SIGNAL RESHAPING FOR GRPO IN WEAK-FEEDBACK AGENTIC CODE REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Code-agent RL often operates under weak online execution feedback: rollout-time signals are reliable and executable, but capture only necessary conditions rather than the target semantic predicate. Using agentic compile-fix as the setting, we study signal reshaping for GRPO with offline reference-conditioned reward construction. Our central claim is that GRPO's within-group comparison is meaningful only after three kinds of signals are reshaped: outcome rewards recover semantic ranking, process signals localize intra-trajectory credit, and rollouts from the same prompt remain execution-comparable. We operationalize these conditions by reshaping signals around GRPO: compile-and-semantic layered rewards reshape trajectory ranking, step-level process scores placed outside group reward normalization act as token loss weights to reshape intra-trajectory update strength, and failure-cause-aware rollout governance preserves within-group comparability. Experiments show a clear end-to-end gain: full signal-reshaped GRPO improves strict compile-and-semantic accuracy from the base model's zero-shot 0.385 to 0.535. Controlled comparisons further explain the source of this gain: binary rewards remove the compile-only middle tier and degrade trajectory control; on top of layered rewards, process-score weighting further improves accuracy from 0.48 to 0.53 and reduces average evaluation steps from 23.50 to 17.02. As a boundary comparison, token-level privileged self-distillation underperforms step-level process weighting, showing that local distribution matching cannot replace action-step credit.

1 INTRODUCTION

Code large language models (LLMs) are moving from one-shot code generation toward interactive software repair in real repositories (Jimenez et al., 2024; Zan et al., 2025; Yang et al., 2024; Wei et al., 2025; Li et al., 2026b). A common paradigm trains the model as a code agent that interacts with repositories through ReAct-style tool use (Yao et al., 2023) and updates its policy from execution feedback with group-relative reinforcement learning, such as Group Relative Policy Optimization (GRPO) (Shao et al., 2024; Luo et al., 2025; Yang et al., 2025; Golubev et al., 2025). GRPO relies on relative rankings among same-prompt rollouts; if these rankings no longer mainly reflect repair progress, advantage updates can optimize noise or surface shortcuts. Agentic repair makes this premise nontrivial: trajectories are no longer pure text sequences, but long interactions composed of context reading, code editing, compilation checks, and explicit termination; rollout outcomes are also jointly determined by tool definitions and execution resources. This means that execution-environment jitter, and whether execution feedback is sufficient to characterize true repair success, can affect the optimization target itself.

This within-group comparison problem is especially pronounced in *weak-feedback agentic repair*, where the policy receives only weak online execution feedback during rollout, while developer references and semantic judges are available only offline during training. Online feedback checks only necessary or surface conditions, not the target semantic predicate. We use compile-fix as the study setting: an uncompilable patch cannot be correct, but a compiling patch may still delete needed code, add an unsafe stub, or bypass the failing path. During training, no executable tests are available, and literal similarity to a reference patch is not correctness. Thus, neither the test oracle in SWE-bench (Jimenez et al., 2024) nor the oracle-patch diff reward in SWE-RL (Wei et al., 2025) can serve directly as an online training reward. Compile-fix is therefore a typical weak-feedback task: execution feedback is informative and necessary, but not sufficient, for semantic correctness.

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.

Used directly for GRPO's same-prompt ranking, within-group comparison can drift from true repair progress. Signal reshaping aims to realign these comparisons with repair intent.

We view weak-feedback agentic repair first as a *signal reshaping* problem, not a call to rewrite the reinforcement-learning objective. GRPO relies on same-prompt rollout comparison, but this comparison is useful only when reward differences mainly arise from policy behavior. In our setting, this requirement yields three signal conditions: (i) **outcome-semantics failure**, where stable online signals often capture only surface acceptability and, when used as $\{0, 1\}$ rewards, induce syntactic shortcuts (reward hacking); (ii) **diffuse process credit**, where a long-horizon outcome reward cannot separate effective repair steps from uninformative repetition, diluting the semantic signal; and (iii) **broken within-group comparability**, where infrastructure jitter, such as compilation queues or container initialization failures, and policy degeneration, such as no tool calls or catastrophic repetition, can mix with genuine repair failures in the same zero-reward bin and distort rankings. Thus, signal reshaping in weak-feedback agentic reinforcement learning reduces to two requirements: *(A) rewards must reflect repair intent at the outcome level and support useful process credit; and (B) rollouts entering within-group comparison must be execution-comparable.* We call modifying training signals around the existing GRPO update *signal reshaping*.

We use signal reshaping as a minimal intervention to test these requirements. A layered compile-and-semantic reward assigns $0/0.5/1$ to non-executable, compile-only, and semantically correct repairs, restoring outcome ranking across trajectories. Step-level process scores serve as token-level loss weights outside group normalization to redistribute update strength within each rollout. Failure-cause-aware rollout governance routes non-learnable exceptions before incomparable samples pollute same-group ranking. The method does not alter GRPO's advantage estimator or clipping operator; it places reshaped signals around them.

Experiments test signal reshaping layer by layer. End-to-end, full signal-reshaped GRPO improves strict compile-and-semantic accuracy from the base model's zero-shot 0.385 to 0.535. Controlled comparisons first test outcome semantics: a pure compilation reward makes the compile rate spike early and then fall back, without improving compile-and-semantic correctness. Under fixed rollout governance, layered rewards preserve roughly 0.30 mass at the intermediate $R = 0.5$ tier, whereas removing this tier jointly degrades trajectory control and termination. From the same post-fork checkpoint, step-level process scores improve accuracy from 0.48 to 0.53 and reduce average evaluation steps from 23.50 to 17.02. Independent-judge and blind-human evaluations corroborate this gain, while an additive-reward control rules out trajectory shortening as the primary cause (Section 4.3, Appendices C.2–C.6).

These results show that signal reshaping can recover trainable GRPO comparisons. As a boundary comparison, we test whether token-level dense supervision can replace step-level process credit via privileged-prompt distillation. Both privileged-prompt π-Distill and on-policy self-distillation (OPSD) (Penaloza et al., 2026) underperform the stable GRPO baseline: π-Distill drifts toward premature `finish`, while OPSD exhibits tool-level degradation. This boundary result shows that Kullback–Leibler (KL)-style local distribution matching cannot replace action-step credit attribution in long trajectories; see Appendix D.2.

Figure 1 summarizes the framework: a weak-feedback execution environment produces multi-turn rollouts; layered rewards reshape outcome ranking, step-level process scores reshape intra-trajectory credit, and rollout governance reshapes within-group comparability. The π-Distill and OPSD (Penaloza et al., 2026) branch on the right provides a boundary comparison for the token-level KL path.

Our main contributions are:

- We formulate GRPO training in weak-feedback agentic repair as a signal-reshaping problem for within-group comparison, identifying three signal conditions: semantically ordered outcomes, localized process credit, and execution-comparable same-prompt rollouts.
- We present a decoupled signal-reshaping framework that separates supervision levels: layered rewards restore outcome ranking, step-level process scores outside group normalization reallocate token update strength, and failure-cause governance keeps rollouts comparable.
- We validate these reshaped signals in compile-fix: layered rewards restore stable training, step-level process scores improve accuracy across training, independent, and human judges, and token-level privileged distillation highlights the boundary of local distribution matching.

