RLPgR: Reinforcement Learning from Dependency-Aware Progress Rewards

Anonymous ACL submission

Abstract

Group-relative policy optimization (GRPO) provides a stable recipe for outcome-based RL on language model reasoning, but its credit assignment is coarse: it broadcasts a single trajectory advantage to every token, regardless of how individual steps contribute to the outcome. We call this the uniform-broadcast failure mode and address it with RLPgR, a dependency-aware rescaling of the existing GRPO advantage. RLPgR parses each rollout into a Progress Dependency Graph (PDG), where each node groups a contiguous span of steps into a milestone-level unit with a locally validatable artifact, and directed edges connect a node to the artifacts it depends on. Using the PDG, RLPgR rescales the group-relative advantage by a small, bounded, sign-preserving factor at the node level. This preserves the outcome-governed rollout direction while each step's update better reflects its dependency role, avoiding the reward-hacking drift of densely shaped process rewards. To reduce PDG construction cost, we train an open Qwen3-8B progress annotator amortizing a one-time proprietary bootstrap. Across model families and sizes on five math reasoning benchmarks, RLPgR matches or surpasses strong outcome-RL and process-RL baselines. Three controlled studies (scaling-strength sensitivity, segment-level baselines, and PDG label perturbation) support the view that the gain comes from structured credit assignment rather than incidental tuning.

1 Introduction

Reinforcement learning with verifiable rewards (RLVR) is now a workhorse for post-training LLMs on verifiable domains such as mathematics and programming (Shao et al., 2024; Guo et al., 2025a). Group-relative policy optimization (GRPO) is a stable critic-free instantiation that optimizes binary correctness signals via a single group-relative trajectory advantage (Shao et al., 2024; He et al., 2025). The price of this stability is a coarse credit-assignment rule. GRPO derives the trajectory-level advantage from final correctness and broadcasts that scalar uniformly to every token in the rollout, regardless of how individual steps contribute to the final outcome (Uesato et al., 2022; Lightman et al., 2023). We call this the uniform-broadcast failure mode of GRPO. For long-chain reasoning, where intermediate conclusions depend sparsely on earlier artifacts, this uniform broadcast applies the same update to root-cause mistakes, locally valid steps that inherit a wrong premise, and explicit recovery steps that backtrack from earlier errors.

A natural reaction is to densify the reward at the step level, either by training a process reward model (PRM) that scores intermediate states (Lightman et al., 2023; Zhang et al., 2024a; Cheng et al., 2025) or by estimating step-level value via tree search and Monte Carlo rollouts (Kazemnejad et al., 2025; Guo et al., 2025b). Both replace the binary outcome signal with a denser scalar at intermediate states, but each pays a different cost. Dense step-level rewards redefine what counts as positive at the step level and are prone to reward hacking via shortcut optimization, length inflation, or local-score harvesting (Cheng et al., 2025; Weng, 2024; Cui et al., 2025), while tree-search value estimation incurs high variance and computational cost (Zhang et al., 2025; Chen et al., 2025).

We argue that the core issue is not the absence of dense rewards but how the existing outcome-derived advantage is distributed within a rollout. Multi-step reasoning is structured so that later steps causally depend on a sparse set of earlier premises rather than the entire prefix (Cao et al., 2023; Mukherjee et al., 2025; Zhang et al., 2024b). Once an early misconception enters the chain, uniform broadcast conflates source errors with propagated errors and weakens incentives for recovery (Yee et al., 2024; Dhuliawala et al., 2024). This motivates redistributing the existing outcome-derived

advantage along a dependency graph, rather than introducing an independent dense reward that competes with outcome supervision.

Guided by this perspective, we propose RLPgR, a dependency-aware correction to GRPO-style advantage broadcast in which the existing group-relative advantage is rescaled by a node-level factor that depends on dependency-driven error roles. We represent each rollout as a Progress Dependency Graph (PDG), where each progress node groups a contiguous span of steps into a single subgoal that ends with a locally validatable artifact, and directed edges encode artifact dependencies. Each node is annotated with one of four statuses, Correct Progress (CP), Internal Error (IE), Inherited Error (HE), or Compound Error (CE), which lets us separate root-cause mistakes from errors that arise purely through dependency propagation. RLPgR multiplies GRPO's outcome-derived advantage by a small, bounded, sign-preserving node-level factor, so each step's update is calibrated to its contribution to the final outcome while final correctness still governs the rollout-level direction, avoiding the reward-hacking drift typical of densely shaped process rewards.

We evaluate on a suite of public math reasoning benchmarks, and instantiate RLPgR on Qwen2.5-Math (1.5B and 7B) and DeepSeek-R1-Distill-Qwen-1.5B (Yang et al., 2024; Guo et al., 2025a; Team, 2025). We report headline numbers in both avg@k and pass@k (Yue et al., 2025a), and directly probe the mechanism with three controls. A scaling-strength study shows that a single coefficient α monotonically controls the magnitude of dependency-aware correction, with larger gains on the 7B backbone. A comparison with recent segment-level credit-assignment methods (Guo et al., 2025b; Kazemnejad et al., 2025) further shows that explicit dependency modeling improves over coarser segmentation. Controlled perturbations of PDG node statuses confirm that the gain tracks PDG label fidelity rather than incidental hyperparameter choices. Finally, under a unified LLM-as-judge protocol, our trained open progress annotator surpasses the commercial GPT-5-mini annotator used for bootstrapping, achieving 58% and 75% pairwise win rates under GPT-5.2 and DeepSeek-V3.2 judges, respectively.

Our contributions are threefold. First, we name and characterize the uniform-broadcast failure mode in GRPO-style training, where the rollout-level advantage is broadcast uniformly across tokens regardless of their causal roles. Second, we propose RLPgR, a dependency-aware correction to GRPO that uses Progress Dependency Graphs (PDGs) to rescale the existing group-relative advantage by a small, bounded, sign-preserving factor at the node level. Third, we train an open Qwen3-8B progress annotator that amortizes a one-time proprietary bootstrap, and provide three mechanism studies attributing the gain to structured credit assignment rather than incidental tuning.

2 Related Work

RL for LLM Reasoning. Reinforcement learning (RL) post-training is widely used to improve the reasoning ability of large language models. Early pipelines optimize a policy with outcome reward models (ORMs) that score the final response (Ouyang et al., 2022), and for verifiable domains such as mathematics RL with verifiable rewards (RLVR) replaces ORMs with automatic correctness checks, which makes it practical to scale optimization to long-chain reasoning. Long-horizon reasoning, however, surfaces training instabilities including chain-of-thought collapse, difficulties in balancing exploration and exploitation, ratio-control sensitivity, and rapid entropy decay (Yuan et al., 2025b; Yue et al., 2025b; Yu et al., 2025; Wang et al., 2025). A growing line of work stabilizes RLVR through group-based rollout optimization (Shao et al., 2024) and critic-free updates (Hu et al., 2025), complemented by variance and control refinements for long chain-of-thought optimization (Yuan et al., 2025b; Yue et al., 2025b) and open-source RL implementations that target practical scalability (Yu et al., 2025). Value-free alternatives further reduce reliance on explicit value learning at the cost of higher compute (Kazemnejad et al., 2025). Overall, these studies indicate that reasoning RL depends jointly on reward quality, stable optimization, and practical scalability. We build on GRPO (Shao et al., 2024) as the most representative critic-free RLVR instance and target the residual issue of how the broadcast advantage is redistributed across tokens.

RL with Process Reward Models. PRMs are motivated by the limitation of outcome-only supervision: a single terminal reward is too coarse to distinguish which intermediate decisions were crucial, unproductive, or erroneous. When used for training, PRMs are framed as a way to improve credit assignment, attributing success or failure to specific steps

rather than uniformly to the trajectory (Richard S Sutton, 1998; Arumugam et al., 2021; Lightman et al., 2023; Cui et al., 2025). In practice, step-level PRMs are noisy and drift under distribution shift, dense step-wise evaluation increases training cost, and imperfect process signals invite shortcut optimization (Setlur et al., 2024; Zhang et al., 2025). Complementary approaches further densify the reward signal through proxy schemes such as RLHF-derived dense rewards or Shapley-style credit allocation, but these signals are typically non-verifiable and harder to align robustly with reasoning correctness (Chan et al., 2024; Cao et al., 2025). Overall, prior work suggests that making process supervision truly effective for RL requires not only finer-grained signals but also more robust credit-assignment mechanisms. RLPgR shares this credit-assignment motivation but rescales the existing group-relative advantage rather than introducing an independent step-level reward, and it operates on a dependency graph that distinguishes root-cause errors, inherited errors, and recovery rather than on a flat sequence of steps. A related line of work models reasoning chains through latent dependency structure rather than as flat sequences of steps, focusing on how later steps depend sparsely on earlier premises (Cao et al., 2023; Mukherjee et al., 2025; Zhang et al., 2024b).

Rubric-Based Rewards. Rubrics are structured evaluation criteria that a judge model checks one by one, and rubric-conditioned judging often yields more accurate and reproducible signals than free-form LLM-as-judge scoring (Gunjal et al., 2025). Recent pipelines generate rubrics at scale by distilling closed-source APIs into open rubric generators and verifiers (Liu et al., 2026). DR Tulu (Shao et al., 2025) extends this idea by co-evolving rubrics with the policy during training, continuously adding newly discovered criteria and pruning ones that become non-discriminative. A complementary line uses rubrics to identify and penalize unreliable reasoning patterns such as "miracle steps" in mathematical solutions (Yuan et al., 2025a), targeting reasoning faithfulness rather than only final-answer correctness. Following Gunjal et al. (2025), we adopt rubric-based judging as the reward signal when training the progress annotator.

3 Method

In this section, we first introduce the definition and construction of the Progress Dependency Graph (PDG), then describe our dependency-aware credit assignment mechanism, and finally present our advantage estimation method.

3.1 Progress Dependency Graph

PDG Definition. Given a raw step sequence $\mathcal{S} = \{s_1, \ldots, s_M\}$ generated by the policy, we employ a progress annotator $\mathcal{M}_{\text{prog}}$ to partition the solution into a sequence of progress nodes, where each node groups a contiguous span of steps into a single subgoal that ends with a locally validatable artifact. For each node v_i, the annotator additionally identifies (i) the step span $\mathcal{I}_i \subseteq \{1, \ldots, M\}$ it covers, (ii) its subgoal g_i, (iii) the intermediate artifact a_i it produces for downstream reasoning, (iv) its dependency set $\mathrm{Dep}(i)$ consisting of predecessor nodes whose artifacts are required premises, (v) its status $s(v_i)$.

We define four types of node status: Correct Progress (CP) indicates the node makes valid progress, Internal Error (IE) indicates the node introduces an internal computational or logical mistake by itself, Inherited Error (HE) indicates the node is locally correct but relies on erroneous upstream premises without performing correction, and Compound Error (CE) indicates both intrinsic and inherited errors occur simultaneously.

High-quality PDGs can be obtained by prompting strong closed-source LLM APIs, but doing so is prohibitively expensive at scale. To reduce the PDG generation cost while maintaining high annotation quality, we train an open-source progress annotator that ultimately produces PDGs that are even competitive with those from the closed-source API, while avoiding the recurring API expenses. We will introduce the training details of the progress annotator in Section 4.1.1.

