reward is the outcome-only reference that `ASTrace` extends; since `ASTrace` changes only the advantage, this comparison isolates the effect of unit-level credit. VeRPO (Wang et al., 2026a) represents dense verifiable rewards, turning partial test success into a density-calibrated reward combined with the execution outcome. CodeScaler (Zhu et al., 2026) represents learned rewards, scoring each program with a trained code reward model. Both give all-fail rollouts a signal by scoring the whole program, so comparing with them tests whether the placement of credit matters beyond its density. Base denotes each released instruct checkpoint without our training.

Benchmarks and evaluation. We evaluate on HumanEval and HumanEval+ (Chen et al., 2021; Liu et al., 2023b), BigCodeBench Full and Hard (Zhuo et al., 2024), LiveCodeBench v6 (Jain et al., 2024), and ClassEval (Du et al., 2023). For each step-400 checkpoint, we draw eight samples per problem at each of the decoding seeds 42, 43 and 44 (temperature 1.0, top-p 1.0, 16,384 output tokens, 32,768-token context) and report unbiased pass@1, pass@5 and pass@8, averaged over problems and seeds. Each method has one training run per backbone, so the seeds measure sampling variability at a fixed checkpoint; Base uses seed 42. We also evaluate them on the competitive-programming benchmarks OJBench (Wang et al., 2025b) and LCB-Pro (Zheng et al., 2026) (Appendix C).

5.2 MAIN RESULTS

Unit-level credit improves pass@1 on both backbones. Table 1 shows that `ASTrace` improves over GRPO on all six benchmarks for both backbones and has the highest mean pass@1 among trained methods, leading in 11 of the 12 benchmark–backbone pairs. The program-level baselines help on specific benchmarks: on Qwen2.5-Coder-7B, VeRPO and CodeScaler improve BigCodeBench over GRPO and CodeScaler leads BigCodeBench Full, but neither exceeds GRPO's mean on either backbone. We attribute this difference to what each signal changes. VeRPO and CodeScaler modify

Table 1: **pass@1 (%) on six code generation benchmarks.** Trained methods report the mean and sample standard deviation over three decoding seeds of one checkpoint; Base is the released instruct checkpoint at one seed. Avg. averages the six benchmarks. Bold and underline mark the best and second-best trained method per backbone.

Method	HumanEval	HumanEval+	BCB Full	BCB Hard	LCB v6	ClassEval	Avg.
Qwen3-4B-Instruct-2507							
Base	89.10	84.60	39.79	19.43	30.79	26.12	48.31
GRPO	91.06±0.09	<u>86.26</u>±0.22	<u>40.03</u>±0.14	<u>20.27</u>±0.37	<u>38.81</u>±0.15	<u>31.29</u>±0.64	<u>51.29</u>
VeRPO	<u>91.44</u>±0.12	86.08±0.45	39.99±0.12	19.85±0.42	36.29±0.45	31.17±0.56	50.80
CodeScaler	80.01±0.38	76.68±0.42	37.43±0.29	13.96±0.59	33.90±0.23	28.46±0.29	45.07
ASTrace	**93.27**±0.23	**88.41**±0.20	**40.17**±0.11	**22.89**±0.53	**40.67**±0.11	**32.92**±0.56	**53.05**
Δ vs. GRPO	+2.21	+2.15	+0.14	+2.62	+1.86	+1.63	+1.76
Qwen2.5-Coder-7B-Instruct							
Base	84.15	80.18	31.14	12.67	18.07	27.75	42.33
GRPO	<u>87.70</u>±0.22	<u>83.33</u>±0.18	37.38±0.19	18.02±0.21	23.17±0.15	<u>31.71</u>±0.29	<u>46.89</u>
VeRPO	86.81±0.08	82.77±0.46	38.39±0.07	18.64±0.20	<u>23.55</u>±0.23	30.88±0.12	46.84
CodeScaler	86.81±0.08	83.00±0.20	**39.94**±0.04	<u>19.06</u>±0.38	19.79±0.75	30.50±0.75	46.52
ASTrace	**87.80**±0.08	**83.92**±0.33	<u>38.79</u>±0.25	**19.17**±0.34	**23.62**±0.22	**33.33**±0.14	**47.77**
Δ vs. GRPO	+0.10	+0.59	+1.41	+1.15	+0.45	+1.62	+0.88

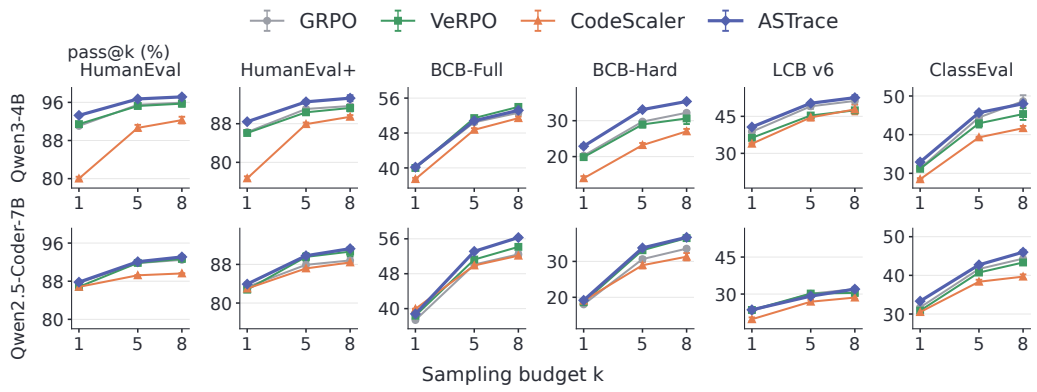


Figure 3: **Performance across sampling budgets.** pass@1, pass@5 and pass@8 at step 400 on all six benchmarks. Rows show Qwen3-4B and Qwen2.5-Coder-7B. Points are means over three decoding seeds of each fixed checkpoint; error bars show the across-seed sample standard deviation.

the terminal reward, so every rollout is also scored on partial success or on a learned judgment, whose agreement with full-suite correctness varies across tasks and models. ASTrace keeps the binary reward as the objective and adds a bounded term on units, which gives failing rollouts a direction toward the units that change test outcomes without changing what is rewarded.