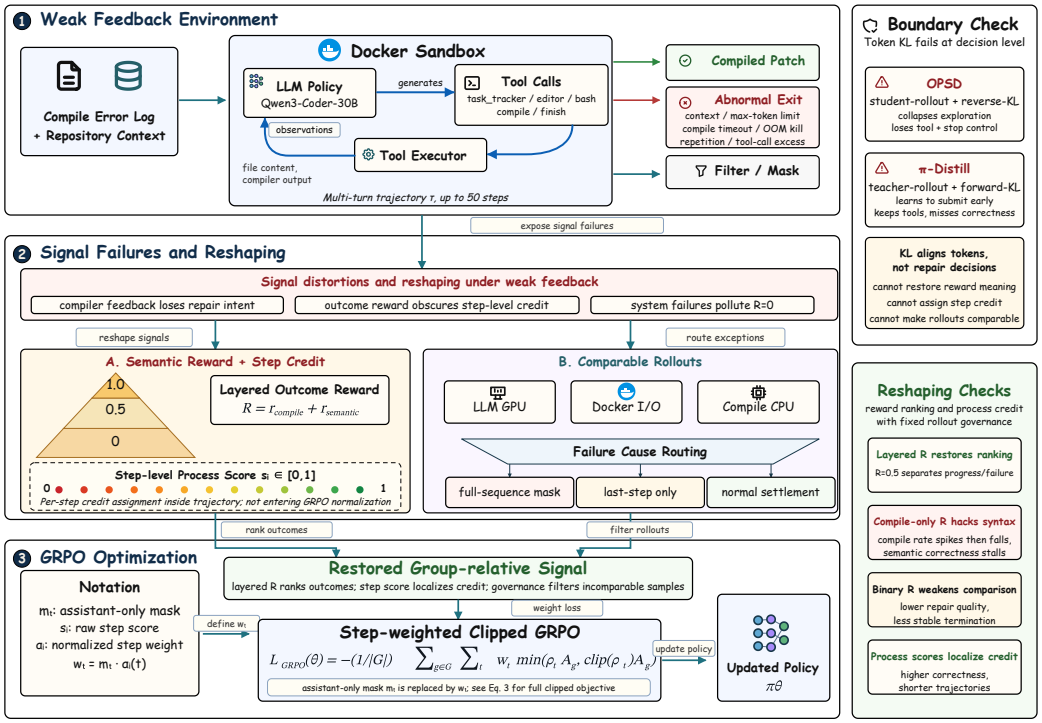


Figure 1: Signal-reshaping framework for GRPO in weak-feedback agentic repair: layered rewards reshape outcome ranking, step-level process scores reshape intra-trajectory credit, and rollout governance reshapes within-group comparability.

The rest of the paper formalizes the setting (Section 2), presents the construction (Section 3), and tests the predictions (Section 4). Related work, limitations, prompts, and the full distillation analysis appear in the appendix.

2 Preliminaries

This section fixes notation for weak feedback, multi-turn trajectories, assistant-only masks, GRPO, and token-level OPD.

Weak feedback. Let $C(\tau)$ denote the execution-level signal observable during rollout and $S(\tau)$ denote the target semantic success predicate. Feedback is *weak* when C is reliable and available online, but captures only a necessary condition for S, not a sufficient one. In compile-fix, C is compilation success: $C(\tau) = 0$ rules out a valid repair, but $C(\tau) = 1$ does not guarantee that the patch repairs the intended error. Semantic correctness S must therefore be supplied by an offline reference-conditioned judgment. The policy sees only compiler feedback during rollout; ground-truth repairs are never visible in the rollout context and are used strictly offline after generation. The issue is not observation noise, but signal incompleteness: online execution feedback is informative, yet insufficient to define the task predicate that the policy should optimize.

2.1 Multi-turn Agentic RL and Assistant-Only Mask

We model agentic code repair as a finite-horizon Markov decision process (MDP) $\langle \mathcal{S}, \mathcal{A}, D, R, T \rangle$. The state $\mathcal{S}$ contains repository state and conversation history; $\mathcal{A}$ contains natural-language outputs and tool calls; D is the external runtime; R is observed after termination; and T is the maximum number of interaction steps.

Given an initial prompt and repository state x, the policy π_θ interacts with the environment to produce a multi-turn trajectory

$$\tau = (a_1, o_1, a_2, o_2, \ldots, a_{T'}, o_{T'}), \tag{1}$$

where a_i are model actions, o_i are environment observations, and $T' \leq T$. The objective is to maximize $\mathbb{E}_\tau[R(\tau)]$.

Assistant-only mask. Serialized trajectories contain model generations and environment echoes, but only model tokens are determined by π_θ. Let $m_t \in \{0, 1\}$ mark whether token t belongs to an assistant generation; $m_t = 1$ tokens enter the policy gradient, while $m_t = 0$ echoes are masked (Jin et al., 2025; Qian et al., 2025).

2.2 GRPO

GRPO (Shao et al., 2024) avoids an explicit value function by comparing multiple samples from the same prompt. For each prompt, K trajectories $\{\tau_k\}_{k=1}^K$ are sampled, and scalar rewards R_k are group-normalized into trajectory-level advantages:

$$\hat{A}_k = \frac{R_k - \bar{R}}{\mathrm{std}(\{R_i\}_{i=1}^K)}, \qquad \bar{R} = \tfrac{1}{K}\textstyle\sum_{i=1}^K R_i, \tag{2}$$

which are broadcast to all assistant tokens in the corresponding trajectory, denoted by $\hat{A}_t \equiv \hat{A}_{k(t)}$. The policy is updated with the clipped policy-gradient objective:

$$\mathcal{L}_{\text{GRPO}}(\theta) = -\,\mathbb{E}_{x,\,\{\tau_k\}_{k=1}^K}\left[\frac{\sum_t m_t\,\min\left(\rho_t\hat{A}_t,\ \mathrm{clip}(\rho_t,\,1-\epsilon_{\text{lo}},\,1+\epsilon_{\text{hi}})\,\hat{A}_t\right)}{\max(1,\ \sum_t m_t)}\right], \tag{3}$$

where y_t is response token t and $\rho_t = \rho_t(y_t) = \pi_\theta(y_t \mid \cdot)/\pi_{\theta_{\text{old}}}(y_t \mid \cdot)$. KL and entropy regularization are added in the standard way (full form in Appendix C.4).

Within-group comparison prerequisites. Equations (2)–(3) assume that R is a semantically meaningful measure of task success and that the K trajectories sampled from the same prompt are comparable at the execution level. In weak-feedback, long-horizon agentic tasks, these assumptions need not hold; terminal advantages are also broadcast over many assistant tokens, motivating process-level credit assignment.

2.3 Token-Level On-Policy Distillation with Privileged Prompts

OPD (Agarwal et al., 2024) matches a teacher distribution on trajectories from the current policy. We use privileged-hint π-Distill and OPSD (Penaloza et al., 2026) as boundary comparisons: π-Distill transfers from hint-conditioned teacher trajectories to a hint-free student, whereas OPSD scores hint-free student trajectories with the hint-conditioned teacher. Both inject masked-sum token KL as a GRPO reward penalty, with directions $D_{\text{KL}}(\pi^T \| \pi^S)$ and $D_{\text{KL}}(\pi^S \| \pi^T)$. This path assumes teacher-preferred behavior aligns with student-feasible behavior and that critical decisions are not diluted by low-information tokens. Full objectives are in Appendix D.3; the corresponding empirical analysis is in Appendix D.2.

3 Method

This section presents the signal-reshaping construction: layered rewards reshape outcome ranking, step-level scores reshape intra-trajectory credit, and rollout governance reshapes within-group comparability. Table 1 maps the three components to the corresponding signal distortions and reshaping mechanisms.

3.1 Task Formalization

Following Section 2.1, each compile-fix sample contains a failure report, repository context, tool interfaces, and a developer repair from which we derive the ground-truth (GT) repair; the GT is not visible during rollout and, in the absence of executable tests, serves as a reference-conditioned semantic proxy for reward construction. The agent runs in a Docker sandbox with file editing (`str_replace_editor`), shell (`bash`), incremental compilation (`compile_code`), task tracking (`task_tracker`), and termination (`finish`) tools; details are in Appendix C. At each step, the policy acts until termination or the step limit.

In this setting, the availability of training signals depends on whether a trajectory can be executed to completion. This condition is coupled with system constraints such as compilation queues,

Table 1: Signal conditions, signal distortions, and reshaping mechanisms.

Condition	Signal distortion	Reshaping mechanism	Section
(A) Semantic reward signal	Surface acceptability conflates true repair	Layered reward $R = r_{\text{compile}} + r_{\text{semantic}}$; compile-only ≤ 0.5	§3.3
	Outcome advantage cannot localize key steps	Step score s_i induces token weight $w_t = m_t \alpha_{i(t)}$	§3.3
(B) Comparable within-group execution	System bottlenecks reshape sampling	Decouple concurrency by resource type	§3.5
	Abnormal exits and genuine failures are mixed into 0-reward failures	Route by failure cause: mask, retain last step, or terminate normally	§3.5

container startup, and context budgets; as a result, execution-level comparability within a group is not guaranteed by default.