3.2 Sign-Preserving Advantage Rescaling

GRPO broadcasts a single trajectory-level advantage to every token, so any rollout that mixes locally correct and incorrect reasoning induces sign-mismatched updates where the proxy advantage disagrees with the latent step quality. The sign-preserving rescaling defined below downweights these mismatched positions while keeping the rollout-level update direction governed by the verifiable outcome. We formalize this alignment property as Lemma A.2 in Appendix A, and Section 4.3 validates it through the smooth degradation observed under controlled PDG label perturbation.

Given a rollout-level PDG $G = (\mathcal{V}, \mathcal{E})$, each

progress node $v \in \mathcal{V}$ is annotated with a status $s(v) \in \{\mathrm{CP}, \mathrm{IE}, \mathrm{HE}, \mathrm{CE}\}$ and directed dependencies $\mathcal{E}$. Our goal is to derive a node-level scaling factor $\kappa(v)$, applied to every node $v \in \mathcal{V}$ in the rollout, as a sign-preserving dependency-aware rescaling of the GRPO advantage broadcast (Section 3.3). The rule is governed by a single scalar α, where the root-cause node in a failed rollout receives a larger correction and other affected nodes receive smaller corrections in either direction, so a stronger α uniformly amplifies every dependency-aware correction the PDG induces. We use two signals from PDG: node status to distinguish error types, and dependency structure to locate root-cause and recovery nodes.

For an incorrect rollout, we define the root-cause node v_{root} as the earliest node on the realized mainline whose status indicates an intrinsic error, i.e., the first v with $s(v) \in \{\mathrm{IE}, \mathrm{CE}\}$. For a correct rollout that contains intermediate mistakes, we identify a recovery node v_{rec} as the first CP node whose parent set contains at least one non-CP node:

$$\begin{aligned} v_{\mathrm{rec}}: \quad & s(v) = \mathrm{CP} \\ & \wedge\ \exists u \in \mathrm{Dep}(v) \text{ s.t. } s(u) \neq \mathrm{CP}, \end{aligned} \tag{1}$$

which captures the earliest explicit correction step supported by dependencies.

Let $c \in \{0, 1\}$ denote rollout correctness from the verifiable outcome. We assign every node a single scalar offset $\delta(v) \in \{-\frac{1}{2}, 0, +\frac{1}{2}, +1\}$ and define

$$\kappa(v) \ = \ 1 + \alpha \cdot \delta(v), \tag{2}$$

with $\delta(v)$ determined by the node's PDG role:

- $\delta(v) = +1$ for the root-cause node v_{root} in a failed rollout ($c = 0$),
- $\delta(v) = +\frac{1}{2}$ for a recovery node v_{rec} in a correct rollout ($c = 1$), or for an HE node in a failed rollout,
- $\delta(v) = -\frac{1}{2}$ for an IE/CE node in a correct rollout, or for a CP node in a failed rollout,
- $\delta(v) = 0$ for all other nodes.

HE nodes receive $\delta = +\frac{1}{2}$ rather than $-\frac{1}{2}$ because they propagate an upstream error without correcting it. The penalty pushes the policy to detect contaminated premises rather than chain on them. The constraint $0 \leq \alpha < 2$ is the only requirement for sign-preservation: $\kappa(v) > 0$ on every node. We use $\alpha = 0.2$ as the conservative default and treat α as the only hyperparameter introduced beyond GRPO. A full sensitivity study over $\alpha \in \{0.2, 0.5, 1.0\}$ is reported in Section 4.3, showing that varying α materially affects performance. Because $\kappa(v) > 0$ on every node, the sign of every per-node update agrees with the rollout-level group-relative advantage $A^{(j)}$ derived from the verifiable outcome reward. After redistribution, the rollout-level optimization direction is therefore still governed by final correctness rather than by the rescaling itself. This sign-preserving structure distinguishes RLPgR from densely shaped process rewards. By re-defining what counts as "positive" at the step level, dense shaped rewards can drift away from the outcome signal and invite reward hacking via length inflation, shortcut steps, or local-score harvesting (Cheng et al., 2025; Weng, 2024; Cui et al., 2025). Concretely, the rescaling lets the optimizer (i) penalize erroneous behaviors even within correct rollouts and amplify explicit correction steps, and (ii) for incorrect rollouts, concentrate penalties on the originating mistake and on inherited-error nodes while shielding locally correct progress from spurious negative credit.

3.3 From Node-Level Scaling to the RLPgR Objective

We follow group-relative optimization (Shao et al., 2024) to compute rollout-level advantages from verifiable outcomes, and then apply the node-level scaling derived in Section 3.2 to get token-level discriminative advantages.

For the same input x, we sample a group of K rollouts $\{y^{(j)}\}_{j=1}^{K}$ and obtain verifiable outcome rewards $r_{\mathrm{out}}^{(j)} \in \{0, 1\}$. The group-relative advantage for rollout j is

$$A^{(j)} = \frac{r_{\mathrm{out}}^{(j)} - \mu}{\sigma + \epsilon}, \tag{3}$$

where

$$\begin{aligned} \mu &= \frac{1}{K} \sum_{k=1}^{K} r_{\mathrm{out}}^{(k)}, \\ \sigma &= \sqrt{\frac{1}{K} \sum_{k=1}^{K} \left(r_{\mathrm{out}}^{(k)} - \mu\right)^2}, \end{aligned} \tag{4}$$

ϵ is a small constant.

Each rollout $y^{(j)}$ is segmented into progress nodes $\{v_i^{(j)}\}_{i=1}^{N_j}$ by the PDG. Let $\kappa\left(v_i^{(j)}\right)$ be the scaling factor derived from PDG status and dependencies (Eq. (2)). For any token t that belongs to

node $v_i^{(j)}$, we define the token-level advantage as

$$\tilde{A}_t^{(j)} = \kappa\left(v_i^{(j)}\right) \cdot A^{(j)}. \tag{5}$$

We optimize a clipped surrogate objective without KL:

$$\mathcal{L}(\theta) = \mathbb{E}_{x\sim\mathcal{D}}\left[\frac{1}{\sum_{j=1}^{K} T_j}\sum_{j=1}^{K}\sum_{t=1}^{T_j} \min\left(\rho_t^{(j)}\tilde{A}_t^{(j)},\ \mathrm{clip}(\rho_t^{(j)}, 1-\epsilon, 1+\epsilon)\tilde{A}_t^{(j)}\right)\right], \tag{6}$$

where

$$\rho_t^{(j)} = \frac{\pi_\theta(y_t^{(j)} \mid x, y_{<t}^{(j)})}{\pi_{\theta_{\text{old}}}(y_t^{(j)} \mid x, y_{<t}^{(j)})}. \tag{7}$$

Through this scaling, RLPgR yields more discriminative token-level advantages than GRPO by differentiating root-cause errors, dependency-propagated errors, and recovery behaviors at the progress-node level. Although DAPO (Yu et al., 2025) suggests that the KL penalty can be removed in GRPO, the effect of omitting it under process supervision remains unexplored. We empirically compare the GRPO-style separate KL term, the PPO-style KL-in-reward variant, and the no-KL setting in Appendix B.1 (Table 6).

4 Experiments

4.1 Experimental Setup

In this subsection, we first describe the two-stage training procedure for the progress annotator and present a quality comparison between Progress Nodes generated by annotators at different training stages and those produced by commercial APIs. Based on these results, we determine the source of Progress Nodes for the subsequent reinforcement learning phase. We then detail the experimental settings used during RL training.

4.1.1 Progress Annotator

To construct PDGs during policy training without relying on expensive commercial APIs, we train an open-source progress annotator and improve its quality by post-training. Specifically, we first initialize the progress annotator by supervised fine-tuning (SFT) on a small set of API-generated PDGs, and then further refine it with reinforcement learning (RL). Since RLPgR relies on accurate node states and dependency links in the PDG, we prioritize high-quality cold-start annotations. We therefore benchmark several proprietary LLM annotators for PDG generation and use GPT-5.2-thinking as a judge to score the accuracy of node status labeling on a 1–5 score. As shown in Table 1, GPT-5-mini matches the status-labeling quality of larger proprietary models at a much lower API cost. We therefore use GPT-5-mini to collect cold-start PDG annotations and to provide rubric-based reward signals during post-training.

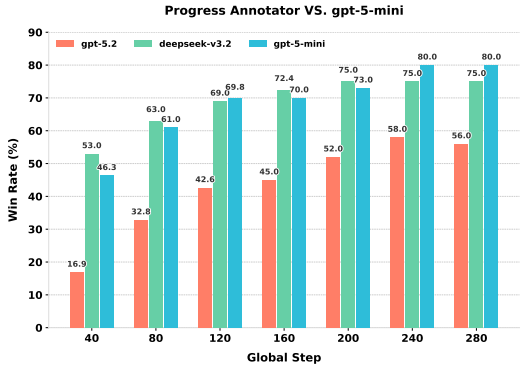


Figure 1: Post-training win rate of the progress annotator against GPT-5-mini generated PDGs at different training steps.

For SFT initialization, we collect 20K trajectories in total. We sample 5K problems with their reference trajectories from PRM800k (Lightman et al., 2023), and additionally collect 15K trajectories using Qwen2.5-Math-1.5B/7B (Yang et al., 2024) and DeepSeek-R1-Distill-Qwen-1.5B (Guo et al., 2025a) to increase trajectory diversity. Based on our PDG definition, we design a prompting template and use GPT-5-mini to generate 20K PDGs as supervision. After filtering by strict format validity, we retain 13K high-quality samples and use them to fine-tune Qwen3-8B (Team, 2025), yielding the initial progress annotator.

Starting from the SFT checkpoint, we further improve the progress annotator via RLVR (Shao et al., 2024) with rubric-based rewards (Gunjal et al., 2025). We design rubrics to evaluate (i) global coherence, i.e., whether the overall PDG structure faithfully represents the reasoning flow, and (ii) node-level accuracy, i.e., whether each node's status and dependencies are correctly identified. GPT-5-mini serves as the judger by comparing the progress annotator's generated PDG against the SFT reference and producing a comparative score. The score is $+1$ if the generated PDG is better, 0 if comparable, and -1 if worse. We additionally impose a -0.5 penalty for format violations to ensure structural validity, and optimize the progress annotator to maximize this reward.

To assess whether the trained progress annotator produces higher-quality PDGs than GPT-5-mini, we perform an independent LLM-as-judge evaluation with two stronger judgers (GPT-5.2 and DeepSeek-V3.2). As shown in Figure 1, our annotator's PDGs achieve a 58% win rate under GPT-5.2 and a 75% win rate under DeepSeek-V3.2 when compared against GPT-5-mini PDGs, suggesting that an open-source annotator can match or surpass the closed-source model used for bootstrapping. The complementary status-accuracy view is reported separately in Table 1, where the open annotator scores 4.85 on the 1–5 status-only rubric, on par with GPT-5-mini's 4.91. Although GPT-5-mini also serves as the in-training judge during annotator RL, GPT-5.2 and DeepSeek-V3.2 provide a stronger and more independent assessment. In our pipeline, GPT-5-mini is used only during bootstrapping. After training, the learned annotator generates PDGs at near-zero marginal cost during policy RL, amortizing the one-time bootstrap cost rather than eliminating closed-source supervision altogether. Due to space constraints, detailed PDG-generation prompts, judging rubrics, and evaluation prompts are provided in the Appendix.