The gains persist at larger sampling budgets. Figure 3 extends the comparison to pass@5 and pass@8 (Appendix G). On Qwen3-4B, ASTrace stays above GRPO at both budgets on every benchmark except ClassEval at pass@8. On Qwen2.5-Coder-7B, all four trained methods fall below Base at pass@8 on four benchmarks, the trade of coverage for reliability reported for RLVR (Yue et al., 2025). ASTrace gives up the least of this coverage, with the highest pass@8 among trained methods on five benchmarks and a tie with GRPO on LiveCodeBench, where it trails at pass@5. Progress credit gives the policy a direction on problems it does not yet solve, where binary GRPO has none, which is consistent with this smaller loss of coverage.

The advantage extends to competitive programming and to C++. Figure 4 evaluates Qwen3-4B on OJBench (Wang et al., 2025b), with Python problems from NOI and ICPC contests, and on LCB-Pro, whose public tests require C++. ASTrace improves pass@1 over GRPO on both, as well as pass@5 and pass@8, and on OJBench the same holds for Qwen2.5-Coder-7B (Appendix D). Training uses only Python and the attribution operates on the Python syntax tree, so the C++ gain comes from the trained policy. The gains come from the easy and medium OJBench problems and the easy LCB-Pro problems.

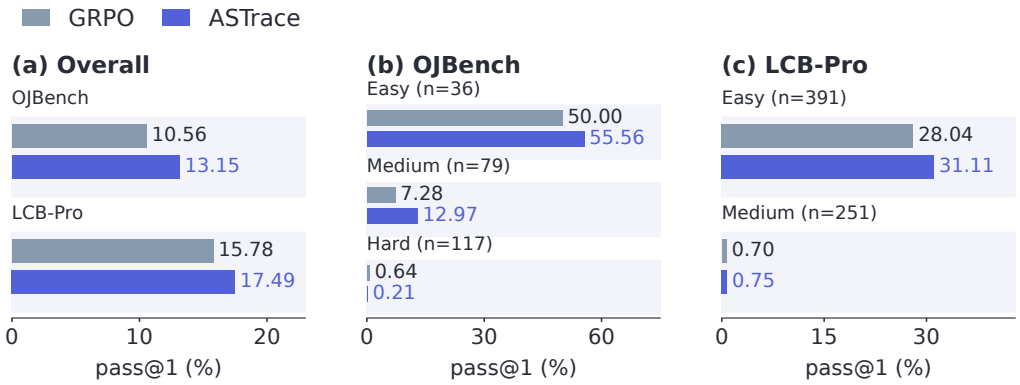


Figure 4: **Overall performance and difficulty slices on challenging benchmarks.** (a) Aggregate pass@1. (b–c) pass@1 by difficulty; n is the number of problems. OJBench is in Python, LCB-Pro in C++.

Table 2: **Component ablation and credit placement.** Qwen3-4B pass@1 (%). P: progress credit; L: failure localization. Green and red mark scores above and below GRPO.

(a) Component ablation

Variant	P	L	LCB v6	ClassEval
GRPO	✗	✗	38.86	32.00
Progress credit	✓	✗	37.36	30.75
Localization	✗	✓	39.64	31.63
`ASTrace`	✓	✓	**40.64**	**32.88**

(b) Credit placement

Placement	LCB v6	ClassEval
GRPO (no credit)	38.86	32.00
Shuffled units	36.64	31.13
Uniform tokens	40.57	32.38
`ASTrace` (localized)	**40.64**	**32.88**

5.3 Ablation Study

The two components are complementary. Progress credit and failure localization act on different rollouts, all-fail and partial-fail, so each could contribute on its own or depend on the other. To separate them, we train each component alone once on Qwen3-4B under the main configuration and evaluate it at decoding seed 42, as shown in Table 2(a). We find that the components are strongest together: their combination improves both benchmarks, localization alone improves LiveCodeBench, and progress credit alone stays below GRPO. The delivery rule of Eq. (7) suggests why. With localization present, the batch mean μ_Z is set mostly by the negative credits of blamed units, so units with positive progress credit lie above it and are pushed up; standardized only against each other, progress credits push down every token below their own mean, even when its unit made tests pass.

Credit on the wrong unit hurts. Because both components add a bounded term to the advantage, part of their benefit could come from the extra signal itself rather than from where it lands. We therefore keep the computed credit and change only its destination, as shown in Table 2(b): *shuffled units* permutes the raw credits among the eligible units of each rollout, and *uniform tokens* spreads their sum evenly over the response, both under the same budget and normalization. All three variants keep each rollout's total raw credit, so their differences reflect placement alone. Shuffling the credits moves both benchmarks below GRPO, since credit on the wrong unit rewards code that does not pass tests and penalizes code that does. Spreading the credit uniformly keeps the direction of each rollout and improves over GRPO on both benchmarks, trailing the localized placement of `ASTrace` by less than the across-seed standard deviations of Table 1.

