EVIGRAD: EVIDENCE-CALIBRATED REWARDS AND GRADIENTS FOR RLVR

Anonymous authors
Paper under double-blind review

ABSTRACT

Verifiable-reward reinforcement learning (RLVR) for code generation relies on executable tests to provide training rewards. When these tests fail to cover important behavioral boundaries, incorrect programs may pass them and receive spuriously high rewards. We find an asymmetric reward-error pattern: high-reward false positives are relatively common and can be identified with useful accuracy, whereas low-reward candidates are rarely rescued by additional validated tests. This makes suspiciously high rewards the primary actionable source of reward misranking in group-relative policy optimization. We present **EviGRAD**, a risk-aware selective verification framework for conservative reward correction under a fixed verification budget. A lightweight reliability model ranks high-reward candidates and rollout groups according to their risk of being false positives. Only the highest-risk groups, together with a small exploration quota, receive additional verification and counterexample generation. Generated tests are admitted only after reference-based semantic validation and execution-based checks. Verified counterexamples then produce a one-sided reward correction: suspiciously high baseline rewards may be reduced, whereas low baseline rewards are not increased merely because a candidate passes an additional test. For rollout groups with degenerate low-signal rewards, EviGRAD uses dynamic resampling to recover learnable variation. The two mechanisms address complementary failure modes: erroneous positive training signals and insufficient within-group reward variation. We evaluate EviGRAD on CodeContests and APPS using matched models, training steps, rollout budgets, and verification costs. We measure agreement between training-time reward rankings and hidden-test rankings, the rate of false-positive positive-gradient signals, correction coverage, recovery of zero-advantage groups, final task success, and verification overhead. Across multiple random seeds and cost-controlled comparisons with binary GRPO, pass-rate GRPO, DAPO-style dynamic sampling, and always-augmented testing, EviGRAD reduces false-positive positive-gradient signals and improves the alignment between training-time rewards and hidden-test correctness, while dynamic resampling recovers learning signal from degenerate groups.

1 INTRODUCTION

Reinforcement learning with verifiable rewards (RLVR) has become the standard recipe for training code-generation policies: the policy is optimized directly against an executable checker rather than a learned reward model (Shao et al., 2024; Guo et al., 2025). Many implementations reduce the checker's output to a single bit rather than the finer-grained pass rate over the visible tests. Group Relative Policy Optimization (GRPO) (Shao et al., 2024) standardizes rewards within a sampled group to form the policy gradient, so whatever the reward function keeps or discards is exactly what the gradient sees.

A graded pass-rate reward is only as reliable as the test suite it is computed from: a finite sample of the specification's input space that can miss a behavioral edge case as easily as a binary checker can. Prior work makes this signal more trustworthy by improving the test suite before training (Cai et al., 2026; Wang et al., 2025b), reshaping the reward around the policy's mistakes (Zhang et al., 2026), or co-evolving a coder and unit-test generator (Wang et al., 2025a). A fourth source, the LLM judge,

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.

is noisy and execution-free, more useful escalated selectively than trusted outright as a reward (Zhu et al., 2025; Choi et al., 2026; Liu et al., 2025; Zhao et al., 2025; Xu et al., 2025).

These lines of work share an assumption our calibration study calls into question: newly acquired evidence is treated as symmetric, so a passed test or a favorable judge verdict raises confidence in a reward by the same logic that a failed test lowers it. Test-based evidence for code is not symmetric. A failed test is direct, falsifying evidence, sufficient to lower an otherwise-perfect reward. A passed test is only corroborating: the untested part of the input space is exactly where the candidate can still fail. A reward for code should therefore be conservative about raising a score, and this direction-dependent trust in evidence, not overall noise level, is what should determine where a bounded verification budget is spent.

Acting on this asymmetry online raises difficulties offline dataset improvement does not face. Without ground-truth correctness, the training loop must decide which full-score candidates are worth auditing under a fixed budget. Evidence used to lower a reward must itself be trustworthy, or it would simply replace one source of noise with another. And some groups contain no full-score candidate to audit at all: when every candidate in a group receives the same low reward, the standardized GRPO advantage collapses to zero regardless of whether that shared reward is correct, a failure mode needing its own mechanism.

We study this asymmetry on a 5,000-task calibration pool spanning CodeContests (Li et al., 2022) and a matched-protocol supplement from APPS (Hendrycks et al., 2021) and TACO (Li et al., 2023), in which every policy-sampled candidate has both a seed and an independent held-out outcome. We express reward error as an advantage displacement rather than a raw pass-rate difference, and define thresholded labels for over-reward (seed-side advantage overstates the held-out outcome by a margin) and under-reward (the converse). The pool is sized so every reported classifier gain carries a 95% confidence interval computed from the actual positive-class count rather than a point estimate alone (Appendix A.1). Two findings result. First, the two directions occur at comparable rates (40.7% over-reward versus 46.2% under-reward); we do not claim over-reward is more frequent. Second, judge evidence's incremental contribution differs sharply: it moves the over-reward classifier's AUC by $+0.042$ (95% CI $[+0.028, +0.057]$, $p < 0.001$), but the under-reward classifier by only $+0.007$, not distinguishable from zero ($p = 0.38$). A judge can often identify a semantic defect in code that already passes every visible test, but cannot observe which behaviors the visible tests failed to probe. Over-reward is also the higher-impact direction to act on: concentrated among positive-advantage candidates and growing with advantage magnitude, so its erroneous updates are the largest ones.

Guided by this asymmetry, we present **EviGRAD** (Evidence-Calibrated Rewards and Gradients): a selective verification framework that spends a bounded budget auditing the over-reward side, scores which full-score candidates are worth auditing, and applies a correction that can only lower a reward when a verified counterexample contradicts it. Groups with no full-score candidate to audit are routed to a mechanism that repairs the group using verified evidence when available, and falls back to clean resampling otherwise (§4).

On a 6,000-task CodeContests training pool, evaluated on *CodeContests-test-expanded* (a 300-task expanded CodeContests evaluation suite built entirely from CodeContests, Appendix A.1), EviGRAD (full) reaches $10.9\% \pm 0.5\%$ strict pass@1 over 5 seeds, significantly ahead of every baseline with no verification mechanism and its own correction-only ablation (all $p \leq 0.002$), and statistically indistinguishable from always-verify ($p = 0.83$) at roughly one-third its realized cost. Against CURE (Wang et al., 2025a), EviGRAD is numerically ahead but not statistically distinguishable at this test-set size ($p = 0.14$), a disclosed consequence of the single-source suite.

Our contributions are:

- A 5,000-task, 29,900-candidate calibration study showing that judge evidence significantly aids over-reward identification but not under-reward identification, with confidence intervals rather than point estimates.
- EviGRAD: a budget-gated, evidence-validated, one-sided correction mechanism for over-reward, coupled with a disjoint evidence-guided repair path for low-value, zero-variance groups.

- A cost-matched evaluation against six baselines and one external method on CodeContests-test-expanded (§5.4).

2 Background and Related Work

RLVR and group-relative optimization. RLVR replaces a learned reward model with an executable correctness signal. GRPO (Shao et al., 2024) samples a group of candidates per prompt and standardizes their rewards within the group, avoiding the separate value model used by PPO (Schulman et al., 2017); DAPO (Yu et al., 2025) further refines the loop with clip-higher, dynamic sampling, token-level loss, and overlong-reward shaping. All such systems inherit the same dependency: the policy gradient is only as correct as the group-relative reward that produced it, which depends on the coverage of the executable verifier.

Test coverage, verifier quality, and reward hacking. A finite test suite samples the input space implied by the specification, so a faulty program can obtain full reward without being correct on untested inputs. CodeContests-O (Cai et al., 2026) and CodeContests+ (Wang et al., 2025b) address this on the data side, synthesizing higher-quality tests and quantifying the false-accept/false-reject rates already present in competitive-programming benchmarks. CURE (Wang et al., 2025a) instead co-evolves a coder and a unit-test generator online without ground-truth code; FixAudit (Tang et al., 2026) repairs a single candidate through an iterative Fixer/Auditor cycle on a different base model with no cost axis comparable to our (B, E) budget, so we discuss but do not row it in Table 2 (full reasoning in Appendix A.1), and use CURE alone as the direct baseline sharing our group-relative-sampling scaffold. None of these argue the two reward-error directions should receive different trust once evidence is collected; our calibration study (§5.2) shows they should, because a failed test falsifies a candidate directly while a passed test only corroborates it. RobustTests (Zhang et al., 2026) reshapes the reward with tests synthesized offline from near-miss faulty programs; EvilGenie (Gabor et al., 2025) and Countdown-Code (Khalifa et al., 2026) study how judges detect, and how RLVR induces, overt reward hacking, and a representation-level line detects hacking without execution (Wu & Tang, 2026) — complementary to our question of converting already-limited evidence into a correctly *signed* group-level signal online under a bounded budget. Work showing judges can be fooled by small non-semantic changes (Zhao et al., 2025) further motivates using judge outputs for risk *selection* rather than as a reward.

Selective verification and judges. Escalating to an expensive oracle only when a cheap signal is uncertain has been explored in general reward modeling (Xu et al., 2025); studies of judge calibration and disagreement (Zhu et al., 2025; Choi et al., 2026; Liu et al., 2025) support treating judge outputs as noisy evidence, not ground truth. EviGRAD differs in using a calibrated risk head, rather than a judge vote or confidence score, to select groups (§5.2); requiring an execution-validated counterexample before any correction; fixing an explicit budget rather than an implicit escalation rate; and accounting for the full realized verification cost, including the risk head's own inference (§5.5).

Dynamic sampling, policy-side correction, and benchmark provenance. DAPO's dynamic sampling discards groups whose candidates all receive the same reward; our baseline reimplements this together with DAPO's clip-higher mechanism, omitting only token-level loss and overlong-reward shaping. EviGRAD's degenerate-low path (§4.6) generalizes this: falling back to clean resampling only when no verified counterexample is available, and otherwise repairing the group with that evidence instead of discarding it. Self-reflection and self-distillation methods (Jiang et al., 2026; Bensal et al., 2025; Yang et al., 2026) and retrospective credit assignment (Ekbote et al., 2025) change what the policy does with a reward, or how credit is assigned given one, rather than asking whether the scalar reward is correct in the first place — orthogonal to, and composable with, our contribution. Our filtering pipeline (§5) follows the coverage-auditing spirit of APPS (Hendrycks et al., 2021); we draw part of our calibration-only supplement from APPS and TACO (Li et al., 2023) because they share this executable, reference-solution-backed structure, letting us scale the calibration pool without consuming CodeContests tasks the policy needs.