3.2 Training Data Construction

The training corpus comes from compilation-failure events in an internal build system. Each raw record contains a static repository snapshot, build configuration, compilation error log, and developer repair. Directly using the developer repair as a supervision target introduces two sources of noise: (i) human commits often merge the error fix with feature changes, refactoring, or style cleanup, so only a small subset of changes actually eliminates the error; and (ii) the same root cause, such as a cascading upstream interface change, can appear repeatedly in near-duplicate form, letting a few frequent errors dominate the data distribution.

We therefore construct *minimal sufficient repairs* through the following steps: (i) **Repair snapshot alignment**: align each failed build with the triggering commit and the subsequent repair commit, extract a normalized unified diff, and obtain a tuple of (compilation error, developer repair, build environment) with repository coordinates; (ii) **Structuring and filtering**: deduplicate by error-log fingerprints, classify errors into coarse/fine categories covering syntax, type, declaration, function, linking, and related failures for stratified sampling, and remove pure-deletion repairs that are more often refactoring than error repair; (iii) **Minimal sufficient repair refinement**: constrain each developer repair to a strict subset that is both necessary and sufficient, removing only irrelevant changes and introducing no new rewrites, and verify it end-to-end through isolated incremental compilation. The resulting minimal sufficient repair is the GT repair used in later semantic judgment and distillation, keeping unrelated changes from contaminating the semantic judgment reference and providing a clean abstraction source for distillation; (iv) **Recent-change clues**: for each reported source file, trace the most recent pre-failure diff, which often identifies root causes such as interface changes, and provide it in both training and evaluation prompts; and (v) **Split**: stratify by primary error type and refined repair size to obtain disjoint training and evaluation sets, then further filter the training set to remove overly simple samples, reducing cases where all K rollouts within a group succeed simultaneously and cause advantage collapse. After refinement, each minimal sufficient repair is called the GT repair and serves as the semantic judgment reference rather than a token-level target for imitation.

3.3 Layered Rewards and Step-Level Process Scores

The build system indicates whether code is accepted, not whether the original error is correctly repaired. Used directly as a binary GRPO reward, it can rank shortcuts such as commenting out error sites, stubs, or unconditional `return` statements together with genuine fixes (Amodei et al., 2016; Skalse et al., 2022). The reward must therefore separate three cases that binary compilation feedback conflates: no executable repair, executable but semantically wrong repair, and executable semantically correct repair.

Scalar reward: layered $\{0, 0.5, 1\}$.

$$R(\tau) = r_{\text{compile}}(\tau) + r_{\text{semantic}}(\tau), \tag{4}$$

where $r_{\text{compile}} \in \{0, 0.5\}$ indicates whether incremental compilation succeeds, and $r_{\text{semantic}} \in \{0, 0.5\}$ is assigned by a strong model (Kimi-K2.5 (Kimi Team, 2026)) that compares the semantic consistency

between the agent patch and the GT repair patch. The criterion is functional equivalence to the GT repair, not lexical similarity. Semantic checking is performed only after compilation succeeds: a compilation failure receives $R = 0$, a compilation success with semantic inconsistency receives $R = 0.5$, and a compilation success with semantic consistency receives $R = 1$.

Separating compilation success from semantic alignment and assigning 0.5 to each has two effects. First, compile-only shortcuts receive at most partial credit, so full reward is reserved for repairs consistent with the GT repair. Second, the intermediate 0.5 level preserves within-group ranking signal early in training, when fully semantic repairs are still sparse.

Step-level process scores. The scalar reward R is sparse: critical edits and repetitive operations inherit the same trajectory advantage. Motivated by process supervision for long reasoning and program generation (Lightman et al., 2024; Wang et al., 2024; Dou et al., 2024), we introduce a step-level score $s_i \in [0, 1]$ for directional correctness and information gain.

The design follows three constraints: (i) s_i only redistributes gradient *magnitude* and does not enter GRPO scalar-reward normalization; (ii) scores are assigned only to trajectories that compile successfully, avoiding attribution on non-executable code; and (iii) the weights are applied on top of the assistant-only mask m_t. In the simplified view, each step score is converted into a loss weight relative to the trajectory average, then written to that step's assistant-token mask. Concretely, let n_i be the number of active assistant tokens in step i and

$$\bar{s} = \frac{\sum_i n_i s_i}{\sum_i n_i}, \qquad \alpha_i^+ = \frac{s_i}{\bar{s}}, \qquad w_t = m_t\, \alpha_{i(t)}. \tag{5}$$

Positive-advantage trajectories use $\alpha_i = \alpha_i^+$. Negative-advantage trajectories use an affine sign-aware inversion $\alpha_i = c\, \max(2 - \alpha_i^+, 0.1)$, with c enforcing $\sum_i n_i \alpha_i = \sum_i n_i$. Centering at 1 preserves mean weight before clipping, while the heuristic 0.1 floor prevents non-positive weights. This reverses step priority under negative advantages, penalizing low-scoring steps more while sparing sound intermediate steps, without altering total trajectory update scale. Steps scored 0 naturally receive $w_t = 0$, acting as a sparse gate that suppresses uninformative actions (Appendix C.7).

Scoring criteria and information gain. Process scores are emitted step by step by a strong model conditioned on the full trajectory context. The judge follows the same semantic-equivalence, functional-consistency, and behavior-preservation criteria as outcome judging, but lowers the comparison granularity from the full patch to a single tool call. To suppress scoring drift, the judge focuses only on modifications related to the target compilation error and explicitly ignores unrelated changes in the GT. It also follows an information-gain principle: low-gain steps such as repeatedly viewing already inspected files are penalized, while steps that trigger compilation verification receive a higher base score to encourage the policy to seek executable feedback.

3.4 GRPO Optimization

The two-level supervision retains GRPO's group advantage estimator and clipping operator, replacing uniform token aggregation m_t with score-weighted aggregation $w_t = m_t \alpha_{i(t)}$ from Eq. (5). The surrogate objective therefore changes to redistribute token gradients across steps. Disabling process scores sets $\alpha_{i(t)} \equiv 1$, recovering the assistant-only mask.

The configuration follows two principles. First, each prompt samples $K = 8$ trajectories with rollout temperature 1 to keep within-group candidates diverse. Second, asymmetric clipping, with a looser upper than lower bound, follows large-scale GRPO/RL stabilization practice (DeepSeek-AI et al., 2025; Yu et al., 2025): it permits larger updates toward beneficial exploration while making degenerate updates more conservative. A small λ_{KL} anchors the policy to the frozen reference model; details are in Appendix C.4.

GRPO itself is not the algorithmic contribution; the contribution lies in decoupled signal placement: GT repairs provide offline semantic references, layered R governs trajectory-level ranking, s_i acts outside group normalization as token loss weights, and governance ensures rollout comparability.

3.5 Rollout Governance

Effective GRPO learning also requires that the K rollouts in a group have the same kind of execution opportunity. Prior work on multi-turn agentic RL shows that environment echoes, tool calls, and

rollout governance directly affect the stability of learnable signals (Jin et al., 2025; Qian et al., 2025; Wang et al., 2025b); in compile-fix, this issue is amplified by container setup and compilation queues.

Decoupling sampling concurrency by execution-resource dimension. Rollouts consume three heterogeneous resources: policy inference on GPUs, sandbox operations under host scheduling, and incremental compilation under CPU and memory pressure. A single concurrency controller lets compilation dominate latency, while excessive compilation concurrency causes contention and timeouts. We therefore decouple the three resource types and tighten the compilation limit, preventing compilation from becoming the rollout bottleneck while improving resource utilization.

Routing trajectories by failure cause. Without failure-cause routing, 0-reward samples mix repair failures, policy degeneration, and trajectories without valid execution opportunity. We separate *unlearnable exceptions* from *informative failures* by exit source and use three handling strategies:

- **Full-trajectory masking**: exceptions that have no causal relation to model decisions and therefore should not enter the training objective. These include environment initialization failures; context overflow during generation, inference interruption, truncation caused by reaching the per-step output limit, and early termination triggered by repeated compilation timeouts; as well as final compilation exceptions caused by the execution environment rather than repair semantics, including compilation timeouts and out-of-memory cases where the compilation process is killed by the system. The entire response is removed from the training objective.
- **Last-step retention**: patterns that arise from sampling itself and whose degeneration signal is concentrated at the end of the trajectory. Currently these include out-of-bound concurrent tool calls within a single turn and catastrophic repetition, both of which are detected during generation and trigger early termination. For such samples, the preceding trajectory is masked and only the final step is retained as a penalty sample, aligning negative feedback with the position that actually reflects policy degradation.
- **Normal termination**: all remaining valid trajectories proceed to compilation and semantic judgment.