6 Analysis

Both components write credit where a change in the program changes test outcomes. To inspect what each route writes, we replay both on ClassEval samples in Figure 5, from all-fail rollouts in (a) and partial-fail rollouts in (b), and find that the credit follows the tests. In (a), the program passes all six signal cases with every unit active and none with all units disabled, although its outcome reward is zero, and restoring the units in the sampled order credits `construct_stop_word_list` with one test, `process` with none and `remove_stop_words` with five. `process` receives zero because its callee `remove_stop_words` is still disabled when it is restored; the tests go to whichever unit completes the call chain, and the Shapley value averages over all $3! = 6$ orders. In (b),

the missed test raises its error inside `condition_judge`, which receives -1 while the other units receive nothing. The strips under each panel spread each credit evenly over its unit's tokens, keeping a unit's total raw credit independent of how many tokens the unit spans.

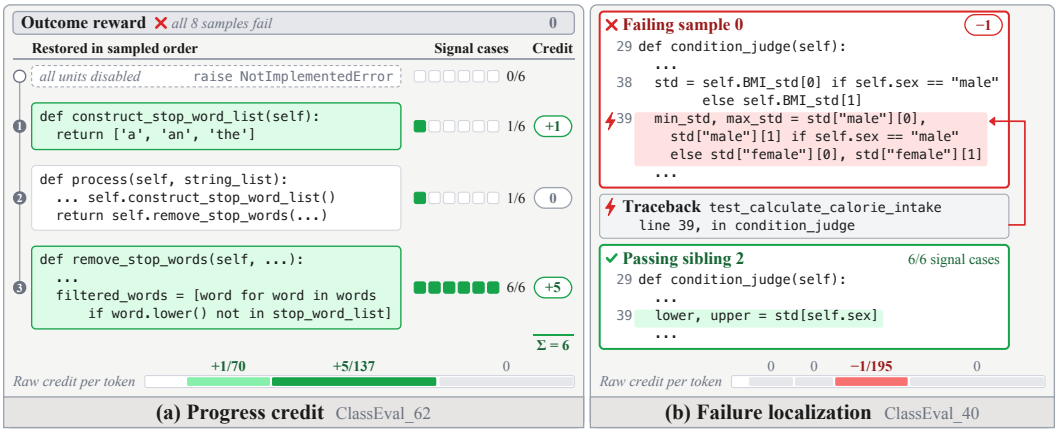


Figure 5: **Process credit on two ClassEval rollouts. (a)** Restoring units yields progress credit. **(b)** A failing test's traceback localizes negative credit.

Table 3: **Mechanism diagnostics.** Left: probes on GRPO samples (Appendix H); net progress means $v(\mathcal{U}) > v(\emptyset)$ for the credited program, and a swap replaces one unit with its counterpart in a passing sibling, on the same 96 failures for both rows. Right: the training steps of Qwen3-4B.

Offline probe	Benchmark	Rate	Training statistic	Value
All-fail rollouts with net progress	ClassEval	42/56 (75.0%)	All-fail / partial-fail rollouts	21.8% / 33.0%
All-fail rollouts with net progress	LCB v6	29/40 (72.5%)	Rollouts routed to localization / progress	11.3% / 1.3%
Swapping the localized unit repairs	ClassEval	56/96 (58.3%)	Extra executions used, of the cap	47.5%
Swapping an unblamed unit repairs	ClassEval	13/96 (13.5%)	Wall-clock relative to GRPO	1.16×

The localized unit is usually the one whose repair fixes the program. A trace records where a test's failure surfaces, and negative credit is most informative when that unit is also where the program can be fixed. To test this, we swap the localized unit of each failing ClassEval sample for the passing sibling's version and rerun the suite, with a swap of an unblamed, signature-compatible unit as the control, as shown in the left half of Table 3. We find that the localized swap repairs 58.3% of the failures against 13.5% for the control, so localization finds the unit that separates a failing program from a passing one about four times as often.

Process credit is sparse and adds little cost. Executing ablated programs during training is practical when the attribution runs selectively and reuses what grading already computes. We record how often each route fires during Qwen3-4B training and what it costs, as shown in the right half of Table 3. We find that localization covers most credited rollouts, since every failing sample of partial-fail rollouts is eligible and needs no execution, while progress credit serves one rollout per all-fail group and reaches 1.3% of rollouts, near its ceiling of about 1.4%. The attribution uses under half of the execution cap, and reusing the terminal run keeps training at $1.16\times$ the GRPO wall-clock (Appendix A). Removing this small share lowers pass@1 (Table 2(a)), since it is the only signal in its all-fail rollouts.

Most all-fail rollouts hold a partially correct program. `ASTrace` rests on the premise that failing programs are often near misses, partially correct programs that unit credit can complete. About three quarters of the probed all-fail rollouts hold a program with net progress (Table 3). On OJBench, `ASTrace` also yields 48 more accepted samples than GRPO and 120 fewer timeouts (Appendix D).

7 Conclusion

`ASTrace` credits syntax-tree units from execution through progress credit in all-fail rollouts and failure localization in partial-fail rollouts, while the outcome reward stays binary. `ASTrace` improves mean pass@1 over GRPO on all six benchmarks for both backbones, and the gain carries over to competitive programming and C++. Swapping in a passing sibling's version of its localized unit repairs a failing program about four times as often as swapping another unit, and `ASTrace` credits about one rollout in eight at $1.16\times$ the GRPO wall-clock. Appendix I discusses limitations.

AI USE STATEMENT

To enhance the clarity and readability of this manuscript, gpt-5.6 was utilized exclusively as a language polishing tool. Its role was strictly confined to proofreading and grammatical correction. gpt-5.6 did not contribute to the generation of any scientific content, experimental design, or new ideas presented in the paper. Its usage is consistent with standard practices for manuscript preparation and did not influence the research itself.

REFERENCES

Javier Castro, Daniel Gómez, and Juan Tejada. Polynomial calculation of the Shapley value based on sampling. *Computers & Operations Research*, 36(5):1726–1730, 2009.