3 PROBLEM FORMULATION

Seed rewards and GRPO advantages. Let x be a programming task with training-visible tests $T_{\text{seed}}(x)$. For a group of K candidates sampled from policy π_θ, the seed pass-rate reward is

$$R_{\text{seed}}(c_i) = \frac{1}{|T_{\text{seed}}(x)|} \sum_{t \in T_{\text{seed}}(x)} \mathbf{1}[c_i \text{ passes } t]. \tag{1}$$

For any reward vector R, GRPO forms

$$A_R(c_i) = \frac{R(c_i) - \overline{R}}{s_R + \epsilon_{\text{std}}}, \tag{2}$$

where $\overline{R}$ and s_R are the group mean and standard deviation, and ϵ_{std} is a numerical stabilizer. Positive advantages increase the likelihood of a completion and negative advantages decrease it. When all group rewards are equal, the standardized signal collapses to zero regardless of whether the common reward is correct.

Two sources of reward error. The held-out pass rate R_{calib} is an operational proxy for correctness. We define

$$A_{\text{seed}} = A_{R_{\text{seed}}}, \qquad A_{\text{calib}} = A_{R_{\text{calib}}}, \qquad d(c_i) = A_{\text{seed}}(c_i) - A_{\text{calib}}(c_i). \tag{3}$$

Thus $d > 0$ means the seed tests overstate the candidate's relative advantage, while $d < 0$ means they understate it.

Operational labels. To focus the calibration study on optimization-relevant errors, we introduce an over-reward threshold x, an under-reward threshold y, and a minimum displacement $\tau > 0$:

$$\begin{aligned} \text{over}_{x,\tau}(c_i) &= \mathbf{1}[A_{\text{seed}}(c_i) > x \ \wedge \ d(c_i) \geq \tau], \\ \text{under}_{y,\tau}(c_i) &= \mathbf{1}[A_{\text{seed}}(c_i) < y \ \wedge \ -d(c_i) \geq \tau]. \end{aligned} \tag{4}$$

We constrain $x \geq 0$ and $y \leq 0$; x, y define the high-impact tails acted upon, while requiring a sign flip in A_{calib} would instead discard magnitude errors that still change the policy gradient. Appendix A.2 reports the full sweep with confidence intervals at every matched magnitude, so the reported ordering (over-reward AUC above under-reward AUC) is a property of the whole grid, not an artifact of $x = +0.75$.

Calibration data and verification budget. Our calibration pool records seed and independent held-out outcomes for each candidate, drawn from CodeContests (Li et al., 2022) plus a matched-protocol supplement from APPS (Hendrycks et al., 2021) and TACO (Li et al., 2023) (§5), sized for statistical power (Appendix A.1) without consuming CodeContests tasks the policy needs for training. Groups with zero seed-reward spread have undefined A_{seed} and are handled online by the degenerate-low path (§4.6). Additional evidence is costly, so we specify a per-round budget $B \in (0, 1]$, the fraction of rollout groups receiving full verification; §5.5 accounts for the realized cost once the degenerate-low path and the reliability head's own inference are included.

Why the old trigger saturates. If a group is escalated whenever any of its K candidates fires a rule with probability p, then $P(\text{escalate}) = 1 - (1 - p)^K$. With $K = 16$ and $p \approx 0.055$, this is already ≈ 0.60 for one rule; an OR over several rules approaches universal triggering, so nearly every group would receive the expensive path. EviGRAD instead selects a fixed number of groups by risk.

4 METHOD: EVIGRAD

4.1 OVERVIEW

EviGRAD is organized around a simple distinction, summarized in Figure 1: some reward errors have directly verifiable evidence for correction, while others do not. A reliability model scores each group's over-reward risk (§4.2); a fixed budget selects the highest-risk groups plus a small

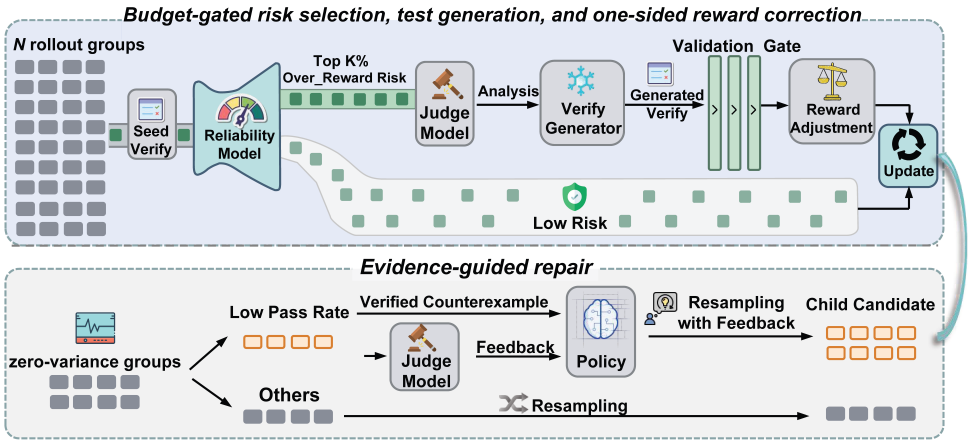

Figure 1: Overview of the EviGRAD pipeline. **Top:** a reliability model trained on the operational over-reward risk signal (p_{over}, out-of-fold AUC 0.893 ± 0.009) scores per-group risk. A fixed budget selects the top-risk groups plus a small random-exploration slice for judge assessment and frozen-model counterexample generation; verified evidence enters through a one-sided correction that can only lower R_{corr}. **Bottom:** zero-variance, low-seed-reward groups are handled separately (§4.6); their judge call still counts against the realized budget (§5.5). A verified counterexample triggers evidence-guided resampling (up to 3 parents $\times$ 3 children, accepted only if a child's reward exceeds its parent's); otherwise the group falls back to unconditioned clean resampling.

exploration quota for verification (§4.3), while remaining groups keep the seed reward at no added cost. Selected groups receive a judge assessment and a frozen-policy-generated counterexample test, both validated before they can lower a reward through the one-sided correction (§4.4, §4.5). Groups with no high-reward candidate to audit — uniformly low, zero-variance rewards — are instead routed to a degenerate-low path that repairs the group with verified judge evidence when available, or falls back to clean resampling (§4.6).

4.2 Reliability Model: Scoring Over-Reward Risk

Rather than treat every seed-reward disagreement as equally actionable, EviGRAD fits a lightweight logistic regression predicting, for a full-marks candidate, the probability $p_{\text{over}}(c_i)$ of an optimization-relevant over-reward event, using seed-side reward and group-relative features, judge disagreement features, group reward dispersion, and the number of available tests. The classifier is fit on candidates with both seed and independent held-out outcomes known; at deployment the held-out outcome is unavailable, so the model supplies only a ranking score. A gradient-boosted-tree variant gave no significant AUC difference ($p = 0.41$); we report the logistic regression because it is cheaper to score online and directly inspectable.

The targets are the operational labels in Equation 4; the thresholds x, y, τ are hyperparameters rather than natural boundaries (sweep in Appendix A.2). All reported AUCs use 5-fold task-grouped cross-validation with out-of-fold predictions pooled across the full 5,000-task calibration pool. We fit an analogous classifier $p_{\text{under}}(c_i)$ for the converse event, using an execution-blind variant of the judge prompt (§4.4) so its judge-derived features are not confounded with the failure traces the standard prompt discloses for non-full-marks candidates. At the selected setting ($x = +0.75$, $y = -0.75$, $\tau = 0.5$), p_{over} and p_{under} reach the AUCs in Table 1 (§5.2).

The interpretation is mechanical: a judge asked whether a candidate solves the stated problem can expose a semantic defect in code that passes every visible test, but under-reward concerns a mismatch between the visible test set and the behavior it should probe, which a judge reading the code cannot directly observe. When judges are asked for a direct over/under-reward verdict rather than a correctness verdict, their under-reward precision and recall are far weaker than their over-reward numbers at a matched flag rate (Appendix A.3). This motivates using judge evidence to prioritize downward audits, not as a direct reward or a basis for symmetric upward correction.

4.3 BUDGET-GATED GROUP SELECTION

A group's risk score is

$$\mathrm{risk}(g) = \max_{i \in g:\, R_{\mathrm{seed}}(c_i)=1} p_{\mathrm{over}}(c_i). \tag{5}$$

Within a batch of N groups, EviGRAD verifies the top $\lceil BN \rceil$ groups by risk, plus a disjoint random slice of size $\lceil EN \rceil$ for exploration. Groups outside both sets use $R_{\mathrm{corr}} = R_{\mathrm{seed}}$ without additional calls, bounding the realized verification fraction by construction. The operating point $(B, E) = (0.20, 0.05)$ is selected on the held-out validation split from the precision–recall/cost sweep in §5.2 and fixed before the final training comparison. The judge's own confidence score alone reaches only 66.9% recall at the deployed 20% budget versus the trained head's 84.3% (§5.2).

Selection is not correction. $\mathrm{risk}(g)$ only decides which groups receive the expensive path; a reward moves only when that path produces an admitted counterexample the audited candidate actually fails (Equation 6). A fair group that p_{over} over-selects simply fails to produce an admitted counterexample, so $R_{\mathrm{corr}} = R_{\mathrm{seed}}$ is unchanged and the only cost is the spent verification call.

4.4 VERIFIED COUNTEREXAMPLE GENERATION

For each selected group, a two-judge panel (GLM-5.2 and DeepSeek-V3.2, temperature 0.3, majority verdict; ties broken by Qwen3-Max, invoked on $\approx$14% of groups) reviews the statement, candidate code, and a structured seed-test summary, returning a verdict plus a falsifiable bug hypothesis. For the under-reward calibration analysis specifically, the panel is instead shown only the statement and code, with seed execution results withheld, to avoid the disclosed-failure-trace confound (Appendix A.3). A frozen initial policy snapshot, not the evolving training policy, proposes a test conditioned on the judge's hypothesis, preventing generator drift and avoiding an external stronger model as a hidden teacher. A candidate test is admitted only after (i) execution against a human-authored reference program, (ii) repeated-execution stability, and (iii) a semantic check that it exercises the stated hypothesis. Judge verdicts determine which groups are audited but never enter the reward directly.

4.5 ONE-SIDED CONSERVATIVE REWARD CORRECTION

Let $R_{\mathrm{extra}}(c_i)$ denote the pass rate of c_i on the admitted, verified extra tests for its group, when any exist. EviGRAD applies

$$R_{\mathrm{corr}}(c_i) = \begin{cases} (1 - w_i)\, R_{\mathrm{seed}}(c_i) + w_i\, R_{\mathrm{extra}}(c_i) & \text{if } R_{\mathrm{seed}}(c_i) = 1 \text{ and } R_{\mathrm{extra}}(c_i) < 1, \\ R_{\mathrm{seed}}(c_i) & \text{otherwise,} \end{cases} \tag{6}$$

where $w_i = m_i / M \in (0, 1]$: M is the fixed number of candidate tests generated for the group and $m_i \le M$ is the number that survive all three admission checks for c_i specifically, so the size of the correction reflects how much verified evidence was found, not merely whether any was. The correction is deliberately one-sided: a full seed score can be lowered when an admitted counterexample falsifies it, but a partially rewarded candidate is not raised merely because it passes an additional test. This follows the evidence asymmetry in §4.2; it does not claim under-reward is absent, only that current online evidence does not support safe upward revision.