To keep the group size K fixed without deleting valid updates, fully masked slots remain in normalization at $R = 0$ with an all-zero loss mask. This shifts the baseline μ slightly downward while strictly preserving pairwise ranking among valid rollouts (Appendix C.5).

4 Experiments

This section evaluates the signal reshaping from Sections 2–3. As the end-to-end result, the full two-stage GRPO trajectory improves strict compile-and-semantic accuracy from 0.385 at the first evaluation point (`rollout/step`$= 0$, i.e., the base model zero-shot) to 0.535, an absolute gain of $+15.0$pp. Q1 tests whether layered rewards provide learnable trajectory-level ordering; Q2 then tests whether step-level process scores improve intra-trajectory credit assignment.

4.1 Research Questions and Compared Systems

We do not ablate rollout-governance components individually. Preliminary runs showed that disabling any single component often breaks execution comparability early in training, causing training collapse or unstable convergence and making reward and credit-assignment effects hard to interpret cleanly. We therefore keep the rollout pipeline fixed and isolate two training-signal conditions:

1. **Q1 (Reward semantics).** Are compile-and-semantic layered rewards the reward-semantic condition needed for stable GRPO training?
2. **Q2 (Process credit).** Once layered rewards produce learnable trajectories, can step-level process scores further improve GRPO updates through finer-grained intra-trajectory credit assignment?

The four GRPO configurations share the same base model, rollout system, data split, and training hyperparameters, and differ only in reward form and whether step-level process scores are used:

- **GRPO-compile-only**: uses a further weakened binary reward ($0/1$, $R = 1$ iff compilation succeeds), removing semantic checks from the reward.
- **GRPO-early**: layered rewards ($0/0.5/1$) with the standard assistant-only loss mask.
- **GRPO-full (main method)**: continued from the GRPO-early fork point with the same layered reward plus step-level process-score weighting.

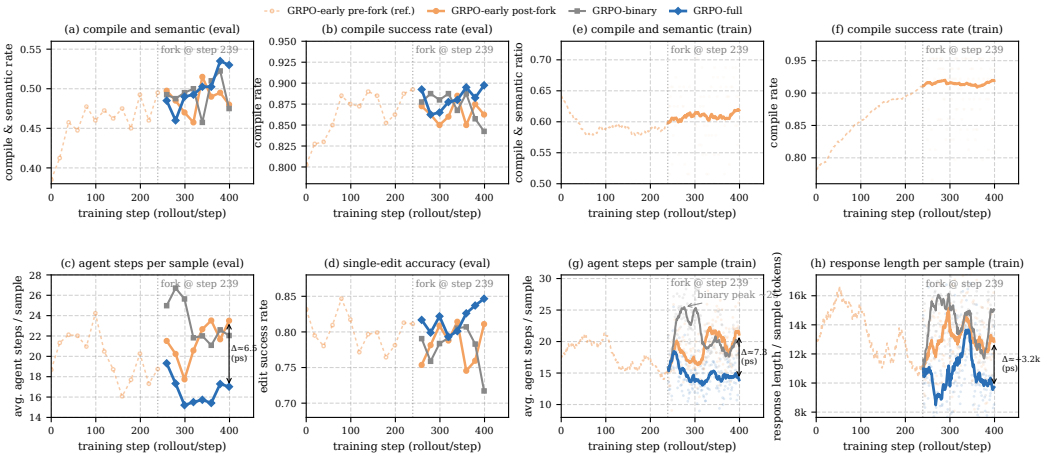
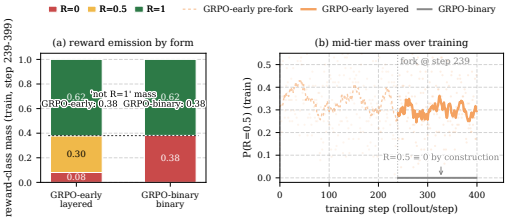
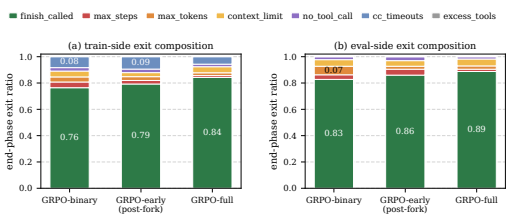


Figure 2: Four GRPO settings on a unified `rollout/step` axis. GRPO-early/binary/full share the fork at `rollout/step`= 239; GRPO-compile-only and GRPO-early (pre-fork segment) both start independently from the base model. Panels (a)–(d): fixed 400-sample evaluation; (e)–(h): train-time signals; curves use a 20-step centered rolling average.



(a) Reward emission for layered vs. binary rewards. Left: reward-level distribution in the shared window; right: middle-tier training dynamics.



(b) Exit composition at the tail of the shared window. Left: training rollouts; right: 400-sample evaluation split.

Figure 3: Mechanism evidence for outcome-level reward semantics: (a) binary rewards remove the $R = 0.5$ tier; (b) weaker comparison semantics coincide with worse termination.

- **GRPO-binary**: continued from the same fork point with binary rewards ($0/1$, $R = 1$ iff compilation succeeds and the repair is semantically correct), collapsing compile-only partial success into failure.

Protocol summary. All methods use the Docker-based compile-fix environment in Section 3.1, Qwen3-Coder-30B-A3B-Instruct (Qwen Team, 2025) as the base model, a 1110-sample training set, and a fixed 400-sample hint-free evaluation split; full hyperparameters are in Appendix C.4. Both splits are difficulty-controlled to focus on non-trivial repairs; Appendix C.1 details the filtering stages and split composition.

4.2 Q1: Layered Rewards Support Stable Training

E1 (Reward semantics): binary rewards remove the middle signal needed for within-group comparison. In the shared post-fork window, layered GRPO assigns about 0.30 reward mass to $R = 0.5$, corresponding to executable but semantically incomplete repairs; binary GRPO lacks this middle tier and folds these samples into $R = 0$. Figure 3(a) shows that the middle tier preserves the ordering information between compile-level progress and complete failure.

E2 (Binary-reward degradation): repair quality decreases with less stable termination behavior. At the end of the post-fork comparison, GRPO-binary reaches 0.475 compile-and-semantic accuracy, below GRPO-early at 0.480. Its auxiliary metrics decline in the same window: compilation success drops from 0.8775 to 0.8425, and single-edit success[1] drops from 0.791 to 0.717. Figure 2(a)–(d) and (g)–(h) show that lower quality is accompanied by longer trajectories; Figure 3(b) gives the same diagnosis: GRPO-binary has the lowest normal-termination ratio and a higher abnormal-exit burden than GRPO-full.

[1] Single-edit success is the fraction of `str_replace_editor` calls that apply an exact string replacement and return success; it measures tool-level edit execution quality.

E3 (Execution-signal reward hacking): pure compile-only binary rewards push the policy toward syntactic shortcuts.
GRPO-compile-only and GRPO-early share the pre-fork window; only the reward differs, determined solely by *is_compiled*. The optimized compile rate spikes early and then declines, with less stability than GRPO-early (Figure 2(f)); meanwhile, compile-and-semantic correctness stagnates or slightly decreases on training and evaluation signals (Figure 2(a,e)), and tool-level single-edit success[1] also drops (Figure 2(d)). This matches the outcome-semantics failure from the introduction: execution feedback is necessary but insufficient for semantic correctness. Used directly as a $\{0, 1\}$ reward, it lets within-group comparison favor syntactic shortcuts that bypass semantic checks, leaving compile success without stable support. Adding semantic-level outcome signals strengthens the target, steadily improves compile-and-semantic success, and stabilizes compile success. Outcome rewards therefore need semantic-level reshaping.

E4 (Stable convergence): layered rewards and rollout governance provide a trainable comparison.
Layered rewards provide outcome semantics, and fixed rollout governance keeps within-group samples comparable; together, they support GRPO's within-group comparison. The GRPO-early training curves in Figure 2(e)–(f) enter a stable range after early variation, without sustained oscillation, divergence, or collapse toward low-success rollouts.

Thus, Q1 asks whether the task can be learned stably; Q2 tests whether step-level process scores further improve intra-trajectory credit assignment under this condition.