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

Jie Cheng, Gang Xiong, Ruixi Qiao, Lijun Li, Chao Guo, Junle Wang, Yisheng Lv, and Fei-Yue Wang. Stop summation: Min-form credit assignment is all process reward model needs for reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2504.15275.

Ganqu Cui, Lifan Yuan, Zefan Wang, et al. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*, 2025.

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

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, et al. StepCoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*, 2024.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, et al. ClassEval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

Chanakya Ekbote, Vijay Lingam, Sujay Sanghavi, Jun Huan, Behrooz Omidvar-Tehrani, Anoop Deoras, and Stefano Soatto. MURPHY: Feedback-aware GRPO with retrospective credit assignment for multi-turn code generation. *arXiv preprint arXiv:2511.07833*, 2025.

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

Yiran Guo, Lijie Xu, Jie Liu, Dan Ye, and Shuang Qiu. Segment policy optimization: Effective segment-level credit assignment in RL for large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2505.23564.

Binyuan Hui et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

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

Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng, Tian Wang, Yongding Tao, Rongyu Cao, Binhua Li, Zhi Jin, Wenpin Jiao, Fei Huang, Yongbin Li, and Ge Li. CodeRL+: Improving code generation via reinforcement with execution semantics alignment. *arXiv preprint arXiv:2510.18471*, 2025.

Abhijit Kumar, Natalya Kumar, and Shikhar Gupta. Execution-grounded credit assignment for GRPO in code generation. *arXiv preprint arXiv:2603.16158*, 2026. ICLR 2026 Workshop on Scaling Post-Training for LLMs (SPOT).

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 2022.

Thanh-Long V. Le, Myeongho Jeon, Kim Vu, Viet Lai, and Eunho Yang. No prompt left behind: Exploiting zero-variance prompts in LLM reinforcement learning via entropy-guided advantage shaping. In *International Conference on Learning Representations*, 2026.

Xin-Ye Li, Ren-Biao Liu, Yun-Ji Zhang, Hui Sun, Zheng Xie, and Ming Li. Exploring pass-rate reward in reinforcement learning for code generation. *arXiv preprint arXiv:2605.02944*, 2026.

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023b.

Zichen Liu, Changyu Chen, et al. Understanding R1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Gongrui Nan, Siye Chen, Jing Huang, Mengyu Lu, Dexun Wang, Chunmei Xie, Weiqi Xiong, Xianzhou Zeng, Qixuan Zhou, Yadong Li, and Xingzhong Xu. NGRPO: Negative-enhanced group relative policy optimization. *arXiv preprint arXiv:2509.18851*, 2025.

Massimiliano Pronesti, Anya Belz, and Yufang Hou. Beyond outcome verification: Verifiable process reward models for structured reasoning. *arXiv preprint arXiv:2601.17223*, 2026.

Rulin Shao, Shuyue Stella Li, Rui Xin, Scott Geng, Yiping Wang, Sewoong Oh, Simon Shaolei Du, Nathan Lambert, Sewon Min, Ranjay Krishna, Yulia Tsvetkov, Hannaneh Hajishirzi, Pang Wei Koh, and Luke Zettlemoyer. Spurious rewards: Rethinking training signals in RLVR. *arXiv preprint arXiv:2506.10947*, 2025.

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

Lloyd S. Shapley. A value for n-person games. In Harold W. Kuhn and Albert W. Tucker (eds.), *Contributions to the Theory of Games, Volume II*, volume 28 of *Annals of Mathematics Studies*, pp. 307–317. Princeton University Press, 1953.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. *arXiv preprint arXiv:2409.19256*, 2024.

Egor Skopin and Evgeny Kotelnikov. Improving small language models for code generation with reinforcement learning from verification feedback. *arXiv preprint arXiv:2605.30478*, 2026.

Erik Štrumbelj and Igor Kononenko. An efficient explanation of individual classifications using game theory. *Journal of Machine Learning Research*, 11:1–18, 2010.

Lingxiao Tang, He Ye, Zhaoyang Chu, Muyang Ye, Zhongxin Liu, Xiaoxue Ren, and Lingfeng Bao. ExecVerify: White-box RL with verifiable stepwise rewards for code execution reasoning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13850–13875. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.631.

Rishabh Tiwari, Aditya Tomar, Udbhav Bamba, Monishwaran Maheswaran, Heng Yang, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. Reward under attack: Analyzing the robustness and hackability of process reward models. *arXiv preprint arXiv:2603.06621*, 2026.

Longwen Wang, Yirui Liu, Xuan'er Wu, Xiaohui Hu, Yuankai Fan, Kaidong Yu, Qizhen Weng, Wei Xi, and Xuelong Li. Beyond binary: Turning partial success into dense verifiable rewards for reinforcement learning in code generation. *arXiv preprint arXiv:2601.03525*, 2026a.

Shuhang Wang, Ziming Li, and Hui Cheng. DHRCL: Training code LLMs with dense hierarchical rewards and curriculum learning. *arXiv preprint arXiv:2607.26457*, 2026b.

Yiting Wang, Guoheng Sun, Wanghao Ye, Gang Qu, and Ang Li. VeriReason: Reinforcement learning with testbench feedback for reasoning-enhanced Verilog generation. *arXiv preprint arXiv:2505.11849*, 2025a.

Yu Wang. The dark room in the reward channel: Dense prediction rewards collapse GRPO-trained LLM agents—and the channel, not the content, decides what works. *arXiv preprint arXiv:2607.21273*, 2026.

Zhexu Wang, Yiping Liu, Yejie Wang, Wenyang He, Bofei Gao, Muxi Diao, Yanxu Chen, Kelin Fu, Flood Sung, Zhilin Yang, Tianyu Liu, and Weiran Xu. OJBench: A competition level code benchmark for large language models. *arXiv preprint arXiv:2506.16395*, 2025b.