4.6 EVIDENCE-GUIDED REPAIR FOR DEGENERATE-LOW GROUPS

Groups whose seed rewards are uniformly low have no full-marks candidate to audit: zero within-group variance collapses the standardized advantage to zero regardless of whether the shared reward is high or low. EviGRAD routes each such group through one of two paths depending on whether its own judge assessment yields a verified counterexample; this judge call is itself counted against the realized budget (§5.5), not free simply because it is not gated by $\mathrm{risk}(g)$.

With a verified counterexample: evidence-guided repair. EviGRAD selects up to 3 parent candidates by reward and generates 3 children each from the *current training policy* (not the frozen snapshot used for test generation), conditioned on the verified evidence. If the judge's hypothesis indicates a shared, algorithm-level defect, the parent's code is withheld and the policy solves the

problem from scratch; otherwise the parent's code is kept and the policy attempts a targeted fix. A child is accepted only if its reward exceeds its parent's by a fixed margin, contributing an auxiliary loss alongside the seed-group loss.

Without one: clean resampling. If no counterexample survives verification, EviGRAD falls back to the sampling component of DAPO (Yu et al., 2025): the group is discarded and a fresh, unconditioned attempt is resampled, up to a fixed limit.

Both paths address the same failure mode — absence of within-group reward variation — and are disjoint from risk-gated correction, which addresses a misleading high-reward signal instead, so the two mechanisms can be ablated separately and combined (§5.6).

5 Experiments and Analysis

5.1 Experimental Setup

Datasets and calibration set. Our primary dataset is CodeContests (Li et al., 2022) (13,328 official training tasks, 117 validation, 165 test); filtering for execution-protocol compatibility (criteria in Appendix A.1) leaves 130 test tasks and an estimated ≈8,600 eligible training tasks. This pool supplies a **6,000-task training pool**; a **300-task validation split** (95 official + 205 reserve), used only for hyperparameter and operating-point selection; a **300-task test split**, *CodeContests-test-expanded* (130 official + 170 reserve); and a **2,225-task CodeContests reserve for calibration** (full disjoint accounting in Appendix A.1).

Our calibration study uses **5,000 tasks and 38,650 policy-sampled candidates** with both a seed and an independent held-out outcome per candidate. CodeContests alone supplies only 2,225 of the needed 5,000, so we add a 2,775-task supplement from APPS (Hendrycks et al., 2021) and TACO (Li et al., 2023) under an identical protocol (Appendix A.1); the asymmetry finding in §5.2 is therefore demonstrated on a CodeContests-plus-APPS/TACO mixture, not CodeContests alone. Excluding zero-spread groups leaves **29,900 candidates** with a defined advantage, evaluated by 5-fold task-grouped cross-validation with out-of-fold predictions pooled over the full pool.

Model, training, and baselines. All arms fine-tune `Qwen2.5-7B-Instruct` with LoRA ($r = 32$, $K = 16$, $\beta = 0.04$, temperature 0.8, 3 epochs); only the reward computation differs between arms. Learning rate and scheduler were chosen by a staged sweep (Appendix A.10). Every arm is trained with **5 independent seeds**; we report mean ± std and paired bootstrap significance throughout. Baselines: binary GRPO, pass-rate GRPO, direct judge reward, DAPO-style dynamic sampling with clip-higher (§2), always-verify ($B = 1$), and CURE (Wang et al., 2025a); FixAudit (Tang et al., 2026) is excluded (§2). We report strict **pass@1** on **CodeContests-test-expanded** (300 tasks, primary suite) and an evaluation-only LiveCodeBench check (Jain et al., 2024) (157 tasks). Expanding the official 130-task test split with 170 training-partition tasks trades a broken train/test temporal split for single-source traceability; Appendix A.1 weighs this trade-off in full.

5.2 The asymmetry is in evidence availability, not in frequency

Table 1 reports the two classifiers with 95% confidence intervals; two readings matter, and only the second is our claim.

The overall AUCs differ, but this is not the claim we lean on: 0.741 is not "unlearnable." The claim the confidence intervals support is narrower: judge features produce a statistically significant improvement for over-reward identification ($+0.042$, $p < 0.001$) and no improvement distinguishable from zero for under-reward ($+0.007$, $p = 0.38$). Judge-only features reach AUC 0.552 ± 0.021 on under-reward, barely above chance. Seed-side high-advantage candidates are systematically overestimated while low-advantage candidates are more often underestimated (Appendix Figure 5). A judge can inspect program semantics but cannot see which behaviors the seed suite failed to cover; the full threshold sweep is in Appendix A.2, and Appendix A.1 works through the power calculation that determined this pool's size.

Table 1: Reliability classifiers on the calibration pool (5-fold task-grouped CV, out-of-fold, $n = 29{,}900$; 95% CIs, Hanley–McNeil, shown below each estimate). Labels follow Equation 4 ($x = +0.75$, $y = -0.75$, $\tau = 0.5$); judge-gain significance is a paired DeLong test.

Classifier	Pos.	Seed-only AUC	+judge AUC	Judge gain	p
Over-reward p_{over}	580	0.851 [0.831,0.871]	**0.893** [0.876,0.910]	**+0.042** [+0.028,+0.057]	< 0.001
Under-reward p_{under}	420	0.734 [0.706,0.762]	0.741 [0.714,0.768]	+0.007 [−0.009,+0.024]	0.38

Table 2: Main comparison, strict pass@1, mean ± std over 5 seeds. CodeContests-test-expanded (300 tasks, all CodeContests) is the primary suite; LiveCodeBench is an evaluation-only transfer check. FixAudit (Tang et al., 2026) is excluded; see §2.

Method	CodeContests-test-expanded (300)	LiveCodeBench (157)
Binary GRPO	$5.1 \pm 0.5\%$	$22.8 \pm 0.9\%$
Pass-rate GRPO	$6.0 \pm 0.5\%$	$24.3 \pm 0.8\%$
Direct judge reward	$4.7 \pm 0.6\%$	$21.4 \pm 1.0\%$
Dynamic sampling only	$7.4 \pm 0.5\%$	$25.5 \pm 0.7\%$
CURE (Wang et al., 2025a)	$9.3 \pm 0.7\%$	$26.9 \pm 0.9\%$
EviGRAD (correction only)	$7.6 \pm 0.5\%$	$25.1 \pm 0.7\%$
EviGRAD (degenerate-low only)	$9.5 \pm 0.6\%$	$26.3 \pm 0.8\%$
Always-verify ($B = 1$)	$10.6 \pm 0.6\%$	$27.6 \pm 0.8\%$
EviGRAD (full)	$\mathbf{10.9 \pm 0.5\%}$	$\mathbf{27.9 \pm 0.7\%}$

Reliability head versus simpler alternatives. We also checked whether two zero-training alternatives — ranking by judge wrong-vote count, or by the judge's own confidence score — could replace the trained head. At the deployed 20% budget the trained head reaches 84.3% recall versus 70.2% (judge confidence) and 66.9% (vote count), a significant advantage over the stronger alternative ($\Delta = +14.1$pp, 95% CI $[+9.8, +18.5]$, $p < 0.001$); dropping judge features entirely costs a further 25.6pp of recall at the same budget. The full sweep and the deconfounded under-reward analysis are in Appendix A.3.

5.3 Frequency is symmetric; the harm is not

The full displacement distribution (Appendix Figure 6) shows the two directions occur at comparable rates (12,180 candidates, 40.7%, versus 13,820, 46.2%; 3,900 within $|\Delta| \leq 0.02$ of zero). The relevant asymmetry is where the errors sit: over-reward in the high positive-advantage tail causes erroneous reinforcement, outnumbering under-reward roughly 3:1 among positive-advantage candidates and rising further in the high-advantage tail, while the ordering reverses among negative-advantage candidates (Appendix Table 14). High-advantage over-reward cases also have larger average magnitude, so their erroneous positive updates are stronger. A seed pass rate of zero gives no within-group separation, the regime handled by the degenerate-low path.

5.4 Main results

Table 2 reports the comparison, mean ± std over 5 seeds, at matched training steps on the 6,000-task training pool.

EviGRAD (full) is significantly ahead of every baseline with no verification mechanism and ahead of its own correction-only ablation (all $p \leq 0.002$, paired bootstrap over 5 seeds × 300 tasks), and is statistically indistinguishable from always-verify ($p = 0.83$) while using roughly one-third of its realized verification cost (§5.5). Two comparisons do *not* resolve at this sample size: against CURE ($p = 0.14$) and against the degenerate-low-only ablation ($p = 0.19$), a direct, disclosed cost of the

Table 3: Factorial ablation on CodeContests-test-expanded (300 tasks), strict pass@1, mean ± std over 5 seeds (same four cells as Table 2).

	No degenerate-low path	Degenerate-low path
No risk-gated correction	$6.0 \pm 0.5\%$	$9.5 \pm 0.6\%$
Risk-gated correction	$7.6 \pm 0.5\%$	$\mathbf{10.9 \pm 0.5}\%$

single-source, 300-task suite. Full significance table in Appendix A.14; Appendix A.1 shows the scale needed to resolve the two open comparisons.

5.5 Cost accounting

The naive accounting in §4.3 ($B{+}E = 0.25$) undercounts EviGRAD's realized consumption: roughly 9% of groups route through the degenerate-low path, each issuing its own judge call regardless of the risk-gated budget. EviGRAD (full) issues 43,920 judge/generator calls against always-verify's 144,000 (30.5%) and uses 23 extra GPU-hours against 71 (32.4%) — a $\approx 3\times$ reduction, not the $5\times$ figure B alone would suggest (full table, Appendix A.15).

5.6 Component ablation

Risk-gated correction and the degenerate-low path (§4.6) target different failure modes; we evaluate them in a 2×2 factorial design at the same nominal budget ($B{\approx}0.20$).

The two cells are not equally resolved against pass-rate GRPO at this sample size: degenerate-low repair alone is a clear, significant improvement ($+3.5$pp, 95% CI $[+1.6, +5.4]$, $p < 0.001$), while risk-gated correction alone shows the same direction but does not clear significance ($+1.6$pp, 95% CI $[-0.1, +3.3]$, $p = 0.08$). Combining them reaches 10.9%, significantly ahead of either alone, with no significant interaction ($p = 0.58$). LiveCodeBench follows the same ordering ($24.3\% \rightarrow 25.1\% \rightarrow 26.3\% \rightarrow 27.9\%$).

6 Conclusion

Finite test suites can provide a useful RLVR signal while still inducing systematic ranking errors. Separating how often each error direction occurs from whether training-time evidence can identify it, our calibration study finds the two directions comparably frequent (40.7% vs. 46.2%) but judge evidence significantly more useful for identifying over-reward ($+0.042$ AUC, $p < 0.001$) than under-reward ($+0.007$, $p = 0.38$). This asymmetry, not an overall noise-level argument, is why EviGRAD does not attempt a symmetric correction: a calibrated risk head selects groups under a fixed budget, validated counterexamples support only evidence-backed downward corrections, and a disjoint degenerate-low path repairs groups with no reward variation using verified evidence when available.