Model	Score	Price ($/1M tok)	
		Input	Output
GPT-4o	4.64	2.50	10.00
Claude-4-sonnet	4.65	3.00	15.00
Gemini-3-flash	4.82	0.50	3.00
GPT-5-mini	4.91	0.25	2.00
Ours	4.85	–	–

Table 1: Mean status score (1–5, higher is better) of PDG node-status annotations, judged by GPT-5.2-thinking and averaged over 100 examples. We additionally report public API token prices (USD per 1M tokens) for each proprietary annotator. `Ours` denotes post-trained Qwen3-8B progress annotator.

4.1.2 Reinforcement Learning Settings

Dataset and Evaluation. We perform RL post-training on NUMINAMATH-COT, and due to compute constraints we uniformly sample 150K prompts from the corpus as training data (Jia et al., 2024). We evaluate on a suite of public mathematical reasoning benchmarks, including MATH500 test split (Hendrycks et al., 2021), Minerva Math (Lewkowycz et al., 2022), OlympiadBench (He et al., 2024), AMC23, and AIME24. Following standard practice for reasoning evaluation, we report both avg@k and pass@k.

Model Backbone and Training Setup. Our policy models are instantiated from Qwen2.5-Math (1.5B and 7B) (Yang et al., 2024) and DeepSeek-R1-Distill-Qwen-1.5B (Guo et al., 2025a). We implement all methods using the open-source VERL training framework (Sheng et al., 2025) and run experiments on 4 nodes, each with 8×NVIDIA H20 (90GB) GPUs. Full hyperparameters and implementation details are provided in Table 9 and Appendix C.2.

Baselines. We compare against outcome-reward and process-reward baselines, including GRPO (Shao et al., 2024), SimpleRL (Zeng et al., 2025), PURE (Cheng et al., 2025), and PRL (Yao et al., 2026). We also compare against recent segment-level credit-assignment baselines, including SPO (Guo et al., 2025b) and VinePPO (Kazemnejad et al., 2025). To strengthen GRPO, we incorporate the asymmetric clipping used in DAPO with range $(0.2, 0.28)$ (Yu et al., 2025). Sensitivity to the scaling coefficient α is reported in Section 4.3. Due to page limits, we provide per-baseline descriptions and method-specific settings in Appendix C.2.

4.2 Comparative Results

Table 2 reports avg@8 for RLPgR alongside outcome-RL and process-RL baselines across multiple mathematical benchmarks. For Qwen2.5-Math (both 1.5B and 7B), we use $\alpha = 1.0$ at the high end of the sweep in Section 4.3. For DeepSeek-R1-Distill-Qwen-1.5B, we use the default $\alpha = 0.2$. Across the three backbones, RLPgR achieves the best mean avg@8 on every family. The per-dataset advantage is most pronounced on Qwen2.5-Math, while on DeepSeek-R1-Distill-Qwen-1.5B RLPgR is best-or-tied on most metrics.

On Qwen backbones, methods that leverage process rewards generally outperform those trained with outcome-only rewards. PURE uses a step-level PRM to provide process rewards and rescales them to penalize the worst step more aggressively (Cheng et al., 2025). PRL leverages the KL divergence between the policy and a reference model to decompose the final outcome reward across reasoning steps, without introducing an additional process reward model (Yao et al., 2026). Empirically, PURE is competitive on Qwen but fails to outperform GRPO on DeepSeek, suggesting that its reward formulation does not al-

Backbone	Method	Benchmarks					
		MATH500	Minerva	Olympiad	AIME24	AMC23	Avg.
Qwen2.5-Math-1.5B	Base	35.2	10.7	22.0	8.3	32.5	21.7
	GRPO	68.5	22.2	32.4	13.3	50.9	37.5
	PURE	69.6	22.7	32.7	13.3	50.9	37.8
	PRL	71.0	28.6	33.6	9.2	45.9	37.7
	RLPgR	73.6	30.5	39.1	10.0	60.0	42.6
Qwen2.5-Math-7B	Base	48.2	15.0	15.4	10.8	37.2	25.3
	GRPO	79.7	32.7	40.7	21.7	61.9	47.4
	SimpleRL	75.5	27.1	36.7	20.8	58.1	43.6
	PURE	81.3	35.4	45.2	23.3	62.2	49.5
	PRL	81.4	37.8	44.9	22.5	63.8	50.1
	RLPgR	82.4	39.0	45.3	30.0	77.5	54.8
DeepSeek-R1-Distill-Qwen-1.5B	Base	80.4	31.7	45.3	22.9	63.1	48.7
	GRPO	84.9	34.6	51.0	29.2	72.2	54.4
	PURE	83.6	33.0	51.6	28.8	66.9	52.8
	PRL	83.0	33.5	52.0	30.0	72.5	54.2
	RLPgR	85.1	35.4	53.2	29.2	71.6	54.9

Table 2: Main **avg@8** results on math reasoning benchmarks.

ways translate process-level supervision into effective learning signals. As a complementary check, pass@8 results in Appendix B.2 show the same ordering as avg@8 across both Qwen2.5-Math-7B and DeepSeek-R1-Distill-Qwen-1.5B, suggesting that RLPgR improves the underlying model capacity rather than reinforcing superficial patterns to boost benchmark performance (Yue et al., 2025a).

During training on DeepSeek-R1-Distill-Qwen-1.5B, generation length under PURE steadily increases and rollouts are frequently truncated. This suggests the model is exploiting response length rather than improving reasoning quality. In contrast, RLPgR builds on group-relative advantages but explicitly adjusts the advantages of implausible nodes, without introducing length-dependent bias in the advantage computation. This lets RLPgR capitalize on process supervision while avoiding length hacking. RLPgR also attains stronger overall performance, suggesting that targeted dependency-aware credit assignment helps the model learn more reliable reasoning patterns.

4.3 Mechanism Studies

We conduct three studies to understand how RLPgR works, varying the strength of node-level scaling, the fidelity of PDG labels, and the granularity of credit assignment.

Table 3 sweeps the single scalar α over $\{0.2, 0.5, 1.0\}$ on Qwen2.5-Math-1.5B and Qwen2.5-Math-7B with all other settings held fixed. The mean avg@8 increases monotonically with α on both backbones, from 40.7 to 42.6 on 1.5B and from 50.2 to 54.8 on 7B. The gain is larger on the 7B model, suggesting that larger backbones benefit from a stronger dependency-aware correction.

Table 4 runs a controlled status-shuffle experiment on Qwen2.5-Math-7B, where for each rollout a fixed fraction of progress-node statuses is randomly relabeled in $\{\mathrm{CP}, \mathrm{IE}, \mathrm{HE}, \mathrm{CE}\}$ before $\kappa(v)$ is computed via Eq. (2), while dependency edges and the rest of the training pipeline are unchanged. Downstream avg@8 degrades smoothly from 50.2 at perturbation ratio 0.0 to 44.5 at 0.3. The smooth decay confirms that PDG node statuses provide a real credit-assignment signal and that RLPgR tolerates moderate label noise without collapsing.

Finally, we compare RLPgR against recent segment-level credit-assignment baselines (SPO, VinePPO) in Table 5 to test whether the gain comes specifically from dependency modeling. Following Guo et al. (2025b), we evaluate on GSM8K with RhoMath 1.1B as the backbone. Baseline numbers are taken from their paper, and RLPgR is rerun under the same training setup. RLPgR yields the highest test accuracy, providing an additional gain over coarser segmentation.

4.4 Ablation Studies and Discussion

Impact of PDG Quality on Training Performance. To study how PDG quality affects policy learning, we train RLPgR policies using progress annotators from different stages of annotator train-

Backbone	α	MATH500	Minerva	Olympiad	AIME24	AMC23	Avg.	Diff
Qwen2.5-Math-1.5B	0.2 (default)	72.5	29.0	35.5	11.7	55.0	40.7	0.0
Qwen2.5-Math-1.5B	0.5	73.0	28.7	38.0	13.3	52.5	41.1	0.4
Qwen2.5-Math-1.5B	1.0	73.6	30.5	39.1	10.0	60.0	42.6	1.9
Qwen2.5-Math-7B	0.2 (default)	81.3	36.8	43.1	24.6	65.0	50.2	0.0
Qwen2.5-Math-7B	0.5	81.6	36.8	48.3	20.0	75.0	52.3	2.2
Qwen2.5-Math-7B	1.0	82.4	39.0	45.3	30.0	77.5	54.8	4.7

Table 3: Sensitivity of RLPgR to the single scaling coefficient α on Qwen2.5-Math backbones. "Diff" is measured relative to the default setting $\alpha = 0.2$.

Shuffle	MATH500	Minerva	Olympiad	AIME24	AMC23	Avg.
0.0	81.3	36.8	43.1	24.6	65.0	50.2
0.1	80.8	36.5	44.2	20.0	62.5	48.8
0.2	79.6	36.0	41.2	20.0	57.5	46.9
0.3	80.6	35.2	39.8	16.7	50.0	44.5

Table 4: Avg@8 of RLPgR on Qwen2.5-Math-7B under controlled status-shuffle perturbation. "Shuffle" is the fraction of PDG node statuses randomly relabeled in $\{\mathrm{CP}, \mathrm{IE}, \mathrm{HE}, \mathrm{CE}\}$ before computing $\kappa(v)$, with the rest of the training pipeline held fixed. The smooth decay confirms that the gain depends causally on PDG label fidelity rather than on incidental hyperparameters.

Method	Performance (%)
Init SFT	40.3
RestEM	42.8
RLOO	44.5
GRPO	44.6
DPO+x	46.4
PPO	50.1
VinePPO	53.4
SPO	56.7
RLPgR	57.1

Table 5: Test accuracy on GSM8K with RhoMath 1.1B, comparing against segment-level and other RL baselines. RLPgR achieves the highest score.

ing: an SFT checkpoint and two post-trained checkpoints at steps 100 and 260. Moving across these checkpoints yields a monotonic +1.1 avg@8 and +2.7 pass@8 improvement from the SFT to the step-260 annotator, with the full per-benchmark breakdown in Appendix B.1, Table 6. The improvement comes from sharper status-label assignment after RL post-training, which in turn produces more discriminative node-level scaling during policy RL. This indicates that high-fidelity PDG construction is critical for translating process supervision into downstream RL gains.

Training Dynamics. Across training, GRPO maintains the highest token entropy, while PURE undergoes an early entropy collapse. The early collapse in PURE narrows the policy distribution before sufficient exploration has taken place. By contrast, RLPgR stays close to GRPO and well above PURE, suggesting that progress-node rewards provide useful process guidance without suppressing exploration as aggressively as step-level supervision. Within RLPgR, annotators that produce more progress nodes yield slightly lower entropy, consistent with a trade-off where denser process feedback can reduce exploration. Due to space constraints, detailed entropy and node-count trajectories are reported in Appendix B.4, Figure 2.

5 Conclusion

We propose RLPgR, a sign-preserving dependency-aware rescaling of the GRPO advantage broadcast for long-chain reasoning. Final correctness still governs the rollout-level direction, and the redistribution avoids the reward-hacking drift typical of densely shaped process rewards. On verifiable mathematical reasoning benchmarks across various backbones, RLPgR is competitive with strong outcome-RL and process-RL baselines, and our mechanism studies show that the improvement scales monotonically with the strength of dependency-aware correction, exceeds segment-level credit-assignment baselines, and degrades smoothly under controlled PDG label perturbation. After a one-time bootstrap stage that uses a strong proprietary annotator, an open Qwen3-8B progress annotator amortizes the recurring proprietary annotation cost during policy RL. Future work includes extending RLPgR to multi-turn agent tasks, and open-ended evaluation.