4.3 Q2: Process Scores Improve Correctness and Efficiency

This section isolates the second-stage mechanism. The GRPO-early post-fork segment and GRPO-full start from the same fork point and use the same layered scalar reward; the only additional signal in GRPO-full is the step-level process score applied to the loss mask.

E5 (Process-level credit assignment): step weighting improves correctness while shortening trajectories.
In the post-fork comparison, GRPO-full improves compile-and-semantic accuracy ($0.480 \rightarrow 0.530$, $+5.0$pp), compilation success ($0.8625 \rightarrow 0.8975$), and single-edit success[1] ($0.811 \rightarrow 0.847$) over GRPO-early, while average evaluation steps fall from 23.50 to 17.02 (Figure 2(g)–(h)). To rule out judge bias, three independent measurement channels corroborate this gain: the training judge yields $+5.00$pp, the independent GLM-5.2 judge (GLM-5 Team, 2026) achieves $+5.75$pp ($p = 0.018$), and a blind human audit on the natural 400-problem split reaches $+6.25$pp ($p = 0.008$; Appendix C.2). An additive-reward ablation with scalar return $R + \beta \operatorname{mean}(s_i)$ shortens trajectories even further yet underperforms GRPO-full by 8.5pp, isolating the token-reweighting interface as the causal mechanism (Appendix C.6).

E6 (Termination behavior): process scores shift the policy toward more stable normal termination.
In the train-side tail, GRPO-full has the highest `finish_called` ratio (0.842) and lowest `max_steps` ratio (0.014), ahead of GRPO-early ($0.793/0.027$) and GRPO-binary ($0.764/0.044$). The evaluation tail shows the same ordering, with GRPO-full at `finish_called` $= 0.888$ and `max_steps` $= 0.018$. The efficiency gain therefore reflects fewer steps and more normal termination within budget.

Together, Q1 and Q2 address two levels of signal reshaping: layered rewards first provide trajectory-level preferences, and step-level process scores then refine intra-trajectory credit. The main benefit does not come from changing GRPO's group-normalized scalar-reward update form, but from first providing comparison semantics and then improving token-level loss weighting over tool steps.

5 Conclusion

This paper treats weak-feedback agentic code repair as a *signal reshaping* problem: online execution feedback is reliable but semantically incomplete, so GRPO stays effective only while within-group comparisons remain meaningful. Our decoupling around the unchanged RL objective targets outcome ranking, process credit, and within-group comparability, and the evidence supports that ordering: pure compile rewards induce syntactic shortcuts; layered rewards restore outcome ranking and semantic stability; step-level weighting then improves correctness and efficiency across training, independent-judge, and human assessments. Controlled ablations attribute the gain to the step reweighting interface rather than trajectory shortening, while token-level privileged self-distillation fails to provide equivalent action-level credit. For agentic tasks with weak online feedback, reshaping signals around the RL objective provides a principled alternative to ungrounded surrogate rewards or mismatched token-level supervision.

Ethics Statement

This work uses compilation-failure records and developer repairs derived from an internal continuous-integration build system. No new human-subject data is collected: the blind human audit in Appendix C.2 is performed by the authors on model outputs, not on human participants, and therefore involves no crowdsourcing and no IRB-eligible activity. Project-specific paths, repository names, and infrastructure identifiers are anonymized in the manuscript, and all agent execution is confined to Docker sandboxes that never touch production systems.

More reliable weak-feedback code-repair agents can reduce developer effort on routine build failures and improve the maintainability of large software systems. At the same time, repair agents optimized only against shallow execution signals may produce behavior-changing repairs; the reward hacking documented in Section 4.2 is exactly this failure mode, and the signal-reshaping construction is designed to mitigate it. Practical deployment should remain staged and should retain sandboxing, provenance tracking, semantic validation, and human code review. A fuller discussion of limitations and societal considerations appears in Appendix B.

Reproducibility Statement

All components needed to reproduce the reported results are specified in the paper. The GRPO objective, layered reward, and step-level weighting rule are given in Eq. (2)–(3) and Eq. (5); the full training hyperparameters, base model, group size, and sampling budget are listed in Appendix C.4. The sandbox image, tool interface, and return conventions are documented in Appendix C, and the rollout-governance routing rules are given in Appendix C.5. The data-construction pipeline, including minimal-sufficient-repair refinement and its per-stage retention rates, is described in Section 3.2 and Appendix C.1, and the dataset and repository scale is reported in Appendix C.3. All prompts used for the policy, the semantic judge, the process-score judge, and the privileged-hint teacher are reproduced verbatim in Appendix E, and the distillation objectives are given in Appendix D.3. The judge used for reward construction is audited against a blind human annotation of 400 samples in Appendix C.2, so that readers can calibrate the measurement channel itself.

The underlying build logs and developer repairs originate from a proprietary internal build system and cannot be released. This limits exact replication of the training corpus; the refinement procedure, filtering criteria, and split construction are therefore specified in enough operational detail to rebuild an equivalent corpus from any compile-fix repository with comparable build instrumentation. Appendix B states this constraint explicitly.

Declaration of LLM Usage

LLMs are a core, non-standard component of the methodology of this paper, and their roles are stated throughout the main text rather than only here. Specifically: (i) the repair policy being trained is an LLM agent (Qwen3-Coder-30B-A3B-Instruct); (ii) the terminal semantic-correctness component r_{semantic} of the layered reward is produced by an LLM judge (Kimi-K2.5), whose reliability is audited against blind human annotation in Appendix C.2; (iii) the step-level process scores s_i used as token loss weights are emitted by an LLM judge; (iv) the privileged-hint teachers used in the π-Distill and OPSD boundary comparison are LLM variants conditioned on hint-augmented prompts; (v) an independent LLM judge (GLM-5.2) provides a second measurement channel in Section 4.3; and (vi) the minimal-sufficient-repair refinement pipeline in Section 3.2 uses an LLM to select a strict subset of each developer patch, with every output verified by compilation. All corresponding prompt templates are given in Appendix E.

In addition, the authors used an LLM as a general-purpose writing aid for grammar correction and phrasing of the manuscript. It was not used to generate research ideas, design experiments, produce or analyze results, or retrieve references. The authors verified every claim, number, and citation in the paper and take full responsibility for its content.

References

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=3zKtaqxLhW.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016.

Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. SEQUENCER: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2021. doi: 10.1109/TSE.2019.2940179. URL https://doi.org/10.1109/TSE.2019.2940179.

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

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Shirong Ma, Mingchuan Zhang, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. doi: 10.1038/s41586-025-09422-z.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4571–4585, 2024. doi: 10.18653/v1/2024.acl-long.251.

Yuqian Fu, Haohuan Huang, Kaiwen Jiang, Yuanheng Zhu, and Dongbin Zhao. Revisiting on-policy distillation: Empirical failure modes and simple fixes. *arXiv preprint arXiv:2603.25562*, 2026.

GLM-5 Team. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, Sergei Skvortsov, and Boris Yangel. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501*, 2025.

Yuki Ichihara, Yuu Jinnai, Tetsuro Morimura, Mitsuki Sakamoto, Ryota Mitsuhashi, and Eiji Uchibe. MO-GRPO: Mitigating reward hacking of group relative policy optimization on multi-objective problems. *arXiv preprint arXiv:2509.22047*, 2026.

Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1161–1173, 2021. doi: 10.1109/ICSE43902.2021.00107. URL https://doi.org/10.1109/ICSE43902.2021.00107.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Rwhi91ideu.

Kimi Team. Kimi K2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 21314–21328, 2022. doi: 10.52202/068431-1549.

Jia Li, Zhuangbin Chen, Yuxin Su, and Michael R. Lyu. VulKey: Automated vulnerability repair guided by domain-specific repair patterns. *arXiv preprint arXiv:2605.01769*, 2026a.

Jia Li, Yuxin Su, and Michael R. Lyu. From laboratory to real-world applications: Benchmarking agentic code reasoning at the repository level. *arXiv preprint arXiv:2601.03731*, 2026b.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, QIANG FU, Xiao Han, Yang Wei, and Deheng Ye. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=hjYmsV6nXZ.

Shih-Yang Liu, Xin Dong, Ximing Lu, Shizhe Diao, Peter Belcak, Mingjie Liu, Min-Hung Chen, Hongxu Yin, Yu-Chiang Frank Wang, Kwang-Ting Cheng, Yejin Choi, Jan Kautz, and Pavlo Molchanov. GDPO: Group reward-decoupled normalization policy optimization for multi-reward RL optimization. *arXiv preprint arXiv:2601.05242*, 2026.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-Zero-Like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Meiqi Guo, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, and Abhinav Rastogi. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*, 2024.