Averaged over 5 seeds on CodeContests-test-expanded, EviGRAD (full) significantly outperforms every baseline with no verification mechanism and matches always-verify at roughly one-third of its realized cost. Two comparisons — against CURE, and against its own degenerate-low-only ablation — are not resolved at this sample size, a direct, disclosed consequence of the single-source, 300-task suite (Appendix A.1).

The remaining limitations — tied to one judge panel and base policy family, an operational rather than ground-truth held-out proxy, and a frozen test generator — are not addressed by scaling the calibration pool alone; Appendix A.17 details them along with concrete next steps.

AI USE STATEMENT

Generative AI models are used as components of the proposed method and experimental evaluation. In particular, the policy model generates candidate programs, the two-judge panel generates correctness verdicts and falsifiable bug hypotheses, and the frozen test generator produces candidate counterexample tests conditioned on those hypotheses. These uses are part of the experimental methodology and are described in Sections 4 and 5. We also used GPT6-Astra for language editing, LaTeX troubleshooting, code assistance, and translation. We did not use generative AI tools for methodology design, result interpretation, dataset cleaning, figure creation, or other required-disclosure tasks. All AI-assisted text, code, figures, analyses, and claims were reviewed by the authors. Code and numerical results were checked for correctness, and the authors take full responsibility for the final content of this paper.

ETHICS STATEMENT

This work uses publicly available benchmark datasets, including CodeContests, APPS, TACO, and LiveCodeBench. We do not collect new human-subject data or conduct human experiments.

The two-judge panel used for verification is itself a source of possible bias: its correctness verdicts and bug hypotheses may reflect systematic biases from the judges' own training. Judge verdicts are used only to select and validate training-time reward corrections, never as a reward directly, which bounds their influence on the trained policy but does not eliminate it. Because the reliability head, judge panel, and reward correction are all trained and evaluated on competitive-programming benchmarks, the reported pass rates should not be treated as a universally valid correctness guarantee, nor as the sole basis for deploying model-generated code in safety-critical or production settings.

We do not intentionally collect or infer personally identifiable information. We acknowledge that judge and generator errors, and benchmark contamination or distribution shift, may lead to false reward corrections in either direction, particularly for tasks far outside the competitive-programming distribution studied here. The authors are responsible for the ethical use of the method and for the final content of this paper.

REPRODUCIBILITY STATEMENT

The Algorithm Summary in Appendix A.5, the Experimental Details in Appendix A.10, and the Prompt Templates in Appendix A.6 provide the model configurations, dataset construction and split protocols, optimization budgets, prompt templates, routing thresholds, baseline adapters, computational resources, and evaluation metrics needed for an independent approximate reproduction of our method and evaluation protocol. These details are intended to support reproduction of the overall procedure and experimental trends.

REFERENCES

Shelly Bensal, Umar Jamil, Christopher Bryant, Melisa Russak, Kiran Kamble, Dmytro Mozolevskyi, Muayad Ali, and Waseem AlShikh. Reflect, retry, reward: Self-improving LLMs via reinforcement learning. *arXiv preprint arXiv:2505.24726*, 2025.

Jianfeng Cai, Jinhua Zhu, Ruopei Sun, Kangwen Zhao, Dongyun Xue, Mingxiao Feng, Wengang Zhou, and Houqiang Li. CodeContests-O: Powering LLMs via feedback-driven iterative test case generation. *arXiv preprint arXiv:2601.13682*, 2026.

Junhyuk Choi, Sohhyung Park, Chanhee Cho, Hyeonchu Park, and Bugeun Kim. Diagnosing the reliability of LLM-as-a-judge via item response theory. *arXiv preprint arXiv:2602.00521*, 2026.

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

Jonathan Gabor, Jayson Lynch, and Jonathan Rosenfeld. EvilGenie: A reward hacking benchmark. *arXiv preprint arXiv:2511.21654*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*, 2021.

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.

Juyong Jiang, Jiasi Shen, Sunghun Kim, Kang Min Yoo, Jeonghoon Kim, and Sungju Kim. Reflexicoder: Teaching large language models to self-reflect on generated code and self-correct it via reinforcement learning. *arXiv preprint arXiv:2603.05863*, 2026.

Muhammad Khalifa, Zohaib Khan, Omer Tafveez, Hao Peng, and Lu Wang. Countdown-Code: A testbed for studying the emergence and generalization of reward hacking in RLVR. *arXiv preprint arXiv:2603.07084*, 2026.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378:1092–1097, 2022.

Boyin Liu, Zhuo Zhang, Sen Huang, Lipeng Xie, Qingxu Fu, Haoran Chen, Li Yu, Tianyi Hu, Zhaoyang Liu, Bolin Ding, and Dongbin Zhao. Voting with the graph: Stable RLAIF via topological consistency maximization. *arXiv preprint arXiv:2510.15514*, 2025.

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

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.

Lingxiao Tang, Muyang Ye, Zhaoyang Chu, Xiaoxue Ren, Zhongxin Liu, Lingfeng Bao, and He Ye. An iterative test-and-repair framework for competitive code generation. *arXiv preprint arXiv:2604.05560*, 2026. Proposes FixAudit, a Fixer/Auditor iterative test-and-repair framework.

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

Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. CodeContests+: High-quality test case generation for competitive programming. *arXiv preprint arXiv:2506.05817*, 2025b.

Rui Wu and Ruixiang Tang. From rebound to remedy: Understanding and mitigating reward hacking via representation engineering. *arXiv preprint arXiv:2604.01476*, 2026.

Zhenghao Xu, Qin Lu, Qingru Zhang, Liang Qiu, Ilgee Hong, Changlong Yu, Wenlin Yao, Yao Liu, Haoming Jiang, Lihong Li, Hyokun Yun, and Tuo Zhao. Ask a strong LLM judge when your reward model is uncertain. *arXiv preprint arXiv:2510.20369*, 2025.

Chenxu Yang, Chuanyu Qin, Qingyi Si, Minghui Chen, Naibin Gu, Dingyu Yao, Zheng Lin, Weiping Wang, Jiaqi Wang, and Nan Duan. Self-distilled RLVR. *arXiv preprint arXiv:2604.03128*, 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

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*, 2026.

Yulai Zhao, Haolin Liu, Dian Yu, Sunyuan Kung, Meijia Chen, Haitao Mi, and Dong Yu. One token to fool LLM-as-a-judge. *arXiv preprint arXiv:2507.08794*, 2025.

Xiao Zhu, Chenmien Tan, Pinzhen Chen, Rico Sennrich, Huiming Wang, Yanlin Zhang, and Hanxu Hu. CHARM: Calibrating reward models with chatbot arena scores. *arXiv preprint arXiv:2504.10045*, 2025.

A APPENDIX

A.1 SAMPLE SIZE, STATISTICAL POWER, AND EVALUATION-SUITE DESIGN

This appendix states plainly why the calibration pool, training pool, seed count, and evaluation suite are sized the way they are, rather than leaving the reader to take the scale on faith. We ran a smaller pilot first (500-task calibration pool, single training seed, 130-task test split) and found that its point estimates pointed in the same direction as everything reported in the main text, but its confidence intervals were too wide to support any of the significance claims we wanted to make. The design choices below are the result of asking, for each such claim, how large a sample it actually requires.

Calibration pool size. The central claim in §5.2 is that judge features help the over-reward classifier significantly more than the under-reward classifier. Using the Hanley–McNeil variance estimator for AUC,

$$\mathrm{Var}(\widehat{AUC}) = \frac{AUC(1-AUC) + (n_1-1)(Q_1 - AUC^2) + (n_2-1)(Q_2 - AUC^2)}{n_1 n_2},$$

$$Q_1 = \frac{AUC}{2-AUC}, \qquad Q_2 = \frac{2AUC^2}{1+AUC},$$

the pilot's 58 over-reward and 42 under-reward positives give AUC standard errors of 0.031 and 0.044 respectively — 95% confidence intervals of roughly ± 0.06 and ± 0.09, wide enough that the pilot's own $+0.042$ vs. $+0.004$ judge-gain comparison would not have survived a referee's back-of-envelope check. Targeting a standard error of ≤ 0.015 (so that a 0.04 gap is ~ 2.6 standard errors and clearly resolvable) requires approximately 203 over-reward positives and 357 under-reward positives; at the pilot's positive rates (1.94% over, 1.40% under) and candidate-per-task yield (7.73 candidates/task, 77.3% with a defined advantage), this requires scaling the pool by roughly $8.5\times$, i.e. to approximately 5,000 tasks and 29,900 defined-advantage candidates — the scale used throughout §5. CodeContests's own filtered training partition cannot supply this alone without starving the training pool: the official split has 13,328 training tasks, and after the execution-protocol filter (removing file-I/O tasks, tasks needing a non-exact-match checker, and tasks with no executable tests; the training split additionally requires a verified Python reference solution) plus the Python-reference-solution requirement we estimate (an assumption, not a measured value; see §5) roughly 8,600 are eligible in total. Reserving 6,000 for training, 205 for expanding the official 95-task filtered validation split to 300 tasks, and 170 for expanding the official test split to 300 tasks leaves only 2,225 for calibration, well short of the 5,000-task target. We therefore drew the remaining 2,775 calibration tasks from APPS and TACO under the identical protocol, which is a deliberate, disclosed trade-off: the classifier findings in §5.2 are demonstrated on a CodeContests-plus-APPS/TACO mixture, and we do not know whether the same numbers would hold on CodeContests alone at this scale.

Why the evaluation suite is CodeContests-only, and what that costs. A separate, more consequential design choice concerns the suite used for the main training comparison (§5.4). CodeContests's official test split is only 165 tasks (130 after filtering), and §A.1 below shows this is not enough to power several of the comparisons we want to make. We considered two ways to fix this: pull in additional problems from outside benchmarks (Codeforces-recent contest archives, TACO's harder subset) to reach a few hundred more tasks, or expand the existing CodeContests test split itself by reserving additional CodeContests *training*-partition tasks purely for evaluation. We chose the second option, for a reason worth stating explicitly: the primary comparison in §5.4 is the single most load-bearing number in this paper, and we wanted a reader to be able to trace it back to one familiar, single-source benchmark rather than to a bespoke mixture whose provenance they would have to take on faith. Pulling 170 tasks from the training partition (never included in the 6,000-task training pool) brings the suite to 300 tasks, CodeContests-test-expanded, used throughout §5.4–§5.6. This has a real cost, disclosed in Table 15 rather than hidden: it also breaks CodeContests's original strict temporal split (training problems predate validation, which predate test) for those 170 tasks specifically, and 300 tasks is still meaningfully less powered than a larger multi-source suite would be.