Limitations

We discuss the main limitations and threats to validity of this work. Our evaluation focuses on verifiable mathematical reasoning with three backbones, Qwen2.5-Math 1.5B and 7B and DeepSeek-R1-Distill-Qwen-1.5B, which may limit external validity across broader domains, architectures, and scales. We choose this setting because mathematical reasoning provides reliable outcome verification and a controlled testbed for studying progress-level credit assignment, and we reduce confounding factors by matching data, training, decoding, and evaluation pipelines across methods. RLPgR also relies on a learned progress annotator bootstrapped with GPT-5-mini for cold-start PDG generation and rubric-based reward judging during annotator RL, which means that applying it to a new domain still requires a competent bootstrap annotator. However, this cost is amortized after training, since the learned annotator can replace repeated proprietary annotation during RL, and the final policy optimization only uses mild, sign-preserving rescaling rather than replacing outcome rewards.

Ethical Considerations

Reinforcement learning post-training that produces longer or more confident chains of reasoning can also produce more convincing yet incorrect explanations, especially when intermediate steps look locally valid but rest on flawed premises. RLPgR is built on top of LLM-based progress annotators and an LLM-as-judge protocol, both of which can inherit systematic biases from the underlying models. We mitigate these risks by restricting evaluation to tasks with verifiable outcomes, by reporting sensitivity to PDG label fidelity and to KL-penalty choices, and by releasing the full prompts used for PDG generation and judging so that downstream users can audit and replace them. Practitioners deploying RLPgR-trained policies in settings without reliable verification should account for amplified residual confidence in incorrect reasoning chains and for the failure modes inherited from upstream annotators.

References

Dilip Arumugam, Peter Henderson, and Pierre-Luc Bacon. 2021. An information-theoretic perspective on credit assignment in reinforcement learning. *Preprint*, arXiv:2103.06224.

Meng Cao, Shuyuan Zhang, Xiao-Wen Chang, and Doina Precup. 2025. Scar: Shapley credit assignment for more efficient rlhf. *Preprint*, arXiv:2505.20417.

Shulin Cao, Jiajie Zhang, Jiaxin Shi, Xin Lv, Zijun Yao, Qi Tian, Lei Hou, and Juanzi Li. 2023. Probabilistic tree-of-thought reasoning for answering knowledge-intensive complex questions. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 12541–12560.

Alex J. Chan, Hao Sun, Samuel Holt, and Mihaela van der Schaar. 2024. Dense reward for free in reinforcement learning from human feedback. *Preprint*, arXiv:2402.00782.

Wenxiang Chen, Wei He, Zhiheng Xi, Honglin Guo, Boyang Hong, Jiazheng Zhang, Nijun Li, Tao Gui, Yun Li, Qi Zhang, and Xuanjing Huang. 2025. Better process supervision with bi-directional rewarding signals. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 14471–14485, Vienna, Austria. Association for Computational Linguistics.

Jie Cheng, Ruixi Qiao, Lijun Li, Chao Guo, Junle Wang, Gang Xiong, Yisheng Lv, and Fei-Yue Wang. 2025. Stop summation: Min-form credit assignment is all process reward model needs for reasoning. *arXiv preprint arXiv:2504.15275*.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Yuchen Zhang, Jiacheng Chen, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, Jiarui Yuan, Huayu Chen, Kaiyan Zhang, Xingtai Lv, Shuo Wang, Yuan Yao, Xu Han, and 6 others. 2025. Process reinforcement through implicit rewards. *Preprint*, arXiv:2502.01456.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2024. Chain-of-verification reduces hallucination in large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3563–3578, Bangkok, Thailand. Association for Computational Linguistics.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. 2025. Rubrics as rewards: Reinforcement learning beyond verifiable domains. *arXiv preprint arXiv:2507.17746*.

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

Yiran Guo, Lijie Xu, Jie Liu, Dan Ye, and Shuang Qiu. 2025b. Segment policy optimization: Effective segment-level credit assignment in rl for large language models. *arXiv preprint arXiv:2505.23564*.

Bingxiang He, Zekai Qu, Zeyuan Liu, Yinghao Chen, Yuxin Zuo, Cheng Qian, Kaiyan Zhang, Weize Chen, Chaojun Xiao, Ganqu Cui, and 1 others. 2025. Justrl: Scaling a 1.5 b llm with a simple rl recipe. *arXiv preprint arXiv:2512.16649*.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, and 1 others. 2024. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3828–3850.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. 2025. Reinforce++: Stabilizing critic-free policy optimization with global advantage normalization. *Preprint*, arXiv:2501.03262.

LI Jia, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, and 1 others. 2024. Numinamath.

Amirhossein Kazemnejad, Milad Aghajohari, Eva Portelance, Alessandro Sordoni, Siva Reddy, Aaron Courville, and Nicolas Le Roux. 2025. Vineppo: Refining credit assignment in rl training of llms. *Preprint*, arXiv:2410.01679.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, and 1 others. 2022. Solving quantitative reasoning problems with language models. *Advances in neural information processing systems*, 35:3843–3857.

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

Tianci Liu, Ran Xu, Tony Yu, Ilgee Hong, Carl Yang, Tuo Zhao, and Haoyu Wang. 2026. Openrubrics: Towards scalable synthetic rubric generation for reward modeling and llm alignment. *Preprint*, arXiv:2510.07743.

Sagnik Mukherjee, Abhinav Chinta, Takyoung Kim, Tarun Anoop Sharma, and Dilek Hakkani Tur. 2025. Premise-augmented reasoning chains improve error identification in math reasoning with LLMs. In *Forty-second International Conference on Machine Learning*.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *Preprint*, arXiv:2203.02155.

et al Richard S Sutton, Andrew G Barto. 1998. *Reinforcement learning: An introduction*. MIT press Cambridge.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. 2024. Rewarding progress: Scaling automated process verifiers for llm reasoning. *Preprint*, arXiv:2410.08146.

Rulin Shao, Akari Asai, Shannon Zejiang Shen, Hamish Ivison, Varsha Kishore, Jingming Zhuo, Xinran Zhao, Molly Park, Samuel G. Finlayson, David Sontag, Tyler Murray, Sewon Min, Pradeep Dasigi, Luca Soldaini, Faeze Brahman, Wen tau Yih, Tongshuang Wu, Luke Zettlemoyer, Yoon Kim, and 2 others. 2025. Dr tulu: Reinforcement learning with evolving rubrics for deep research. *Preprint*, arXiv:2511.19399.

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

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

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. 2022. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*.

Shenzhi Wang, Le Yu, Chang Gao, Chujie Zheng, Shixuan Liu, Rui Lu, Kai Dang, Xionghui Chen, Jianxin Yang, Zhenru Zhang, Yuqiong Liu, An Yang, Andrew Zhao, Yang Yue, Shiji Song, Bowen Yu, Gao Huang, and Junyang Lin. 2025. Beyond the 80/20 rule: High-entropy minority tokens drive effective reinforcement learning for llm reasoning. *Preprint*, arXiv:2506.01939.

Lilian Weng. 2024. Reward hacking in reinforcement learning. *lilianweng.github.io*.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, and 1 others. 2024.

Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

Jiarui Yao, Ruida Wang, and Tong Zhang. 2026. Prl: Process reward learning improves llms' reasoning ability and broadens the reasoning boundary. *arXiv preprint arXiv:2601.10201*.

Evelyn Yee, Alice Li, Chenyu Tang, Yeon Ho Jung, Ramamohan Paturi, and Leon Bergen. 2024. Faithful and unfaithful error recovery in chain of thought. In *First Conference on Language Modeling*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, and 16 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *Preprint*, arXiv:2503.14476.

Youliang Yuan, Qiuyang Mang, Jingbang Chen, Hong Wan, Xiaoyuan Liu, Junjielong Xu, Jen tse Huang, Wenxuan Wang, Wenxiang Jiao, and Pinjia He. 2025a. Curing miracle steps in llm mathematical reasoning with rubric rewards. *Preprint*, arXiv:2510.07774.

Yufeng Yuan, Yu Yue, Ruofei Zhu, Tiantian Fan, and Lin Yan. 2025b. What's behind ppo's collapse in long-cot? value optimization holds the secret. *Preprint*, arXiv:2503.01491.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. 2025a. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *arXiv preprint arXiv:2504.13837*.

Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, Xiangpeng Wei, Xiangyu Yu, Gaohong Liu, Juncai Liu, Lingjun Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Chi Zhang, and 8 others. 2025b. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks. *Preprint*, arXiv:2504.05118.

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

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024a. Rest-mcts*: Llm self-training via process reward guided tree search. *Advances in Neural Information Processing Systems*, 37:64735–64772.

Muru Zhang, Ofir Press, William Merrill, Alisa Liu, and Noah A. Smith. 2024b. How language model hallucinations can snowball. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 59670–59684. PMLR.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2025. The lessons of developing process reward models in mathematical reasoning. *Preprint*, arXiv:2501.07301.

A Mechanism Intuition for Sign-Preserving Rescaling

This appendix gives a mechanism-level intuition for why a small, sign-preserving, dependency-aware rescaling of the GRPO advantage can improve gradient alignment with the latent ideal direction in expectation. We frame the argument under an oracle PDG assumption to isolate the role of sign-mismatch correction. The realistic, noisy PDG regime is studied empirically in Section 4.3 via controlled status-shuffle perturbation.

Section 3.3 established that in GRPO-style training, each rollout receives a single trajectory-level advantage that is broadcast to all its tokens, which inevitably misassigns credit whenever a trajectory mixes tokens whose latent ideal update direction agrees or disagrees with $\hat{A}_t$. Two prevalent cases are False Positive ($\hat{A}_t > 0, A_t^* < 0$) and False Negative ($\hat{A}_t < 0, A_t^* > 0$), where the proxy advantage $\hat{A}_t$ contradicts the latent ground truth A_t^*. Here A_t^* encodes the ideal direction in which a policy should update on token t to maximize verifiable correctness, rather than a local judgment of whether the step is internally consistent. An HE token in a failed rollout therefore has $A_t^* < 0$ and falls into $\mathcal{C}$, since the ideal direction is to detect the contaminated upstream premise rather than chain on it.

Assumption A.1 (Mismatch Identification)**.** Let $A^*(s_t, a_t)$ denote the latent (ground-truth) token-level advantage, and let $\hat{A}_t$ be the proxy advantage assigned by GRPO (broadcast within each rollout). Define the *consistent* and *mismatched* token sets:

$$\mathcal{C} = \{t : \mathrm{sign}(\hat{A}_t) = \mathrm{sign}(A^*(s_t, a_t))\}, \quad (8)$$

$$\mathcal{M} = \{t : \mathrm{sign}(\hat{A}_t) \neq \mathrm{sign}(A^*(s_t, a_t))\}. \quad (9)$$

We assume the PDG is an oracle for this partition, i.e., it perfectly identifies whether each token t belongs to the mismatch set $\mathcal{M}$ or the complement set $\mathcal{C}$ (see Section 4.1.1 for empirical evidence). Accordingly, we set the scaling coefficient as $\kappa_t \geq 1$ for $t \in \mathcal{C}$ and $\kappa_t = \lambda \in [0, 1)$ for $t \in \mathcal{M}$.

