# EXECUTION-GROUNDED CREDIT ASSIGNMENT FOR GRPO IN CODE GENERATION*

**Abhijit Kumar**†
abhijitkumar4293@gmail.com

**Natalya Kumar**
natalya2kumar@gmail.com

**Shikhar Gupta**
shik1470@gmail.com

## ABSTRACT

Critic-free reinforcement learning with verifiable rewards (RLVR) improves code generation by optimizing unit-test pass rates, but GRPO-style updates suffer from coarse credit assignment: a single outcome signal is spread uniformly across long programs even when failure stems from a localized semantic error.

We propose **Execution-Grounded Credit Assignment (EGCA)**, which localizes GRPO updates using execution traces. For programs that satisfy algorithmic constraints but fail tests, EGCA executes the candidate and a canonical reference solution (curated once offline; used for analysis, not supervision) under identical instrumentation, identifies the earliest semantic divergence, and assigns advantage only to the corresponding token span while masking downstream tokens.

EGCA is a drop-in modification requiring no critic, auxiliary loss, or learned verifier, yielding 82.1% pass@1 on HumanEval (+3.1 over GRPO) and 68.9% on MBPP (+1.5) with 18% wall-clock overhead.

## 1 INTRODUCTION

Reinforcement learning with verifiable rewards (RLVR), where generated programs are evaluated by unit tests, has become a standard post-training approach for improving code generation models. Critic-free objectives such as GRPO (Shao et al., 2024) are appealing: they avoid a value function and directly optimize functional correctness. As base models improve, however, the nature of failures shifts. Modern models increasingly produce code that is syntactically valid, structurally plausible, and fully executable, yet still fails unit tests due to subtle semantic mistakes—an incorrect condition, a misplaced update, or a misinterpreted invariant.

Unit tests provide a reliable correctness signal, but it is temporally coarse: it applies to the entire program rather than to the specific decisions that caused failure. Group-based policy gradients distribute this signal uniformly, so near-correct solutions receive gradients too diffuse to correct localized reasoning errors.

This paper addresses the problem of semantic credit assignment in critic-free RLVR for code generation and targets the near-correct regime where further gains depend on precise attribution rather than coarse feedback.

The following are our key contributions:

1. We show that credit assignment—not reward sparsity—is the main bottleneck in critic-free RL for code generation once models already produce syntactically valid and structurally reasonable programs.
2. We introduce EGCA, which routes each sample through deterministic failure-mode gates (syntax/constraint/logic) and, for near-correct candidates, localizes the earliest execution divergence against a reference trace to concentrate the GRPO advantage on the causal token span.

*Accepted to the ICLR 2026 Workshop on Scaling Post-Training for LLMs (SPOT).
†Lead author and corresponding author; primary contribution to this work.

