CREDIT WHERE IT MATTERS: DEPENDENCY-AWARE POLICY OPTIMIZATION FOR TERMINAL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Terminal-using agents benefit from reinforcement learning (RL) in coding, debugging, and other multi-step terminal tasks. In these tasks, later commands often depend on information or intermediate results produced by earlier commands. However, existing trajectory-level and step-level credit assignment methods do not explicitly trace the read-write dependencies through which commands affect the final outcome. Consequently, training signals could still be assigned to irrelevant operations, weakening learning from relevant steps. In this paper, we propose Dependency-Aware Group Policy Optimization (DepGPO), which uses execution dependencies between commands to guide credit assignment for terminal agents. Specifically, we construct a command dependency graph from execution traces and trace backward from the resources inspected by the task verifier. We then assign credit to relevant writes and their supporting reads along these paths, and use it to redistribute trajectory advantages across steps. Extensive comparative experiments and ablation studies demonstrate that DepGPO improves task performance and training stability on complex terminal tasks.

1 INTRODUCTION

Terminal-using agents (Anthropic, 2025; OpenAI, 2025; Google, 2025; Wang et al., 2025) have become a major application of language models for coding, debugging, and other multi-step terminal tasks (Yang et al., 2024; Merrill et al., 2026; Ivison et al., 2026; Gandhi et al., 2026). These tasks involve multi-turn interactions with a persistent environment, where later commands use information and intermediate results produced by earlier ones. Recent work improves these agents through reinforcement learning (RL), with outcome rewards determined by task verifiers that check whether the resulting environment satisfies task requirements (Shen et al., 2026; Ivison et al., 2026; Yao et al., 2026). A common approach uses group-based algorithms such as GRPO (Shao et al., 2024) and DAPO (Yu et al., 2025), which compare outcome rewards across sampled trajectories. In their outcome-based formulations, a trajectory-level advantage is shared across all generated steps. However, commands can play different roles: an inspection may guide a later repair, whereas a write may be overwritten before its output is used. This mismatch calls for step-level credit assignment that reflects how individual commands contribute to task completion.

To refine this within-trajectory credit assignment, recent work has developed several step-level methods. GiGPO compares actions taken from the same environment state (Feng et al., 2025), and HGPO further accounts for historical context when forming groups (He et al., 2026). GAGPO uses grouped state values to propagate outcome supervision through time (Zhu et al., 2026). Graph-based methods, including GEPO (Yuan et al., 2025), GraphGPO (Cheng et al., 2026), and G2PO (Wang et al., 2026b), construct state-transition graphs and derive credit from structural importance, goal progress, or state values. HCAPO provides another route by using a language model to reassess intermediate actions in hindsight (Tan et al., 2026). Although these methods enable finer-grained credit assignment, they do not explicitly trace how information and intermediate results flow between commands. In fact, terminal execution traces can expose these read-write dependencies, linking individual commands to the resources checked by the verifier and providing direct evidence for credit assignment.

To examine the limitations of existing credit assignment and assess whether execution dependencies can support finer-grained attribution, we conduct a pilot study (Figure 1). With four rollouts per task from a model trained by supervised fine-tuning, 61% of 1,376 task groups yield identi-

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

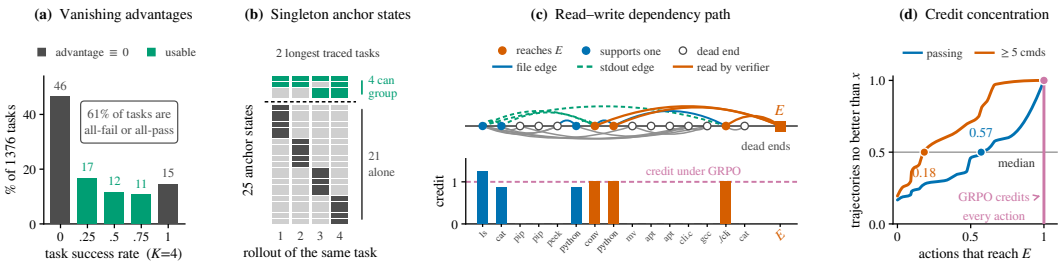

Figure 1: **Pilot analysis of credit assignment in terminal tasks.** **(a)** With $K = 4$, an SFT policy yields zero group-relative advantages on 61% of tasks. **(b)** Sparse anchor-state recurrence in our terminal implementation of GiGPO. **(c)** Example command dependencies and raw credits; E denotes the resources read by the verifier. **(d)** Cumulative distributions of the fraction of commands connected to E in passing Terminal-Bench 2.0 reference trajectories ($n = 51$) and the subset with at least five commands ($n = 18$). Pink guides illustrate GRPO's uniform step weights.

cal binary rewards and zero group-relative advantages (Figure 1(a)). DAPO can filter these groups through dynamic sampling, but allocating the remaining nonzero advantages across commands remains unresolved. For GiGPO, exact matching in our terminal implementation yields only four recurring anchor states out of 25 in the two traced tasks with the most anchors, limiting step-level comparisons (Figure 1(b)). We further analyze Terminal-Bench 2.0 reference solutions and denote the set of files and other resources read by the task verifier by E. Execution traces reveal command dependency paths leading to this resource set (Figure 1(c)). Across 51 passing reference trajectories, the median fraction of commands on these paths is 0.57, falling to 0.18 for the 18 trajectories with at least five commands (Figure 1(d)). Together, these findings highlight the need to distinguish command contributions more carefully and show that execution dependencies provide observable evidence for allocating outcome-based learning signals to relevant steps.

Building on these pilot findings, we develop **Dep**endency-Aware **G**roup **P**olicy **O**ptimization (DepGPO), which redistributes trajectory advantages using command dependencies connected to task verification. Specifically, we construct a directed command graph from recorded resource reads and writes and subsequent uses of command outputs, then trace backward from the verifier's read set E. For file writes, credit is the fraction of written lines that reach E. A line reaches E if it remains in a file read by the verifier, or if a later command on a dependency path to E reads it before it is overwritten. Reads then accumulate credit from the relevant writes they support, with each contribution decaying with shortest-path distance in the dependency graph. We then aggregate command credits within each agent step and normalize them into nonnegative, unit-mean step factors. Multiplying these factors by the original trajectory advantage produces per-step advantages for the clipped policy objective. In this design, group-relative outcomes determine the sign of the learning signal, while execution dependencies determine its allocation across steps.

Our main contributions are as follows:

- We propose Dependency-Aware Group Policy Optimization (DepGPO), which uses execution dependencies between commands to redistribute group-relative advantages through distinct credit assignment rules for relevant writes and their supporting reads.
- We analyze training dynamics through controlled experiments and show that DepGPO delays training degradation and improves the stability of reinforcement learning for terminal agents.
- We conduct experiments with Qwen3.5-9B and Qwen3.6-27B using both SETA and TMAX training data, demonstrating consistent improvements in task performance and training stability.

2 Related Work

Reinforcement Learning with Outcome Supervision. Outcome-supervised RL uses response- or trajectory-level rewards as the training signal (Le et al., 2022; Yu et al., 2024). PPO estimates advantages with a learned value model (Schulman et al., 2017), while GRPO uses group-relative rewards without a separate critic (Shao et al., 2024). Such methods have been applied to terminal agents (Yang et al., 2026; Yao et al., 2026), including Endless Terminals with PPO (Gandhi et al.,

2026) and SETA with GRPO, task synthesis, and environment evolution (Shen et al., 2026). However, sparse terminal rewards provide limited guidance for intermediate commands, and continued policy updates can become unstable (Yu et al., 2025; Liang et al., 2026; Jin et al., 2026).

Training Stability. Methods for stabilizing RL maintain exploration, constrain policy updates, or reduce rollout–training mismatch. DAPO combines asymmetric clipping with dynamic sampling (Yu et al., 2025), while DPPO directly constrains policy divergence (Qi et al., 2026). For terminal agents, TMAX further combines DPPO with an FP32 language-model head and larger rollout groups to reduce numerical mismatch and training collapse (Ivison et al., 2026). These techniques improve optimization stability but do not identify which intermediate commands contribute to the outcome.

Process Feedback and Step-Level Credit Assignment. Fine-grained supervision evaluates intermediate behavior through process rewards or step-level comparisons (Zhang, 2026; Dihan & Khan, 2026; Han et al., 2026). Process reward models can use human step labels (Lightman et al., 2024), sampled continuations (Wang et al., 2024), or outcome-derived implicit rewards (Cui et al., 2025); AgentPRM extends process evaluation to multi-turn agents (Xi et al., 2026). In terminal tasks, River uses rule-based behavioral penalties (Yao et al., 2026), while T1 combines verifier-based rewards with value-model credit assignment (Yang et al., 2026). GiGPO instead compares actions from shared environment states (Feng et al., 2025), while GraphGPO (Cheng et al., 2026) and G2PO (Wang et al., 2026b) derive step credit from state-transition graphs. These approaches provide finer feedback, but learned scores require supervision, predefined penalties cover behaviors, and state-based methods depend on representations that may omit file or process changes.

Execution Traces for Terminal Agents. Execution traces expose agent actions and their effects on the environment and have been studied for observability, failure attribution, and execution provenance (AlSayyad et al., 2026; Zhang et al., 2026; Wang et al., 2026a). CodeTracer reconstructs hierarchical state transitions and localizes failure onset for diagnostic replay (Li et al., 2026), while Shepherd supports branching from recorded intermediate states and computes advantages from additional rollouts (Yu et al., 2026). DepGPO instead uses the execution trace itself as credit-assignment evidence: it follows read–write dependencies from verifier-inspected resources to identify relevant writes and their supporting reads, then redistributes group-relative advantages accordingly.

3 METHOD

3.1 PROBLEM SETUP AND OVERVIEW

Let π_θ denote the language model policy of a terminal agent, where θ represents the model parameters. Given a task x, the policy generates a response containing terminal commands, whose execution yields an environment observation. A response and its resulting observation form one interaction step. Together, the T_i steps of the i-th task attempt form a trajectory τ_i. Since a step may contain several commands or none, we use commands as the units of dependency analysis, while policy optimization operates on the tokens generated in each response.

At the end of each trajectory, a task verifier evaluates the resulting environment and returns a reward R_i. We collect K trajectories for the same task using a fixed sampling policy $\pi_{\theta_{\text{old}}}$. Following GRPO (Shao et al., 2024), we compute the group-relative advantage

$$\widehat{A}_i = \frac{R_i - \mu_x}{\sigma_x + \epsilon}, \tag{1}$$

where μ_x and σ_x are the mean and population standard deviation of the K rewards, and $\epsilon > 0$ prevents division by zero. This advantage is positive when the reward exceeds the group mean and negative when it falls below the mean. Under outcome supervision, GRPO applies the same trajectory advantage to every token generated during that attempt.

DepGPO redistributes this advantage using execution dependencies within each trajectory. We build a graph whose nodes are commands from that trajectory, then compute credit for relevant writes and their supporting reads. Credits of commands executed in the same step are added and normalized into step factors. These factors determine per-step advantages, which are assigned to the tokens generated in their respective steps. The following sections describe graph construction, credit assignment, and policy optimization.

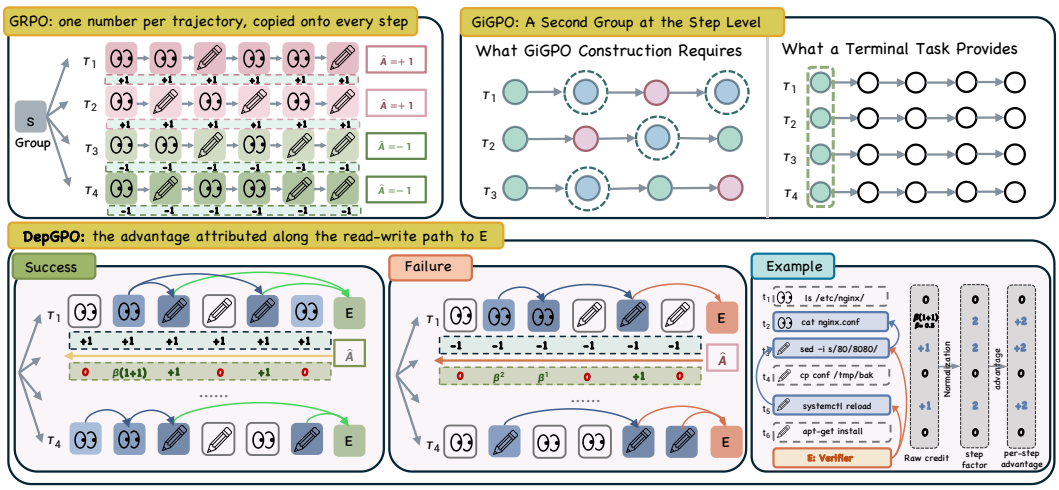

Figure 2: Overview of DepGPO. Unlike GRPO's uniform trajectory-level assignment and GiGPO's generic step-level grouping, DepGPO traces read–write dependencies to the verifier-inspected resources E and redistributes the group-relative advantage according to dependency-derived step factors. The example illustrates the transformation from raw credit to the final per-step advantage.