Xiao Xia, Dan Zhang, and Tianrui Sun. CodeRM-NT: Reward model for code RL without unit tests. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 43316–43333. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.2150.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. *arXiv preprint arXiv:2503.02951*, 2025.

An Yang et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Qiying Yu et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Lifan Yuan et al. Free process rewards without process labels. *arXiv preprint arXiv:2412.01981*, 2024.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *Advances in Neural Information Processing Systems*, 2025.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. AceCoder: Acing coder RL via automated test-case synthesis. *arXiv preprint arXiv:2502.01718*, 2025.

Ruiyi Zhang, Peijia Qin, Qi Cao, and Pengtao Xie. DreamPRM-Code: Function-as-step process reward model with label correction for LLM coding. *arXiv preprint arXiv:2512.15000*, 2025.

Ruiyi Zhang, Peijia Qin, Qi Cao, Eric Xue, and Pengtao Xie. FunPRM: Function-as-step process reward model with meta reward correction for code generation. *arXiv preprint arXiv:2601.22249*, 2026a.

Yiwen Zhang, Xiaodong Yan, Zhenyu Huang, Deng Zhao, Liang Jiang, Qing Cui, Zujie Wen, Zhiqiang Zhang, and Jun Zhou. Robust code RL via faulty-code-driven test case synthesis and dense reward shaping. *arXiv preprint arXiv:2608.24135*, 2026b. To appear at EMNLP 2026.

Zihan Zheng, Zerui Cheng, Zeyu Shen, Shang Zhou, Kaiyuan Liu, Hansen He, Dongruixuan Li, Stanley Wei, Hangyi Hao, Jianzhu Yao, et al. Livecodebench pro: How do olympiad medalists judge llms in competitive programming? *Advances in Neural Information Processing Systems*, 38, 2026.

Yuhang Zhou, Yixin Cao, and Guangnan Ye. From correctness to utility: Gain-based prefix evaluation for LLM reasoning. *arXiv preprint arXiv:2606.07190*, 2026.

Xiao Zhu, Xinyu Zhou, Boyu Zhu, Hanxu Hu, Mingzhe Du, Haotian Zhang, Huiming Wang, and Zhijiang Guo. CodeScaler: Scaling code LLM training and test-time inference via reward models. *arXiv preprint arXiv:2602.17684*, 2026.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

APPENDIX CONTENTS

A. Full Training Configuration

B. Training Data

C. Evaluation Details

C.1 Graders and pinned revisions

C.2 Statistical procedure

D. Additional Challenging-Benchmark Results

E. Implementation Notes

F. Length Statistics

G. Results Across Decoding Seeds

H. Unit-Level Structure of Failing Programs

I. Limitations

A FULL TRAINING CONFIGURATION

For the original Qwen3-4B GRPO–`ASTrace` comparison, both arms run the same veRL 0.8.0 job (Sheng et al., 2024) on the same data, row order, seed and step budget. The `ASTrace` arm rebinds veRL's `compute_advantage` inside the trainer actor, so the stock GRPO advantage is computed first and the process channel is added to it; the control arm runs the unmodified trainer entry point. Every `ASTrace` knob is a field of a single configuration object with an environment-variable override. Table 4 lists the values used for these runs. The main evaluation additionally includes Qwen2.5-Coder-7B, VeRPO and CodeScaler. The first 200-step stage takes 5:27:27 for GRPO and 6:20:12 for `ASTrace` on 8×RTX PRO 6000 Blackwell, a wall-clock ratio of $1.16\times$, and `ASTrace` uses $6{,}074$ extra executions, 47.5% of the 64×200 cap.

Table 4: **Qwen3-4B GRPO and `ASTrace` training configuration.** Identical for both arms except the advantage wrapper; both stages use binary terminal scores before the process channel is added.

Knob	Value
backbone	`Qwen/Qwen3-4B-Instruct-2507` (Yang et al., 2025)
framework	veRL 0.8.0, FSDP2, vLLM with TP=1, `gpu_memory_utilization` 0.70
algorithm	filter-off GRPO, `filter_groups.enable=False`
on-policyness	strict: mini-batch = batch, `ppo_epochs=1`
group size / problems per step	K=16 / 32, i.e. B=512 rows per step
steps	400
learning rate	1×10^{-6} constant, no warm-up
KL and entropy	none: `use_kl_loss=False`, `use_kl_in_reward=False`, `entropy_coeff=0`
sampling	temperature 1.0, top-p 1.0, top-k off, max response 16,384, max prompt 2,048
length control	none in either arm, in both stages
process weight	λ=0.1, per-token cap κ=1.0
decomposition	`max_segments` 4, `max_signal_cases` 6, `n_perm` 1
execution budget	`budget_evals` 64 per step, `rollouts_per_group` 1
sandbox	`sandbox_mode warm`, `workers` 64, `terminal_servers` 8, `test_timeout_s` 10, `memory_mb` 2048, `capture_detail True`
hardware	8×RTX PRO 6000 Blackwell