Lemma A.2 (Alignment Improvement)**.** *Let g^* denote the latent, ideal policy-gradient direction induced by token-level advantages A_t^*. Let g_{grpo} and g_{ours} be the stochastic gradient estimators computed by GRPO and by our reweighted method, respectively. Under Assumption A.1, our estimator strictly improves alignment with the ideal direction:*

$$\mathbb{E}\left[\langle g_{\mathrm{ours}}, g^* \rangle\right] > \mathbb{E}\left[\langle g_{\mathrm{grpo}}, g^* \rangle\right], \quad (10)$$

whenever $\mathbb{P}(|\mathcal{M}| > 0) > 0$.

Proof. Let $u_t = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ be the score function. Define token-summed estimators:

$$g(\alpha) = \sum_{t \in \mathcal{T}} \alpha_t u_t, \quad (11)$$

where $\alpha_t \in \{\hat{A}_t,\ \kappa_t \hat{A}_t,\ A_t^*\}$ corresponds to g_{grpo}, g_{ours}, and g^*, respectively. Consider the alignment difference

$$\begin{aligned} \Delta &\triangleq \mathbb{E}[\langle g_{\mathrm{ours}}, g^* \rangle] - \mathbb{E}[\langle g_{\mathrm{grpo}}, g^* \rangle] \\ &= \mathbb{E}\left[\left\langle \sum_{t \in \mathcal{T}} (\kappa_t - 1)\hat{A}_t u_t,\ \sum_{t \in \mathcal{T}} A_t^* u_t \right\rangle\right]. \quad (12) \end{aligned}$$

Under the simplifying assumption that token-wise scores are approximately uncorrelated, i.e., $\mathbb{E}[\langle u_t, u_s \rangle] \approx 0$ for $t \neq s$, we keep the diagonal terms and split the sum over $\mathcal{C}$ and $\mathcal{M}$:

$$\begin{aligned} \Delta &= \mathbb{E}\left[\sum_{t \in \mathcal{T}} (\kappa_t - 1)\hat{A}_t A_t^* \|u_t\|^2\right] \\ &= \mathbb{E}\left[\sum_{t \in \mathcal{C}} (\kappa_t - 1)\hat{A}_t A_t^* \|u_t\|^2\right] \\ &\quad + (\lambda - 1)\,\mathbb{E}\left[\sum_{t \in \mathcal{T}} \hat{A}_t A_t^* \|u_t\|^2 \mathbf{1}\{t \in \mathcal{M}\}\right]. \end{aligned} \quad (13)$$

For $t \in \mathcal{C}$, $\hat{A}_t A_t^* > 0$ and $\kappa_t - 1 \geq 0$, so the first term is non-negative. For $t \in \mathcal{M}$, $\hat{A}_t A_t^* < 0$ and $\lambda - 1 < 0$, so the second term is strictly positive whenever $\mathbb{P}(|\mathcal{M}| > 0) > 0$. Hence $\Delta > 0$. □

Caveat on the Oracle Assumption. Lemma A.2 is a mechanism-level intuition rather than a standalone theoretical guarantee for the deployed RLPgR pipeline. The oracle assumption that the PDG perfectly identifies the mismatch partition $\mathcal{M}$ is an idealization. In practice, the PDG is produced by a learned annotator and may misassign node statuses, especially under distribution shift. The complementary, realistic regime is studied empirically in Section 4.3, where the controlled status-shuffle experiment shows that downstream avg@8 degrades smoothly from 50.2 at zero perturbation to 44.5 at 30% random relabeling, indicating that the gain depends on PDG label fidelity but does not collapse under moderate label noise.

B Additional Experiment Results

B.1 Per-Benchmark Ablations on KL Design and PDG Annotator Quality

Table 6 reports the per-benchmark breakdown underlying the inline summary in Section 4.4 for Qwen2.5-Math-1.5B, ablating (i) the KL-penalty design in GRPO-style optimization and (ii) the PDG annotator quality used to derive node-level scaling factors. All settings are evaluated under the same multi-sample decoding protocol, and we report both avg@8 and pass@8 on five public math benchmarks.

Effect of PDG Annotator Quality. Holding the objective fixed (No KL), stronger progress annotators translate into better downstream policy performance. Moving from the SFT annotator to the post-trained checkpoints yields monotonic improvements in both avg@8 (39.6 $\rightarrow$ 39.8 $\rightarrow$ 40.7) and pass@8 (58.4 $\rightarrow$ 58.8 $\rightarrow$ 61.1). The gains are more pronounced on harder subsets where error attribution and recovery matter more (e.g., Minerva and AMC23), while MATH500 remains relatively stable. This trend supports our claim that higher-fidelity PDG construction produces more accurate node statuses, which in turn yields more informative advantage reweighting during policy learning.

KL Design Choice. Comparing the two KL variants with the same annotator (260-step), adding the KL as an explicit auxiliary term yields slightly stronger avg@8 than folding KL into the reward (40.6 vs. 40.3), and improves pass@8 (59.9 vs. 58.4), with gains on Minerva and AIME24. At the same time, removing KL entirely remains competitive and achieves the highest avg@8 and pass@8 in our ablation when paired with the 260-step annotator (avg@8 = 40.7, pass@8 = 61.1). Overall, these results suggest that (a) the choice of KL formulation can matter for stability and final accuracy, and (b) PDG quality is the dominant factor, since improvements from better annotators hold across KL settings and can outweigh the differences between KL designs.

B.2 Pass@8 Results

We additionally report pass@8 numbers for RLPgR on Qwen2.5-Math-7B and DeepSeek-R1-Distill-Qwen-1.5B in Table 7, evaluated under the default RLPgR configuration. The ordering across baselines mirrors the avg@8 ordering reported in Table 2, suggesting that the gains observed under avg@8 reflect changes to the underlying model capacity rather than to a single decoding mode.

B.3 All-Zero Group Treatment Ablation

The default GRPO update drops a group whose K rollouts are all correct or all incorrect, because the group-relative advantage $A^{(j)}$ degenerates to zero and any node-wise rescaling $\kappa(v) \cdot A^{(j)}$ also vanishes. To probe whether the dependency-aware progress signal can still be useful in this regime, we modify the training setup so that all-zero-advantage groups are retained, and replace the default scaling rule on those groups with an additive bias term controlled by the same scalar α, so that the per-node update no longer vanishes when $A^{(j)} = 0$. All other components, including the PDG construction, the four-status taxonomy, and the optimizer, are kept unchanged.

Table 8 reports avg@8 on Qwen2.5-Math-7B. Holding the scaling coefficient fixed at $\alpha = 0.5$, retaining the all-zero-advantage groups with an additive bias term yields a separable $+1.5$ avg@8 improvement over dropping them ($52.3 \rightarrow 53.8$), on top of the $+2.1$ gain from the same α shift under the default drop convention ($50.2 \rightarrow 52.3$). This indicates that the dependency-aware progress signal carries useful information even on prompts where the verifiable outcome alone provides no learning signal, so the all-zero-advantage regime is a soft design choice rather than a hard failure mode of RLPgR. The additive-bias variant is no longer sign-preserving by construction. We treat this as a controlled extension that complements the main result table. The headline numbers in Table 2 use the standard GRPO convention of dropping all-zero-advantage groups.

B.4 Training Dynamics

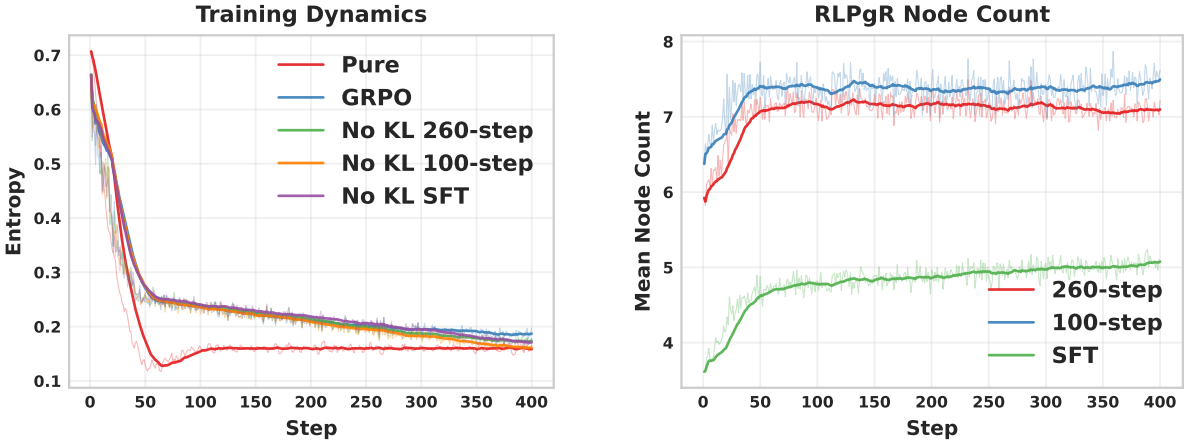
Figure 2a compares token-entropy trajectories across GRPO, step-level process supervision PURE, and node-level process supervision RLPgR (instantiated with three progress annotators of different fidelities). GRPO maintains the highest entropy throughout training, while PURE exhibits an early entropy collapse. RLPgR preserves higher entropy than PURE and stays close to GRPO, suggesting that progress-node rewards provide useful process guidance without suppressing exploration as aggressively as step-level supervision. Within RLPgR, Figure 2b shows that annotators producing more progress nodes per rollout tend to yield slightly lower entropy, consistent with a trade-off

Setting		Results					
Annotator	**KL Penalty**	**MATH500**	**Minerva**	**Olympiad**	**AIME24**	**AMC23**	**Avg.**
			Avg@8				
260 Step	KL in Reward	71.7	27.9	34.8	12.1	55.0	40.3
260 Step	KL Term	71.6	28.0	34.6	15.8	52.8	40.6
SFT	No KL	72.9	27.5	34.9	10.4	52.5	39.6
100 Step	No KL	72.3	28.1	35.5	13.3	49.7	39.8
260 Step	No KL	72.5	29.0	35.5	11.7	55.0	40.7
			Pass@8				
260 Step	KL in Reward	88.4	43.8	54.1	23.3	82.5	58.4
260 Step	KL Term	88.8	46.3	55.0	26.7	82.5	59.9
SFT	No KL	88.2	43.4	55.6	30.0	75.0	58.4
100 Step	No KL	88.8	42.6	56.0	26.7	80.0	58.8
260 Step	No KL	87.8	44.9	54.7	33.3	85.0	61.1

Table 6: Per-benchmark breakdown for the ablations on (i) KL-penalty design and (ii) PDG annotator quality, summarized inline in Section 4.4, on Qwen2.5-Math-1.5B.

Backbone	**Method**	**Benchmarks**					
		MATH500	**Minerva**	**Olympiad**	**AIME24**	**AMC23**	**Avg.**
Qwen2.5-Math-7B	Base	84.2	41.2	42.2	26.7	80.0	54.9
	GRPO	92.0	49.3	61.2	50.0	82.5	67.0
	SimpleRL	91.6	48.5	58.5	46.7	80.0	65.1
	PURE	90.8	49.6	65.8	50.0	82.5	67.7
	PRL	93.6	52.6	65.2	43.3	85.0	67.9
	RLPgR	93.4	52.2	65.5	46.7	87.5	69.0
DeepSeek-R1-Distill-Qwen-1.5B	Base	94.4	50.4	65.8	50.0	85.0	69.1
	GRPO	96.0	51.1	65.5	60.0	92.5	73.0
	PURE	94.6	50.7	64.6	60.0	92.5	72.5
	RLPgR	96.0	52.2	66.4	60.0	92.5	73.4

Table 7: Main **pass@8** results on math reasoning benchmarks under the default RLPgR configuration.