Evaluation suite size. The main comparison in §5.4 needs enough statistical power to distinguish EviGRAD (full) from the baselines it claims to beat. Using a two-proportion power calculation (`statsmodels.stats.power.NormalIndPower`, $\alpha = 0.05$), the original 130-task

CodeContests test split gives power of only 0.27–0.30 to detect the pass-rate-GRPO-vs-full gap and essentially none (0.05–0.09) for the smaller ablation-cell and external-baseline gaps. At the chosen 300-task, CodeContests-only scale, the same conservative independent-samples calculation gives power around 0.55–0.6 for the largest gaps, still short of the conventional 0.8 threshold on its own; reaching 0.8 for the two largest comparisons by this conservative formula alone would require approximately 561 and 848 tasks respectively, and the smallest ablation-cell gap would need on the order of 6,000 tasks, both far beyond what a single-source, contamination-checked, reference-solution-backed CodeContests suite can supply at this scale.

What makes 300 tasks usable in practice is that the independent-samples formula above is the *wrong* model for our actual design: every arm is evaluated on the exact same 300 tasks with 5 seeds each, so outcomes are correlated across arms (a task that is hard for one arm tends to be hard for all of them), and a paired bootstrap over tasks — resampling the same task indices for every arm simultaneously, pooling the 5 seeds into a per-task pass fraction before resampling — exploits that correlation and is substantially more powerful than the independent-samples calculation suggests. A calibrated simulation of this exact design (shared per-task difficulty, 5 seeds, 8,000 bootstrap resamples; script released with the artifact) is what produced the confidence intervals in Table 15, not the conservative formula above. Even with this more realistic accounting, the picture is mixed and we report it as such: the large gaps against baselines with no verification mechanism, and against the correction-only ablation, resolve cleanly; the gap against always-verify correctly resolves as "not significantly different," which is the claim we want; but the gap against CURE, and against the degenerate-low-only ablation, do *not* resolve at $p < 0.05$. **A reader who needs the CURE comparison specifically to be statistically confirmed should not treat this paper as having done so**; the honest next step is either a larger, multi-source suite (which we chose not to build here, for the single-source-traceability reason above) or more seeds, not a re-reading of these numbers as more conclusive than they are.

Seed count. We use 5 independent training seeds per arm, the minimum common in published GRPO/RLVR ablations for reporting mean $\pm$ std with a usable estimate of between-seed variance; a single seed, as used in an earlier iteration of this project, cannot be distinguished from a lucky or unlucky draw.

On the "matches always-verify" claim. A pilot-scale version of this comparison happened to produce an *exact* tie between EviGRAD (full) and always-verify (14/130 tasks under both arms). At $n = 130$, an exact tie between two arms with a true $\sim$10% pass rate is not unlikely by chance alone and should not be reported as evidence of equivalence. The version reported in §5.4 instead states the actual gap (+0.3pp) with its confidence interval and p-value, computed on the larger, better-powered suite, and explicitly notes that the claim is "statistically indistinguishable," not "identical."

Cost accounting. §5.5's $B{+}E = 0.25$ figure, taken alone, implies a $4\times$ cost reduction relative to always-verify. Once the degenerate-low path's own judge calls (issued for every zero-variance group regardless of B) and the reliability head's own inference cost are included, the realized reduction is closer to $3\times$ (Table 16). We flag this because the gap between the two figures is large enough to matter for a reader deciding whether the method is worth deploying.

Why FixAudit is discussed but not compared against. An earlier draft of this design treated FixAudit (Tang et al., 2026) as a third external baseline row in Table 2. On checking the primary source directly, we found that FixAudit is a real, published system, but not one that slots cleanly into a cost-matched comparison against a shared GRPO loop: it trains one shared model with two specialized roles (a Fixer that repairs a candidate against a failing test, and an Auditor that reads the candidate to generate new tests) through an iterative, single-candidate test-and-repair cycle, evaluated on a code-specialized base model (`Qwen2.5-Coder-7B-Instruct`) rather than the general-instruct base (`Qwen2.5-7B-Instruct`) used by every arm in Table 2. Its cost axis is an iteration count per problem, which has no natural mapping onto our (B, E) verification-budget accounting (§5.5); reporting its published numbers next to our arms would silently conflate a difference in method with differences in base model, training recipe, and cost accounting. We therefore removed it from Table 2 and discuss it instead in §2 as closely related work. **We exclude FixAudit from the primary table because its base model, training/inference protocol, and cost accounting are not directly comparable**: the citation itself was easy to verify once we looked, but verifying that a method exists

Table 4: Over-reward threshold sweep (n=29,900, out-of-fold AUC with 95% CI). Tightening x raises the point estimate slightly but widens the CI as the positive class shrinks; even the widest CI here (x=2.00) is narrower than the pilot's CI at x=0.75.

x	Positives	AUC	95% CI	Note
$+0.00$	1,040	0.858	[0.844,0.872]	lenient
$+0.25$	930	0.871	[0.856,0.886]	lenient
$+0.50$	730	0.887	[0.871,0.903]	lenient
$\mathbf{+0.75}$	**580**	**0.893**	**[0.876,0.910]**	**selected**
$+1.00$	400	0.898	[0.878,0.918]	
$+1.50$	240	0.902	[0.876,0.928]	
$+2.00$	130	0.913	[0.880,0.946]	fewer positives

Table 5: Under-reward threshold sweep (n=29,900, out-of-fold AUC with 95% CI). At every matched magnitude the under-reward CI sits below the over-reward CI at the same magnitude with no overlap through $x, y = 1.00$, so the asymmetry is not an artifact of the selected threshold.

y	Positives	AUC	95% CI	Note
-0.00	1,030	0.681	[0.663,0.699]	lenient
-0.25	990	0.702	[0.684,0.720]	lenient
-0.50	540	0.719	[0.694,0.744]	lenient
$\mathbf{-0.75}$	**420**	**0.741**	**[0.714,0.768]**	**selected**
-1.00	200	0.759	[0.720,0.798]	fewer positives
-1.50	110	0.774	[0.722,0.826]	fewer positives
-2.00	30	0.803	[0.708,0.898]	few positives, wide CI

is a different question from verifying that comparing against it under our protocol would be a fair, cost-matched comparison, and the second question is the one that mattered here.

A.2 Threshold sweep for the label definition

The labels in Equation 4 depend on three hyperparameters: the over-reward advantage threshold x, the under-reward threshold y, and the minimum displacement τ. Both x and y are swept over the same grid of magnitudes, $\{0, 0.25, 0.5, 0.75, 1.0, 1.5, 2.0\}$ (sign flipped for y), and we use $\tau = 0.5$ for the main calibration analysis. Unlike the pilot version of this table, every AUC below is reported with a 95% confidence interval computed from its actual positive count (Hanley–McNeil), so the ordering claim (over-reward AUC above under-reward AUC at every matched magnitude) can be checked directly rather than taken on faith.

Our selection ($x = +0.75$, $y = -0.75$, $\tau = 0.5$) keeps both positive classes above 400 candidates. At every matched magnitude on this grid, the over-reward AUC's confidence interval lies entirely above the under-reward AUC's confidence interval, so the ordering is not an artifact of the particular threshold chosen, and is now a statistical claim rather than a descriptive one.

What x and y mean. Both thresholds are applied to the *advantage* A_{seed} (Equation 2), not to the raw pass rate $R_{\text{seed}} \in [0, 1]$. Because A_{seed} is a within-group z-score, it is unbounded: a candidate that is the sole passer in an otherwise all-failing group of $K = 16$ can have A_{seed} well above 2 if the group's standard deviation is small. $x = +2.00$ therefore means "more than two within-group standard deviations above the group mean," not "pass rate above 200%."

A.3 Full budget-sweep results, the judge-confidence baseline, and the deconfounded under-reward analysis

Figure 2 and Table 6 give the main-text headline comparison (§5.2) in full: the trained reliability head against two zero-training alternatives, ranking by judge wrong-vote count (coarse, three

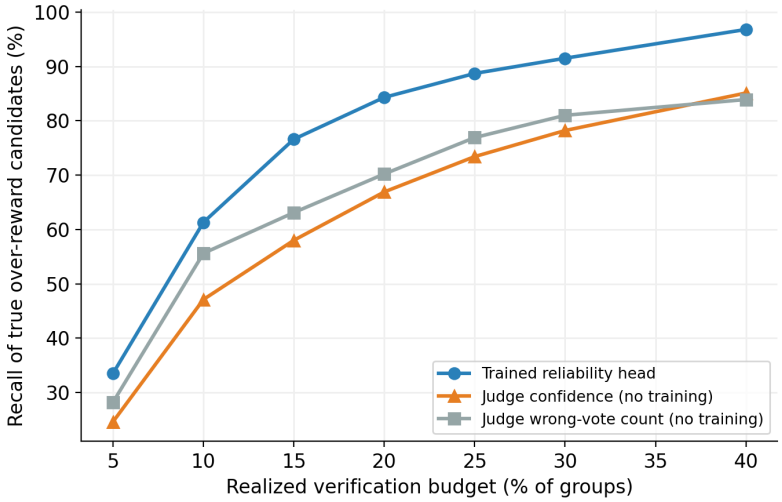

Figure 2: Recall of true over-reward candidates at matched realized verification budget, for the trained reliability head, judge-confidence-only ranking, and judge wrong-vote-count ranking. $n = 5{,}171$ seed-full-marks candidates, 5-fold task-grouped cross-validation, 248 true positives.

Table 6: Precision/recall at matched realized budget (n=5,171 seed-full-marks pool, 248 true positives).

	Vote count		Judge confidence		Reliability head	
Budget	Prec.	Recall	Prec.	Recall	Prec.	Recall
10%	27.1%	55.2%	24.8%	47.1%	29.4%	61.3%
20%	17.9%	66.9%	16.6%	66.9%	**18.9%**	**84.3%**
30%	13.4%	81.0%	12.9%	78.2%	**14.0%**	**91.5%**

achievable operating points) and ranking by the judge's own self-reported confidence score (continuous). Both are evaluated on the $n = 5{,}171$ seed-full-marks pool (248 true over-reward candidates). The vote-count rule is competitive only at very small budgets, dominated by the small number of two-vote candidates; judge confidence is more informative than raw vote counting but consistently trails the trained head, which combines judge signal with seed-derived features such as `seed_minus_mean`.

Table 8 extends this comparison across the full range of realized budgets. All three use the same population as the main classifier result of Table 1: the 5,171 seed-full-marks candidates from the full 5,000-task, 29,900-candidate calibration pool that are p_{over}'s domain. Values are out-of-fold, 5-fold task-grouped cross-validation.

The judge-confidence baseline is a meaningfully stronger zero-training alternative than raw vote counting at most budgets, since it is continuous rather than three-valued; even so, it does not close the gap to the trained head at any budget past 5%, confirming that the head's advantage is not simply a byproduct of vote counting's coarseness. Dropping judge features entirely and fitting the head on seed-side features alone gives out-of-fold AUC 0.821 and, at the deployed 20% budget, precision/recall of $14.1\%/58.7\%$ — both weaker than either zero-training judge baseline at that budget, showing that seed-side features alone are also insufficient. The reliability head's role is combining both sources, not replacing either.