B TRAINING DATA

Unit-level credit needs problems whose solutions decompose into several callable units and whose tests call those units, and the binary reward needs test suites that a correct program passes. We

therefore select training problems by the structure of their reference solutions and verify every reference by execution. Stage 1 draws on KodCode-V1 (Xu et al., 2025), keeping its function-level subsets (Data_Structure, Package, Docs, Algorithm and Leetcode) and excluding the subsets whose programs are monolithic or read stdin (Taco, Codeforces, Apps and Code_Contests). We keep a problem when its pytest suite imports the tested functions from the solution module, its reference solution is SEMANTIC (§4.1), and its question and reference fit within $6{,}000$ and $8{,}000$ characters. After deduplicating on the normalized question, we execute the $43{,}101$ remaining references against their own suites, discard every failure, and sample $16{,}000$ training and 500 validation problems from the rest. Because this filter favors problems that decompose, the stage-1 mixture leans toward easy, documentation-style problems (Table 5). Stage 2 raises the difficulty: it keeps $11{,}400$ stage-1 problems and adds $4{,}600$ competitive-programming problems, $2{,}100$ from KodCode's Codeforces subset and $2{,}500$ from Code_Contests, restricted to those that ship as functions with pytest suites and SEMANTIC references so that the same execution and attribution apply. Their references pass their own suites in $6{,}546$ of $6{,}654$ cases (98.4%) at a 30 s timeout, and we sample only from the passing set under per-source and per-difficulty quotas. A TF-IDF screen against the 175 LiveCodeBench v6 problems finds a maximum cosine similarity of 0.36, below our threshold of 0.80. In both stages, the prompt fixes the import contract, the expected signatures, a single fenced Python block and no use of stdin, and all models train on the same data.

Table 5: **Training mixture.** Share of the $16{,}000$ training problems by source and difficulty; stage 2 lists the sampling quotas.

Source	Stage 1 (%)	Stage 2 quota (%)
Data_Structure	37.1	19.4
Docs	28.8	18.8
Algorithm	22.3	21.9
Leetcode	10.5	10.0
Package	1.4	1.2
Competitive programming (new)	—	28.8
easy	62.5	45
medium	22.0	35
hard	15.6	20

C Evaluation Details

C.1 Graders and pinned revisions

All benchmarks are graded offline on CPU from saved generations with $n{=}8$ samples per problem and decoding seed, temperature 1.0, top-p 1.0, $16{,}384$ new tokens and a $32{,}768$-token context. The main comparison uses decoding seeds 42, 43, and 44 for every trained method and backbone; the untrained instruct checkpoints use seed 42 only. OJBench evaluates both backbones and decoding seed 42, and LCB-Pro evaluates Qwen3-4B and decoding seed 42. No checkpoint is selected by a benchmark, and grading never regenerates samples. Table 6 lists the graders. The OJBench run pins dataset revision `61cf9986`, OJBench commit `5e94480b` and DMOJ commit `f098cd3a` on PyPy 3.10, and all 232 problems and $1{,}856$ samples per arm were judged. LCB-Pro uses 706 C++ problems with public tests; since training uses Python problems only and the attribution operates on the Python AST, it measures transfer of the trained policy to a second language.

Table 6: **Benchmarks and graders.**

Benchmark	Problems	Grader
HumanEval (Chen et al., 2021) and HumanEval+ (Liu et al., 2023b)	164	dataset `check` function
BigCodeBench Full and Hard (Zhuo et al., 2024)	1,140 / 148	`unittest` suites
LiveCodeBench v6 (Jain et al., 2024)	175	stdin and functional runner, pinned revision
ClassEval (Du et al., 2023)	100	`unittest` suites
OJBench (Wang et al., 2025b), Python	232	official DMOJ harness
LCB-Pro (Zheng et al., 2026), C++ with public tests	706	public-test runner

C.2 Statistical procedure

The main tables and Figure 3 summarize three decoding seeds per fixed checkpoint, with sample standard deviations across seeds. These describe sampling variability, while independent training-seed variability remains unmeasured. The difficulty and contest slices of §D carry no separate intervals. HumanEval+ shares its problems with HumanEval, and BigCodeBench Hard is a subset of BigCodeBench Full, so the columns of Table 1 are not independent, and its mean is a descriptive summary, not a test statistic.

D Additional Challenging-Benchmark Results

Table 7: **Challenging competitive-programming benchmarks.** Step-400 checkpoints, decoding seed 42, eight samples per problem. OJBench uses 232 Python problems; LCB-Pro uses 706 C++ problems with public tests. Scores are in %.

Benchmark	Backbone	Method	pass@1	pass@5	pass@8
OJBench	Qwen3-4B	GRPO	10.56	16.30	18.10
		ASTrace	**13.15**	**18.92**	**20.26**
OJBench	Qwen2.5-Coder-7B	GRPO	3.07	4.52	5.17
		ASTrace	**3.93**	**5.65**	**6.47**
LCB-Pro	Qwen3-4B	GRPO	15.78	25.28	27.48
		ASTrace	**17.49**	**26.28**	**28.33**

Table 8 breaks the Qwen3-4B OJBench comparison down by difficulty and by contest family, and Table 9 gives the judge's verdict counts. The hard slice, 117 of 232 problems, is essentially unsolved by all three arms, with at most two problems per arm solved at pass@8. The ICPC slice loses pass@5 and pass@8 under ASTrace even though its pass@1 rises. Relative to GRPO, ASTrace has 120 fewer timeouts and 48 more accepted samples, alongside 65 more compile errors, the length tail of §F. The coverage changes decompose into gains and losses: GRPO over base is +7 problems (9 gained, 2 lost), ASTrace over base is +12 (13 gained, 1 lost), and ASTrace over GRPO is +5 (8 gained, 3 lost).

Table 8: **OJBench slices.** pass@1 and pass@8 per slice over n problems, at the step-400 checkpoints with n=8 samples. Best in **bold**. Slice estimates carry no separate confidence intervals.

Slice	Metric	n	Base	GRPO	**ASTrace**	Δ ASTrace −GRPO
easy	pass@1	36	48.96	50.00	**55.56**	+5.56
medium	pass@1	79	6.49	7.28	**12.97**	+5.70
hard	pass@1	117	0.43	**0.64**	0.21	−0.43
hard	pass@8	117	0.85	**1.71**	**1.71**	0.00
NOI	pass@1	159	10.14	9.67	**12.97**	+3.30
NOI	pass@8	159	15.09	16.98	**20.75**	+3.77
ICPC	pass@1	73	9.76	12.50	**13.53**	+1.03
ICPC	pass@8	73	15.07	**20.55**	19.18	−1.37

Table 9: **Official DMOJ verdicts.** Counts of the five listed verdict categories over the 1,856 judged samples per arm.