(a) Entropy comparison across training paradigms.

(b) Node count comparison across progress annotators.

Figure 2: Training dynamics of RLPgR compared with GRPO and PURE. (a) Entropy during training for different paradigms. (b) Number of progress nodes generated by different annotators during RLPgR training.

where denser process feedback can reduce exploration. Taken together, these results indicate that node-level process supervision can capture the performance gains of process-guided training while

Setting	MATH500	Minerva	Olympiad	AIME24	AMC23	Avg.	Diff
$\alpha = 0.2$ (baseline)	81.3	36.8	43.1	24.6	65.0	50.2	0.0
$\alpha = 0.5$	81.6	36.8	48.3	20.0	75.0	52.3	+2.1
$\alpha = 0.5$ (keep all groups)	82.2	37.5	47.4	26.7	75.0	53.8	+3.6

Table 8: Avg@8 of RLPgR on Qwen2.5-Math-7B with and without the all-zero-group treatment. "Diff" is measured against the default $\alpha = 0.2$ baseline. Under $\alpha = 0.5$, retaining the all-zero-advantage groups with an additive bias term improves avg@8 by 1.5 points over dropping them ($52.3 \rightarrow 53.8$).

avoiding the sharp reduction in exploration induced by step-level supervision.

B.5 Additional Win-Rate Analysis for Progress Annotator

Figure 3 reports a more fine-grained LLM-as-judge win-rate curve of our progress annotator across training steps under three different judges. Overall, the win rate exhibits a clear rapid-improvement stage followed by a plateau stage. From 40 to 140 steps, performance increases sharply, indicating that the annotator quickly acquires core PDG-construction capabilities early in training. After 240 steps, the curves stabilize and improvements become marginal, with DeepSeek-v3.2 and GPT-5-mini judges reporting high win rates around 70–80% with mild fluctuations, suggesting that most gains are consolidated and subsequent training mainly refines details. Considering stability across judges, checkpoints around 240–280 steps offer a favorable trade-off between peak performance and cross-judge consistency.

A notable observation is that DeepSeek-v3.2 and GPT-5-mini judges are highly aligned in both trend and magnitude, supporting the robustness of our conclusions across multiple evaluation models. In contrast, GPT-5.2 yields systematically lower win rates (typically 49–59% after 200 steps), implying that different judges may emphasize different aspects of PDG quality (e.g., strictness of correctness criteria, preference for conservative decisions, or sensitivity to formatting). Therefore, we report results with multiple judges to provide a comprehensive and reliable view of annotator quality.

C Additional Experiment Details

C.1 Runtime Environment and Hyperparameters

Progress annotator (Annotator RL) and policy post-training (Policy RL) hyperparameters and training settings are summarized in Table 9. All settings are chosen to trade off training stability, throughput, and GPU memory footprint while maintaining competitive final performance. We conduct both Annotator RL and Policy RL in the `verl` framework with AdamW, enabling gradient checkpointing throughout to fit long-context training. Unless otherwise specified, we use a unified learning rate of 1×10^{-6} and sample rollouts with temperature 1, Top-$k = 20$, and Top-$p = 0.95$. During evaluation, we use stochastic rollouts with temperature 0.6 and Top-$p = 0.95$.

For Annotator RL, we train for 3 epochs with batch size 64 and a maximum sequence length of 12,288 tokens to support long-horizon progress annotation. For Policy RL, we train for 1 epoch with batch size 256, using backbone-specific context lengths (4,096 for Qwen and 9,216 for DeepSeek) to match each model's effective context window. We apply PPO-style clipping with a clip ratio in $[0.2,\ 0.28]$ and set the KL regularization coefficient to 0.001 to control policy drift from the reference model. All experiments are run on 32 NVIDIA H20 GPUs (90GB each, 2,880GB total GPU memory).

C.2 Additional Details for Baselines

GRPO Group Relative Policy Optimization (GRPO) is a value-free policy-gradient baseline that replaces an explicit critic with within-prompt relative comparisons across multiple sampled rollouts. Starting from the Base policy, we draw a group of K responses $\{y^{(k)}\}_{k=1}^{K}$ for each prompt x and evaluate each completion with a scalar task reward $R^{(k)}$ (e.g., preference-model reward or verifiable correctness). We then construct a relative advantage by centering (and optionally normalizing) rewards within the group, $A^{(k)} = (R^{(k)} - \mu_R)/(\sigma_R + \epsilon)$, which reduces variance without learning $V(s)$. Policy updates follow a PPO-style clipped surrogate over tokens, using the sequence log-ratio $r^{(k)} = \exp(\log \pi_\theta(y^{(k)} \mid x) - \log \pi_{\theta_{\text{old}}}(y^{(k)} \mid x))$ and the standard clipping rule, while constraining drift from a fixed reference policy via a KL penalty. GRPO thus provides a strong and simple RL baseline that isolates the effect of relative credit assignment from critic learn-

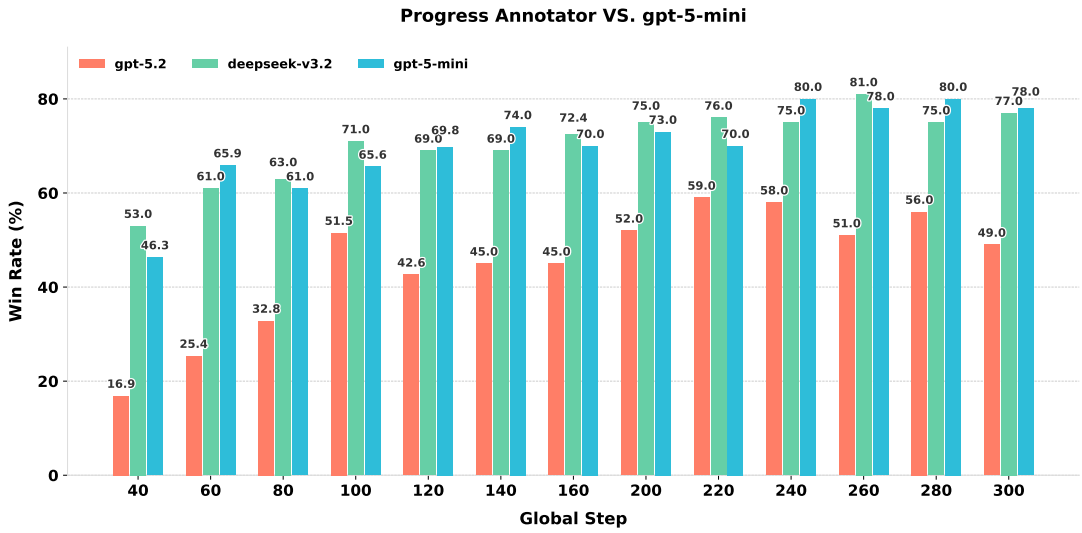


Figure 3: Detailed post-training win rate of the progress annotator against GPT-5-mini generated PDGs at different training steps.

ing.

Simple-RL Simple-RL provides a systematic recipe for zero-RL training, where a base model is directly optimized with reinforcement learning using a lightweight, rule-based reward (e.g., correctness on math problems), without relying on preference data or a learned reward model. The core training loop uses GRPO-style group sampling. For each prompt, the policy generates multiple candidate solutions, receives a binary verifiable reward per completion, and updates the policy with a clipped policy-gradient objective under a reference-KL constraint. In our baseline, we follow this protocol as closely as possible, matching decoding, rollout length limits, and optimization settings, so that improvements can be attributed to algorithmic differences rather than reward shaping or formatting heuristics.

PURE Process sUpervised Reinforcement lEarning (PURE) addresses a common failure mode of PRM-driven training. When process rewards are aggregated with the conventional sum-form return, models can exploit high-scoring intermediate steps and exhibit severe reward hacking. To mitigate this, PURE replaces the discounted cumulative return with a min-form credit assignment, defining the return from time t as the minimum future process reward, $G_t = \min(r_t^p, \ldots, r_T^p)$, which bounds values and concentrates learning pressure on the weakest step along the chain of reasoning. Starting from the SFT policy, we score each generated solution with a PRM to obtain per-step rewards $\{r_t^p\}$, compute min-form returns (and corresponding advantages), and apply a GRPO-style clipped policy update with the same reference-KL constraint used in other RL baselines.

PRL Process Reward Learning (PRL) targets the sparsity of outcome-only supervision by converting a trajectory-level objective into a dense, stepwise training signal derived from an entropy-regularized RL formulation. Rather than learning an external process reward model, PRL decomposes the objective into (i) a terminal reward component and (ii) a per-token regularization term that measures deviation from a reference policy, yielding a process advantage that can be optimized with standard policy-gradient machinery. Concretely, PRL defines a stepwise advantage by subtracting a cumulative log-ratio (or KL-shaped) term from the trajectory-level signal, and applies `stopgrad` to stabilize optimization when backpropagating through long sequences.

C.3 Prompts Used for Our Method

To ensure reproducibility and to make our PDG construction and evaluation procedures fully transparent, we report the exact prompts used in our pipeline. The three prompts shown below correspond to (i) PDG generation, (ii) pairwise winrate comparison between two PDG outputs, and (iii) benchmark-style scoring of multiple candidates. All prompts enforce strict output formats (JSON-only or single-token decisions) to support automatic parsing and large-scale evaluation.

Hyperparameters	Annotator RL	Policy RL	
		Qwen	DeepSeek
Batch Size	64	256	
Learning Rate	1×10^{-6}		
Temperature	1		
Top-K	20		
Top-P	0.95		
Epochs	3	1	
Optimizer	AdamW		
Gradient Checkpointing	Enabled		
Clip Ratio	$[0.2,\ 0.28]$		
KL Coeff	0.001		
Training Framework	verl		
Sequence Length	12288	4096	9216
GPUs	32 × NVIDIA H20		
Total GPU Memory	2880GB (90GB per GPU)		

Table 9: Hyperparameters and training configurations for progress annotator and Policy post-training. We emphasize that the KL Coeff only plays a role in the KL term and KL in reward settings, and the coefficient for both settings is 0.001.

Placeholders such as `$Question`, `$Solution`, and `$ProgressNodes` are programmatically substituted with the task instance at runtime.

Progress Annotator Prompt for Progress Generation. This prompt is used to construct a Progress Dependency Graph (PDG) from a raw step-by-step rollout. It defines the PDG entities (progress node, artifact, dependency) and prescribes a stepwise auditing protocol. This protocol covers segmentation into milestone-level nodes, extraction of artifacts and dependency edges, and a dual-layer counterfactual verification that separates (a) whether the node's inputs are correct from (b) whether the node's internal reasoning is correct given those inputs. The prompt then maps the two verification flags to one of four status labels (CP/IE/HE/CE) and requires a single valid JSON object as output, enabling the generated PDG to be consumed directly by downstream training and analysis code.