Michael Luo, Sijun Tan, Colin Cai, Roy Hao, Lambda Lu, Tarun Venkat, Tianjun Zhang, Manan Roongta, Tianhao Wu, Justin Wong, Ion Stoica, and Raluca Ada Popa. DeepSWE: Training a fully open-sourced, state-of-the-art coding agent by scaling RL. Together AI Blog, 2025. URL https://www.together.ai/blog/deepswe. Agentica Project, UC Berkeley Sky Computing Lab and Together AI.

Yuanjie Lyu, Chengyu Wang, Jun Huang, and Tong Xu. From correction to mastery: Reinforced distillation of large language model agents. *arXiv preprint arXiv:2509.14257*, 2025.

Youssef Mroueh. Reinforcement learning with verifiable rewards: GRPO's effective loss, dynamics, and success amplification. *arXiv preprint arXiv:2503.06639*, 2025.

Zhongqiang Pan, Chuanyi Li, Wenkang Zhong, Yi Feng, Bin Luo, and Vincent Ng. RepoRepair: Leveraging code documentation for repository-level automated program repair. *arXiv preprint arXiv:2603.01048*, 2026.

Emiliano Penaloza, Dheeraj Vattikonda, Nicolas Gontier, Alexandre Lacoste, Laurent Charlin, and Massimo Caccia. Privileged information distillation for language models. *arXiv preprint arXiv:2602.04942*, 2026.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru WANG, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. ToolRL: Reward is all tool learning needs. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=eOLdGbXT6t.

Qwen Team. Qwen3-Coder-30B-A3B-Instruct. Hugging Face model, 2025. https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*, 2023.

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

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

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=0XBuaxqEcG.

André Silva, Sen Fang, and Martin Monperrus. RepairLLaMA: Efficient representations and fine-tuned adapters for program repair. *IEEE Transactions on Software Engineering*, 2025. doi: 10.1109/TSE.2025.3581062. URL https://doi.org/10.1109/TSE.2025.3581062.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 9460–9471, 2022. doi: 10.52202/068431-0687.

Mingyang Song and Mao Zheng. A survey of on-policy distillation for large language models. *arXiv preprint arXiv:2604.00626*, 2026.

Alex Stein, Furong Huang, and Tom Goldstein. GATES: Self-distillation under privileged context with consensus gating. *arXiv preprint arXiv:2602.20574*, 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 (Volume 1: Long Papers)*, pp. 9426–9439, 2024. doi: 10.18653/v1/2024.acl-long.510.

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, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=OJd3ayDDoF.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025b.

Zihan Wang, Chi Gui, Xing Jin, Qineng Wang, Licheng Liu, Kangrui Wang, Shiqi Chen, Linjie Li, Zhengyuan Yang, Pingyue Zhang, Yiping Lu, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. RAGEN-2: Reasoning collapse in agentic RL. *arXiv preprint arXiv:2604.06268*, 2026.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, LINGMING ZHANG, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=ULblO61XZ0.

Hongling Xu, Qi Zhu, Heyuan Deng, Jinpeng Li, Lu Hou, Yasheng Wang, Lifeng Shang, Ruifeng Xu, and Fei Mi. KDRL: Post-training reasoning LLMs via unified knowledge distillation and reinforcement learning. *arXiv preprint arXiv:2506.02208*, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Zhuolin Yang, Zihan Liu, Yang Chen, Wenliang Dai, Boxin Wang, Sheng-Chieh Lin, Chankyu Lee, Yangyi Chen, Dongfu Jiang, Jiafan He, Renjie Pi, Grace Lam, Nayeon Lee, Alexander Bukharin, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Nemotron-Cascade 2: Post-training LLMs with cascade RL and multi-domain on-policy distillation. *arXiv preprint arXiv:2603.19220*, 2026.

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, Yanhao Li, Yue Liu, Zhenxing Hu, Kaitai Zhang, Shuyi Wang, Huarong Chen, Flood Sung, Yang Liu, Yang Gao, Zhilin Yang, and Tianyu Liu. Kimi-Dev: Agentless training as skill prior for SWE-agents. *arXiv preprint arXiv:2509.23045*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, 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 *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=2a36EMSSTp.

Lifan Yuan, Wendi Li, Huayu Chen, Ganqu Cui, Ning Ding, Kaiyan Zhang, Bowen Zhou, Zhiyuan Liu, and Hao Peng. Free process rewards without process labels. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 73511–73525. PMLR, 2025. URL https://proceedings.mlr.press/v267/yuan25c.html.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

Jian Zhao, Runze Liu, Kaiyan Zhang, Zhimu Zhou, Junqi Gao, Dong Li, Jiafei Lyu, Zhouyi Qian, Biqing Qi, Xiu Li, and Bowen Zhou. GenPRM: Scaling test-time compute of process reward models via generative reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 34932–34940. AAAI Press, March 2026. doi: 10.1609/aaai.v40i41.40797.

Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. ProcessBench: Identifying process errors in mathematical reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1009–1024. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.acl-long.50/.

Appendix for "Signal Reshaping for GRPO in Weak-Feedback Agentic Code Repair"

TABLE OF CONTENTS

A Related Work **15**

B Limitations **18**

C Environment and Tool-Interface Implementation Details **19**

C.1 Retention Rates of the Data-Construction Pipeline 20

C.2 Semantic Judge Validation: 400-Sample Human Audit 20

C.3 Dataset and Repository Scale 22

C.4 GRPO Training Hyperparameters and Sampling Budget 23

C.5 Rollout Governance Implementation Details 23

C.6 Process-as-Reweighting vs. Process-as-Additive-Reward Ablation 25

C.7 Validation of Step-Level Process Scores and Loss-Mask Weights 25

D Failure Analysis of Token-Level Policy Distillation **25**

D.1 Research Position on Distilling Privileged Information 25

D.2 Negative Evidence: Token-Level Policy Distillation Does Not Replace Process Credit 26

D.3 On-Policy Distillation Implementation Details 29

D.4 Full OPSD Sweep and Additional Diagnostics 30

E Core Prompt Design **30**

E.1 System Prompt: Role Definition and Tool Instructions 30

E.2 User Task Prompt: Initial Observation 31

E.3 Semantic-Consistency Judgment Prompt: Terminal Reward r_{semantic} 32

E.4 Process-Level Semantic Scoring Prompt: Step-Level Weights in GRPO-full 32

E.5 Hint-Type Construction and Selection 33

E.6 C3 Repair-Strategy Hint Generation Prompt 35

E.7 Privileged-Hint Injection Format 36

A RELATED WORK

Existing automatic program repair work often formulates repair as one-shot patch generation from buggy code to fixed code, including neural patch generation and recent LLM-based repair at function and repository scales (Chen et al., 2021; Jiang et al., 2021; Silva et al., 2025; Pan et al., 2026; Li et al., 2026a). The central object remains a single generated patch, and training and evaluation mainly ask whether that patch is correct, rather than modeling a repair process with tool interaction, multi-step decision-making, and trajectory-level credit assignment.

Repository-level benchmarks and tool interfaces first change the object of repair: from generating a patch to completing an executable repair trajectory in a real repository. SWE-bench asks models

to modify repository code for real GitHub issues (Jimenez et al., 2024), while SWE-agent and OpenHands further emphasize agent-computer interfaces, file editing, shell/test calls, and termination decisions (Yang et al., 2024; Wang et al., 2025a); the ReAct reasoning-action framework also provides a general paradigm for such multi-turn tool interaction (Yao et al., 2023). In this setting, the model output is no longer only the final patch, but also includes context reading, error localization, file editing, shell/test invocation, feedback interpretation, and termination. Agentic code repair therefore moves code repair from static patch prediction to a long-horizon decision problem jointly defined by tool interaction, execution feedback, and termination decisions.