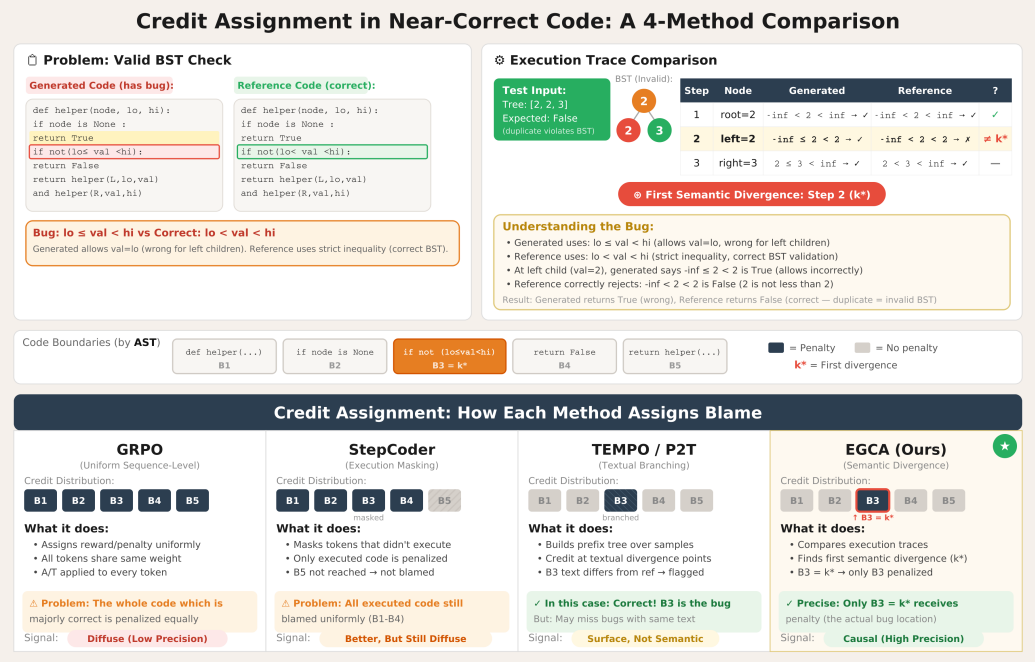


Figure 1: **Motivation: credit smear vs. localized updates.** Sequence-level RLVR objectives apply unit-test outcomes uniformly across long programs, penalizing large spans of correct code for a localized semantic bug. EGCA concentrates gradient mass on the earliest semantically divergent span (identified via execution) while masking downstream tokens, improving credit assignment in the near-correct regime.

3. We show the approach is agnostic to the debugger's code-generation ability: the trained student surpasses a 1.5B-parameter debugger by +8.2 points, ruling out knowledge distillation as the mechanism.

## 2 Related Work

We situate EGCA against five lines of work that densify credit in RLVR for code. The shared limitation is that none reliably localizes failure to semantically causal regions of fully executing programs.

**Richer outcome signals.** Methods such as RLTF (Liu et al., 2023) enrich outcome feedback from execution beyond binary pass/fail, but the signal remains outcome-anchored and does not identify where within a program failure originated.

**Execution-aware masking.** StepCoder (Dou et al., 2024) masks unexecuted tokens during updates, reducing spurious blame. However, when programs execute to completion, all tokens are executed, and masking provides no disambiguation among them.

**Group-structure credit.** Prefix-tree methods such as TEMPO/P2T (Tran et al., 2025) derive token-level updates from textual branching points within sample groups. This provides cleaner credit than sequence-level baselines, but textual divergence does not necessarily coincide with the causal location of semantic failure in code.

**Learned evaluators.** Process reward models (Li et al., 2025; Lightman et al., 2023) train step-level scorers for RL shaping. These meaningfully densify supervision but inherit challenges around label noise and distribution shift from the learned evaluator.

**Execution semantics and actor–critic.** CodeRL+ (Jiang et al., 2025) adds auxiliary execution-alignment objectives, and actor–critic methods (Le et al., 2022; Shojaee et al., 2023) provide dense shaping via learned value functions. Both depart from the critic-free regime.

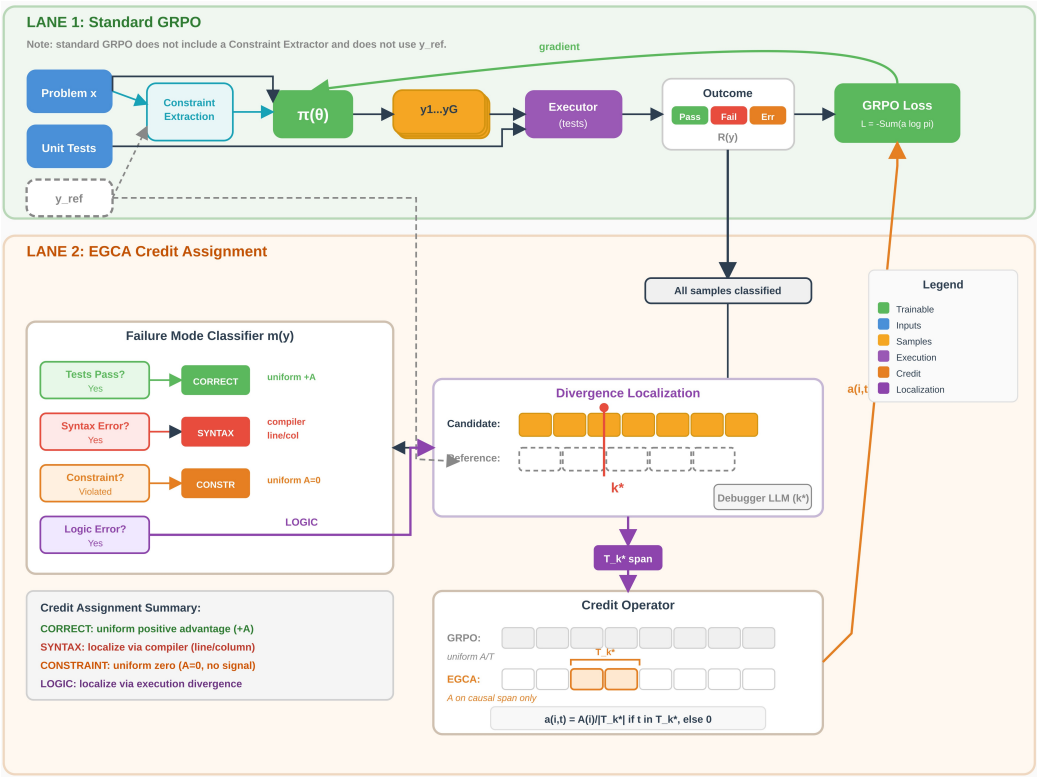


Figure 2: **EGCA pipeline.** We extract constraints from a canonical reference, sample and execute a group of programs, route each into SYNTAX/CONSTRAINT/LOGIC/CORRECT via deterministic gates, and apply token-level GRPO by localizing advantage (compiler span for SYNTAX, earliest reference-trace divergence for LOGIC) while masking downstream tokens.