LLM-as-Judge Prompt for Win-Rate Evaluation. This prompt is used for the pairwise evaluation reported in our annotator win-rate analysis. Given the same math problem, golden answer, rollout, and two candidate PDGs, the judge compares them using both node-level criteria (status accuracy; coherence/completeness of the node's step set) and global criteria (segmentation validity, boundary clarity, coverage, dependency correctness/necessity, and faithful error attribution). The prompt enforces a strict decision output. The judge must return exactly one letter (`A`/`B`/`C`) indicating which PDG is better or whether it is a tie. This design yields a robust, easily aggregatable preference signal for tracking annotator improvements across training steps.

LLM-as-Judge Prompt for Benchmark Scoring. This prompt is used for multi-candidate scoring (e.g., Table 1). The judge receives a set of anonymized PDG candidates (C01, C02, ...) for the same problem and rollout, and assigns an integer `status_score` in $[1, 5]$ to each candidate based on the correctness of node-status assignment. The prompt returns a single JSON object mapping candidate IDs to scores and short notes, which supports automated aggregation.

D Case study of generated PDG data

To concretize our PDG annotation scheme, we include two JSON-formatted examples of generated `progress_nodes` together with their rendered PDG diagrams, a wrong case in Figure 4 and a correct case in Figure 5. In a PDG, each node aggregates one or several consecutive solution steps into a single progress unit with a clear subgoal (`goal`) and a verifiable intermediate product (`artifact`). Directed edges encode `dependency_ids`, i.e., which upstream artifacts are required premises for producing the current node's artifact.

Node `status` is determined by the dual-layer verification protocol described in Section 3.1. `is_input_valid` checks whether the

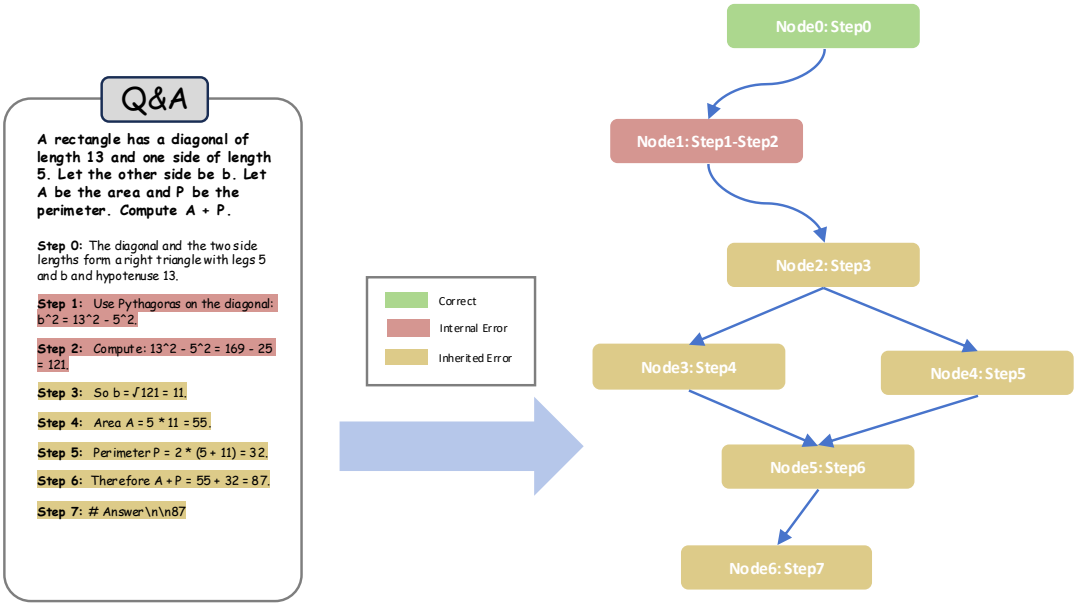


Figure 4: The corresponding visualization of the wrong case.

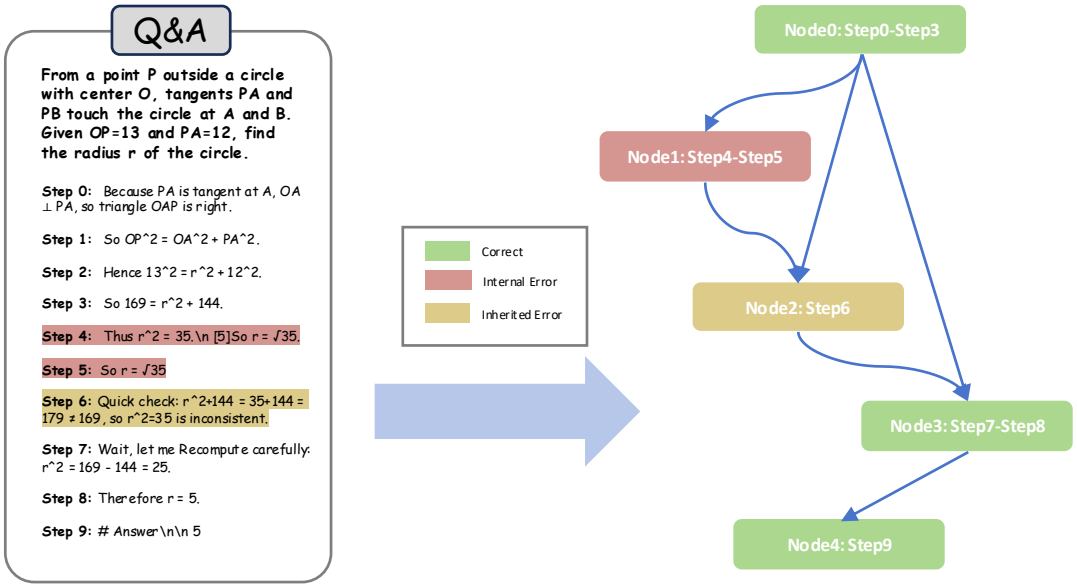


Figure 5: The corresponding visualization of the correct case.

required dependency artifacts are correct, while `is_logic_valid` checks whether the current node's internal transformation is valid conditional on its inputs. A node is labeled (a) **Correct Progress (CP)** when both checks pass, (b) **Internal Error (IE)** when inputs are valid but the node introduces a mistake, (c) **Inherited Error (HE)** when the node is locally correct but relies on at least one incorrect upstream artifact, and (d) **Compound Error (CE)** when both inherited and internal mistakes occur simultaneously. For HE/CE nodes, we additionally record `source_error_id` to trace the originating internal-error node that caused the downstream error propagation.

In the wrong-case example (Figure 4), an IE node miscomputes an intermediate quantity (a self-contained arithmetic error), which then contaminates all subsequent computations. Downstream nodes are labeled HE because they remain locally consistent given the wrong premise, illustrating how a single root-cause mistake can propagate along dependency edges and lead to an incorrect final answer. In the correct-case example (Figure 5), an early IE node introduces an incorrect interme-

diate value, and a subsequent HE node performs a consistency check that exposes the mismatch. The trajectory then contains a recovery (correction) node. This recovery node is a CP node that appears downstream of earlier non-CP nodes and successfully re-derives the correct quantity by revisiting a valid dependency (matching our definition of v_{rec} in Section 3.2). This pattern is important for RLPgR, as we explicitly encourage such recovery behaviors via a positive scaling on recovery nodes in correct rollouts (Eq. (2)).

Progress annotator prompt for progress generation task.

```
# Role Definition

You are an expert Mathematical Logic Auditor. Your task is to dissect a raw step-by-step
solution (Rollout) generated by an AI model into a structured "Progress Dependency Graph" and
perform a rigorous "Causal Error Analysis".

# Core Definitions

    1. **Progress Node**: An atomic logical unit that advances the problem-solving state. It
    may group multiple consecutive raw steps if they serve the same immediate sub-goal.
    2. **Artifact**: The explicit, verifiable result produced by a Progress (e.g., "Equation:
    2x+y=10", "Value: x=5"). This is the interface for future dependencies.
    3. **Dependency**: The IDs of *previous* Progress nodes that provided the necessary inputs
    for the current node.

# Auditing Protocol (Step-by-Step)

## Phase 1: Segmentation
    Group consecutive steps into a single Progress.
    - **Constraint**: Ensure each Progress produces a clear Artifact. Do not split atomic
    calculations.
## Phase 2: Dependency & Artifact Extraction
    For each Progress:
    1. **Goal**: Identify the intent (e.g., "Define variables", "Solve equation").
    2. **Artifact**: Extract the key mathematical result.
    3. **Deps**: List IDs of previous nodes that provided *necessary inputs*.
## Phase 3: Dual-Layer Verification (CRITICAL)
    You must determine the validity of each Progress by performing a **Counterfactual Logic
    Check**:
    1. **Check Input (`is_input_valid`)**: Are the Artifacts referenced from dependency nodes
    mathematically correct?
    - *Note*: If the dependency is the Question itself, input is Valid.
    2. **Check Logic (`is_logic_valid`)**: **Assuming the inputs are accepted as is (even if
    they are mathematically wrong)**, is the reasoning/calculation *within* this current node
    correct?
   - *Example*: Previous node output `x=5` (Wrong, true is 3). Current node calculates `x+1=6`.
    - Here, `is_input_valid` = False.
    - But `is_logic_valid` = True (because 5+1 is indeed 6).
## Phase 4: Status Assignment
    Assign exactly one status based on the verification above:
    1. **`Correct Progress`**: `is_input_valid=True` AND `is_logic_valid=True`. (And it's an
    intermediate step).
    2. **`Internal Error`** (The Root Cause): `is_input_valid=True` AND `is_logic_valid=False`.
    The mistake happens *here* (calculation error, hallucination, wrong formula).
    3. **`Inherited Error`** (The Victim): `is_input_valid=False` AND `is_logic_valid=True`.
    The reasoning is correct relative to the wrong input.
   - *Requirement*: You MUST specify `source_error_id` (the ID of the ancestor Internal Error).
    4. **`Compound Error`**: `is_logic_valid=False` AND `is_input_valid=False`. The step both
    inherits an wrong input and introduces a new logical mistake: even when treating the given
    (incorrect) input as premises, the reasoning in this step is invalid.
   - *Requirement*: You MUST specify `source_error_id` (the ID of the ancestor Internal Error).
## Phase 5: Check Final Correctness
    **Final Correctness**: Determine the overall correctness of the result based on the given
    golden answer.

# Output Format

    Output ONLY a valid JSON object describing the graph.
```