Verdict	Base	GRPO	**ASTrace**
accepted	186	196	**244**
time-limit exceeded	403	398	278
compile error	4	17	82
invalid return	225	213	231
wrong answer	1,017	1,014	1,000

The LCB-Pro difficulty slices are those of Figure 4(c): the pass@1 gain is concentrated on the 391 easy problems, medium-problem scores remain below one percent, and neither method solves any of the 64 hard problems in eight attempts.

E Implementation Notes

Verdict handshake. Besides the scalar score, the terminal reward function returns three extra keys per row: the full verdict vector as a JSON string, the sandbox exit status, and a detail record that lists, for each failing test, the traceback line numbers inside the extracted solution module and the last functions of that module entered under a `sys.settrace` tracer, whose measured overhead is within 1% of the untraced path. veRL copies extra keys into the non-tensor batch, so the driver rebuilds the terminal result per row instead of regrading it; a missing key raises unless an explicit regrade flag is set. This is what makes $v(\mathcal{U})$ free in Eq. (4) and keeps the terminal path identical in the two arms.

Spans to tokens. Token offsets are reconstructed by decoding each sampled token id individually and accumulating lengths, which is exact for byte-level BPE; if the accumulation disagrees with the full decode, the tokenizer's offset mapping is used instead. A span's credit is split evenly over the tokens it covers, so a unit's raw credit does not depend on its token length, and overlapping spans add; the per-token normalization and clip of Algorithm 1 act on the result. Uncredited tokens stay at zero, and the token-level rewards and returns tensors are left unchanged.

Decomposition mechanics. Units are enumerated in source order; the $k_{\max}$ longest are retained, and dropped units are present in every coalition, so the attribution is exact over the retained players. Ablation edits are applied right to left so that source offsets stay valid while the module is rebuilt.

Test selection and execution budgets. To limit execution cost, we select at most six signal cases per group. Let $r_\tau = K^{-1} \sum_{i=1}^{K} \mathbb{1}[V_{i,\tau} = \text{pass}]$ be the fraction of the group's K responses that pass test τ. We choose the $\min(6, |\mathcal{T}|)$ tests with the smallest $|r_\tau - \frac{1}{2}|$, breaking ties by suite order; $|\mathcal{T}|$ is the number of tests. We prioritize tests with greater outcome variation across sampled programs and recompute the selection for each group at every step. We use the same selection rule for partial-fail rollouts in §4.3.

For progress attribution, we evaluate $k + 1$ coalition scores along one restoration order instead of the 2^k scores needed for exact Shapley attribution. We reuse the original program's recorded test outcomes to obtain $v(\mathcal{U})$. This assumes that the selected tests produce the same outcomes when run separately as within the full suite. We execute the baseline with no retained units active and the $k - 1$ intermediate coalitions, giving at most $k_{\max} = 4$ new executions per path. We select at most one SEMANTIC response per all-fail group, rotating the selection across steps. We evaluate a path when it fits the remaining budget of 64 executions per step.

Rotation across steps. The scheduler's step counter and rotation cursor are not checkpointed, so the stage-2 continuation restarts the rotation over all-fail rollouts. Each step logs the regime and route counts, the number of served groups, the extra executions against the budget, and the batch scoring time.

F Length Statistics

Table 10 collects output-length and execution-failure statistics for the Qwen3-4B OJBench evaluation of Appendix D, over all n=8 samples per problem. The extracted code shows the same tail as the raw response: its p95 grows from $5{,}614$ characters at base and $4{,}227$ under GRPO to $52{,}113$ under `ASTrace`.

Table 10: **The length tail.** Output-length and execution-failure statistics per arm. Larger is worse in every row.

Statistic (OJBench)	Base	GRPO	**ASTrace**
median raw completion (chars)	9,343	16,379	29,620
odd code fences (of 1,856)	11	48	239
compile errors (of 1,856)	4	17	82

G Results Across Decoding Seeds

Tables 11 and 12 report the complete higher-budget results. Trained methods use decoding seeds 42, 43, and 44 of the same checkpoint, with eight samples per problem per seed; the base model is the released instruct checkpoint evaluated at seed 42. The Avg. column averages the six benchmark means. On Qwen2.5-Coder, the untrained checkpoint has higher pass@8 than `ASTrace` on HumanEval, HumanEval+, BigCodeBench Full and ClassEval, despite lower pass@1 on every benchmark. Thus, improvements over trained baselines do not imply uniform coverage gains over the starting checkpoint.

Table 11: **pass@5 (%) across decoding seeds.** Values are means and across-seed sample standard deviations; Base model uses one seed. Avg. is the unweighted benchmark average. Bold and underline compare trained methods within each backbone.