**Concurrent directions.** Concurrent work explores complementary directions: RLEF (Gehring et al., 2024) grounds code LLMs in execution feedback via RL; multi-turn rewards (Jain et al., 2025) study single-step reward shaping across turns; and MURPHY (Ekbote et al., 2025) applies multi-turn GRPO for self-correction. EGCA differs in localizing credit to the earliest divergence within a single generation.

**Summary and gap.** Existing methods densify feedback or reduce noise, but do not reliably attribute failure to *semantically causal* regions of otherwise well-formed, constraint-following programs. EGCA targets this gap: for near-miss programs that execute but fail due to a localized reasoning error, it identifies the causal span via reference-trace comparison and concentrates the GRPO advantage there.

# 3 METHOD

## 3.1 PROBLEM SETTING

Let $x$ denote a programming problem and let $\pi_\theta$ be a code generation policy that samples programs $y \sim \pi_\theta(\cdot \mid x)$. Programs are evaluated using unit tests, producing a base verifiable score $\hat{R}(y) \in [0, 1]$ (fraction of tests passed). We incorporate algorithmic constraints by defining the reward used for optimization as

$$R(y) = \hat{R}(y)\, \mathbb{I}_{\mathcal{C}}(y), \tag{1}$$

so that a program receives full reward only when it both passes all tests and satisfies the extracted constraints.

We adopt *Group Relative Policy Optimization (GRPO)* (Shao et al., 2024). For each problem $x$, we sample a group of $G$ programs $\{y_i\}_{i=1}^{G}$ and compute group-relative advantages

$$A_i = R(y_i) - \frac{1}{G}\sum_{j=1}^{G} R(y_j). \tag{2}$$

### 3.2 Canonical Solutions as a Structural Pivot

For each problem $x$, we assume access to a *canonical reference solution* $y^{\mathrm{ref}}$, curated once offline. The reference is *not* used as a target for imitation. Instead, it serves as a non-parametric pivot for (i) extracting algorithmic constraints, (ii) defining a reference execution behavior, and (iii) anchoring semantic comparisons. This requirement limits applicability to settings where at least one correct solution exists (e.g., competitive programming and function synthesis with tests); extending EGCA to open-ended generation without references remains future work.

### 3.3 Constraint-Guided Sampling

**Constraint extraction.** A debugging-oriented teacher model extracts from $(x, y^{\mathrm{ref}})$ a set of algorithmic constraints

$$\mathcal{C} = \{c_1, \ldots, c_M\}, \tag{3}$$

where each constraint specifies a permitted or forbidden structural property (e.g., control-flow form, permitted data structures, or complexity targets). Constraints are non-executable, non-token-level, and solution-agnostic.

**Sampling with constraints.** Constraints are injected as soft guidance via a prompt suffix:

$$y_i \sim \pi_\theta(\cdot \mid x \parallel \mathcal{C}). \tag{4}$$

This biases sampling toward programs that are structurally comparable to $y^{\mathrm{ref}}$, increasing the density of samples for which semantic divergence is meaningful.

**Constraint satisfaction.** We define a deterministic indicator $\mathbb{I}_{\mathcal{C}}(y) = 1$ if $y$ satisfies all constraints in $\mathcal{C}$, and 0 otherwise.

In addition, semantic divergence is only well-defined when a candidate is *comparable* to the reference at a coarse structural level. We therefore compute a comparability indicator $\mathbb{I}_{\mathrm{cmp}}(y) \in \{0, 1\}$ via normalized AST/CFG validation (Section 3.4). Candidates that fail this gate are treated as constraint violations in $m(\cdot)$.

### 3.4 Structural Validation (Comparability Gate)

We parse $y$ and $y^{\mathrm{ref}}$ into ASTs, construct normalized CFGs, compute structural similarity scores, and declare a candidate comparable if scores exceed fixed thresholds, yielding $\mathbb{I}_{\mathrm{cmp}}(y) \in \{0, 1\}$.