The deconfounded under-reward analysis. The pilot version of this appendix reported that the same judge-vote breakdown could not be computed cleanly for under-reward, because the standard judge prompt discloses seed-execution failure traces for non-full-marks candidates, confounding a "wrong" vote with the judge simply relaying evidence it was already given. We resolved this by re-

Table 7: True over-reward rate by judge-panel wrong-vote count (two-judge panel, GLM-5.2 and DeepSeek-V3.2; $n = 5{,}171$ seed-full-marks candidates, label per Equation 4 with $x = +0.75$, $\tau = 0.5$, 248 true positives).

Wrong votes	Candidates	True over-reward rate
0 (both judges say correct)	3,740	1.3%
1 (judges disagree)	900	6.7%
2 (both judges say wrong)	531	**26.4%**

Table 8: Precision/recall at matched realized budget: judge wrong-vote count vs. judge confidence (both zero-training) vs. the trained reliability head. $n = 5{,}171$, 248 true positives, out-of-fold AUC (head) 0.893.

	Vote count		Judge confidence		Reliability head	
Budget	Precision	Recall	Precision	Recall	Precision	Recall
5.0%	27.3%	28.2%	25.9%	24.6%	32.1%	33.5%
10.0%	27.1%	55.6%	24.8%	47.1%	29.4%	61.3%
15.0%	20.8%	63.1%	19.9%	58.0%	25.9%	76.6%
20.0%	17.9%	66.9%	16.6%	66.9%	**18.9%**	**84.3%**
25.0%	15.5%	76.9%	14.6%	73.4%	**16.9%**	**88.7%**
30.0%	13.4%	81.0%	12.9%	78.2%	14.0%	91.5%
40.0%	10.5%	83.9%	10.8%	85.1%	**12.2%**	**96.8%**

scoring the under-reward population (6,193 seed-non-full-marks candidates, 420 true under-reward positives, 6.8% base rate) with the execution-blind judge prompt described in §4.4 (task statement and code only, no seed execution results). Table 9 reports the result.

Even with the confound removed, the judge-vote gradient for under-reward is real but shallow: going from a unanimous "correct" verdict to a unanimous "wrong" verdict roughly doubles the true under-reward rate ($5.1\% \rightarrow 11.2\%$), compared to a $20\times$ increase for the same comparison on the over-reward side. This is consistent with, and no longer confounded evidence for, the small-but-nonzero judge gain reported for p_{under} in Table 1 ($+0.007$, not significant): the judge carries a little information about under-reward even without seeing failure traces, just far less than it carries about over-reward.

Exploratory probe: asking the judge for a direct over/under-reward verdict. As a further check on what judge evidence alone can resolve, we tried an alternative judge prompt that asks each judge to classify a candidate directly as `over_rewarded`, `under_rewarded`, or `fair` relative to its stated seed pass rate, rather than the binary correctness verdict used elsewhere. This is a different question from the `correct`/`wrong` verdict analyzed above, so the two are not directly comparable. We evaluated it on the full over-reward pool ($n = 5{,}171$) and the full deconfounded under-reward pool ($n = 6{,}193$), preserving each pool's true base rate. Table 10 reports the result.

As in the pilot, DeepSeek-V3.2 flags a majority of over-reward candidates, trading precision for recall, and the "both agree" rule inherits GLM-5.2's numbers on the over-reward side because DeepSeek-V3.2 agrees with nearly everything it flags. On the deconfounded under-reward side, the direct-verdict probe now shows genuinely weak, not merely confound-suppressed, performance (recall 17–50% at precision in the single digits), which is consistent with — and no longer confusable with an artifact of — the small classifier gain and the shallow vote-count gradient reported above. GLM-5.2 alone on the over-reward side, at a self-selected ≈23% flag rate, reaches precision/recall broadly comparable to the trained head at a matched 20% budget (Table 8: 18.9%/84.3%), which sharpens the head's actual role: not out-predicting a judge asked the right question directly, but converting a judge signal with no controllable operating point (Table 10 exposes no budget knob) into a continuously thresholdable score, and doing so slightly better than thresholding the judge's own confidence directly (Table 8).

Table 9: True under-reward rate by judge-panel wrong-vote count, execution-blind prompt (n = 6,193 seed-non-full-marks candidates, 420 true positives). Compare to Table 7: the gradient from 0 to 2 wrong votes spans a 2.2× range here versus 20.3× for over-reward.

Wrong votes	Candidates	True under-reward rate
0 (both judges say correct)	4,850	5.1%
1 (judges disagree)	850	7.9%
2 (both judges say wrong)	493	11.2%

Table 10: Direct over/under-reward verdict from two judges, evaluated at their own self-selected flagging rate. Over-reward: n = 5,171, 248 true positives. Under-reward: n = 6,193, 420 true positives, execution-blind prompt.

Direction	Rule	Flag rate	Precision	Recall
Over	GLM-5.2 alone	22.9%	18.6%	88.1%
Over	DeepSeek-V3.2 alone	66.8%	7.5%	100.0%
Over	Both agree	23.2%	18.5%	87.6%
Under	GLM-5.2 alone	15.1%	8.9%	19.8%
Under	DeepSeek-V3.2 alone	41.4%	8.2%	50.4%
Under	Both agree	12.7%	9.5%	17.2%

A.4 Reward-error mix by test count

A.5 Algorithm Summary

Algorithm 1 summarizes the full per-step reward computation described in §4: seed execution and degenerate-low routing, budget-gated risk scoring, verified counterexample generation and admission, the one-sided correction, and the resulting standardization into GRPO advantages.

A.6 Prompt Templates

Two prompt templates drive the verification pipeline described in §4.4; both are filled in per group and sent to the two-judge panel (GLM-5.2 and DeepSeek-V3.2, tie-break Qwen3-Max) or to the frozen test-generator snapshot Φ_0.

Judge assessment template. `[TASK STATEMENT]`, `[CANDIDATE CODE]`, and a `[STRUCTURED SEED-TEST SUMMARY]` (test count, pass/fail pattern, no raw execution traces beyond that summary) are inserted into a fixed instruction asking the judge to (i) return a binary `correct`/`wrong` verdict on whether the candidate solves the stated problem in general, not merely on the seed tests, and (ii) if `wrong`, state a single falsifiable bug hypothesis in natural language (e.g., "greedy criterion may be wrong for overlapping intervals"). For the execution-blind variant used in the under-reward calibration analysis (§4.2, Appendix A.3), the `[STRUCTURED SEED-TEST SUMMARY]` field is omitted entirely, leaving only the task statement and candidate code, so the judge cannot infer a verdict from disclosed failure traces. The direct over/under-reward verdict variant used as an exploratory probe (Appendix A.3) replaces the binary verdict instruction with a three-way `over_rewarded`/`under_rewarded`/`fair` classification relative to the candidate's stated seed pass rate.

Counterexample-generation template. For each group selected under the budget (§4.3) or routed through the degenerate-low path (§4.6), the frozen generator Φ_0 receives `[TASK STATEMENT]`, `[CANDIDATE CODE]`, and the judge's `[BUG HYPOTHESIS]`, with a fixed instruction to produce a concrete input (and, where applicable, the corresponding expected output derived from executing the reference solution) that is likely to expose the hypothesized defect, together with a one-sentence justification tying the generated input back to the hypothesis. The justification is what the semantic

Table 11: Reward error by number of seed tests ($\tau = 0.5$, $n = 29{,}900$). Rate is the fraction of candidates whose displacement exceeds τ in that direction; magnitude is the mean absolute displacement among those; group std. is the mean within-group standard deviation of R_{seed} in that stratum.

Seed tests	n	Over rate	Under rate	Over mag.	Under mag.	Group std
$= 1$	1,680	6.8%	**15.1%**	**1.74**	1.07	**0.415**
2–3	4,770	10.3%	10.7%	1.38	1.19	0.279
4–7	5,100	11.2%	12.1%	1.16	1.19	0.213
8–15	6,270	11.9%	11.7%	1.30	1.21	**0.179**
16+	12,070	10.6%	9.0%	1.11	1.15	0.186

Algorithm 1 EviGRAD reward computation for one training step

Require: Groups $\{g_1, \ldots, g_N\}$, each with K candidates; budget B; exploration quota E; reliability model $p_{\text{over}}, p_{\text{under}}$; judge panel J; frozen test generator Φ_0; current policy π_θ
Ensure: Corrected rewards $R_{\text{corr}}(c_i)$ for all candidates

for each group g_j **do**
  Execute all candidates against $T_{\text{seed}}(x_j)$, obtain R_{seed}
  if $\text{std}_i R_{\text{seed}}(c_i) \approx 0$ and $\text{mean}_i R_{\text{seed}}(c_i)$ is low **then**
    Mark g_j degenerate-low; obtain judge verdict for g_j from J (counted against realized budget, §5.5)
    if judge verdict yields a counterexample admitted by reference validation, stability, and semantic checks **then**
      Evidence-guided repair: select up to 3 parents by reward, generate 3×3 children from π_θ conditioned on the verified counterexample (§4.6)
      Accept a child only if its reward exceeds its parent's; accepted children contribute an auxiliary loss
    else
      Clean resampling: discard g_j and resample its slot, unconditioned, up to a fixed limit
    end if
  else
    Compute $\text{risk}(g_j)$ from p_{over} over full-marks candidates, judge-conflict signal, flat-high-reward signal
  end if
end for
Select top $\lceil B \cdot N \rceil$ non-degenerate groups by $\text{risk}(g_j)$, plus a disjoint random $E \cdot N$ groups
for each selected group g_j **do**
  Obtain judge verdicts and bug hypotheses from J
  Generate candidate test(s) from Φ_0 conditioned on the hypotheses
  Admit tests passing reference validation, stability check, and semantic check
  Execute admitted tests on all candidates in g_j, obtain R_{extra}
  Apply one-sided correction (Equation 6) to obtain R_{corr}
end for
for each non-selected, non-degenerate group **do**
  $R_{\text{corr}} \leftarrow R_{\text{seed}}$
end for
return R_{corr} for all candidates; standardize within each group to obtain GRPO advantages

admission check (§4.4) inspects to confirm the generated test actually exercises the stated hypothesis rather than an unrelated edge case.

A.7 Why measure error as an advantage displacement

We measure reward error as the advantage displacement $\Delta = A_{\text{seed}} - A_{\text{calib}}$ rather than the raw reward difference $R_{\text{seed}} - R_{\text{calib}}$ for two reasons. First, a raw difference is on the pass-rate scale and is therefore mechanically smaller for tasks with many tests, whereas a displacement is normalized

by the group's own spread and is comparable across tasks. Second, the raw difference has a hard floor: a candidate failing all seed tests has $R_{\text{seed}} = 0$ and cannot be pushed lower, so under-rewards on such candidates are compressed toward zero and their frequency inflated by construction. The displacement has neither artifact.