3.2 Command Dependency Graph

We extract file accesses, port and process events, and standard output from command execution traces. For file writes, we identify written line ranges from differences in file contents before and after execution, when this information is available. In contrast, file-read records identify the accessed file paths but do not specify which lines are used during computation. Accordingly, we track writes at line level and record reads at file level. Based on these records, commands with at least one recorded write are classified as write commands, including those that also read; all remaining commands are classified as read commands. The two categories are mutually exclusive, and each command receives credit only under the rule for its assigned category.

For each trajectory, we construct a directed command dependency graph G containing both agent commands and the verifier command. Edges capture dependencies through file contents and standard output and may connect commands across interaction steps.

File edges. File edges record the use of content that earlier commands wrote to files. A file edge $c_u \rightarrow c_v$ means that command c_v reads a file containing lines written by the earlier command c_u that have not yet been overwritten. We process commands in execution order and record the latest writer of each tracked line. For the current command, we first connect the earlier writers of these remaining lines to the current command based on its file reads. We then update the records for lines written by the current command. An overwrite changes only the records for the replaced lines. It does not create an edge by itself, because replacing existing content does not establish that the later command read that content. If the later command also reads the file before replacing its contents, that read creates the corresponding file edge.

Standard-output edges. Standard-output edges record the reuse of information printed by commands in the terminal. To construct these edges, we match typed values, such as file paths and ports, in a later command's text against earlier standard output. For each match, we add an edge from the command that first printed the value to the command that reuses it. During this process, we exclude values already supplied in the task description, since their reuse does not establish a dependency on an earlier command. Because both file and standard-output edges follow execution order, the resulting graph G is acyclic.

We also trace the verifier to identify the resources inspected during evaluation. We denote these resources by E, the verifier's read set. This set includes files, ports, and processes. File existence and metadata checks are included, while files written by the verifier itself are excluded from the file set. These resources provide the starting point for tracing the writes relevant to verification.

3.3 Dependency-Aware Credit Assignment

Write credit.

We assign file-write credit by the fraction of written lines that reach E. A line reaches E directly if its file is inspected by the verifier and the line has not been overwritten by a later agent command. Otherwise, it may contribute indirectly if a later write command reads the file before the line is overwritten and produces an output that reaches E, either directly or through further writes.

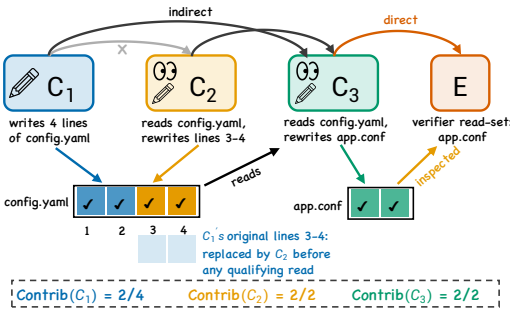

Figure 3: Illustration of write-credit assignment through direct and indirect dependencies.

We apply a conservative rule to indirect contributions: the consuming command must write to a file other than the input file and must not overwrite the input lines being counted. The first condition excludes same-file edits, which produce no output elsewhere; earlier lines can still receive direct credit if they remain in a verifier-inspected file. The second excludes lines overwritten by the consumer from downstream contribution.

Credit assignment proceeds backward from E, starting with writes whose contents remain in inspected resources and therefore receive direct credit. For each identified write, we trace its file reads to earlier written lines satisfying the indirect-contribution conditions, and recursively repeat this process for their writers until no additional writers are found. These relations form a separate reachability graph G_E, whose nodes are agent commands and an endpoint representing E, excluding the verifier command. Complete line-level rules are provided in Appendix A.3.

Let $\mathcal{W}_c$ be the set of tracked lines written by command c, identified by file and line position, and let $\mathcal{W}_c^E \subseteq \mathcal{W}_c$ contain those that contribute to E directly or indirectly. For $|\mathcal{W}_c| > 0$, the raw credit is

$$q(c) = \mathrm{Contrib}(c) = \frac{|\mathcal{W}_c^E|}{|\mathcal{W}_c|}, \tag{2}$$

where $|\cdot|$ denotes set cardinality. Each qualifying line contributes one count regardless of the number of downstream consumers or dependency length; downstream contribution scores are not multiplied along the chain. Thus, $\mathrm{Contrib}(c) \in [0, 1]$ measures the fraction of lines written by c that contribute to E, without distance decay.

Ports and processes lack a meaningful line-level representation and are therefore handled separately. A command represented only by such resources receives credit one if its opened port or created process is inspected by the verifier, and zero otherwise. Resource-specific cases, unavailable line ranges, and commands involving multiple resource types are detailed in the Appendix A.4.

Read credit.

Read commands can support relevant writes by revealing information used later. Because a read command produces no recorded file write, its support begins with a standard-output edge; once its output is reused, the path may continue through file or standard-output edges in G. We use these paths to identify the writes supported by each read command.

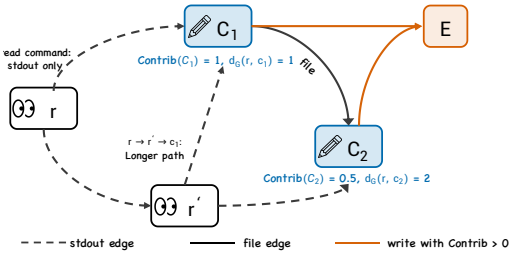

Figure 4: Illustration of read-credit assignment through dependency paths in G.

For a read command r, let $\mathcal{D}_G(r)$ denote the agent write commands reachable from r in G, and let $d_G(r, c)$ be the shortest-path length to a reachable write c. Its raw credit is

$$q(r) = \sum\nolimits_{c \in \mathcal{D}_G(r)} \beta^{d_G(r,c)}\, \mathrm{Contrib}(c), \tag{3}$$

where $\beta \in (0, 1]$ controls decay with dependency distance, measured in graph edges rather than interaction steps. If multiple paths connect r to the same write c, its contribution is counted once

using the shortest path. Contributions from distinct positive-credit writes are instead summed, while a read with no path to any positive-credit write receives zero.

Both read and write credits are nonnegative scores of relevance to verification rather than trajectory success. They determine where the learning signal is assigned, while the group-relative advantage determines whether responses are reinforced or penalized during policy optimization.

3.4 Advantage Redistribution and Policy Optimization

From operation credit to step factors. The preceding rules assign raw credit to traced write and read operations. We aggregate all credits within the same interaction step:

$$q_{i,t} = \sum_{c \in \mathcal{W}_{i,t}} q(c) + \sum_{r \in \mathcal{R}_{i,t}} q(r), \tag{4}$$

where $\mathcal{W}_{i,t}$ and $\mathcal{R}_{i,t}$ are the traced write and read operations in step t of trajectory i. Let $\mathcal{S}_i$ contain the steps with at least one traced operation. Step credits are compared only within the same trajectory, so the resulting weights reflect relative relevance rather than absolute credit scale.

We use the minimum step credit as a within-trajectory reference and normalize each excess above it by the mean excess:

$$m_i = \min_{s \in \mathcal{S}_i} q_{i,s}, \qquad f_{i,t} = \frac{q_{i,t} - m_i}{\frac{1}{|\mathcal{S}_i|} \sum_{s \in \mathcal{S}_i} (q_{i,s} - m_i)}, \quad t \in \mathcal{S}_i, \tag{5}$$

where m_i is the minimum credit and $|\mathcal{S}_i|$ is the number of compared steps. Thus, $f_{i,t}$ measures relative relevance within trajectory i: minimum-credit steps receive zero, larger credits receive proportionally larger factors, and the mean factor over $\mathcal{S}_i$ is one.

Steps without traced operations are excluded from this comparison and retain the default factor of one. If all compared credits are identical, the denominator in Eq. 5 is zero; if tracing yields no usable credit, no relative comparison is available. In either case, all step factors are set to one, recovering the original trajectory-level advantage assignment.

From step factors to per-step advantages. The step factors redistribute the trajectory-level advantage $\widehat{A}_i$ across interaction steps. Directly using $f_{i,t}\widehat{A}_i$, however, would give longer responses greater influence because the policy loss is averaged over generated tokens. We therefore normalize the factors by their token-weighted mean.

Let $L_{i,t}$ be the number of generated tokens from step t included in the loss, excluding environment observations. For a trajectory with T_i steps, we define the token-weighted mean factor as

$$Z_i = \frac{\sum_{t=1}^{T_i} L_{i,t} f_{i,t}}{\sum_{t=1}^{T_i} L_{i,t}}, \tag{6}$$

and use it to normalize the step factors when defining the final per-step advantage:

$$\widehat{A}_{i,t}^{\mathrm{Dep}} = \frac{f_{i,t}}{Z_i} \widehat{A}_i. \tag{7}$$

Because the same Z_i scales every step, this preserves their relative weights while making the token-weighted mean factor one. Thus, DepGPO redistributes the original trajectory advantage across tokens without changing its overall scale.

Each generated token in step t receives $\widehat{A}_{i,t}^{\mathrm{Dep}}$. Larger factors strengthen the learning signal, whereas a zero factor removes that step's direct contribution to the policy loss. Since all factors are nonnegative, the redistribution preserves the sign of $\widehat{A}_i$, and a zero advantage remains zero. If all steps contribute the same number of loss tokens, $Z_i = 1$ and $\widehat{A}_{i,t}^{\mathrm{Dep}} = f_{i,t}\widehat{A}_i$.

Policy optimization. DepGPO retains the GRPO policy optimization procedure and changes only how the group-relative advantage is distributed across interaction steps. Index the policy-generated tokens in a training minibatch by $n = 1, \ldots, N$. Let y_n denote token n and h_n its context, including

the task, preceding observations, and previously generated tokens. If token n belongs to step t of trajectory i, it receives $a_n = \widehat{A}^{\mathrm{Dep}}_{i,t}$. The corresponding probability ratio is

$$\rho_n(\theta) = \frac{\pi_\theta(y_n \mid h_n)}{\pi_{\theta_{\mathrm{old}}}(y_n \mid h_n)},$$

where $\pi_{\theta_{\mathrm{old}}}$ is the fixed rollout policy and π_θ is the policy being optimized.

Following the clipped surrogate update of the GRPO baseline, we define

$$\ell_{\mathrm{clip}}(\rho, a) = \min\{\rho a,\ \mathrm{clip}(\rho, 1-\epsilon_{\mathrm{lo}}, 1+\epsilon_{\mathrm{hi}})a\}\,, \tag{8}$$

where ϵ_{lo} and ϵ_{hi} specify the lower and upper clipping ranges. The clipping limits policy changes once the ratio moves beyond the allowed range in the direction favored by the advantage.

For negative advantages, the baseline further applies dual clipping, lower-bounding the surrogate objective by ηa with $\eta > 1$ to prevent excessively negative values from large probability ratios. Retaining this update rule, we optimize

$$\mathcal{L}_{\mathrm{Dep}}(\theta) = -\frac{1}{N}\sum\nolimits_{n=1}^{N} \begin{cases} \max\{\ell_{\mathrm{clip}}(\rho_n(\theta), a_n),\ \eta a_n\}\,, & a_n < 0, \\ \ell_{\mathrm{clip}}(\rho_n(\theta), a_n), & a_n \geq 0. \end{cases} \tag{9}$$

The minus sign converts surrogate-objective maximization into loss minimization, and $1/N$ averages over generated tokens. Environment observations remain in the context but contribute no loss terms.

During each update, the dependency-derived factors and per-step advantages are held fixed. Positive advantages reinforce sampled tokens and negative advantages penalize them, with dependency factors controlling the signal magnitude across steps. Because these factors are nonnegative, they change the magnitude but not the sign of the original GRPO signal. The updated policy then collects new trajectories, after which group-relative advantages and dependency credits are recomputed.

Thus, DepGPO preserves the GRPO objective and clipping configuration while replacing uniform trajectory-level advantage assignment with dependency-aware step-level redistribution. Token mapping and numerical fallback details are provided in Appendix A.5.

4 Experiments

4.1 Experimental Setup

Models and datasets. We evaluate DepGPO with Qwen3.5-9B (Qwen Team, 2026a) and Qwen3.6-27B (Qwen Team, 2026b), using SETA (Shen et al., 2026) and TMAX (Ivison et al., 2026) separately, for four model–data configurations. We evaluate the resulting policies on Terminal-Bench 2.0 and 2.1 (Merrill et al., 2026) under the same protocol within each configuration.