The next step is to place such long trajectories inside an RL objective. Existing code RL work usually converts unit tests, execution results, or compilation feedback into scalar rewards and then uses deep reinforcement learning to optimize code generation models (Le et al., 2022; Liu et al., 2023; Shojaee et al., 2023; Dou et al., 2024); general LLM post-training also includes variants such as PPO-style RLHF (Schulman et al., 2017), DPO (Rafailov et al., 2023), KL distillation (Penaloza et al., 2026), and DAPO (Yu et al., 2025). In recent long-chain reasoning and code-agentic RL, GRPO/R1-style same-prompt multi-sample relative optimization has become a common backbone (Shao et al., 2024; DeepSeek-AI et al., 2025; Wei et al., 2025; Luo et al., 2025; Golubev et al., 2025); we therefore take GRPO as the main object of analysis. Compared with one-shot patch generation, long trajectories do not by themselves require the three kinds of signal reshaping. These conditions arise from GRPO's within-group comparison mechanism: rollouts for the same prompt yield learnable advantages only when the outcome reward provides reliable semantic ordering, the process signal can localize effective actions within the trajectory, and rollout execution does not lose comparability because of resource contention or system exceptions. We organize related work accordingly: outcome rewards define "which trajectory is better," process weights define "which tokens are more important within the same trajectory," and rollout governance preserves sample comparability before group normalization.

Outcome-level reward semantics. CodeRL, RLTF, execution-based code RL, and StepCoder mainly convert tests, execution results, or compilation outcomes into pass/fail rewards (Le et al., 2022; Liu et al., 2023; Shojaee et al., 2023; Dou et al., 2024). When the test oracle is strong enough, pass/fail can approximate semantic correctness; in compile-fix, however, successful compilation is only a necessary condition. A compile-only binary reward collapses syntactic executability and functional semantic repair into the same positive class, and therefore cannot distinguish patches that merely remove compilation errors from patches that actually repair the intended behavior. Another line of work uses reference-patch similarity or repository-level benchmark outcomes as proxy signals (Jimenez et al., 2024; Zan et al., 2025; Wei et al., 2025; Li et al., 2026b), but patch similarity can conflate implementation style with functional equivalence.

Our layered outcome reward changes only the result semantics used for within-group comparison; it does not change the form of GRPO advantages. Compared with compile-only pass/fail, we keep an intermediate tier: non-compiling repairs receive 0, compiling but semantically inconsistent repairs receive 0.5, and repairs semantically consistent with the minimal sufficient ground-truth repair receive 1. Thus, GRPO's within-group comparison no longer observes only whether compilation passes, but can also use the partial progress of a repair that compiles while remaining semantically incomplete. Compared with patch-similarity rewards, r_{semantic} focuses on reference-conditioned functional consistency rather than surface edit similarity. The layered reward therefore restores semantic-correctness ordering for outcome rewards in weak-feedback compile-fix.

Process reward models. PRMs and their automated variants mitigate credit assignment in multi-step decision making by scoring intermediate reasoning or action steps (Lightman et al., 2024; Wang et al., 2024; Luo et al., 2024; Zheng et al., 2025; Yuan et al., 2025; Zhao et al., 2026). Their standard interface is to add step scores as process rewards to the optimization objective. For GRPO, these process rewards are aggregated into the rollout-level scalar reward (or equivalently into the trajectory return), then normalized together with outcome rewards by the same-prompt group mean and variance, thereby changing the advantages of different rollouts (Shao et al., 2024; Mroueh, 2025). As a result, reward scales and variances from different sources can interact. Recent multi-reward GRPO analyses explicitly show that vanilla GRPO advantages can be dominated by high-variance reward components, potentially ignoring lower-variance objectives and inducing reward hacking (Ichihara et al., 2026); summing multiple reward sources before group normalization can also collapse distinctions among reward combinations, reduce the resolution of the training signal, and lead to worse convergence or early failures (Liu et al., 2026). In addition, when PRM scores enter RFT as sum-form process rewards,

models have been observed to hack high-scoring steps and even collapse early in training (Cheng et al., 2025). Thus, in long-horizon execution tasks, directly folding process scores into the scalar reward can amplify process-scoring noise into group-relative preferences and destabilize training.

We separate outcome ranking from process attribution. The layered compile-and-semantic reward R ranks different rollouts under the same prompt; the step-level process score s_i does not enter GRPO's within-group scalar reward and does not change cross-trajectory advantages. It is converted into token loss weights only inside compilation-successful trajectories with valid process scores, increasing update strength on key edit and verification steps while down-weighting repeated inspection, uninformative explanation, and other low-information operations. Thus, our claim is not that the process judge contains more semantic information than a PRM, but that it uses a different optimization interface: s_i is not an additional reward source, and only reweights token gradients after R has fixed the update direction. This preserves the density of semantic process supervision while avoiding the redefinition of trajectory-level ranking by step count, tool-call frequency, or score scale through group normalization (Amodei et al., 2016; Skalse et al., 2022).

Comparability of long-trajectory rollouts. GRPO (Shao et al., 2024) and DAPO (Yu et al., 2025) are mainly validated in reasoning settings with no external execution environment or weak environment coupling (DeepSeek-AI et al., 2025), and recent analyses show that group-relative comparison itself can introduce bias (Liu et al., 2025). In agentic tasks, prior work improves rollout signals through within-group variance filtering (Wang et al., 2026), environment-echo token masking (Jin et al., 2025), or tool-call-level reward organization (Qian et al., 2025; Wei et al., 2025). Our comparability concern is closer to execution opportunity: container startup, host scheduling, concurrent compilation, context growth, and system exceptions can make the K rollouts for the same prompt face different execution constraints. Our rollout governance is therefore not a new reward source, but execution control before within-group comparison: concurrency decoupling reduces the effect of system bottlenecks on the sampling distribution, and failure-cause routing separates environment exceptions, sampling degeneration, and ordinary repair failures, thereby preventing environmental noise from contaminating GRPO's within-group ranking.

Granularity of additional supervision. On-policy distillation and privileged-prompt supervision have recently been used for mathematical reasoning and short-horizon tool interaction (Agarwal et al., 2024; Xu et al., 2025; Song & Zheng, 2026; Yang et al., 2026). These methods implicitly require the teacher-preferred token distribution to guide the student toward better task decisions. In weak-feedback, long-trajectory code-agent settings, this correspondence is unstable: repair success is often determined by a few edit, verification, and termination actions, while many tokens are formatting, explanations, or tool echoes. When π-Distill and OPSD (Penaloza et al., 2026) write token-level KL into the trajectory reward through a `masked-sum`, the KL is diluted by low-information tokens and also changes GRPO's within-group reward scale as trajectory length varies, interfering with the trajectory ranking that should be determined by outcome reward. Our boundary comparison (Appendix D.2) shows two manifestations of this granularity mismatch: reverse KL tends toward low entropy and degraded termination behavior, while forward KL slides into a premature-finish shortcut; both underperform the GRPO baseline under strict no-hint evaluation (Fu et al., 2026). Privileged supervision therefore does not replace the process credit provided by step-level process scores; at the same time, it does not itself provide outcome semantics or within-group comparability. Tool-call-level or trajectory-level supervision may be a more suitable direction (Lyu et al., 2025; Stein et al., 2026).

Detailed comparison with prior reinforcement learning and credit-assignment works. Table 2 summarizes how our signal-reshaping interface differs mechanically from representative prior works in code RL and agent training.

Prior works share individual ingredients such as compiler feedback, masking, or credit localization, but within one-shot generation or test-oracle settings. In contrast, our contribution lies in decoupled signal design for multi-turn weak-feedback GRPO: separating trajectory-level ranking from intra-trajectory credit, routing exceptions before group comparison, and placing a semantic consistency tier above syntax compilation.

Table 2: Interface positioning relative to prior code-RL and agent-training works.

Prior Work	Where Its Signal Sits	This Paper's Interface Placement
StepCoder (Dou et al., 2024)	Single-generation FGO mask: masks out code tokens unexecuted by unit tests	Mask determines execution comparability of full multi-turn rollouts before group comparison; action steps scored in time.
RLTF (Liu et al., 2023)	Spatial error localization: fixed negative reward $R_{\text{fine}} = -0.3$ on compiler error span	Temporal credit: continuous step scores s_i scale action tokens after group advantage formation via sign-aware reweighting.
Search-R1 (Jin et al., 2025)	Token ownership mask: separates environment text from policy-generated tokens	Failure-cause routing determines rollout settlement; surviving trajectories receive action-step weighting.
ToolRL (Qian et al., 2025)	Format reward below execution feedback	Semantic consistency tier placed strictly above syntax compilation proxy ($R \in \{0, 0.5, 1\}$).