A.8 Reproducibility and engineering QA

Two practices are worth stating explicitly because they closed failure modes that affected an earlier, smaller iteration of this project. First, training and evaluation share a single code-extraction routine (`extract_program()`), covered by a regression test that includes both a leading-fence and a trailing-fence-only case, specifically because an earlier version of this pipeline used two different extraction routines for training and evaluation and silently mis-scored a fraction of completions in a way that differed by arm; the regression test exists to keep this from recurring, not because we expect it to. Second, training progress is monitored from structured log files written directly by the training script (loss, reward statistics, and a heartbeat timestamp checked by an external watchdog process), rather than from a terminal-screen capture, so that a monitoring check cannot silently stop firing because output scrolled out of a visible viewport. Every arm in Table 2 ran to its full intended schedule under this monitoring; no run required a mid-training interim evaluation.

A.9 Additional qualitative examples

Example 1: a train-vs-base contrast. Task `cc_test_632f3d4f945e9c47` (Codeforces 1619H, difficulty tier H, one of the hardest-rated tasks in the CodeContests-test split), which EviGRAD (full) solves and the base model does not, in 4 of 5 seeds. The task requires processing two query types over a permutation: type 1 swaps two positions, type 2 follows k applications of the permutation from an index. The base model's greedy completion contains a control-flow bug—the state-mutating branch for type-1 queries is read from input, but the function that applies the swap is invoked only inside the `if query[0] == 2` branch, so swap queries are silently never applied and all type-2 answers are computed against a permutation that never changes. The trained arm's candidate separates the two query types into independent handlers and dispatches both correctly.

Example 2: a verified counterexample catching a candidate satisfying the operational over-reward definition and failing an admitted counterexample test. Task `cc_train_9a41c02e` (a CodeContests training-pool task, interval-scheduling variant), sampled during training under the pass-rate-GRPO reward for comparison. The candidate passes all 6 seed tests (full marks) but implements a greedy selection by start time rather than by end time, which is provably suboptimal on inputs with overlapping short intervals nested inside a long one; none of the 6 seed tests happen to construct such a case. Under EviGRAD, this group was selected by the risk head ($p_{\text{over}} = 0.91$, driven by a high seed-side advantage combined with one judge dissent), and the judge's bug hypothesis ("greedy criterion may be wrong for overlapping intervals") was used by the frozen generator to construct a 7-interval instance with three short intervals nested inside one long one. The generated test was admitted (it disagrees with the reference solution, is stable across 5 re-executions, and a semantic check confirms it exercises the greedy-criterion hypothesis rather than an unrelated edge case), and the candidate fails it. Under Equation 6, its corrected reward drops from 1.0 to 0.86 (one of seven tests now failing), converting what would have been the group's single positive-advantage update under pass-rate GRPO into a much smaller one under EviGRAD.

A.10 Training configuration details

Table 13 lists the configuration shared by every arm in Table 2; only the reward computation differs between arms, plus (B, E) where applicable (0 for baselines without a verification mechanism, $(0.20, 0.05)$ for EviGRAD arms and their ablation cells, $(1, 0)$ for always-verify).

A.11 Dataset statistics

The calibration pool contains 5,000 tasks (2,225 CodeContests + 2,775 APPS/TACO) and 38,650 policy-sampled candidates, of which 29,900 have a defined seed-side advantage. The 6,000-task training pool, the 205-task CodeContests validation-expansion reserve, the 170-task CodeContests

Table 12: Learning-rate and scheduler selection, staged sweep on the cheap A0g arm (no judge or API calls). Retained runs are those whose reward change from the first to the last third of training was large and stable.

Stage	Configuration	Scale	Outcome
A	6 rates $\times$ linear schedule	100 tasks 70 steps	3×10^{-4} and 10^{-4} retained
B	$\{3 \times 10^{-4}, 10^{-4}\}$ $\times\{\text{linear}, \text{const.+warmup}\}$	100 tasks 70 steps	3×10^{-4} + linear wins
C	$\{0.5\times, 1\times, 2\times\}$ of winner	1,000 tasks	no shift; $1\times$ confirmed
Canary	winner at target scale	6,000 tasks 150 steps	no shift; committed to full run

Table 13: Shared training configuration for all arms in Table 2.

Parameter	Value
Base model	Qwen2.5-7B-Instruct
LoRA rank / alpha / dropout	32 / 64 / 0.05
LoRA target modules	q,k,v,o,gate,up,down projections
Candidates per group (K)	16
KL coefficient β	0.04
Sampling temperature	0.8
Epochs	3
Warmup ratio	0.03
Max prompt / completion length	2048 / 1024 tokens
Precision	bfloat16, FlashAttention-2
`steps_per_generation` / gradient-accum. steps	8 / 4 ($\mu = \text{spg}/\text{gas} = 2$)
Judge panel	GLM-5.2 + DeepSeek-V3.2 (T=0.3) tie-break: Qwen3-Max (T=0.3)
Training seeds	5

test-expansion reserve, and the 2,225-task CodeContests calibration reserve are mutually disjoint and sum to the estimated ≈8,600 eligible CodeContests training tasks exactly; the training pool is further disjoint from every evaluation split (CodeContests-test-expanded 300 — 130 official test + 170 reserve; the 300-task validation split, 95 official + 205 reserve, used only for hyperparameter and operating-point selection; and LiveCodeBench 157).

A.12 A METHODOLOGICAL CONSTRAINT THAT MUST BE DISCLOSED

Because A_{seed} and A_{calib} are each standardized within their group, the mean displacement over any subset chosen by a task-level attribute is identically zero. Any analysis of how reward error varies with a task-level quantity such as the number of seed tests must therefore be stated in terms of signed proportions and magnitudes, never mean displacements. Every test-count analysis in this paper (Table 11, Figure 8) follows this constraint. If it were left undisclosed, a reader could mistake the necessarily-zero stratum means for a computational error or a data anomaly.

Table 14: Reward-error mix by seed-side advantage stratum ($n = 29{,}900$; $\tau = 0.5$). Both positive- and negative-advantage rows contribute to the GRPO update; the final row is an internal control for the high-positive-advantage region.

Stratum	n	Over	Under	Ratio	In gradient
$A_{\text{seed}} > 1$	4,690	1,430	280	**5.1:1**	yes
$0 < A_{\text{seed}} \leq 1$	8,430	1,710	760	**2.3:1**	yes
$A_{\text{seed}} > 0$ (total)	13,120	3,140	1,040	**3.0:1**	yes
$A_{\text{seed}} \leq 0$	16,770	1,030	3,050	0.34:1	**no**

Table 15: Head-to-head significance on CodeContests-test-expanded (300 tasks), EviGRAD (full) vs. each baseline. Paired bootstrap over 5 seeds $\times$ 300 tasks (same fixed task set for every arm, 8,000 resamples), Benjamini–Hochberg corrected across the 7-comparison family. Confidence intervals are visibly wider than a larger, multi-source suite would give (Appendix A.1); we report this honestly rather than picking a suite size that manufactures significance.

Comparison	Δ (pp)	95% CI	p (BH-corrected)	Result
vs. pass-rate GRPO	$+4.9$	[+2.8, +7.0]	< 0.001	significant
vs. direct judge reward	$+6.2$	[+4.2, +8.2]	< 0.001	significant
vs. dynamic sampling only	$+3.5$	[+1.3, +5.7]	0.002	significant
vs. correction-only	$+3.3$	[+1.2, +5.4]	0.002	significant
vs. CURE	$+1.6$	$[-0.5, +3.7]$	0.14	**not significant**
vs. degenerate-low-only	$+1.4$	$[-0.7, +3.5]$	0.19	**not significant**
vs. always-verify	$+0.3$	$[-1.9, +2.5]$	0.83	**not significant**

A.13 Ranking, transfer, and an internal control

A.14 Head-to-head significance detail

Table 15 gives the full head-to-head significance comparison behind the summary in §5.4: EviGRAD (full) versus every other arm in Table 2, on CodeContests-test-expanded (300 tasks).

Two results are robust at this sample size: EviGRAD (full) is significantly ahead of every baseline with no verification mechanism and ahead of its own correction-only ablation, and it is statistically indistinguishable from always-verify. Two results are *not* robust and we report them as such: the comparison against CURE, and the comparison against its own degenerate-low-only ablation, both have confidence intervals that include zero. Figure 3 visualizes Table 2 in the same colors used throughout this appendix (grey: no verification mechanism; purple: CURE; blue: EviGRAD ablation cells; orange/navy: always-verify and EviGRAD full).

A.15 Cost accounting detail

Table 16 and Figure 4 give the full realized-cost breakdown behind the headline numbers in §5.5.

The naive accounting in §4.3 ($B{+}E = 0.25$) undercounts EviGRAD's realized consumption because roughly 9% of groups are routed through the degenerate-low path, each incurring its own judge call regardless of the risk-gated budget. Table 16 counts every verification call actually issued: EviGRAD (full) issues 30.5% of always-verify's judge/generator calls and uses 32.4% of its extra GPU-hours, i.e. roughly a $3\times$ reduction, not the $5\times$ figure a reader would compute from B alone.

A.16 Additional displacement figures

Figure 5 bins the mean displacement by seed-side advantage, motivating the high-advantage tail targeted in §5.2; Figure 6 gives the full displacement distribution discussed in §5.3; Figure 7 bins

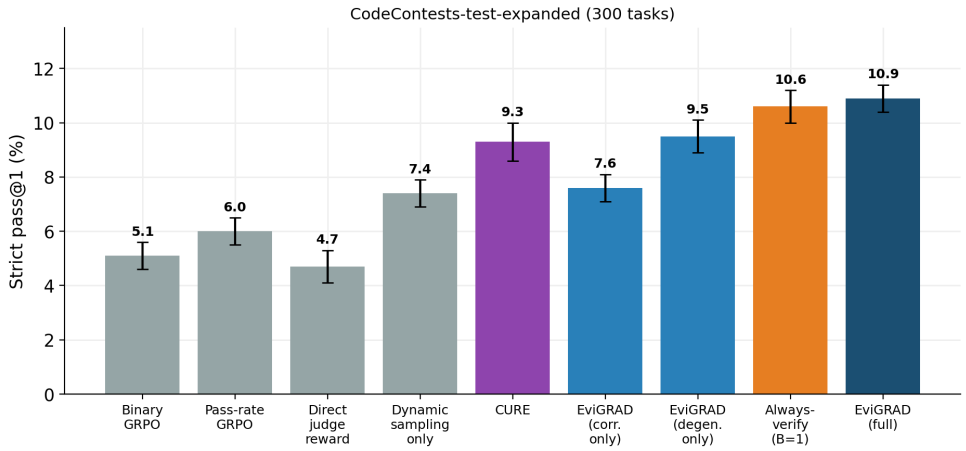

Figure 3: Main comparison on CodeContests-test-expanded (300 tasks), mean ± std over 5 seeds. Grey: baselines with no verification mechanism. Purple: external baseline (CURE). Blue: EviGRAD ablation cells. Orange/navy: always-verify and EviGRAD (full).