Training pipeline and task selection. For each configuration, we first perform SFT on the corresponding terminal trajectories and use the resulting checkpoint to select tasks for RL. Specifically, we sample four attempts per candidate task and retain tasks with both successful and failed outcomes, excluding all-pass and all-fail groups whose group-relative advantages are zero under binary rewards. The retained pool is then split into RL training and validation sets and fixed across all methods within that configuration. Thus, methods are compared from the same SFT checkpoint and on the same selected task distribution.

Baselines. We compare DepGPO with GRPO (Shao et al., 2024), DAPO (Yu et al., 2025), and DPPO (Qi et al., 2026), together with GiGPO (Feng et al., 2025) and GraphGPO (Cheng et al., 2026) for step-level credit assignment. For the controlled comparison with GRPO, DepGPO keeps the initial checkpoint, selected task pool, policy objective, and optimization settings fixed, modifying only the redistribution of group-relative advantages across steps. Other baselines retain their respective method-specific components and optimization procedures.

Training and evaluation. For GRPO and DepGPO, each training batch contains four tasks with $K = 8$ trajectories per task, giving 32 trajectories in total. We use a learning rate of 10^{-6}, limit each trajectory to 10 interaction steps for SETA and 56 for TMAX and 24,576 generated response tokens, and set the DepGPO dependency-decay coefficient to $\beta = 0.5$. During evaluation, we allow up to

Table 1: Task performance on Terminal-Bench 2.0 and 2.1. Results are reported as pass@1 (%), mean ± SD over three evaluation rounds; blue shading and underlining denote the best and second-best results within each setting, respectively.

Method	Qwen3.5-9B				Qwen3.6-27B			
	SETA		TMAX		SETA		TMAX	
	TB2.0	TB2.1	TB2.0	TB2.1	TB2.0	TB2.1	TB2.0	TB2.1
SFT	15.28±0.81	16.48±1.35	14.01±2.38	10.56±2.82	25.92±0.72	26.37±1.83	23.45±2.97	24.27±0.78
GRPO	11.09±0.26	11.01±0.67	14.91±2.26	17.83±0.57	25.09±1.11	23.37±0.67	23.82±1.95	24.94±0.45
DAPO	13.11±0.13	13.26±0.67	15.13±1.32	13.86±2.36	27.34±2.16	28.16±1.11	26.74±2.34	25.09±0.13
DPPO	15.36±0.91	15.81±0.47	17.23±1.28	18.95±2.36	28.24±1.53	28.01±0.34	26.07±1.35	26.67±1.94
GiGPO	11.24±0.39	11.39±1.04	12.96±2.38	11.31±2.63	16.55±0.94	16.70±2.70	15.96±1.70	18.65±2.55
GraphGPO	10.79±0.22	10.79±0.39	10.56±0.22	11.46±1.03	17.00±0.91	16.63±0.22	18.20±0.39	16.85±1.70
DepGPO	**25.24**±1.01	**24.64**±2.36	**20.45**±0.98	**22.17**±1.80	**37.75**±1.25	**37.90**±3.97	**35.06**±1.35	**36.70**±0.13

56 interaction steps and estimate pass@1 from five independent attempts per task. We repeat the full evaluation three times and report mean ± SD of the overall pass@1. Complete SFT configurations, task-selection counts and splits and method-specific settings are provided in Appendix B.

4.2 Main Results

Overall performance. Table 1 shows that DepGPO achieves the best pass@1 across all eight settings. Compared with the strongest competing RL baseline in each setting, the gains range from 3.22 to 10.03 percentage points; relative to GRPO, they range from 4.34 to 14.53 points. These consistent improvements suggest that explicitly tracing read–write dependencies provides information that is not captured by trajectory-level or state-based credit alone, allowing the same outcome signal to be concentrated on commands more closely connected to task verification.

Scaling across models and training data. The gains persist for both Qwen3.5-9B and Qwen3.6-27B and with either SETA or TMAX. Notably, the 27B model shows large improvements on both training sources, reaching 37.75/37.90 on SETA and 35.06/36.70 on TMAX. One plausible explanation is that the larger model can make better use of the finer-grained learning signal provided by dependency-aware redistribution, while the smaller model is more sensitive to the underlying training distribution. Indeed, the 9B gain over the strongest baseline is substantially larger on SETA than on TMAX, suggesting that model capacity and task distribution jointly affect how much benefit can be extracted from step-level credit.

4.3 Training Stability

Reward dynamics. Figure 5(a) shows that GRPO improves initially but degrades with continued training, while GiGPO deteriorates earlier and remains at a low reward level. In contrast, DepGPO improves overall and maintains higher rewards in the late training window despite fluctuations. This difference may reflect their credit-assignment mechanisms: GRPO applies the same group-relative advantage to every step, including commands weakly related to the outcome, whereas GiGPO relies on matching states whose limited recurrence in terminal tasks restricts step-level grouping.

Optimization behavior. The entropy and gradient statistics provide further evidence of this divergence. GiGPO's policy entropy drops sharply after a brief increase, while GRPO's rises substantially later in training; DepGPO remains comparatively stable (Figure 5(b)). Both baselines also approach near-zero gradient norms in later stages, whereas DepGPO retains nonzero gradients (Figure 5(c)). Together with the reward curves, these patterns indicate more stable optimization under DepGPO in the runs shown.

Credit redistribution. Figure 5(d) reports the standard deviation of DepGPO's step factors, whose mean is fixed at one by normalization. The standard deviation generally increases during training, indicating increasingly differentiated weights across interaction steps while preserving their average scale. This pattern suggests that DepGPO becomes increasingly selective in where it allocates the learning signal as training progresses.

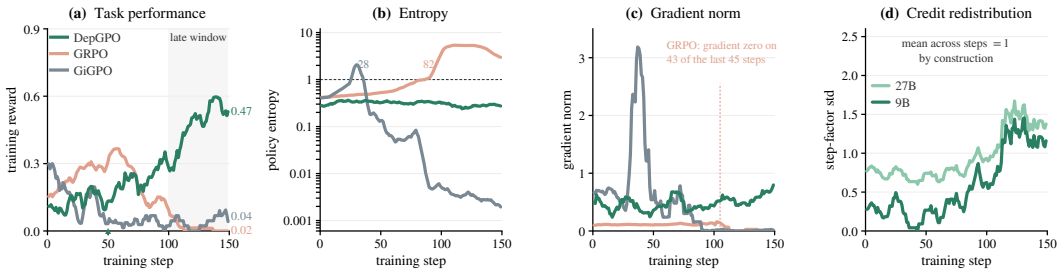

Figure 5: Panels (a)–(c) compare training reward, policy entropy, and gradient norm of DepGPO, GRPO, and GiGPO on Qwen3.5-9B trained with SETA, with shading marking the late training window and entropy plotted on a logarithmic scale. Panel (d) further shows the standard deviation of DepGPO's step factors across interaction steps, whose mean is fixed at one by normalization.

4.4 Ablation Study

We ablate Qwen3.5-9B trained on SETA and evaluate on Terminal-Bench 2.0 and 2.1. **Binary write contribution** sets positive $\mathrm{Contrib}(c)$ values to one and leaves zeros unchanged, preserving verifier filtering while removing graded credit. **Without verifier filtering** assigns credit one to all write commands, including those that do not reach E. For both variants, read credit is recomputed using the original read-credit rule.

Table 2: Ablation study of Qwen3.5-9B trained on SETA. Pass@1 (%), mean ± SD. Red arrows denote performance drops relative to Full DepGPO.

Variant	TB2.0	TB2.1
GRPO	11.09±0.26	11.01±0.67
Binary write contribution	23.07±0.85 ↓2.17	22.55±0.91 ↓2.09
Without verifier filtering	18.65±1.03 ↓6.59	17.75±1.19 ↓6.89
Without supporting-read credit	21.42±0.94 ↓3.82	20.15±1.13 ↓4.49
Direct writes only	23.45±0.91 ↓1.79	23.00±1.04 ↓1.64
Full DepGPO	**25.24**±1.01	**24.64**±2.36

Write credit and verifier filtering. Binary write contribution reduces performance by 2.17 and 2.09 percentage points on Terminal-Bench 2.0 and 2.1, showing that graded credit provides useful distinctions among relevant writes. Removing verifier filtering causes a further drop of 4.42 and 4.80 points relative to the binary variant, suggesting that identifying writes connected to verifier-inspected resources is even more important than distinguishing their contribution strength.

Supporting-read credit. Removing supporting-read credit lowers performance by 3.82 and 4.49 points. This drop shows that credit should extend beyond environment-modifying commands: information-gathering reads can guide subsequent relevant writes, and assigning them credit propagates the learning signal to these supporting decisions.

Indirect write contribution. Using direct write contributions alone reduces performance by 1.79 and 1.64 points. This result shows that some useful writes contribute through later write commands even when their own contents do not remain in the resources inspected by the verifier.

Overall, the ablations show that verifier filtering removes unrelated writes, graded credit distinguishes contribution strength, supporting-read credit assigns signal to information-gathering commands, and indirect credit captures earlier writes that support verifier-relevant outputs.

5 Conclusion

We introduce DepGPO, which uses execution dependencies to redistribute group-relative advantages across interaction steps in terminal tasks. Specifically, DepGPO traces written content to the verifier's read set, assigns graded credit to relevant writes, and uses dependency paths to assign credit to their supporting reads. Experiments on Terminal-Bench 2.0 and 2.1 demonstrate improved task performance, while training analyses show more stable learning. These findings support observable read-write dependencies as an effective basis for fine-grained credit assignment in terminal agents.

AI USE STATEMENT

We used generative AI tools primarily for language polishing, grammar correction, and improving the clarity and organization of the manuscript. The core research idea and experimental design originated with the authors. All AI-assisted content was reviewed and edited by the authors, who take full responsibility for the final manuscript, including its claims, references, implementation, and reported results.

ETHICS STATEMENT

This study uses publicly available datasets and benchmarks for training and evaluation. Agent-generated commands are executed in isolated sandbox environments to reduce the risk of unintended effects on external systems. These safeguards apply to the controlled experimental setting; the reported results do not establish the safety of unrestricted deployment of terminal agents.

REPRODUCIBILITY STATEMENT

We provide the key implementation code in the Supplementary Material to support reproducibility. The main text describes the DepGPO formulation, training objective, experimental setup, and primary empirical results. Appendix A provides the complete implementation details for dependency graph construction, verifier-based write and read credit assignment, step-factor normalization, token-level advantage redistribution, fallback handling, and policy optimization. Appendix B documents the SFT and RL configurations, training-task selection and data splits, checkpoint selection, benchmark evaluation, and score aggregation protocols. Appendix D details the training diagnostics and stability analysis, including the plotting and smoothing protocol, while Appendix E reports sensitivity to the dependency-decay coefficient together with computational-cost and dependency-trace statistics. Together, the manuscript, appendices, and accompanying code specify the procedures used to train, evaluate, and analyze DepGPO.

REFERENCES

Adam AlSayyad, Kelvin Yuxiang Huang, and Richik Pal. Agenttrace: A structured logging framework for agent system observability. *arXiv preprint arXiv:2602.10133*, 2026.

Anthropic. Claude Code: Best practices for agentic coding. `https://www.anthropic.com/engineering/claude-code-best-practices`, April 2025. Accessed: 2026-09-14.

Xin Cheng, Shuo He, Lang Feng, HaiYang Xu, Ming Yan, Lei Feng, and Bo An. Beyond trajectory-level attribution: Graph-based credit assignment for agentic reinforcement learning. *arXiv preprint arXiv:2605.26684*, 2026.

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, Hao Peng, Yu Cheng, Zhiyuan Liu, Maosong Sun, Bowen Zhou, and Ning Ding. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*, 2025.

Mahir Labib Dihan and Md Ashrafur Rahman Khan. Swe-shepherd: Advancing prms for reinforcing code agents. *arXiv preprint arXiv:2604.10493*, 2026.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for LLM agent training. In *Proceedings of Advances in Neural Information Processing Systems*, 2025.

Kanishk Gandhi, Shivam Garg, Noah D Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling rl environments for terminal agents. *arXiv preprint arXiv:2601.16443*, 2026.

Google. Gemini CLI. `https://github.com/google-gemini/gemini-cli`, 2025. GitHub repository, accessed: 2026-09-14.

Hao Han, Jin Xie, Xuehao Ma, Weiquan Zhu, Ziyao Zhang, ZhiLiang Long, Hongkai Chen, and Qingwen Ye. Swe-trace: Optimizing long-horizon swe agents through rubric process reward models and heuristic test-time scaling. *arXiv preprint arXiv:2604.14820*, 2026.