### 3.5 Failure Modes and Credit Operator

Let $m(y) \in \{\textsc{correct}, \textsc{constraint}, \textsc{syntax}, \textsc{logic}\}$ denote a deterministic failure-mode classifier defined by the following priority order:

$$m(y) = \begin{cases} \textsc{syntax} & y \text{ raises a compile/runtime error,} \\ \textsc{constraint} & \mathbb{I}_{\mathcal{C}}(y) = 0 \ \vee \ \mathbb{I}_{\mathrm{cmp}}(y) = 0, \\ \textsc{correct} & \hat{R}(y) = 1 \ \wedge \ \mathbb{I}_{\mathcal{C}}(y) = 1, \\ \textsc{logic} & \text{otherwise.} \end{cases} \tag{5}$$

(Note that the priority ordering ensures $\mathbb{I}_{\mathrm{cmp}}(y) = 1$ for any sample reaching the CORRECT or LOGIC cases.)

We handle syntactic failures before CFG-based validation, since syntax errors can prevent reliable AST/CFG construction; compiler/interpreter diagnostics then provide a precise localization signal for token-level credit assignment.

For a sampled completion $y_i$ of length $T_i$, EGCA defines token-level advantages $a_{i,t}$ as a piecewise function of $m(y_i)$ and a small set of diagnostic spans.

**Syntax span.** If $m(y_i) = \text{SYNTAX}$, the compiler/interpreter returns a location; we map it to a token span $\mathcal{T}_{\text{err}} \subset \{1, \dots, T_i\}$ and localize credit to that span, bypassing semantic divergence analysis.[1]

**Divergence span.** If $m(y_i) = \text{LOGIC}$, we localize the earliest semantic divergence against a reference execution to obtain a boundary index $k^*$ and an associated token span $\mathcal{T}_{k^*} \subset \{1, \dots, T_i\}$ (defined below).

**Token-level advantage operator.** We then set

$$a_{i,t} = \begin{cases} \dfrac{A_i}{T_i} & m(y_i) = \text{CORRECT}, \\ \dfrac{A_i}{T_i} & m(y_i) = \text{CONSTRAINT}, \\ \dfrac{A_i}{|\mathcal{T}_{\text{err}}|}\,\mathbf{1}[t \in \mathcal{T}_{\text{err}}] & m(y_i) = \text{SYNTAX}, \\ \dfrac{A_i}{|\mathcal{T}_{k^*}|}\,\mathbf{1}[t \in \mathcal{T}_{k^*}] & m(y_i) = \text{LOGIC}. \end{cases} \tag{6}$$

The operator is normalized so that $\sum_{t=1}^{T_i} a_{i,t} = A_i$ for every mode, while only localizing credit when blame can be attributed to a specific span.

### 3.6 Execution-Grounded Divergence Localization

We apply divergence localization only when $m(y_i) = \text{LOGIC}$, i.e., for candidates that are both constraint-satisfying and comparable to the reference.

**Semantic divergence.** Let $d$ be the first failing unit test input. We define execution boundaries

$$B(y_i) = (b_1, \dots, b_K), \tag{7}$$

each mapping to a token span $\mathcal{T}_k \subset \{1, \dots, T_i\}$. Executing both programs yields state traces

$$\tau(y_i, d) = (S_1, \dots, S_K), \qquad \tau(y^{\text{ref}}, d) = (S_1^{\text{ref}}, \dots, S_K^{\text{ref}}). \tag{8}$$

We define the earliest semantic divergence boundary

$$k^* = \min\{k : S_k \neq S_k^{\text{ref}}\}. \tag{9}$$

Because static alignment alone cannot reliably map trace mismatches to fault regions, a debugging-oriented LLM localizes $k^*$ over the aligned structure and paired traces; it is not used as a correctness oracle.

### 3.7 Final GRPO Objective

The overall GRPO objective with token-level advantages is

$$\mathcal{L}(\theta) = -\sum_{i=1}^{G} \sum_{t=1}^{T_i} a_{i,t} \log \pi_\theta\big(y_{i,t} \mid x, y_{i,<t}\big). \tag{10}$$

No teacher gradients, auxiliary losses, or imitation terms are introduced.

[1] Our implementation uses Python AST for CFG construction; therefore programs that fail to parse are routed to SYNTAX mode. Extending structural checks with tolerant parsing is possible but not required for the method.