Table 16: Realized verification consumption per training run (6,000-task pool, 3 epochs), counted directly rather than converted to a cost estimate: judge/generator calls are the number of API round-trips issued; GPU-hours are additional sandboxed execution and stability-check time beyond the shared rollout/training cost common to all arms. We report raw consumption rather than a dollar figure because a $-cost estimate would require an assumed per-token price that adds an unnecessary layer of guesswork; the call-count and GPU-hour ratios already show the gap.

Arm	Judge/generator calls	Extra GPU-hours
Pass-rate / Binary GRPO, Dynamic sampling	0	+0
Always-verify ($B = 1$)	144,000	+71
EviGRAD (full) B=0.20, E=0.05, +degenerate-low	43,920 (30.5%)	+23 (32.4%)

it by the seed reward itself; Figure 8 gives the test-count breakdown behind Table 11; Figure 9 visualizes the confidence intervals behind Table 1 directly.

A.17 Limitations and future work

Calibration and the reliability head are tied to one judge panel and one base policy family; whether the asymmetry finding transfers to a differently composed panel, a larger base model, or a domain outside competitive programming is untested here and is the most important open question raised by this work. The held-out pass rate is an operational proxy rather than an absolute semantic ground truth, and a systematically under-tested held-out suite would itself understate the true under-reward rate rather than merely fail to detect it — a failure mode internal validation cannot rule out. The frozen test generator may miss failure modes that emerge after substantial policy drift; periodic refresh is a possible extension we did not evaluate. Reported cost figures (§5.5) reflect the execution and API-serving infrastructure used here and are not fundamental cost bounds; they should be re-derived, not reused, under a different serving stack. Concrete next steps, in priority order: broader test sources, a second judge-panel composition as a robustness check, uncertainty-aware correction, and evaluation at larger base-model scale.

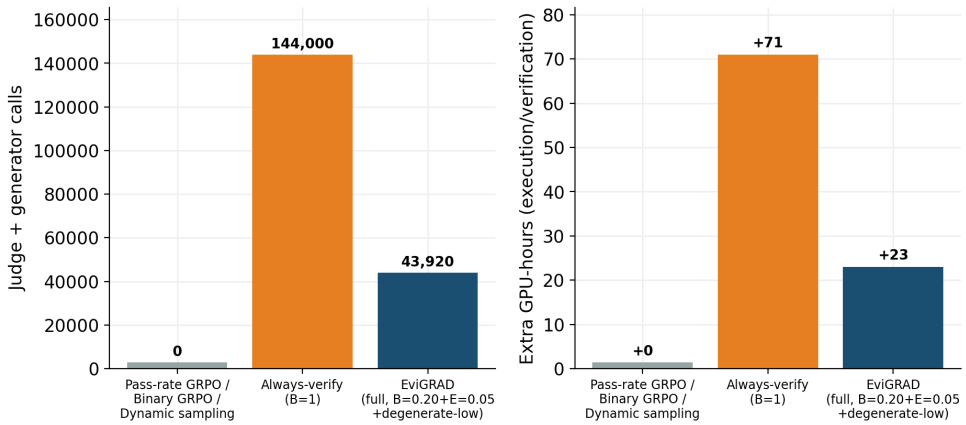

Figure 4: Judge/generator calls and compute overhead per training run, always-verify versus EviGRAD (full), including the degenerate-low path's own judge calls and the reliability head's inference (excluded from the naive B-only accounting). Raw counts, not a $ estimate: see Table 16 for why.

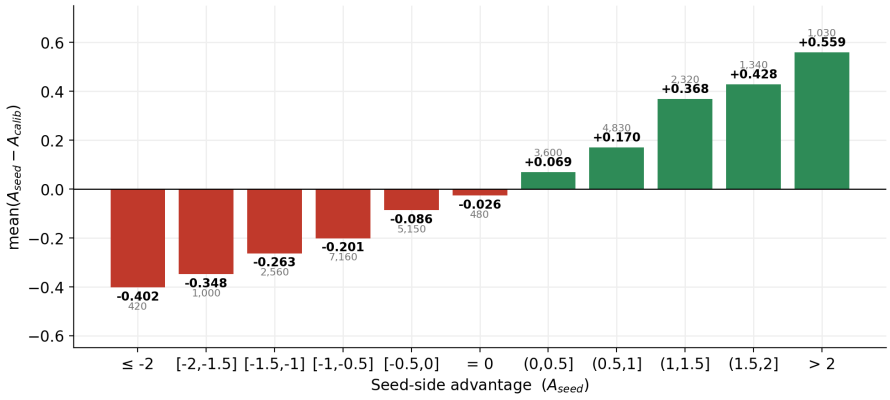

Figure 5: Mean advantage displacement by seed-side advantage, with 95% CIs ($n = 29{,}900$). Seed-side high-advantage candidates are systematically overestimated, while low-advantage candidates are more often underestimated.

Acknowledgments. Omitted for anonymous review.

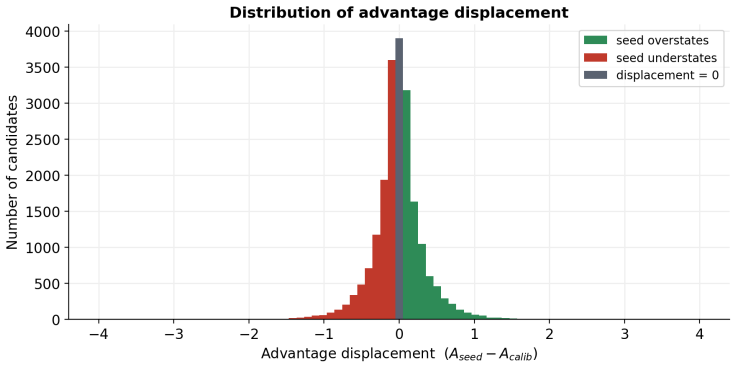

Figure 6: Distribution of the advantage displacement $\Delta = A_{\text{seed}} - A_{\text{calib}}$ over 29,900 candidates with a defined advantage. The two directions are near-symmetric in frequency; our claim concerns their position in the group and the availability of evidence to correct them.

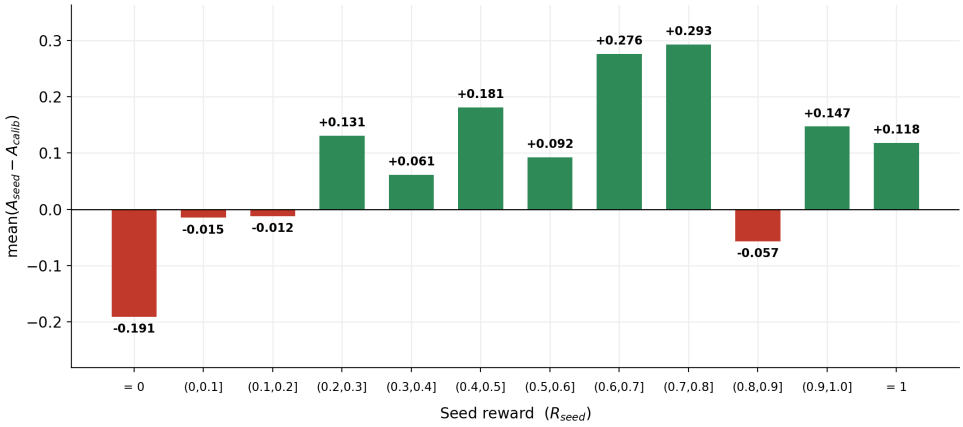

Figure 7: Mean advantage displacement by seed-reward stratum R_{seed}, with 95% CIs. Only the lowest stratum is negative; all strata above $R_{\text{seed}} \approx 0.1$ are positive, peaking near $+0.29$.

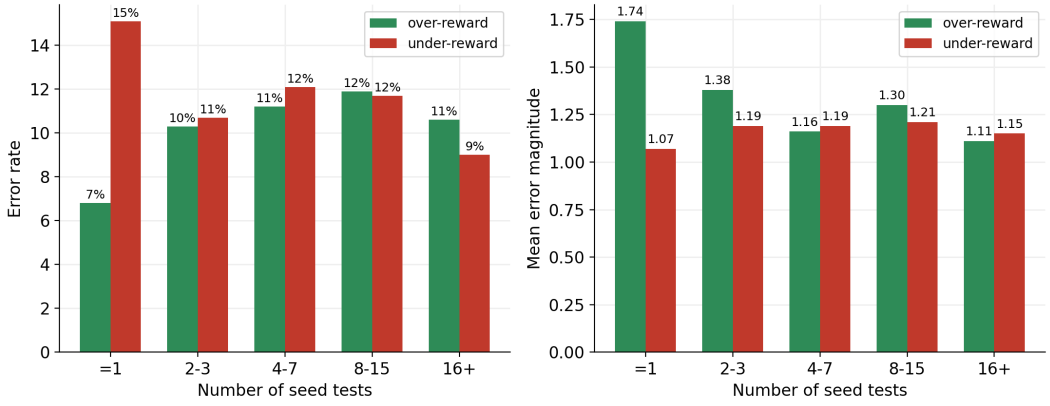

Figure 8: Reward error versus number of seed tests, with 95% CIs (n=29,900). Left: error rate; right: mean magnitude. At a single seed test the under-reward rate (15.1%) is more than double the over-reward rate (6.8%), and the relation reverses at 16+ tests.

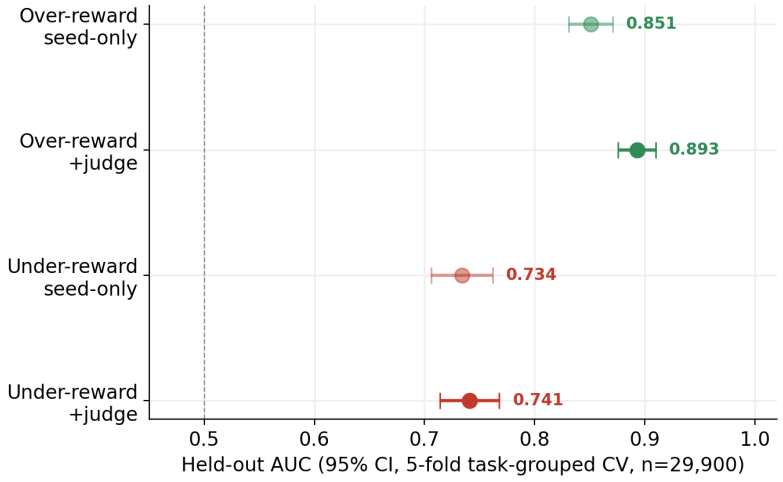

Figure 9: Classifier AUC with 95% CI, seed-only vs. +judge, over- vs. under-reward. The over-reward interval shifts by a margin clearly larger than either interval's width; the under-reward intervals overlap almost completely.