Shuo He, Lang Feng, Xin Cheng, Lei Feng, Bo An, et al. Hierarchy-of-groups policy optimization for long-horizon agentic tasks. In *Proceedings of International Conference on Learning Representations*, volume 2026, pp. 27572–27593, 2026.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. Tmax: A simple recipe for terminal agents. *arXiv preprint arXiv:2606.23321*, 2026.

Renren Jin, Pengzhi Gao, Yuqi Ren, Zhuowen Han, Tongxuan Zhang, Wuwei Huang, Wei Liu, Jian Luan, and Deyi Xiong. Revisiting entropy in reinforcement learning for large reasoning models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 25300–25322, 2026.

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

Han Li, Yifan Yao, Letian Zhu, Rili Feng, Hongyi Ye, Jiaming Wang, Yancheng He, Pengyu Zou, Lehan Zhang, Xinping Lei, et al. Codetracer: Towards traceable agent states. *arXiv preprint arXiv:2604.11641*, 2026.

Taoran Liang, Yang Liu, Shang Luo, Yingguang Yang, Rongrong Zhang, Yingzong Min, Yulin Huang, Jianshen Zhang, Yongzhi Qi, Kefu Xu, et al. Granularity-adaptive credit assignment for long-horizon llm agent reinforcement learning. *arXiv preprint arXiv:2609.12424*, 2026.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *Proceedings of The Twelfth International Conference on Learning Representations*, 2024.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *Proceedings of International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

OpenAI. Codex CLI. `https://github.com/openai/codex`, 2025. GitHub repository, accessed: 2026-09-14.

Penghui Qi, Xiangxin Zhou, Zichen Liu, Tianyu Pang, Chao Du, Min Lin, and Wee Sun Lee. Rethinking the trust region in llm reinforcement learning. *arXiv preprint arXiv:2602.04879*, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents. `https://qwen.ai/blog?id=qwen3.5`, February 2026a. Accessed: 2026-09-21.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27b dense model. `https://qwen.ai/blog?id=qwen3.6-27b`, April 2026b. Accessed: 2026-09-21.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, Zhichen Zeng, Jiajun Li, Shi Dong, Yueming Yuan, Boyuan Ma, Qizheng Zhang, Jiwei Fu, Yuzhen Mao, Wendong Fan, Ping Nie, Philip Torr, Bernard Ghanem, Changran Hu, Jonathan Lingjie Li, Urmish Thakker, and Guohao Li. Seta: Scaling environments for terminal agents. *arXiv preprint arXiv:2607.10891*, 2026.

Hui-Ze Tan, Xiao-Wen Yang, Hao Chen, Jie-Jing Shao, Yi Wen, Yuteng Shen, Weihong Luo, Xiku Du, Lan-Zhe Guo, and Yu-Feng Li. Hindsight credit assignment for long-horizon llm agents. *arXiv preprint arXiv:2603.08754*, 2026.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 9426–9439, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for ai software developers as generalist agents. In *Proceedings of International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Yiqi Wang, Jiaqi Zhang, Taotao Cai, Zirui Liu, Qingqiang Sun, Zequn Sun, Zhangkai Wu, M Zhang, and Yanming Zhu. From agent traces to trust: Evidence tracing and execution provenance in llm agents. *arXiv preprint arXiv:2606.04990*, 2026a.

Yunan Wang, Minghui Song, Zihan Zhang, Shaohan Huang, Haizhen Huang, Furu Wei, Weiwei Deng, Feng Sun, and Qi Zhang. Group-graph policy optimization for long-horizon agentic reinforcement learning. *arXiv preprint arXiv:2606.22995*, 2026b.

Zhiheng Xi, Chenyang Liao, Guanyu Li, Zhihao Zhang, Wenxiang Chen, Binghai Wang, Senjie Jin, Yuhao Zhou, Jian Guan, Wei Wu, et al. Agentprm: Process reward models for llm agents via step-wise promise and progress. In *Proceedings of the ACM Web Conference 2026*, pp. 4184–4195, 2026.

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

Junyao Yang, Yucheng Shi, Zhongzhi Li, Ruhan Wang, Zongxia Li, Haitao Mi, and Leowei Liang. T1: Terminal agent reinforcement learning for long-horizon tasks. *arXiv preprint arXiv:2609.11042*, 2026.

Yihang Yao, Bo Pang, Xuan Phi Nguyen, Ding Zhao, Shafiq Joty, and Semih Yavuz. Learning generalizable behaviors for terminal agents. *arXiv preprint arXiv:2608.22631*, 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: an open-source LLM reinforcement learning system at scale. In *Proceedings of Advances in Neural Information Processing Systems*, 2025.

Simon Yu, Derek Chong, Ananjan Nandi, Dilara Soylu, Jiuding Sun, Christopher D Manning, and Weiyan Shi. Shepherd: Enabling programmable meta-agents via reversible agentic execution traces. *arXiv preprint arXiv:2605.10913*, 2026.

Zishun Yu, Yunzhe Tao, Liyu Chen, Tao Sun, and Hongxia Yang. B-coder: Value-based deep reinforcement learning for program synthesis. In *Proceedings of The Twelfth International Conference on Learning Representations*, 2024.

Jiazhen Yuan, Wei Zhao, and Zhengbiao Bai. Graph-enhanced policy optimization in llm agent training. *arXiv preprint arXiv:2510.26270*, 2025.

Chenchen Zhang. From reasoning to agentic: Credit assignment in reinforcement learning for large language models. *arXiv preprint arXiv:2604.09459*, 2026.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. Agentracer: Who is inducing failure in the llm agentic systems? In *Proceedings of International Conference on Learning Representations*, volume 2026, pp. 11377–11399, 2026.

Siyuan Zhu, Chao Yu, Rongxin Yang, Zongkai Liu, Jinjun Hu, Qiwen Chen, and Yibo Zhang. Gagpo: Generalized advantage grouped policy optimization. *arXiv preprint arXiv:2605.13217*, 2026.

A METHOD AND IMPLEMENTATION DETAILS

DepGPO redistributes the group-relative advantage of a trajectory using execution dependencies that connect agent commands to verifier-inspected resources. The method traces relevant read and write operations, converts their dependencies into command-level credit, and aggregates these credits into step factors that reweight the advantage over policy-generated tokens. This appendix details the complete procedure, from dependency construction and verifier-based attribution to credit computation, normalization, policy optimization, fallback handling, and end-to-end training.

A.1 NOTATION AND RECORDED EVENTS

For task x, let τ_i be the i-th sampled trajectory and T_i its number of interaction steps. Step t consists of an agent response and the resulting environment observation. The set $\mathcal{C}_{i,t}$ contains the commands associated with that response, and $\mathcal{C}_i = \bigcup_t \mathcal{C}_{i,t}$ contains all agent commands in the trajectory. The verifier command is denoted by v_i and is not a policy-generated action. Its execution provides the terminal reward R_i and the resource-access records used to construct the verifier's read set E_i. We omit the trajectory index when discussing one trajectory.

Table 3: Key notation used in the DepGPO credit-assignment procedure.

Symbol	Definition
$\mathcal{C}_{\mathrm{w}}, \mathcal{C}_{\mathrm{r}}$	Disjoint sets of agent write commands and read commands.
$G,\ G_E$	Command dependency graph and write-reachability graph.
E	Files, ports, and processes inspected by the verifier under the extraction rules.
$\mathcal{W}_c$	Tracked file records written by command c, resolved at line level when available.
$D_c,\ I_c$	Records of c that qualify for direct and indirect contribution.
$\mathrm{Contrib}(c)$	Contribution score of write command c.
$q(c),\ q(r),\ q_{i,t}$	Raw write credit, raw read credit, and their aggregate in interaction step t.
$d_G(r,c),\ \beta$	Shortest directed dependency distance and its decay coefficient.
$\mathcal{S}_i,\ f_{i,t}$	Set of steps containing traced commands and the factor assigned to step t.
$m_i,\ \Delta_i$	Minimum step credit and total credit above the minimum within trajectory i.
$L_{i,t},\ Z_i,\ w_{i,t}$	Included token count, token-weighted normalization factor, and final per-step weight.
$\widehat{A}_i,\ \widehat{A}^{\mathrm{Dep}}_{i,t}$	Original group-relative advantage and redistributed advantage for step t.
$M_{i,j}$	Loss mask for token position j of trajectory i.

Read and write records. We record file accesses, port and process events, and standard output. For file writes, differences between the file contents before and after execution identify the written line ranges when available. In contrast, a file-read record identifies the accessed path without identifying which lines the program uses. Accordingly, writes are tracked at line level, while reads are recorded at file level.

A command with at least one recorded write is a write command, including a command that also reads. This rule also applies to recorded deletions and non-file writes. All remaining commands are read commands. Each traced command receives either write credit or read credit according to its command type. The reads of a write command are still used to construct dependency edges and identify indirect contributions of earlier writes.

Record granularity. When line ranges are available, each tracked written line is identified by its file path and line position. When line ranges are unavailable, the implementation uses the whole-file marker `WHOLE_FILE`$= -1$. In that case, the contribution fraction counts file-level fragments. We distinguish this fallback from line-level tracking when reporting examples and statistics.

Path normalization and resource identity. Resource paths are normalized before filtering or matching. Relative paths are resolved against the tracked working directory, while accesses relative to untracked directory descriptors are discarded rather than inferred. Agent and verifier records undergo the same normalization and filtering procedure before dependency construction and extraction of E. Resources outside the configured work roots and known runtime artifacts are filtered as system noise. Symbolic links are not resolved, so resource identity remains path-based and different aliases of the same underlying file are treated as distinct resources. We regard this as a limitation of the current tracing implementation.

Line ranges and file updates. When available, written line ranges are obtained from the new-side hunks of a per-command file diff and recorded in the file coordinates at the time of execution. Ownership is maintained over $(\text{file}, \text{line position})$ pairs and is not rebased after later insertions or deletions. Consequently, subsequent line shifts may introduce limited positional mismatch in ownership tracking. Renames are represented as deletion of the source path and a write to the destination path, without transferring ownership between the two paths.

Whole-file fallback. When a file-level write is recorded but no usable line range can be recovered, the implementation represents that file with the marker `WHOLE_FILE`$= -1$, which contributes one tracked item to $\mathcal{W}_c$. This fallback covers cases such as binary or oversized files, empty-file creation, deletion-only updates, and write sets exceeding the tracing cap. Accordingly, $\mathrm{Contrib}(c)$ is coarser for such writes than for line-resolved edits, because an unresolved whole-file record contributes a single item to $\mathcal{W}_c$.

Multiple writes within one command. For a command that writes multiple files or disjoint regions, all tracked $(\text{file}, \text{position})$ records are pooled into a single set $\mathcal{W}_c$. Thus, $|\mathcal{W}_c|$ is computed across all tracked writes of the command, and unrelated written content can reduce its contribution fraction. Any command with at least one recorded write is treated as a write command, even if it also performs reads; these reads remain available for dependency construction and for identifying indirect contributions of earlier writers. Non-file writes such as ports carry no line range and are handled by the binary rules described in Section A.4.

A.2 Construction of the Command Dependency Graph

The graph $G = (V, \mathcal{E}_G)$ contains the agent commands and the verifier command, with $V = \mathcal{C} \cup \{v\}$. A directed edge $c \to u$ records that a later command u uses information associated with an earlier command c. Edges may connect commands from different interaction steps, but never commands from different trajectories.

File dependencies. Commands are processed in execution order. Immediately before command u is processed, let $o_{u^-}(p, \ell)$ denote the latest recorded writer associated with tracked line position ℓ in file p. File content that has not been written by any agent command has no agent writer and therefore creates no command dependency. When u reads a file p, we connect every earlier command that still owns at least one tracked position in p to u. Formally,

$$(c, u) \in \mathcal{E}_{\text{file}} \quad \Longleftrightarrow \quad \exists (p, \ell) \text{ such that } p \in \mathrm{ReadFiles}(u), \quad o_{u^-}(p, \ell) = c. \tag{10}$$

These read dependencies are established before applying the writes of u. The writes then update ownership of the affected positions, while unaffected positions retain their previous writers. Consequently, multiple earlier commands may remain represented in the same file. An overwrite alone creates no dependency edge between writers; a read of the earlier content is required. Thus, the graph captures recorded read-after-write dependencies rather than write-after-write or write-after-read relations induced solely by execution order.

This processing order applies to the recorded command as a whole. It does not reconstruct the order of individual instructions within a shell command or its subprocesses.

Standard-output dependencies. Standard-output edges capture the reuse of printed information in later command text. The matcher extracts typed values, such as paths and ports, and connects a later command to the first earlier command that printed a matching value of the same type. Values already present in the task description are excluded. This exclusion prevents prompt-provided information from automatically becoming evidence of dependence on an earlier command. When both edge rules connect the same pair of commands, their resource evidence is retained under one mixed edge.