Table 1: Main performance (pass@1, %) on HumanEval and MBPP.

| Method | HumanEval | MBPP |
|---|---|---|
| DeepSeek-Coder (6.7B) base | 78.6 | 65.4 |
| SFT | 71.9 | 60.3 |
| Vanilla PPO | 78.0 | 65.6 |
| GRPO | 79.0 | 67.4 |
| RLTF | 77.9 | 64.5 |
| StepCoder-mask | 78.7 | 67.0 |
| CodeRL+ | 81.6 | 67.4 |
| **EGCA (Ours)** | **82.1** | **68.9** |

# 4 EXPERIMENTS

## 4.1 BENCHMARKS AND DATA

We train our models on APPS+, a curated version of the APPS dataset designed for RL-based code generation. Each problem provides a natural-language specification, a set of executable unit tests, and a canonical reference solution. We focus on Python programs, which dominate APPS+ and allow reliable execution and tracing.

### 4.1.1 CANONICAL REFERENCE SOLUTIONS

Each APPS+ problem includes canonical reference solutions used as described in Section 3.2; they are never used as training targets.

### 4.1.2 EVALUATION BENCHMARKS

We evaluate on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), reporting pass@1. No benchmark problems appear in training.

## 4.2 TRAINING SETUP AND IMPLEMENTATION DETAILS

We initialize from DeepSeek-Coder-Instruct-6.7B (Guo et al., 2024) and use Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as the default debugger/localizer. We train with GRPO ($G$=16 rollouts, AdamW, lr $= 5 \times 10^{-7}$, $\beta = 0.05$, $\varepsilon = 0.2$) on APPS+ following StepCoder's RL protocol (Dou et al., 2024). Full hyperparameters are in Appendix H.

## 4.3 BASELINES

We compare against representative reinforcement learning–based post-training baselines for code generation, differing primarily in how execution feedback and credit assignment are handled. All baselines use the same backbone model (DeepSeek-Coder-Instruct-6.7B), APPS+ training split, decoding parameters, execution environment, and training budget. The baselines include GRPO with uniform token-level credit assignment, StepCoder, which masks unexecuted code regions during policy updates, RLTF, which leverages multi-granularity unit test feedback without explicit localization of semantic errors, and CodeRL+, which augments reinforcement learning with an auxiliary execution-semantics alignment objective. We follow each method's original training objective and do not introduce constraint guidance, structural validation, or execution-grounded localization into any baseline.

## 4.4 MAIN RESULTS

Table 1 reports pass@1 on HumanEval and MBPP after training on APPS+ with DeepSeek-Coder-Instruct-6.7B. All methods use identical rollout budgets ($G$=16), decoding parameters, and execution environments.

Table 2: Code-generation capability of debugger LLMs (pass@1, %).

| Model | HumanEval | MBPP |
|---|---|---|
| Qwen2.5-Coder-1.5B-Instruct | 70.7 | 69.2 |
| Qwen2.5-Coder-7B-Instruct | 84.8 | 79.2 |

Table 3: EGCA performance when varying the debugger/localizer LLM (pass@1, %).

| Debugger model | HumanEval | MBPP |
|---|---|---|
| Qwen2.5-Coder-1.5B-Instruct | 78.9 | 66.1 |
| Qwen2.5-Coder-7B-Instruct | 82.1 | 68.9 |
| Claude 4.5 Sonnet | 83.7 | 67.8 |

EGCA achieves $82.1\%$ on HumanEval and $68.9\%$ on MBPP. Against vanilla GRPO ($79.0/67.4$), this corresponds to gains of $+3.1$ and $+1.5$ absolute points. Against StepCoder, which masks unexecuted tokens during updates, EGCA improves by $+3.4$ and $+1.9$. This gap suggests that executed-token masking alone is insufficient in the near-correct regime: when programs run to completion, the key question is not which tokens were irrelevant, but which executed decision caused the first semantic deviation.

EGCA also outperforms CodeRL+ ($81.6/67.4$) by $+0.5$ on HumanEval and $+1.5$ on MBPP, despite introducing no auxiliary loss or critic. The improvement derives entirely from reweighting where gradient mass is applied within the standard GRPO objective.

### 4.5 Ruling Out Teacher Leakage

EGCA uses canonical reference solutions and a debugging-oriented LLM to localize divergences, raising a natural concern: do gains stem from distilling teacher knowledge rather than improved credit assignment? We include three controls that suggest the benefit primarily comes from localization.

**Distillation baselines underperform.**

**Student exceeds the debugger's generation capability.** When Qwen2.5-Coder-1.5B-Instruct serves as the debugger, EGCA achieves $78.9\%$ on HumanEval. The same model, used directly as a code generator under the same evaluation protocol, achieves $70.7\%$. The student thus surpasses the debugger by $+8.2$ points, indicating that EGCA extracts a localization signal rather than code-writing competence.