LLM-as-judge prompt for evaluation task.

```
### You are an expert evaluator.

You will be given:
- A math problem
- A step-by-step assistant solution
- Two JSON objects of Progress Dependency Graph that contains:
  (1) a list of progress nodes, and
  (2) a boolean field indicating whether the final answer is correct (e.g.,
  "final_correctness": true/false).
  Each progress node includes:
  (a) a status label, one of:
      - "Correct Progress" (internally correct and not affected by earlier errors)
      - "Internal Error" (contains an internal mistake)
      - "Inherited Error" (no internal mistake, but depends on earlier incorrect progress)
      - "Compound Error" (both internal and inherited errors apply)
  (b) the node's goal and produced outcome/artifact
  (c) the step IDs included in this node
  (d) the progress IDs this node depends on
- The node and global criteria

Task:
Use the node criteria for node-level evaluation, use the global criteria for global-level
evaluation, and additionally verify whether the "final_correctness" field is labeled correctly.
Based on these criteria and the accuracy of the final result judgment, determine which of the
two Progress Dependency Graphs is more consistent and better.

## NODE CRITERIA

1) Status accuracy:
   The node's status label must be consistent with:
   - the actual correctness of the solution steps that belong to this node, and
   - the correctness of any dependency progress (if applicable).
   Requirements:
   - "Correct Progress" must NOT contain any incorrect step.
   - "Internal Error" must contain at least one internal mistake within this node's steps.
   - "Inherited Error" must contain no internal mistake in this node's steps, but must depend
   on at least one earlier incorrect progress.
   - "Compound Error" must have both an internal mistake and an inherited dependency error.

2) Step set coherence and completeness:
   All steps included in this node must serve the SAME node goal and produce the stated
   outcome/artifact.
   The node must NOT include irrelevant extra steps, and must NOT miss any necessary steps for
   achieving its stated goal/outcome.

If either condition (1) or (2) is not satisfied, the node is invalid.
If the information provided is insufficient to confidently verify (1) or (2), the node is
invalid.
If the JSON object is missing or only contains simple numbers or dict, the node is invalid.

## GLOBAL CRITERIA

Choose the Progress Dependency Graph for THIS Prompt+Answer by these criteria:

1) Check segmentation validity: each node should correspond to a clear single milestone; if a
node mixes unrelated subgoals or merely paraphrases without real advancement, the segmentation
is invalid.
2) boundary clarity: each node should form a closed unit with explicit premises and an
explicit, verifiable output (intermediate claim/equation/value/decision); otherwise treat it
as invalid or incomplete.
3) overall coverage consistency: the progress chain should have no missing key milestones
(gaps) and no duplicated coverage of the same key milestone across multiple nodes (overlaps).
4) dependency correctness and necessity: dependency edges must connect only truly used key
prerequisites; redundant/missing edges, inverted directions, or cycles indicate an incorrect
dependency graph.
```

```
5) attribution consistency: Inherited Error must trace back to a wrong prerequisite from
dependencies; Internal Error must trace back to an inherent error within the node; if both
exist, do not use inherited to hide internal issues.
6) misuse of error propagation: dependency errors must not be used as a blanket excuse to skip
validating the node's own legality (conditions, transformations, case splits, etc.);
independent node errors must be explicitly identified.
7) for pseudo-progress / reward hacking: fancy restatement, irrelevant derivations, or
plausible-looking intermediate results that are not logically connected to the milestone; if
present, mark as incorrect or invalid progress.

### INPUT (CONTEXT FOR THIS COMPARISON)
Question:
$Question

Golden Answer:
$GoldenAnswer

Step-by-step Solution:
$Solution

ProgressNode1:
$ProgressNodes1

ProgressNode2:
$ProgressNodes2

### OUTPUT FORMAT (STRICT)
Output ONLY one uppercase letter, do not output any other characters:
- A = Template A is better
- B = Template B is better
- C = Tie
```

LLM-as-judge prompt for benchmark scoring.

```
# Role Definition
You are a strict evaluator. Your task is to grade candidate "Progress Dependency Graph" (PDG)
outputs for the Mathematical Logic Auditor task.
All candidate identifiers are anonymized (C01, C02, ...). Do not guess their origin.

# Scoring Dimensions
For EACH candidate, output TWO integer scores in [1, 5]:
status_score
- Judge whether each node's `status` is correctly assigned:
  Correct Progress / Internal Error / Inherited Error / Compound Error
- Also judge whether `final_correctness` is correct (must agree with the Golden Answer).

# Core Definitions (for judging)
## Counterfactual validity
- `is_input_valid`: Are the artifacts coming from the referenced dependency nodes
mathematically correct?
- `is_logic_valid`: Accept the given inputs as premises (even if wrong). Under that assumption,
is the reasoning/calculation inside the node correct?

## Status mapping
- Correct Progress: is_input_valid = True  AND is_logic_valid = True
- Internal Error:   is_input_valid = True  AND is_logic_valid = False
- Inherited Error:  is_input_valid = False AND is_logic_valid = True
- Compound Error: is_input_valid = False AND is_logic_valid = False

# Evaluation Protocol (Step-by-Step)
## Phase 1: Read Context
Use the Math Question, the raw Model Rollout, and the Golden Answer.

## Phase 2: Score status_score (1-5)
Check whether each node's `status` matches what the rollout implies under the counterfactual
rules above.
Check whether `final_correctness` matches the Golden Answer.

## Phase 3: Apply the rubric (separately for each axis)
- 5: essentially perfect
- 4: minor issues only
- 3: noticeable issues but partially correct
- 2: mostly wrong
- 1: empty / null / unparseable / error / clearly wrong

# Robustness Rule
If a candidate is empty, null, not JSON-like, or indicates an error, set:
status_score = 1.

# Task Context
Math Question:
{question}

Model Rollout (raw steps):
{rollout}

Golden Answer:
{golden}

Candidates:
{cand_blob}

# Output Format
Output ONLY a valid JSON object. No markdown. No extra text.

Schema:
{{
  "candidates": {{
    "C01": {{"status_score": 1, "notes": "short reason"}}
  }}
}}
```

D.1 A Wrong Case


```
{
    "progress_nodes": [
      {
        "progress_id": 0,
        "step_ids": [
          0
        ],
        "goal": "Translate the rectangle-diagonal setup into a right-triangle relationship.",
        "artifact": "The sides 5 and b with diagonal 13 satisfy 5^2 + b^2 = 13^2 (right triangle with hypotenuse 13).",
        "dependency_ids": [],
        "verification": {
          "rationale": "A rectangle's diagonal forms a right triangle with its side lengths as legs, so 5, b, and 13 satisfy
          the Pythagorean relationship.",
          "is_input_valid": true,
          "is_logic_valid": true
        },
        "status": "Correct Progress",
        "source_error_id": null
      },
      {
        "progress_id": 1,
        "step_ids": [
          1,
          2
        ],
        "goal": "Set up b^2 using Pythagoras and compute its value.",
        "artifact": "Claim: b^2 = 13^2 - 5^2 = 121.",
        "dependency_ids": [
          0
        ],
        "verification": {
          "rationale": "From 5^2+b^2=13^2, b^2=13^2-5^2 is correct, but 169-25 equals 144, not 121.",
          "is_input_valid": true,
          "is_logic_valid": false
        },
        "status": "Internal Error",
        "source_error_id": null
      },
      {
        "progress_id": 2,
        "step_ids": [
          3
        ],
        "goal": "Compute b from the claimed b^2.",
        "artifact": "From b^2=121, b=\\sqrt{121}=11.",
        "dependency_ids": [
          1
        ],
        "verification": {
          "rationale": "Assuming b^2=121, taking \\sqrt{121}=11 is correct.",
          "is_input_valid": false,
          "is_logic_valid": true
        },
        "status": "Inherited Error",
        "source_error_id": [
          1
        ]
      },
      {
        "progress_id": 3,
        "step_ids": [
          4
        ],
        "goal": "Compute the area A.",
        "artifact": "Using b=11, A=5\\times 11=55.",
        "dependency_ids": [
          2
        ],
        "verification": {
          "rationale": "Given b=11, computing 5\\times 11=55 is internally consistent.",
          "is_input_valid": false,
          "is_logic_valid": true
        },
        "status": "Inherited Error",
        "source_error_id": [
          1
        ]
      },
      {
        "progress_id": 4,
        "step_ids": [
          5
        ],
        "goal": "Compute the perimeter P.",
        "artifact": "Using b=11, P=2(5+11)=32.",
        "dependency_ids": [
          2
```

```
        ],
        "verification": {
          "rationale": "Given b=11, computing 2(5+11)=32 is internally consistent.",
          "is_input_valid": false,
          "is_logic_valid": true
        },
        "status": "Inherited Error",
        "source_error_id": [
          1
        ]
      },
      {
        "progress_id": 5,
        "step_ids": [
          6
        ],
        "goal": "Combine A and P to get A+P.",
        "artifact": "A+P=55+32=87 (under b=11).",
        "dependency_ids": [
          3,
          4
        ],
        "verification": {
          "rationale": "Assuming A=55 and P=32, adding them yields 87 correctly.",
          "is_input_valid": false,
          "is_logic_valid": true
        },
        "status": "Inherited Error",
        "source_error_id": [
          1
        ]
      },
      {
        "progress_id": 6,
        "step_ids": [
          7
        ],
        "goal": "State the final answer.",
        "artifact": "Final answer: 87.",
        "dependency_ids": [
          5
        ],
        "verification": {
          "rationale": "This repeats the previously computed value.",
          "is_input_valid": false,
          "is_logic_valid": true
        },
        "status": "Inherited Error",
        "source_error_id": [
          1
        ]
      }
    ],
    "final_correctness": false
  }
```

D.2 A Correct Case


```
{
    "progress_nodes": [
      {
        "progress_id": 0,
        "step_ids": [0, 1, 2, 3],
        "goal": "Use the tangent-radius perpendicularity to set up a right-triangle equation for r.",
        "artifact": "Right triangle OAP gives OP^2 = r^2 + PA^2, so 169 = r^2 + 144.",
        "dependency_ids": [],
        "verification": {
          "rationale": "Radius OA is perpendicular to tangent PA, so triangle OAP is right. Applying Pythagoras with OP=13 and
          PA=12 correctly yields 169=r^2+144.",
          "is_input_valid": true,
          "is_logic_valid": true
        },
        "status": "Correct Progress",
        "source_error_id": null
      },
      {
        "progress_id": 1,
        "step_ids": [4, 5],
        "goal": "Solve for r from 169 = r^2 + 144.",
        "artifact": "Claim: r^2 = 35 and r = \\sqrt{35}.",
        "dependency_ids": [0],
        "verification": {
          "rationale": "From 169=r^2+144, subtracting 144 should give r^2=25, not 35. This is an arithmetic error inside the
          node.",
          "is_input_valid": true,
          "is_logic_valid": false
        },
        "status": "Internal Error",
        "source_error_id": null
      },
      {
        "progress_id": 2,
        "step_ids": [6],
        "goal": "Check the computed r^2 against the defining equation to test consistency.",
        "artifact": "Using r^2=35 implies r^2+144=179 \\neq 169, so r^2=35 is inconsistent with 169=r^2+144.",
        "dependency_ids": [0, 1],
        "verification": {
          "rationale": "Although r^2=35 is wrong (bad input from node 1), the consistency check is computed correctly:
          35+144=179, which does not equal 169.",
          "is_input_valid": false,
          "is_logic_valid": true
        },
        "status": "Inherited Error",
        "source_error_id": [1]
      },
      {
        "progress_id": 3,
        "step_ids": [7, 8],
        "goal": "Correct the arithmetic and compute the true radius.",
        "artifact": "From 169=r^2+144, r^2=25 and r=5.",
        "dependency_ids": [0, 2],
        "verification": {
          "rationale": "This recomputes directly from the correct equation 169=r^2+144 and performs 169-144=25, hence r=5.",
          "is_input_valid": true,
          "is_logic_valid": true
        },
        "status": "Correct Progress",
        "source_error_id": null
      },
      {
        "progress_id": 4,
        "step_ids": [9],
        "goal": "State the final answer.",
        "artifact": "Final answer: 5.",
        "dependency_ids": [3],
        "verification": {
          "rationale": "It reports r=5 obtained in the corrected computation.",
          "is_input_valid": true,
          "is_logic_valid": true
        },
        "status": "Correct Progress",
        "source_error_id": null
      }
    ],
    "final_correctness": true
  }
```