Every edge follows command execution order, making G acyclic. A read command has no recorded write, and therefore has no outgoing file edge under this construction. Its support for later writes begins with a standard-output edge; subsequent edges on the path may be file edges, standard-output edges, or mixed edges.

A.3 Verifier Read Set and Indirect Write Contributions

The verifier read set defines the resource endpoints used for attribution. After the path normalization and filtering described above, let $\mathcal{F}_v$ denote the files whose contents are read by the verifier, $\mathcal{B}_v$ the

files inspected through existence or metadata probes, and $\mathcal{S}_v$ the files written by the verifier itself. The file component of the endpoint set is

$$E_{\text{files}} = (\mathcal{F}_v \setminus \mathcal{S}_v) \cup (\mathcal{B}_v \setminus \mathcal{S}_v). \tag{11}$$

Records marked as shadowed by the recorder are removed from the read and probe sets before they are merged. Port and process checks form the remaining components of E and are not subject to this file-specific filtering. Thus, a resource may belong to E even when the verifier only probes its existence or metadata rather than reading its contents.

Direct contribution. Let $\mathcal{W}_c$ denote the tracked file records written by command c, represented by file and line position when line-level information is available. Let $o_{\text{end}}(p, \ell)$ denote the writer associated with tracked position ℓ in file p at the end of agent execution. The directly contributing records are

$$D_c = \{(p, \ell) \in \mathcal{W}_c : p \in E_{\text{files}},\ o_{\text{end}}(p, \ell) = c\}. \tag{12}$$

A tracked record therefore receives direct credit when it remains attributed to c in a file inspected by the verifier. For line-resolved records, this corresponds to written lines that remain in the inspected file at the end of agent execution. The same file-level inspection convention in Eq. 11 applies to verifier probes.

Indirect contribution. A written record that does not remain in an inspected file may still contribute through a later write command. For $(p, \ell) \in \mathcal{W}_c$, let $\mathcal{U}_c(p, \ell)$ contain later commands u satisfying all of the following conditions:

1. u is a write command and reads file p;
2. immediately before this read is processed, (p, ℓ) is still attributed to c;
3. u writes to at least one file other than p; and
4. the writes of u do not replace the input position (p, ℓ) being counted.

These conditions provide a conservative rule for indirect attribution. The third condition requires information to propagate beyond the input file, while the fourth excludes earlier content that is replaced by the consuming command itself. Failure to qualify through one command does not preclude direct contribution or indirect contribution through another qualifying downstream write.

We next determine which qualifying consumers ultimately contribute to verification. For this purpose, we construct a separate reachability graph G_E. It contains agent-command nodes and an endpoint representing E, but excludes the verifier command itself. A qualifying consumption creates an edge from the earlier writer to the later writer, while write commands with directly contributing records are connected to E. Matching non-file resource events provide the corresponding endpoint connections.

Let $\mathcal{S}_E$ denote the set of write commands that have a directed path to E in G_E. A written record (p, ℓ) of command c is therefore counted as an indirect contribution if at least one of its qualifying downstream consumers belongs to $\mathcal{S}_E$:

$$I_c = \{(p, \ell) \in \mathcal{W}_c : \mathcal{U}_c(p, \ell) \cap \mathcal{S}_E \neq \varnothing\}, \qquad \mathcal{W}_c^E = D_c \cup I_c. \tag{13}$$

Here, I_c contains the indirectly contributing records of c, while $\mathcal{W}_c^E$ combines its direct and indirect contributions. Because this combination is a set union, a record that satisfies both conditions is counted only once.

Operationally, reverse traversal starts from writes connected to resources in E and repeatedly follows qualifying input relations to earlier writers until no additional writers are found. The input reads of a write command are used only to trace contributions of earlier writes; the command itself remains scored by the write-credit rule.

Each tracked record contributes at most once to $\mathcal{W}_c^E$, even if it has multiple qualifying consumers or satisfies both the direct and indirect conditions. An input record is attributed to its earlier writer, whereas a downstream output record belongs to the later writer, and each command is normalized by its own set $\mathcal{W}_c$. Qualifying records are not weighted by the contribution scores of downstream writers, and no distance decay is applied in the write-reachability calculation.

A.4 Complete Command Credit

For a write command with tracked file records, the contribution score is the fraction of its written records that contribute to E. The denominator is the number of tracked records produced by that command, rather than the size of any file it reads or edits. When line-level ranges are unavailable, `WHOLE_FILE`$= -1$ is treated as one tracked record and therefore remains in the file-based branch. The complete write credit is

$$q(c) = \mathrm{Contrib}(c) = \begin{cases} |\mathcal{W}_c^E|/|\mathcal{W}_c|, & |\mathcal{W}_c| > 0, \\ 1, & |\mathcal{W}_c| = 0 \text{ and } c \in \mathcal{S}_E, \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

The second branch applies to write commands with no tracked file record that nevertheless reach E, such as commands represented only by port or process events. For such non-file resources, a match with a verifier-inspected resource gives credit one and no match gives zero. If a command already has a file-based contribution score, a matching non-file resource does not replace that score, preserving the graded file contribution for commands with mixed resource records.

For a read command r, let $\mathcal{D}_G(r)$ denote the set of agent write commands reachable from r in G, excluding the verifier. Let $d_G(r, c)$ be the shortest directed path length from r to a reachable write c. Its raw credit is

$$q(r) = \sum\nolimits_{c \in \mathcal{D}_G(r)} \beta^{d_G(r,c)} \,\mathrm{Contrib}(c), \qquad \beta \in (0, 1]. \tag{15}$$

Each reachable write contributes at most one term, with the shortest path determining the decay when multiple paths connect r to the same write. Contributions from distinct writes are summed, so read credit may exceed one. A read command with no path to any positive-credit write receives zero.

Both write and read credits are nonnegative scores that quantify recorded relevance to verification rather than trajectory success. They determine where the learning signal is assigned, while the signed group-relative advantage determines whether the corresponding sampled responses are reinforced or penalized. Thus, high credit identifies greater recorded relevance without implying that the associated content is correct.

A.5 Advantage Redistribution and Policy Objective

From command credits to step factors. Each traced command receives either write credit or read credit according to the rules above. Let $\mathcal{C}_{i,t}$ contain all traced commands in step t of trajectory τ_i, and let $\mathcal{S}_i = \{t : \mathcal{C}_{i,t} \neq \varnothing\}$ denote the set of steps containing at least one traced command. We aggregate command credits within each step as

$$q_{i,t} = \sum\nolimits_{c \in \mathcal{C}_{i,t}} q(c), \tag{16}$$

where $q(c)$ denotes the write or read credit assigned to command c according to its command type. Credits from multiple commands in the same step are summed rather than averaged. The resulting step credits are compared only within the same trajectory.

For nonempty $\mathcal{S}_i$, define

$$m_i = \min_{s \in \mathcal{S}_i} q_{i,s}, \qquad \Delta_i = \sum\nolimits_{s \in \mathcal{S}_i} (q_{i,s} - m_i). \tag{17}$$

The step factors are then

$$f_{i,t} = \begin{cases} |\mathcal{S}_i|(q_{i,t} - m_i)/\Delta_i, & t \in \mathcal{S}_i,\ \Delta_i > 10^{-9}, \\ 1, & \text{otherwise.} \end{cases} \tag{18}$$

Thus, when the compared step credits are nondegenerate, subtracting m_i uses the lowest observed credit as the within-trajectory reference, while division by $\Delta_i/|\mathcal{S}_i|$ normalizes the mean factor over $\mathcal{S}_i$ to one. The minimum-credit steps therefore receive factor zero, even when their raw credits are positive.

Steps without traced commands retain an initial factor of one and are not included in the within-trajectory comparison. If $\mathcal{S}_i$ is empty, tracing yields no usable credit, or $\Delta_i \leq 10^{-9}$, all step factors are set to one and the baseline advantage assignment is retained. In particular, if no write command has positive contribution to E, all write credits are zero and the propagated read credits are also zero, triggering the same fallback.

From step factors to final weights and advantages. Let $M_{i,j} \in \{0, 1\}$ indicate whether token position j enters the policy loss. Policy-generated tokens have mask one, while environment observations have mask zero. The factor of step t is assigned to the corresponding policy-generated token span. Positions outside the mapped spans retain an initial factor of one; if such positions enter the loss, they are included in the subsequent token-level normalization.

Because the policy loss averages over generated tokens, steps with longer responses contribute more loss terms. We therefore normalize the step factors by their token-weighted mean. When all included tokens are covered by mapped step spans, let $L_{i,t}$ denote the number of loss-included tokens from step t. For positive included length and normalization denominator,

$$Z_i = \frac{\sum_{t=1}^{T_i} L_{i,t} f_{i,t}}{\sum_{t=1}^{T_i} L_{i,t}}, \qquad w_{i,t} = \frac{f_{i,t}}{Z_i}, \qquad \widehat{A}_{i,t}^{\mathrm{Dep}} = w_{i,t} \widehat{A}_i. \tag{19}$$

Here, $f_{i,t}$ is the step factor before token-level normalization, $w_{i,t}$ is the final weight, and $\widehat{A}_{i,t}^{\mathrm{Dep}}$ is the redistributed advantage for step t. Every loss-included token in that step receives the same redistributed advantage. A step without traced commands has $f_{i,t} = 1$ before token-level normalization and therefore receives final weight $1/Z_i$, which is not necessarily one.

In the implementation, Z_i is computed directly over all policy-generated tokens included in the loss. Each token inherits the factor of its mapped interaction step; any included token without a valid step mapping retains the default factor of one. When every included token is mapped to a step, this token-level computation reduces exactly to the expression above.

What normalization preserves. For a nondegenerate step-level redistribution,

$$\sum_{t \in \mathcal{S}_i} f_{i,t} = |\mathcal{S}_i|. \tag{20}$$

Since steps outside $\mathcal{S}_i$ retain factor one, the mean step factor over the trajectory is also one. After token-level normalization,

$$\frac{\sum_t L_{i,t} w_{i,t}}{\sum_t L_{i,t}} = 1, \qquad \frac{\sum_t L_{i,t} \widehat{A}_{i,t}^{\mathrm{Dep}}}{\sum_t L_{i,t}} = \widehat{A}_i. \tag{21}$$

Thus, the step-level normalization preserves the mean step factor, while the token-level normalization preserves the token-weighted mean final weight. Consequently, the token-weighted mean redistributed advantage remains equal to the original group-relative advantage $\widehat{A}_i$.

These identities apply to the advantage coefficients rather than to the resulting optimization dynamics. The policy gradient, its norm, and the clipped objective may still change because different tokens receive different coefficients. In particular, unequal response lengths can yield $Z_i \neq 1$ even when the unweighted mean step factor is one.

All valid weights are nonnegative and therefore preserve the sign of $\widehat{A}_i$. If $\widehat{A}_i = 0$, all redistributed advantages remain zero. DepGPO therefore redistributes the learning signal supplied by the GRPO baseline without creating a nonzero signal when the original group-relative advantage is zero.

Policy objective. Index all policy-generated tokens included in a training minibatch by $n = 1, \ldots, N$. Let y_n denote token n, h_n its original context, and a_n the redistributed advantage of the interaction step containing that token. The corresponding probability ratio is

$$\rho_n(\theta) = \frac{\pi_\theta(y_n \mid h_n)}{\pi_{\theta_{\mathrm{old}}}(y_n \mid h_n)}, \tag{22}$$

where $\pi_{\theta_{\mathrm{old}}}$ is the policy used to collect the trajectories. This indexing affects only the loss reduction; each token retains the context of its original trajectory.

Following the clipped surrogate update of the GRPO baseline, we define

$$\ell_{\text{clip}}(\rho, a) = \min\{\rho a,\ \text{clip}(\rho, 1-\epsilon_{\text{lo}}, 1+\epsilon_{\text{hi}})a\}\,,$$
$$\ell_{\text{dc}}(\rho, a) = \begin{cases} \max\{\ell_{\text{clip}}(\rho, a), \eta a\}\,, & a < 0, \\ \ell_{\text{clip}}(\rho, a), & a \geq 0, \end{cases} \tag{23}$$
$$\mathcal{L}_{\text{Dep}}(\theta) = -\frac{1}{N}\sum\nolimits_{n=1}^{N} \ell_{\text{dc}}(\rho_n(\theta), a_n). \tag{24}$$

Here, ϵ_{lo} and ϵ_{hi} define the lower and upper clipping ranges, and $\eta > 1$ is the dual-clipping threshold for negative advantages. The dependency-derived factors and redistributed advantages are held fixed during each policy update. DepGPO changes only the advantages assigned to tokens, while retaining the clipping rule and token-mean reduction of the GRPO baseline.