Supervised fine-tuning on teacher-generated code (Teacher SFT) achieves $60.9/58.1$, worse than the base model. Using the teacher as a dense reward signal (Teacher-critique RL) achieves $76.3/66.1$, still $5.8$ points below EGCA on HumanEval. Neither imitation nor teacher-as-judge matches the gains from execution-grounded localization.

**Scaling the debugger shows diminishing returns.** Moving from a $1.5$B to 7B debugger improves EGCA by $+3.2$ on HumanEval, while a substantially stronger debugger (Sonnet 4.5) adds only $+1.6$ more. If EGCA were primarily distilling solver competence, gains would be expected to track debugger capability; instead, improvements saturate as localization quality stabilizes.

## 5 Limitations and Future Work

EGCA's localization pipeline assumes access to a debugger LLM that can parse execution traces and reason about semantic divergence between two programs. This model need not generate correct code itself—our experiments confirm that even a 1.5B-parameter debugger whose own pass@1 falls well below the student's still yields meaningful gains—but it must understand code well enough to produce coherent constraints and to compare candidate versus reference traces. When the debugger

Table 4: Distillation controls (pass@1, %).

| Method | HumanEval | MBPP |
|---|---|---|
| Teacher SFT | 60.9 | 58.1 |
| Teacher-critique RL | 76.3 | 66.1 |
| **EGCA (Ours)** | **82.1** | **68.9** |

lacks this minimum competence, constraint quality degrades, structural comparability judgments become noisy, and the localization signal loses precision.

A related consequence is that EGCA's value is stage-dependent. The method targets the *near-correct regime*: programs that compile, execute, satisfy structural constraints, and fail only due to localized logical errors. In our training distribution roughly 35% of samples fall into this LOGIC mode; the remaining 65% are handled by uniform or compiler-grounded updates that do not require divergence localization. As the base policy improves, the fraction of near-correct samples grows and EGCA's advantage compounds. Conversely, for weak initializations where most failures are syntactic or structural, the localization machinery triggers rarely and the method offers little beyond standard GRPO. EGCA is therefore most impactful as a later-stage refinement technique applied after the policy already produces broadly reasonable code.

Finally, all experiments here use a 6.7B policy. Scaling EGCA to larger base models, longer programs, and multi-file generation remains open. Larger policies will produce more near-correct samples by default, amplifying the regime where EGCA operates, but the debugger LLM and trace infrastructure must scale accordingly. When multiple structurally distinct solutions are valid, EGCA's constraint extraction and comparability gate may exclude correct but divergent approaches. Correct solutions using structurally divergent approaches may fail the comparability gate; in practice this appears rare in our setting, and such cases could be handled by bypassing constraint checks for programs that pass all tests.

## 6 Conclusion

Credit assignment—not reward sparsity—is the binding constraint on critic-free RL for code generation once models reliably produce executable, structurally sound programs. EGCA addresses this by converting a coarse unit-test outcome into a token-level policy-gradient operator grounded in runtime semantics: it routes each sample through deterministic failure-mode gates, localizes the earliest execution divergence against a reference trace for near-correct candidates, and concentrates the GRPO advantage on the causal span while masking everything downstream. The method introduces no critic, no auxiliary loss, and no learned verifier; it changes only which tokens receive gradient mass within the standard objective.

Empirically, EGCA achieves 82.1% on HumanEval and 68.9% on MBPP, improving over vanilla GRPO by +3.1 and +1.5 points and over the strongest baseline (CodeRL+) by +0.5 and +1.5, with 18% wall-clock overhead. Ablations (Appendix F) confirm that gains require localizing to the *causally correct* span: random or late-divergence targeting collapses toward the uniform baseline, and softening the mask monotonically erodes performance. Teacher-leakage controls show the student surpassing its own debugger by over 8 points, ruling out knowledge distillation as the primary mechanism.

The core insight is simple: for near-correct code, knowing *where* a program first goes wrong is more valuable than knowing *that* it goes wrong. EGCA operationalizes this insight within GRPO, converting execution traces into precise credit without leaving the critic-free regime.

## REFERENCES


Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. Stepcoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 4571–4585, 2024.

Pratik Ekbote et al. Murphy: Multi-turn grpo for self-correction in code generation. *arXiv preprint*, 2025.

Jonas Gehring et al. Rlef: Reinforcement learning from execution feedback for code generation. *arXiv preprint*, 2024.

Daya Guo, Qihao Zhu, Dejian Yang, Runxin Xu, Yuxiang Wu, Fei Huang, and Furu Wei. Deepseek-coder: When the large language model meets programming. *arXiv preprint arXiv:2401.14196*, 2024.