Method	HumanEval	HumanEval+	BCB Full	BCB Hard	LCB v6	ClassEval	Avg.
Qwen3-4B-Instruct-2507							
Base	93.66	89.29	49.79	27.64	41.56	38.55	56.75
GRPO	95.50±0.15	91.05±0.33	50.46±0.43	29.75±0.34	49.06±1.00	44.45±1.21	60.04
VeRPO	95.24±0.45	90.36±0.35	**51.42**±0.29	28.95±0.85	45.34±0.81	42.93±1.08	59.04
CodeScaler	90.65±0.63	87.91±0.33	48.73±0.37	23.20±0.59	44.57±1.23	39.30±0.17	55.73
`ASTrace`	**96.72**±0.08	**92.53**±0.40	50.81±0.49	**33.10**±0.39	**50.31**±0.44	**45.69**±0.43	**61.53**
Δ vs. GRPO	+1.22	+1.48	+0.35	+3.35	+1.25	+1.24	+1.49
Qwen2.5-Coder-7B-Instruct							
Base	91.72	88.62	49.87	29.67	27.97	38.93	54.46
GRPO	91.80±0.28	87.92±0.63	50.10±0.50	30.64±0.53	30.14±0.13	41.65±0.86	55.38
VeRPO	91.84±0.15	89.56±0.77	51.24±0.15	33.11±0.17	**30.20**±0.37	40.73±0.33	56.11
CodeScaler	89.24±0.04	87.17±0.15	49.89±0.12	28.94±0.88	26.90±0.75	38.35±0.51	53.42
`ASTrace`	**92.09**±0.21	**89.80**±0.38	**53.12**±0.19	**33.76**±0.33	29.16±0.22	**42.72**±0.29	**56.78**
Δ vs. GRPO	+0.29	+1.88	+3.02	+3.12	-0.98	+1.07	+1.40

Table 12: **pass@8 (%) across decoding seeds.** Values are means and across-seed sample standard deviations; Base model uses one seed. Avg. is the unweighted benchmark average. Bold and underline compare trained methods within each backbone.

Method	HumanEval	HumanEval+	BCB Full	BCB Hard	LCB v6	ClassEval	Avg.
Qwen3-4B-Instruct-2507							
Base	94.51	90.24	51.84	29.73	43.43	43.00	58.79
GRPO	95.93±0.35	91.67±0.35	52.63±0.46	32.21±0.39	51.24±1.44	**48.67**±1.53	62.06
VeRPO	95.73±0.61	91.26±0.70	**53.98**±0.45	30.63±1.56	47.43±0.57	45.33±1.53	60.73
CodeScaler	92.28±0.70	89.43±0.35	51.40±0.38	27.03±0.68	47.81±2.01	41.67±0.58	58.27
`ASTrace`	**97.15**±0.35	**93.29**±0.61	53.19±0.56	**35.36**±0.39	**52.57**±0.99	48.00±0.00	**63.26**
Δ vs. GRPO	+1.22	+1.62	+0.56	+3.15	+1.33	-0.67	+1.20
Qwen2.5-Coder-7B-Instruct							
Base	96.34	92.68	56.84	35.14	30.86	50.00	60.31
GRPO	92.48±0.35	88.82±0.70	52.43±0.53	33.56±0.39	**32.00**±0.00	44.33±1.53	57.27
VeRPO	92.68±0.00	90.65±0.93	54.15±0.13	36.49±0.00	30.48±0.66	43.33±0.58	57.96
CodeScaler	89.63±0.00	88.41±0.00	52.13±0.13	31.31±1.03	28.57±0.57	39.67±0.58	54.95
`ASTrace`	**93.09**±0.35	**91.26**±0.35	**56.29**±0.37	**36.71**±0.39	**32.00**±0.57	**46.00**±0.00	**59.23**
Δ vs. GRPO	+0.61	+2.44	+3.86	+3.15	0.00	+1.67	+1.96

H Unit-Level Structure of Failing Programs

Table 13 measures the structure behind the finding of §1. It uses the ClassEval generations of the GRPO step-200 checkpoint (n=8 samples per problem, the protocol of §C), replayed offline in the sandbox; all 648 samples in the 57 all-fail and 24 mixed problems reproduce their saved verdicts. In all-fail problems, units are segmented as in §4 with at most four retained units, and credit is the exact Shapley value over all coalitions on each group's six signal tests. In mixed problems, each

failing sample is localized as in §4, and its localized unit is replaced by the same-named unit of the first passing sibling before the full suite is rerun. The lower block reports the same two probes at the group level, one sampled program per structure-eligible all-fail group with exact Shapley credit on ClassEval and the single-permutation training estimator on LiveCodeBench v6, and the paired control on the 96 partial-fail rollout failures for which an unblamed, signature-compatible unit could be swapped instead; these are the numbers cited in §6.

Table 13: **Failing programs are mostly not wrong as a whole.** ClassEval, GRPO step-200 samples.

Group	Measure	Count	Share
all-fail	programs with ≥ 2 units	441	—
all-fail	full program passes more signal tests than the empty program	350/441	79.4%
all-fail	at least one unit with positive credit	355/441	80.5%
mixed	failing samples	130	—
mixed	localized unit has a same-named unit in the passing sibling	116/130	89.2%
mixed	swapping only the localized unit repairs the sample	68/116	58.6%
mixed	localized through the traceback	26/130	20.0%
mixed	of which repaired by the swap	11/26	42.3%
all-fail, ClassEval	rollouts whose sampled program has $v(\mathcal{U}) > v(\emptyset)$, exact	42/56	75.0%
all-fail, LCB v6	rollouts whose sampled program has $v(\mathcal{U}) > v(\emptyset)$, one path	29/40	72.5%
mixed, paired	swapping the localized unit repairs the sample	56/96	58.3%
mixed, paired	swapping an unblamed unit repairs the sample	13/96	13.5%

I LIMITATIONS

For efficiency, progress credit considers at most four AST units and six signal cases per rollout; larger caps may give finer-grained credit at a higher execution cost. Failure localization credits the unit where an error surfaces, but can differ from where a fault originates; dependency or data-flow tracing could refine it further. The attribution operates on Python syntax trees, so the C++ results reflect transfer of the trained policy; extending attribution to another language requires a parser and a test harness. Finally, our evaluation covers the function, class, and contest level. Repository-level and long-horizon coding tasks involve larger programs with more units, where process credit has more structure to exploit. They are a natural next step if computational resources allow.