Fallback and integration. Table 4 summarizes the conditions under which unit dependency factors are used. Step factors are derived from the rollout records, expanded to the corresponding token positions, and normalized using the policy-loss mask. The resulting weights are applied after the baseline group-relative advantages are computed and before the actor update. The current integration also applies the same weights to the returns tensor. Because the GRPO path used here is critic-free, this does not introduce a separate critic target; the returns tensor carries the same outcome-derived signal used by the policy update.

Table 4: Fallback conditions that retain the baseline GRPO advantage assignment. These cases use unit dependency factors, or disable dependency weighting in the baseline configuration, without removing the trajectory from the baseline reward and group-relative advantage computation.

Condition	**Result**
No step information or unavailable credit module	Unit factors.
Missing usable verifier or agent-command records	Unit factors; the missing-trace case is recorded.
Empty command-credit output	Unit factors.
$\mathcal{S}_i = \varnothing$	Unit factors.
$\Delta_i \leq 10^{-9}$	Unit factors, including equal-credit and all-zero cases.
Exception caught by the calling trainer	Unit factors and an error diagnostic.
Baseline configuration	Dependency weighting is disabled.

A.6 End-to-End Procedure

The following procedure summarizes the complete DepGPO training pipeline. Resource extraction and dependency-based credit assignment are computed from the recorded trajectories and are held fixed during the corresponding policy updates.

1. Sample K trajectories for each task using $\pi_{\theta_{\text{old}}}$, recording step-to-command mappings and policy-generated token spans.
2. Execute the task verifier for each trajectory to obtain the terminal reward R_i and the verifier resource-access records.
3. For each trajectory, extract the verifier endpoint set E, construct the command dependency graph G, and build the qualifying write-consumption relations that define G_E.
4. Traverse G_E backward from E to identify directly and indirectly contributing write records, count each qualifying record once, and compute write credit.
5. For each read command, identify reachable write descendants in G and sum their contribution scores with shortest-path decay to obtain read credit.
6. Aggregate command credits within each interaction step, compute the corresponding step factors, and apply the documented unit-factor fallbacks when necessary.
7. Expand the step factors to policy-generated tokens and normalize them over the policy-loss mask to obtain the final weights. After the baseline group-relative advantages are computed, apply these weights to obtain the redistributed per-step advantages.

8. Update the policy parameters θ using Eq. 24, and then collect the next group of trajectories with the updated policy.

B Experimental Setup and Reproducibility

This appendix provides the complete experimental setup used to evaluate DepGPO and the baselines, with particular attention to training consistency, comparison fairness, and reproducibility. We first describe the common training pipeline and controlled settings shared across methods, followed by the training data, task selection and splits, model configurations, optimization hyperparameters, evaluation protocol, and implementation details.

All methods follow the same two-stage training procedure, consisting of supervised fine-tuning (SFT) on terminal trajectories followed by reinforcement learning (RL) on selected training tasks. SETA (Shen et al., 2026) and TMAX (Ivison et al., 2026) are used separately with Qwen3.5-9B (Qwen Team, 2026a) and Qwen3.6-27B (Qwen Team, 2026b). Within each model and data setting, all methods share the same SFT checkpoint, selected task pool, training and validation split, and agent scaffold, ensuring matched starting policies and task distributions. For the direct comparison between GRPO and DepGPO, we additionally keep the policy objective and optimization settings unchanged, with DepGPO modifying only how the group-relative advantage is redistributed across interaction steps. Other baselines retain their respective method-specific algorithmic components while using the same shared experimental setting whenever applicable.

B.1 Supervised Fine-Tuning

We fine-tune each base model on SETA and TMAX separately, producing an SFT checkpoint for the corresponding RL setting. The SFT configuration uses AdamW with a learning rate of 10^{-5}, optimizer coefficients $(0.9, 0.95)$, weight decay 0.01, and gradient clipping at 1.0. The learning rate increases during the first 3% of training and remains constant afterward. Each run completes one epoch over its training corpus.

Training updates all model parameters in bfloat16 and uses Fully Sharded Data Parallel (FSDP), gradient checkpointing, and sequence packing. Multi-turn conversations are tokenized with the preceding context retained, while the loss includes only assistant tokens. Conversations exceeding the sequence limit are truncated from the left. Table 5 summarizes the corpus sizes and sequence limits. SETA and TMAX use different sequence budgets, while the settings share the optimizer configuration above. The conversation counts refer to SFT examples and are distinct from the task counts used for RL.

Table 5: Supervised fine-tuning configurations. The 9B and 27B columns use Qwen3.5-9B and Qwen3.6-27B, respectively.

Setting	SETA/9B	SETA/27B	TMAX/9B	TMAX/27B
Training conversations	79,882	79,882	16,355	16,355
Validation conversations	3,690	3,690	166	166
Maximum sequence length	8,192	8,192	32,768	32,768
Per-device micro-batch	1	1	1	1
Devices	6	4	4	4
Epochs	1	1	1	1
Seed	1	1	1	1

B.2 Selecting Training Tasks

With binary outcome rewards, a sampled group containing only successes or only failures has zero centered group-relative advantages. Such groups therefore provide no outcome-based policy-gradient signal for GRPO or for DepGPO's redistribution of those advantages. To increase the frequency of informative groups, we select tasks on which the corresponding SFT policy produces both successful and unsuccessful attempts.

For each candidate task x, we sample four trajectories with the SFT checkpoint of that model and data setting. Let $z_{x,k} \in \{0, 1\}$ be the verifier outcome of attempt k. We compute

$$\widehat{p}_{\mathrm{SFT}}(x) = \frac{1}{4}\sum_{k=1}^{4} z_{x,k}, \qquad \mathcal{P} = \{x : 0 < \widehat{p}_{\mathrm{SFT}}(x) < 1\}, \tag{25}$$

where $\mathcal{P}$ denotes the retained task pool. With four attempts, this rule retains success rates of 0.25, 0.50, and 0.75. Tasks with zero or four successful attempts are excluded. The retained pool is then fixed across all RL methods in that setting. This selection increases the availability of mixed-outcome tasks under the initial policy, although later training groups can still contain identical rewards.

Table 6: Task selection using four attempts per candidate task. Each setting uses its own SFT checkpoint.

Setting	SETA/9B	SETA/27B	TMAX/9B	TMAX/27B
Candidate tasks	1,376	1,376	2,998	2,998
Zero successful attempts	639	279	2,200	1,950
Four successful attempts	200	501	258	400
Retained tasks	**537**	**596**	**540**	**648**
RL training tasks	483	536	486	583
RL validation tasks	54	60	54	65

For SETA/9B, the excluded groups contain $639 + 200 = 839$ tasks, accounting for approximately 61% of the $1{,}376$ candidates. This is the proportion reported in the pilot analysis. On the same candidate collection, the 27B checkpoint produces fewer zero-success groups and more four-success groups, leaving 596 tasks for RL. Because selection depends on the initial policy, task pools are constructed separately for each model and data setting. Fairness across methods is maintained by reusing the same pool within a setting.

In the TMAX completion, the 9B counts retain 540 of 2,998 tasks (18.01%), while the 27B counts retain 648 tasks (21.61%). The 27B pool contains 583 training tasks and 65 validation tasks after the same deterministic split.

We split each retained pool by sorting task identifiers and assigning the first $\lfloor 0.9|\mathcal{P}| \rfloor$ tasks to RL training and the remainder to RL validation. This deterministic rule gives the counts in Table 6. The RL validation tasks are held out from policy updates and are used for checkpoint selection. They are separate from the conversation-level validation splits listed for SFT.

B.3 Reinforcement Learning

RL starts from the SFT checkpoint corresponding to the model and training source. Each update samples four training tasks and eight trajectories per task, giving 32 trajectories. The training budget is 150 policy updates. Table 7 gives the common settings for the controlled GRPO and DepGPO comparison, including the settings that differ across model sizes or training sources.

Across all settings, each update samples four tasks with $K = 8$ trajectories per task, yielding 32 trajectories, and training runs for 150 policy updates. We use AdamW with a learning rate of 10^{-6}, a constant schedule, clipping range $(0.2, 0.28)$, dual-clip threshold $\eta = 3.0$, no KL penalty or entropy bonus, group-SD normalization, a response-token budget of 24,576, and $\beta = 0.5$ for DepGPO. The SETA settings allow up to 10 interaction steps per trajectory, while TMAX uses 56, with sampling temperatures of 1.0 and 0.7, respectively. For Qwen3.5-9B, we use a 120-second command timeout and sequence-parallel degree one; Qwen3.6-27B uses a 400-second timeout and sequence-parallel degree four. All settings use four NVIDIA H200 GPUs.

A trajectory ends when the agent declares completion or reaches its interaction or token limit. The task verifier then returns a binary outcome, with no intermediate environment reward. DepGPO uses the command and verifier records to compute write and read credit, aggregates command credits into step factors, and normalizes those factors into final weights. The weights multiply the group-

Table 7: RL configurations for the GRPO and DepGPO comparison. Each column starts from its corresponding SFT checkpoint. Method-specific baseline components are retained separately.

Setting	SETA/9B	SETA/27B	TMAX/9B	TMAX/27B
Tasks per update	4	4	4	4
Trajectories per task K	8	8	8	8
Trajectories per update	32	32	32	32
Policy updates	150	150	150	150
Optimizer	AdamW	AdamW	AdamW	AdamW
Learning rate	10^{-6}	10^{-6}	10^{-6}	10^{-6}
Learning-rate schedule	Constant	Constant	Constant	Constant
Clip range $(\epsilon_{\mathrm{lo}}, \epsilon_{\mathrm{hi}})$	$(0.2, 0.28)$	$(0.2, 0.28)$	$(0.2, 0.28)$	$(0.2, 0.28)$
Dual-clip threshold η	3.0	3.0	3.0	3.0
KL penalty / entropy bonus	0/0	0/0	0/0	0/0
Group-SD normalization	Enabled	Enabled	Enabled	Enabled
Response-token budget	24,576	24,576	24,576	24,576
Decay β (DepGPO)	0.5	0.5	0.5	0.5
Maximum interaction steps	10	10	56	56
Sampling temperature	1.0	1.0	0.7	0.7
Command timeout (seconds)	120	400	120	400
Sequence-parallel degree	1	4	1	4
Devices	4	4	4	4

relative advantages before the policy update, following Appendix A.5. The outcome reward and the underlying policy objective remain unchanged in the GRPO comparison.

B.4 Checkpoint Selection

We periodically evaluate checkpoints on the RL validation tasks from the corresponding retained pool. For each method, we select the checkpoint with the highest validation success rate and use that checkpoint for the final benchmark evaluation. The same selection criterion applies to DepGPO and all baselines. Consequently, a baseline that deteriorates later in training can still be evaluated at an earlier checkpoint when that checkpoint has the highest validation score. The complete training curves are analyzed separately to assess training stability.

Checkpoint selection uses RL validation performance. Terminal-Bench 2.0 and 2.1 scores are reserved for reporting the selected policies, with the data-separation protocol described in Appendix B.6.

B.5 Benchmark Evaluation and Score Aggregation

We evaluate the selected checkpoints separately on Terminal-Bench 2.0 and 2.1. Each attempt begins from the task's initial environment and allows at most 56 interaction steps. We conduct five independent attempts per task and repeat the complete evaluation three times. Within each round, pass@1 is estimated by averaging the five binary outcomes for each task and then averaging across tasks.

For benchmark version v, let J_v be the number of tasks and let $z_{j,h,a}^{(v,m)} \in \{0, 1\}$ denote the outcome for method m on task j, evaluation round h, and attempt a. The task and benchmark scores for each round are

$$\widehat{p}_{j,h}^{(v,m)} = \frac{1}{5} \sum_{a=1}^{5} z_{j,h,a}^{(v,m)}, \qquad s_h^{(v,m)} = \frac{100}{J_v} \sum_{j=1}^{J_v} \widehat{p}_{j,h}^{(v,m)}. \tag{26}$$

With $H = 3$ evaluation rounds, we report

$$\overline{s}^{(v,m)} = \frac{1}{H}\sum_{h=1}^{H} s_h^{(v,m)}, \qquad \mathrm{SD}^{(v,m)} = \sqrt{\frac{1}{H-1}\sum_{h=1}^{H}\left(s_h^{(v,m)} - \overline{s}^{(v,m)}\right)^2}. \tag{27}$$

The table 1 show $\overline{s} \pm \mathrm{SD}$ in percentage points. This SD measures evaluation variability for a fixed trained checkpoint. Variability across independently trained policies requires separate training runs. In addition, averaging individual attempts estimates pass@1; treating a task as solved whenever any of its five attempts succeeds would instead estimate pass@5.

B.6 Data Separation and Verifier Information

Task selection uses the training-source collections, and RL validation tasks are excluded from policy updates. Terminal-Bench evaluation tasks are kept separate from the training and validation task pools and are not used to choose checkpoints. Conversation-level SFT splits and task-level RL splits serve different purposes, so their separation is assessed at the appropriate unit.