Bo Hui, Jian Yang, Zeyu Cui, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Rishabh Jain et al. Multi-turn rewards for code generation: Single-step reward shaping across turns. *arXiv preprint*, 2025.

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.

Hung Le, Aditi Thakur, Aohan Zeng, Anuj Sharma, Zhen Yang, Tao Chen, Rami Alouni, Yuxuan Zhou, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning, 2022. NeurIPS 2022; OpenReview: https://openreview.net/forum?id=WaGvb7OzySA.

Qing Li, Xinyi Dai, Xiaohan Li, Wei Zhang, Yifan Wang, Ruixiang Tang, and Yong Yu. Codeprm: Execution feedback-enhanced process reward model for code generation. In *Findings of ACL*, pp. 8169–8182, 2025.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Xinrun Liu, Hongyi Zhang, Yixin Wan, Bo Zhang, Kaixuan Song, Mingjie Yang, Haonan Wang, Luyi Yang, Weiwen Zhang, Zihan Lin, Xianfeng Du, Zhen Zhang, Jian Xiao, Chao Zhang, Xu Chen, and Jie Zhou. Reinforcement learning from unit test feedback. Transactions on Machine Learning Research (TMLR), 2023. OpenReview: https://openreview.net/forum?id=R40uZX1R29.

Zihan Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junjie Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Yuxiang Wu, Daya Guo, Jie Zhou, and Furu Wei. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Parshin Shojaee, Miltiadis Allamanis, Marc Brockschmidt, and Charles Sutton. Ppocoder: Learning code generation with policy gradient methods. *arXiv preprint arXiv:2301.13816*, 2023.

Nhat Tran, Daphne Ippolito, Jian Mao, Shizhe Diao, Yuxiang Lin, Hao Li, Ruoyu Zhang, Haoran Xu, Jiatao Gu, Andrew Liu, et al. Token-level credit assignment without critic: Prefix tree group relative policy optimization. *arXiv preprint arXiv:2509.18314*, 2025.

# A Additional Details for Reproducibility

## A.1 Appendix A. Reproducibility Summary

**Compute and schedule.** We fine-tune using 8× NVIDIA A100 80GB GPUs with global batch size 64, training for 3 epochs with 0.3-epoch warmup and a linear learning-rate decay to zero.

## A.2 Appendix B. RL Optimization and Generation Settings

**Optimizer hyperparameters.** During GRPO training, we use a policy learning rate of $5 \times 10^{-7}$ with AdamW. Following the critic-free GRPO formulation (Shao et al., 2024), no separate value network is trained.

**Note on learning rate.** Our policy learning rate is more conservative than DeepSeekMath and CodeRL+ (both report $10^{-6}$). We found this smaller learning rate necessary for stable training when combined with token-level credit localization.

**Rollouts per prompt.** For each training example, we sample $G = 16$ rollouts using nucleus sampling with temperature $= 0.8$, top-$p = 0.9$, and maximum output length $= 8192$ tokens.

**KL regularization and clipping.** We apply a token-level KL penalty with coefficient $\beta = 0.05$ and clip the policy ratio with $\varepsilon = 0.2$.

**Evaluation-time decoding.** For inference, we use temperature $= 0.2$ and top-$p = 0.95$.

## A.3 Appendix C. Execution Environment and Reward Collection

**Runtime.** Reward collection and evaluations are conducted in a deterministic Python 3.10 sandbox with standard library execution.

## A.4 Appendix D. EGCA Components (Implementation Details)

This appendix records implementation-specific details for EGCA.

**D.1 Execution boundary construction.** We segment program execution into an ordered sequence of boundary blocks $B = \{B_1, \ldots, B_K\}$, where each $B_k$ corresponds to a contiguous token span and an execution snapshot captured immediately after that span executes.

**D.2 Failure-mode routing.** Each sampled program is assigned to one of four mutually exclusive modes: (i) SYNTAX—parse/compile failure, (ii) CONSTRAINT—violates extracted algorithmic constraints or fails structural comparability, (iii) LOGIC—executes but fails tests, (iv) CORRECT—passes all tests. Divergence localization is triggered only for LOGIC mode.

**D.3 Divergence localization.** For near-correct samples (LOGIC mode), we identify the earliest boundary index $k^*$ at which the candidate execution diverges from the reference execution under aligned inputs. The localizer LLM (Qwen2.5-Coder-7B-Instruct by default) operates over aligned traces; it is not used as a correctness oracle.