After a training trajectory is completed, the task verifier returns the terminal outcome and records which resources it inspects. DepGPO uses these access records to construct E and then computes the dependency-based weights for the finished trajectory. These weights are applied only after the trajectory has been generated; the policy does not receive E or the verifier access records as additional input during action generation. Thus, DepGPO still learns from a terminal reward, while execution dependencies are used only to determine how that reward-derived advantage is distributed across steps. Whether the agent can directly access verifier scripts through the terminal is determined by the agent scaffold and is independent of this weighting procedure.

C Task-Level Paired Comparison

This appendix examines how the aggregate performance gains of DepGPO are distributed across individual tasks. We compare DepGPO and GRPO using Qwen3.5-9B trained on SETA, with evaluation on Terminal-Bench 2.0 and 2.1. The analysis examines the frequency and magnitude of improvements, ties, and regressions.

Comparison protocol. For each benchmark version v, we compare the two methods on the same 89 tasks. Let $\bar{p}_j^{(v,m)}$ denote the observed success rate of method m on task j, expressed as a proportion. The task-level difference and its benchmark average are

$$\Delta_j^{(v)} = 100\left(\bar{p}_j^{(v,\mathrm{DepGPO})} - \bar{p}_j^{(v,\mathrm{GRPO})}\right),$$
$$\overline{\Delta}^{(v)} = \frac{1}{89}\sum_{j=1}^{89}\Delta_j^{(v)}. \tag{28}$$

Positive, zero, and negative differences indicate improvements, ties, and regressions, respectively. Figure 6 sorts tasks independently within each benchmark by descending $\Delta_j^{(v)}$. The horizontal axis therefore represents the rank of the observed difference, not task difficulty or a shared task ordering across benchmark versions.

Coverage of improvements. On Terminal-Bench 2.0, DepGPO improves on 31 tasks, ties on 53, and performs worse on 5, with a mean improvement of 14.2 percentage points (pp). On Terminal-Bench 2.1, the corresponding counts are 33, 54, and 2, with a mean improvement of 13.6 pp. Among tasks with nonzero differences, improvements account for 31 of 36 tasks (86.1%) on Terminal-Bench 2.0 and 33 of 35 tasks (94.3%) on Terminal-Bench 2.1. Thus, when the observed success rates differ, the difference predominantly favors DepGPO. These counts describe tasks with higher success rates, not necessarily tasks newly solved by DepGPO.

Magnitude of improvements and regressions. The positive differences vary substantially across tasks, with several large improvements alongside smaller gains. On Terminal-Bench 2.0, the highlighted examples include approximately 80 pp on `extract-moves-from-video` and 67 pp each on `git-leak-recovery` and `pypi-server`. On Terminal-Bench 2.1,

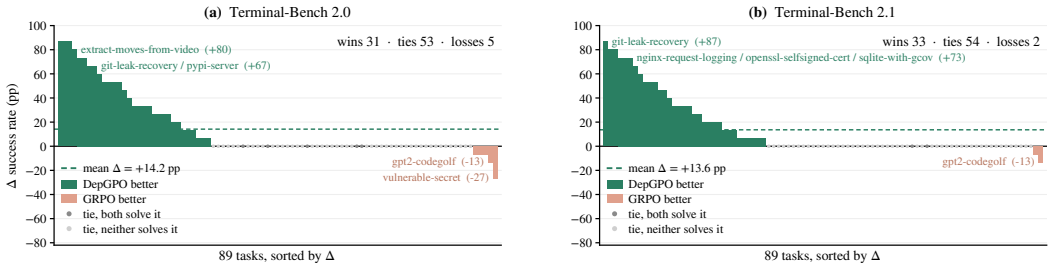

Figure 6: Task-level success-rate differences between DepGPO and GRPO using Qwen3.5-9B trained on SETA and evaluated on Terminal-Bench 2.0 (a) and 2.1 (b). Each panel independently sorts 89 tasks by Δ, computed as DepGPO minus GRPO in percentage points. Green and orange bars indicate improvements and regressions, gray markers denote ties, and dashed lines show the mean difference over all tasks. Annotations highlight selected task-level differences.

`git-leak-recovery` improves by approximately 87 pp, while `nginx-request-logging`, `openssl-selfsigned-cert`, and `sqlite-with-gcov` each improve by approximately 73 pp. These examples show that substantial improvements occur on multiple tasks.

Regressions remain present and should be considered alongside these gains. On Terminal-Bench 2.0, the highlighted decreases are approximately 27 pp on `vulnerable-secret` and 13 pp on `gpt2-codegolf`. The latter also decreases by approximately 13 pp on Terminal-Bench 2.1. The positive benchmark averages therefore reflect improvements that outweigh the observed regressions, not uniform gains on every task.

Ties and differences between benchmark versions. Ties constitute the majority of tasks in both panels, giving a median task-level difference of zero despite the positive mean differences. The average improvement consequently summarizes gains on a subset of tasks, while many tasks retain the same observed success rate. A zero difference alone does not establish that a task is solved reliably: equal success rates can also reflect shared failures. Moreover, equal task-level success rates do not imply identical command sequences or execution dependencies.

Terminal-Bench 2.1 has two more improved tasks and three fewer regressed tasks than Terminal-Bench 2.0, yet its mean improvement is slightly smaller. This illustrates that the number of improved tasks and the magnitude of their improvements capture different aspects of performance. The two benchmarks show a similar overall pattern, but the independently sorted panels do not establish that the same tasks improve in both versions.

Relation to dependency-aware credit assignment. These task-level results are consistent with the motivation of DepGPO. Its credit rules distinguish writes connected to verifier-inspected resources and assign credit to the reads and earlier writes that support them. This redistribution may help on tasks where useful information and intermediate outputs pass through several commands, because those dependencies provide a basis for assigning the group-relative learning signal. However, the outcome differences alone do not identify which dependency paths explain an individual task's improvement or regression; that attribution requires inspecting its execution traces.

Overall, the paired comparison complements aggregate pass@1 by showing that DepGPO improves substantially more tasks than it degrades, with several large gains and many unchanged outcomes. These results support task-dependent benefits for Qwen3.5-9B trained on SETA, without implying universal improvement or statistical significance for every individual task.

D Additional Training Stability Analysis

This appendix provides additional analysis of the training behavior shown in Figure 5. We clarify the diagnostic signals used in the figure and examine how training reward, policy entropy, gradient norm, and dependency-based credit redistribution evolve during optimization. Panels (a)–(c) compare DepGPO, GRPO, and GiGPO using Qwen3.5-9B trained on SETA. Panel (d) compares the

evolution of DepGPO's step-factor dispersion for Qwen3.5-9B and Qwen3.6-27B, both trained on SETA. For visualization, all training curves are smoothed using a centered moving average over up to 11 adjacent training steps, with truncated windows at the boundaries. Policy entropy is smoothed in $\log_{10}$ space and then transformed back to its original scale, while all other metrics are smoothed on their original scales. Checkpoint selection is conducted independently using the RL validation protocol described in Appendix B.4.

D.1 Diagnostics and Plotting Protocol

The four diagnostics in Figure 5 capture complementary aspects of training. Training reward reflects the terminal verifier outcomes obtained by the sampled trajectories at each policy update. Policy entropy characterizes the concentration of the policy distribution over generated tokens, while gradient norm measures the magnitude of the actor update. For DepGPO, we additionally report the standard deviation of the step factors, which measures how differently the group-relative advantage is allocated across interaction steps.

Table 8: Training diagnostics used in Figure 5. These statistics characterize optimization behavior and are not used for checkpoint selection.

Metric	**Interpretation**
Training reward	Terminal task performance observed during training.
Policy entropy	Concentration of the policy distribution over generated tokens.
Gradient norm	Magnitude of the actor update during policy optimization.
Step-factor std.	Variation in DepGPO's allocation of the group-relative advantage across interaction steps.

For visualization, all training curves are smoothed using a centered moving average over up to 11 adjacent training steps, with truncated windows at the boundaries. Policy entropy is smoothed in $\log_{10}$ space and then transformed back to its original scale, while all other metrics are smoothed on their original scales. The shaded late-training region highlights the stage in which the methods exhibit clearly different optimization behavior; it does not affect training or checkpoint selection. Policy entropy is shown on a logarithmic scale to make variation in low-entropy regimes more visible.

D.2 Training Reward Dynamics

Figure 5(a) shows distinct late-training behavior across the three methods. GRPO improves during the earlier stage but subsequently loses much of this gain, while GiGPO deteriorates earlier and remains at a comparatively low reward level. DepGPO instead fluctuates while maintaining an overall higher reward during the late stage.

These differences are consistent with the credit-assignment mechanisms of the three methods. GRPO assigns the same group-relative advantage to all generated steps in a trajectory. Consequently, commands with weak or no recorded connection to the verified outcome can receive the same signed learning signal as commands that directly support task completion.

GiGPO introduces step-level credit through comparisons among actions associated with matching environment states. However, as shown by the pilot analysis in Figure 1, repeated states are sparse in the terminal setting considered here, limiting the number of additional step groups that can be formed. DepGPO does not require such state recurrence. Instead, it uses observed read–write dependencies within each trajectory to determine where the available group-relative advantage is applied.

The reward curves therefore provide evidence that DepGPO maintains stronger late-stage training performance in the runs shown. They do not by themselves establish that credit assignment is the only cause of the observed degradation of the baselines.

D.3 Policy Entropy and Gradient Norm

The entropy curves provide a complementary view of how the policies evolve. As shown in Figure 5(b), GiGPO exhibits an early increase followed by a sharp decline in policy entropy. GRPO follows a different pattern, with entropy increasing substantially later in training. DepGPO remains comparatively stable over the same training horizon.

The gradient statistics show a corresponding difference in optimization behavior. In Figure 5(c), both baselines exhibit very small gradients during the later stage, whereas DepGPO continues to receive nonzero updates. In the unsmoothed logs, the GRPO gradient norm is exactly zero on 43 of the final 45 updates. Together with the reward curves, this indicates that the baseline runs enter late-training regimes in which further policy updates provide limited effective learning, while DepGPO continues to update.

Entropy and gradient norm should not be interpreted as performance metrics in isolation. A particular entropy value is not inherently preferable, and a larger gradient norm does not necessarily imply a better policy. Their relevance here lies in their temporal relationship with the reward curves: the baseline reward degradation coincides with substantial changes in policy entropy and diminishing gradients, whereas DepGPO exhibits less pronounced degeneration across these diagnostics.

D.4 Evolution of Credit Redistribution

Figure 5(d) examines DepGPO's redistribution mechanism directly. For each trajectory, command-level credit is aggregated into step factors $f_{i,t}$. As defined in Appendix A.5, these factors are normalized so that their mean across interaction steps is one. The mean therefore does not indicate how unevenly the learning signal is distributed, motivating the use of the step-factor standard deviation.

A small standard deviation indicates relatively similar weights across interaction steps, whereas a larger value indicates a more differentiated allocation. For both Qwen3.5-9B and Qwen3.6-27B trained on SETA, the step-factor standard deviation generally becomes larger later in training, although the trajectories fluctuate across updates. This pattern indicates that DepGPO increasingly differentiates among interaction steps as optimization proceeds.

The increasing dispersion should not be interpreted as an increase in the overall magnitude of the learning signal. The step-level normalization fixes the mean factor at one, while the subsequent token-level normalization preserves the token-weighted mean redistributed advantage. Thus, the change in standard deviation reflects a change in *where* the group-relative learning signal is allocated rather than a uniform amplification of that signal.

The 9B and 27B curves also show that differentiated credit allocation is observed at both model scales. Their exact trajectories need not match: model capacity, sampled trajectories, and the dependency structures observed during training can all affect the distribution of step credit. Accordingly, Figure 5(d) is used as a descriptive comparison rather than evidence that model size alone causes the difference between the two curves.

D.5 Relationship Between Credit Redistribution and Stability

Taken together, the diagnostics show a consistent pattern in the analyzed runs. DepGPO maintains higher late-stage training reward and nonzero policy updates while its step-level allocation becomes increasingly differentiated. This behavior is consistent with the intended role of dependency-aware credit assignment: the group-relative advantage determines whether a trajectory is reinforced or penalized, while execution dependencies determine how strongly that signal is assigned to different interaction steps.

The normalization procedure is important for this interpretation. Because the mean step factor is fixed and token-level normalization preserves the average advantage scale, the observed stability cannot be explained simply by a uniform increase in the original GRPO learning signal. DepGPO instead changes the relative contribution of tokens from different interaction steps while preserving the sign and overall scale of the trajectory-level advantage.