**D.4 Token-weighted credit operator.** Let $A_i$ denote the scalar advantage for sample $i$ under the base GRPO objective. EGCA converts $A_i$ into token-wise advantages $a_{i,t} = w_{i,t} A_i$, where $w_{i,t}$ is concentrated within the localized boundary block and zero elsewhere for LOGIC/SYNTAX modes. Setting all $w_{i,t} = 1/T_i$ recovers the uniform baseline.

## A.5 Appendix E. Overhead Accounting

To make the runtime footprint auditable, we report:

Table 5: Overhead accounting (averages across APPS+ training).

| Metric | Value |
|---|---|
| Fraction in SYNTAX mode | $\sim 8\%$ |
| Fraction in CONSTRAINT mode | $\sim 12\%$ |
| Fraction in LOGIC (near-correct) mode | $\sim 35\%$ |
| Fraction in CORRECT mode | $\sim 45\%$ |
| Average boundaries $K$ per sample | 12.4 |
| Wall-clock overhead vs. vanilla GRPO | $+18\%$ |
| Localization trigger rate | $35\%$ of samples |

### A.6 Appendix F. Additional Ablations (Credit Assignment Validity)

Table 6: Ablations on localization target under identical training budgets.

| Target | HumanEval | MBPP |
|---|---|---|
| Earliest divergence (EGCA default) | 82.1 | 68.9 |
| Last divergence | 80.2 | 67.5 |
| Random executed boundary | 79.6 | 67.1 |
| No localization (uniform) | 79.0 | 67.4 |

**F.1 Localization target.**

### A.7 Appendix G. Qualitative Case Study: Localized Semantic Errors

We provide a qualitative analysis showing how execution-grounded localization yields a concentrated learning signal for near-correct programs.

**Problem: `longest_increasing_subsequence` (MBPP-style).**

```
Write a function that returns the length of the longest strictly increasing
subsequence in a given list of integers.

Example:
>>> longest_increasing_subsequence([10, 9, 2, 5, 3, 7, 101, 18])
4

def longest_increasing_subsequence(nums):
    if not nums:
        return 0

    n = len(nums)
    dp = [1] * n

    for i in range(1, n):
        for j in range(i):
            if nums[i] >= nums[j]:  # BUG: should be strictly greater (>)
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

**GRPO output (fails unit tests).**

```
def longest_increasing_subsequence(nums):
    if not nums:
        return 0
```

```
    n = len(nums)
    dp = [1] * n

    for i in range(1, n):
        for j in range(i):
            if nums[i] >= nums[j]:  # BUG: should be strictly greater (>)
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

Unit test result: FAILED on input `[1, 3, 3, 5]` (expected 3; got 4).

**EGCA output (passes unit tests).**

```
def longest_increasing_subsequence(nums):
    if not nums:
        return 0

    n = len(nums)
    dp = [1] * n

    for i in range(1, n):
        for j in range(i):
            if nums[i] > nums[j]:  # CORRECT
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

**Execution trace divergence (first semantic mismatch).** For input `[1, 3, 3, 5]`, the first semantic divergence occurs at ($i$=2, $j$=1) when comparing equal elements (3 vs. 3).

Table 7: Trace comparison for `[1,3,3,5]`. The first divergence occurs when the candidate uses $\geq$ while the reference uses $>$.

| Step | $i$ | $j$ | `nums[i]` | `nums[j]` | Divergence? |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 3 | 1 | No |
| 2 | 2 | 0 | 3 | 1 | No |
| 3 | 2 | 1 | 3 | 3 | Yes |

**Credit assignment contrast.** Under uniform sequence-level GRPO, the negative advantage is spread across all tokens in the program, so the comparison operator receives the same penalty weight as many unrelated tokens. EGCA maps the divergence to the span containing the comparison operator and concentrates the full advantage on that span while masking downstream tokens.

### A.8 Appendix H. Full Hyperparameter Table

Table 8: Hyperparameters used in our experiments.

| Hyperparameter | Value |
|---|---|
| SFT learning rate | $2 \times 10^{-5}$ |
| SFT epochs | 3 |
| SFT warmup | 0.3 epochs |
| SFT LR schedule | Linear decay to zero |
| GRPO policy learning rate | $5 \times 10^{-7}$ |
| GRPO optimizer | AdamW |
| Rollouts per prompt ($G$) | 16 |
| Train sampling temperature | 0.8 |
| Train sampling top-$p$ | 0.9 |
| Max generation tokens | 8192 |
| KL coefficient ($\beta$) | 0.05 |
| Clip epsilon ($\varepsilon$) | 0.2 |
| Eval decoding temperature | 0.2 |
| Eval decoding top-$p$ | 0.95 |
| Hardware | 8$\times$ A100 80GB |
| Global batch size | 64 |