These observations establish an association between dependency-aware redistribution and more stable late-stage optimization in the runs examined here. They do not establish that any individual

dependency edge, command credit, or step factor causally determines a particular change in reward, entropy, or gradient norm. Such causal attribution would require interventions on individual trajectories or dependency paths. The purpose of this analysis is therefore to characterize the optimization behavior accompanying DepGPO and to complement the final benchmark results in Table 1.

E Sensitivity and Computational Cost

This appendix examines the sensitivity of DepGPO to the dependency-decay coefficient β and characterizes the additional computation introduced by dependency tracing and credit assignment. Unless otherwise specified, we use Qwen3.5-9B trained on SETA, matching the controlled setting used for the main ablation and stability analyses.

E.1 Sensitivity to Dependency Decay

The coefficient β controls how quickly read credit decays with dependency distance. Smaller values emphasize reads that are close to relevant writes, whereas larger values retain more credit along longer dependency paths. We vary β while keeping the remaining training configuration unchanged.

Table 9: Sensitivity to the dependency-decay coefficient β using Qwen3.5-9B trained on SETA. Results are pass@1 (%), mean $\pm$ SD.

β	TB2.0	TB2.1
0.25	24.27±1.19	23.75±1.63
0.50	**25.24±1.01**	**24.64±2.36**
0.75	24.87±0.91	24.12±1.75
1.00	23.97±1.28	23.45±1.88

Performance remains relatively stable across a broad range of decay strengths. Moderate decay performs best in this setting, with $\beta = 0.5$ reaching 25.24/24.64 on Terminal-Bench 2.0/2.1. Reducing the coefficient to $\beta = 0.25$ yields 24.27/23.75, suggesting that overly rapid decay can suppress useful information propagated through multi-command dependency paths. Increasing the coefficient to $\beta = 0.75$ produces 24.87/24.12, remaining close to the default setting, whereas removing distance attenuation with $\beta = 1.0$ decreases performance to 23.97/23.45.

These results suggest that the benefit of DepGPO does not depend on a narrowly tuned decay value. At the same time, some attenuation is useful: assigning nearly equal importance to both short and long dependency paths can increase the influence of more distant supporting reads, while overly aggressive decay can remove informative longer-range dependencies. We therefore use $\beta = 0.5$ as a balanced default throughout the main experiments.

E.2 Computational Overhead

DepGPO introduces additional computation after trajectory execution. For each completed trajectory, the implementation parses the recorded resource accesses, constructs the command dependency graph, traces verifier-relevant writes, propagates credit to supporting reads, and maps the resulting step factors back to policy-generated tokens. These operations use the recorded execution trace and require no additional language-model forward or backward passes.

Table 10 compares the resulting cost with GRPO under the same Qwen3.5-9B and SETA training configuration. Wall-clock measurements include rollout collection, dependency processing, and the policy update.

The additional wall-clock cost is modest relative to the complete terminal-agent training loop. DepGPO increases the average update time by approximately 7.0%, while peak GPU memory changes by only 1.0%. This difference reflects the fact that dependency construction and credit computation operate primarily on execution records rather than model activations. The additional processing takes approximately *3.8 seconds per trajectory* in this setting and does not introduce extra language-model inference.

Table 10: Computational cost of DepGPO relative to GRPO using Qwen3.5-9B trained on SETA.

Metric	GRPO	DepGPO	Difference
Relative wall-clock time / update	1.00×	1.07×	+7.0%
Peak GPU memory	1.00×	1.01×	+1.0%
Dependency processing / trajectory	–	3.8 s	–
Additional model forward passes	0	0	0

The overhead is therefore concentrated in trace processing rather than neural network computation. This distinction is particularly relevant for terminal tasks, where rollout execution itself includes repeated environment interactions and command execution. As a result, dependency analysis contributes only a limited fraction of the overall training time.

E.3 Dependency-Trace Statistics

We further examine how frequently the dependency mechanism is active during training. Table 11 summarizes execution-trace statistics collected from the Qwen3.5-9B SETA run. These statistics describe the traces used for credit assignment before token-level normalization.

Table 11: Dependency-trace statistics for Qwen3.5-9B trained on SETA.

Statistic	Value
Commands per trajectory	6.8
Trajectories with usable dependency credit	82.4%
Trajectories using the unit-factor fallback	17.6%
Write commands with positive credit	38.7%
Read commands with positive credit	27.9%
Positive-credit writes with indirect contribution	18.6%
Mean shortest read-to-write dependency distance	1.74

Usable dependency credit is available for 82.4% of trajectories, indicating that the redistribution mechanism is active for most sampled interactions rather than being dominated by fallback cases. Only 17.6% of trajectories revert to unit factors because no usable relative credit can be constructed.

Both components of the credit mechanism are exercised frequently. Approximately 38.7% of traced write commands receive positive credit, while 27.9% of read commands receive positive supporting-read credit. This indicates that the resulting signal is not restricted to direct environment-modifying operations: a substantial fraction of information-gathering commands can be connected to later verifier-relevant writes.

Indirect contribution is also non-negligible. Among positive-credit writes, 18.6% receive at least part of their credit through downstream write dependencies. This supports the distinction between direct persistence in verifier-inspected resources and earlier writes whose information is consumed by later relevant commands. The mean shortest distance from a credited read to its reachable writes is 1.74 dependency edges, showing that supporting information commonly extends beyond an immediately adjacent command.

E.4 Summary

Overall, the sensitivity analysis indicates that DepGPO remains effective across a range of dependency-decay coefficients, with moderate decay providing the strongest performance in the examined setting. The trace statistics further show that dependency-based redistribution is applicable to most training trajectories and frequently assigns credit to both relevant writes and supporting reads. At the same time, the additional computation remains small relative to the full terminal-agent training loop because the method operates on recorded execution traces and requires no additional model evaluations. Together, these results suggest that the dependency mechanism provides fine-grained credit assignment without introducing a large computational burden.

F Qualitative Credit-Assignment Examples

This appendix illustrates how DepGPO assigns credit in individual terminal trajectories. We present three representative cases covering supporting reads, indirect write contributions, and commands that receive no credit because they do not contribute to verifier-inspected resources. The first and third examples are rollouts of the Qwen3.5-9B agent used in our pilot study, while the second is a reference solution executed under the same tracer. All command credits are computed using the same implementation and $\beta = 0.5$ as in training.

The tables show selected commands rather than complete trajectories. Raw command credit $q(c)$ is computed from the complete recorded trace. During training, these credits are subsequently aggregated into step factors and token-normalized weights as described in Appendix A.

F.1 Supporting Reads

The first example shows why a command can receive credit even when it does not modify the environment. In *count-dataset-tokens*, the agent must compute the number of tokens in a specified subset of a dataset and write the result to `/app/answer.txt`. In the trajectory shown below, a Python command computes and prints `63841`, after which the next command writes that value to the answer file. The verifier expects a different total, so the trajectory fails. One of the four trajectories in its rollout group succeeds, giving this trajectory a negative group-relative advantage of $\hat{A} = -0.58$.

Table 12: Supporting-read example from *count-dataset-tokens*. Only the commands relevant to this dependency are shown.

Step	Command	Role	$q(c)$
t_{17}	`python3 <<'EOF' ... EOF`	Supporting read	0.50
t_{18}	`echo "63841" > /app/answer.txt`	Relevant write	1.00
t_{19}	`cat /app/answer.txt`	Read with no downstream use	0

Although t_{17} records no file write, its standard output is used directly by the following write:

$$t_{17} \xrightarrow{\text{stdout: } 63841} t_{18} \longrightarrow E.$$

Because t_{18} receives full write credit, t_{17} receives supporting-read credit

$$q(t_{17}) = \beta\, q(t_{18}) = 0.5.$$

By contrast, t_{19} reads the answer only after it has been written. No later positive-credit write depends on this read, so it receives zero credit. The example therefore distinguishes a read that helps produce a relevant write from a read that merely inspects the resulting state.

Because the trajectory has a negative group-relative advantage, the dependency-derived factors do not turn these credits into rewards. Instead, they determine where the negative learning signal is concentrated. The direction of the update still comes from the trajectory outcome.

F.2 Indirect Write Contribution

The second example shows why a write can matter even when the file it creates is never inspected by the verifier. In *bn-fit-modify*, the task requires learning and modifying a Bayesian network and producing three CSV files checked by the verifier. The reference solution first writes an R script, `modify_bn.R`, and then executes that script to produce the required outputs.

The script written by t_3 is not itself in E. However, all 102 tracked lines are still available when t_4 reads the script, and t_4 writes different files that reach E. The resulting dependency is

$$t_3 \xrightarrow{\text{file: } \texttt{modify_bn.R}} t_4 \longrightarrow E.$$

All tracked lines written by t_3 therefore qualify as indirect contributions:

$$\mathrm{Contrib}(t_3) = \frac{102}{102} = 1.$$

Table 13: Indirect-write example from *bn-fit-modify*. The trajectory is a reference solution and therefore has no rollout-group advantage.

Step	Command	Contribution type	$q(c)$
t_2	`R -e '...install_version("bnlearn", ...)'`	No recorded dependency	0
t_3	`cat > modify_bn.R <<'EOF' ... EOF`	Indirect write	1.00
t_4	`Rscript modify_bn.R`	Direct write	1.00

The execution step t_4 receives direct write credit because its outputs are among the resources inspected by the verifier.

The package-installation command in t_2 modifies only system-library locations outside the traced task resources and has no recorded dependency to E, so it receives zero credit.

This case captures an important pattern in terminal workflows. Scripts, temporary files, and intermediate configurations may disappear before verification while still determining the contents of the final outputs. Restricting credit to writes that persist directly in E would miss such intermediate contributions.

F.3 Overwritten and Irrelevant Commands

The final example illustrates how a successful trajectory can contain both useful and unnecessary operations. In *constraints-scheduling*, the agent must read three calendars and write a meeting satisfying the required constraints to `/app/meeting_scheduled.ics`. Two of the four trajectories in the rollout group succeed, giving the trajectory shown here a positive group-relative advantage of $\hat{A} = +1.00$.

During the trajectory, the agent writes an initial version of the output, reads it back, deletes it, writes another version, reads that version, and later writes the final output.

Table 14: Example of verifier-relevant and irrelevant commands from *constraints-scheduling*.

Step	Command	Dependency outcome	$q(c)$
t_4	`cat <<'EOF' > .../meeting_scheduled.ics`	Deleted before contribution	0
t_5	`cat /app/meeting_scheduled.ics`	No downstream relevant write	0
t_6	`rm /app/meeting_scheduled.ics`	No credited output	0
t_7	`printf '...' > .../meeting_scheduled.ics`	Reaches E	1.00
t_8	`cat /app/meeting_scheduled.ics`	No downstream relevant write	0
t_{10}	`cat <<'EOF' > .../meeting_scheduled.ics`	Reaches E	1.00

The first version written in t_4 is removed before it contributes to a qualifying downstream write or remains in the final inspected state. It therefore receives zero credit. Similarly, the read-backs in t_5 and t_8 inspect the agent's own output but do not support a later positive-credit write.

In contrast, the relevant records produced by t_7 and t_{10} reach the final verifier-inspected file and receive full write credit. Thus, even within a successful trajectory, DepGPO does not automatically treat every action as equally responsible for the outcome.

This distinction is particularly important for long terminal trajectories, which often contain exploration, discarded edits, repeated inspections, and failed intermediate attempts. Uniform trajectory-level credit assigns the same positive signal to all of these operations, whereas dependency-aware credit separates commands connected to the verified state from those without such recorded connections.

F.4 From Command Credit to the Training Signal

The three examples illustrate complementary parts of the credit-assignment mechanism. Direct write credit identifies modifications that reach verifier-inspected resources. Indirect write credit recovers

earlier modifications whose information is consumed by later relevant writes. Supporting-read credit extends attribution to information-gathering commands that enable those writes.

Raw command credits are first aggregated within each interaction step. DepGPO then normalizes the resulting step credits into nonnegative step factors and performs the token-level normalization described in Appendix A. If $w_{i,t}$ is the resulting final weight, the advantage assigned to tokens in step t is

$$\hat{A}_{i,t}^{\mathrm{Dep}} = w_{i,t}\hat{A}_i.$$

The dependency-derived weights are nonnegative, so they change the magnitude but not the sign of the original group-relative advantage. When $\hat{A}_i > 0$, larger dependency weights strengthen reinforcement on the corresponding steps; when $\hat{A}_i < 0$, they strengthen the penalty. If $\hat{A}_i = 0$, all redistributed advantages remain zero. Among traced steps, lower dependency credit leads to smaller relative factors, while steps without traced operations retain the documented default factor.

Taken together, these examples show that DepGPO does not introduce a separate notion of task correctness at the command level. The terminal outcome still determines whether the trajectory is reinforced or penalized. Execution dependencies instead determine how that outcome-derived signal is distributed across the commands that produced the final state.