B LIMITATIONS

This study uses C/C++ compile-fix tasks as the main setting because they combine reliable executable feedback, long tool-use trajectories, and semantically incomplete rewards in a controlled way. The goal is not to cover all code-repair benchmarks, but to isolate when GRPO's within-group comparisons remain trainable under weak feedback. Extending the same signal-reshaping principles to other programming languages, repositories with executable tests, and agentic tasks whose feedback is less tied to compilation is an important direction for further validation.

In the absence of executable test oracles, the reward and evaluation protocol use minimal sufficient repairs and an LLM-based semantic judge to provide scalable semantic feedback. The refinement pipeline only removes unrelated changes from developer patches, does not add or rewrite repairs, and verifies the refined patch by compilation; the semantic judge then checks functional consistency against this verified repair reference. This protocol makes it possible to train and evaluate on realistic compile-fix samples that lack tests. Future work can further calibrate the semantic judgments on public test-based benchmarks or human-audited subsets.

The experiments use one base model and one repair environment family to control for model capability and environment differences in the mechanism analysis. Consistent with Section 4, the main experiments keep rollout governance fixed as an execution-control condition rather than disabling its components one by one. Preliminary runs indicated that disabling any individual component often breaks execution comparability early in training, leading to training collapse and failure to converge stably. Therefore, the main text compares layered rewards and step-level process scores under a fixed rollout pipeline. Broader model sweeps, public benchmark replications, and component-level governance diagnostics can further characterize the scope of the framework.

The distillation comparison in Appendix D.2 serves as a boundary experiment: it establishes that strong-judge step weighting outperforms the evaluated 30B privileged self-teacher token-KL setup, but does not claim that step supervision unconditionally surpasses token distillation when teacher model capacity is varied. Regarding the negative-advantage reweighting rule, the floor value $\alpha_i^- \geq 0.1$ is an empirical heuristic designed to guarantee non-zero updates; while it triggered on only 1.31% of scored steps, full sensitivity sweeps across alternative floor values remain for future investigation.

For reproducibility, the paper reports the tool interface, prompt templates, reward definitions, rollout handling rules, hyperparameters, aggregate metrics, and diagnostic plots. We plan to provide anonymized or synthetic examples, the semantic-judgment and process-scoring prompts, plotting scripts, and the full configuration needed to reproduce the analysis on compatible compile-fix data.

Compute resources. All reported GRPO variants use the same FSDP training stack, Qwen3-Coder-30B-A3B-Instruct base model, $K = 8$ rollouts per prompt, global batch size 64, and fixed evaluation split; the full configuration is given in Appendix C.4. The main costs come from long-context policy inference, Docker sandbox execution, and repeated incremental compilation. Rollout governance therefore separates GPU inference, sandbox I/O, and compilation queues, as detailed in Appendix C.5. The wall-clock spans of the four GRPO comparison runs are: GRPO-compile-only 89.2h, GRPO-binary 90.6h, GRPO-early 163.9h, and GRPO-full 115.6h. Due to the high computational burden

Table 3: Tool interfaces in the Docker-sandboxed repair environment.

Category	Tool	Function and return
File inspection and editing	`str_replace_editor`	Four subcommands under a single tool: `view` reads a file by line range or lists a directory to depth two; `create` writes a new file; `str_replace` replaces a uniquely matched string within an optional line range; and `insert` inserts content after a specified line. Long outputs are truncated according to the budget.
Command execution	`bash`	Executes arbitrary shell commands in a persistent shell session inside the container, including navigation, directory inspection, scripted search, and background jobs. It supports soft timeouts and standard-input (STDIN) injection, and returns standard output and exit code, with hard truncation for overlong output.
Incremental compilation	`compile_code`	Triggers an incremental build through `build.sh` under the current repository state, classifies failure causes into `code_error`, `timeout`, `out_of_memory`, `system_limit`, and `environment_error`, and returns structured error lists and logs. This is the only executable-feedback source for the outcome-level reward component r_{compile}.
Task tracking	`task_tracker`	Maintains a structured task list through `view` and `plan`, with states `todo`, `in_progress`, and `done`, allowing the policy to explicitly organize repair plans and progress over long multi-turn trajectories.
Explicit termination	`finish`	A trajectory-termination action actively emitted by the policy, indicating that the repair is complete and should be settled. After it is triggered, no subsequent actions are accepted and reward computation begins directly.

of full post-training runs with sandbox execution at 30B scale, reporting single matched runs per configuration is standard practice; statistical robustness is demonstrated through independent judge and blind human audits at the matched comparison point rather than repeat-run variance.

Broader impacts. More reliable weak-feedback code-repair agents can reduce developer effort on routine build failures and improve the maintainability of large software systems. At the same time, code-repair agents without semantic validation or signal governance may still produce behavior-changing repairs if they rely only on shallow execution signals. The signal-reshaping approach in this paper is designed to mitigate this shallow-signal optimization problem and to support more controlled engineering deployment. Practical deployment should be staged and should retain sandboxing, provenance tracking, semantic validation, and necessary human code review.

C Environment and Tool-Interface Implementation Details

The agent execution environment is based on a containerized repair image. The original repository is mounted read-only inside the container at `/testbed`, while a writable working copy is provided at `/workspace/repair`. The tool interface contains five operators: `str_replace_editor` for file reading, inspection, and string replacement; `bash` for command execution; `compile_code` for incremental compilation and compiler-log feedback; `finish` for explicit termination and repair submission; and `task_tracker` for auxiliary task tracking, which does not affect trajectory termination. Before calling `finish`, the agent must have made at least one successful call to `compile_code`. Table 3 summarizes the subcommands and return conventions.

The semantic judge uses Kimi-K2.5 as the backend. Its input is {compilation error description, model-generated patch list, minimal-sufficient GT repair patch list}, where the GT repair is the minimal sufficient repair defined in Section 3.2. Its output includes intent consistency, confidence, and detailed analysis. Scoring is based strictly on semantic equivalence and functional consistency, without requiring identical code style or implementation, and lexical similarity is not used as the reward. The implementation keeps only two semantic outcomes, `consistent` and `inconsistent`: the former maps to $R = 1$ under the layered reward, while the latter, skipped semantic checks, or semantic-judge failures retain compilation-only partial credit $R = 0.5$ on compiled samples. Semantic checking is retried at most two times during training and at most five times during evaluation, both with exponential backoff. The reliability of this judge is audited on a 400-sample human-annotated subset stratified by method and judge label in Appendix C.2.

Minimal sufficient repairs and recent-change clues. The data-refinement pipeline only removes content from the original developer patch; it never adds, rewrites, or reorders patch lines. An LLM is given the compilation error, error types, and original diff, and is asked to return a strict subset together with an error-to-line mapping and removal rationale. The generated patch is first checked with `git apply -check`/`git apply` and then verified by running the build script in the failing build

Table 4: Per-stage sample retention in the data-construction pipeline.

Stage	Main processing	Retained samples	Stage ret.	Cumul. ret.
Raw compilation-error directories	collection from internal compilation infrastructure	12,618	–	100.0%
Candidate filtering	Multi-process remote patch extraction, 15+ error-type classification, deduplication, and token-length filtering	6,190	49.1%	49.1%
Minimal-sufficient repair verification	LLM patch refinement and sandbox compilation-loop verification	4,105	66.3%	32.5%
Context augmentation	Historical-diff context augmentation	4,105	100.0%	32.5%
Final train/evaluation pool	Difficulty-aware filtering	1,510	36.8%	12.0%

directory. If the refined patch fails to apply or compile, the pipeline retries once in a conservative mode; if it still fails, it falls back to the original developer patch. Only compile-verified outputs are marked as successful refinement and compilation. Recent-change clues are produced by extracting source files from the compiler error log, finding the previous commit for each file at the corresponding repository revision with `git log`, and retrieving the pre-failure diff with `git diff`; these clues are included in the sample's initial prompt text, so training and evaluation use the same injection path.

C.1 Retention Rates of the Data-Construction Pipeline

Table 4 summarizes the per-stage retention from internal compilation-infrastructure logs to the final curated train/evaluation pool. Stage retention is computed relative to the previous stage, and cumulative retention is computed relative to the initial 12,618 raw compilation-error directories. The historical-diff augmentation step adds context to each sample and does not further filter the sample pool.

Difficulty-aware filtering and train/evaluation split composition. To clarify the difficulty-aware filtering stage in Table 4: both the training and evaluation splits are difficulty-selected, but under distinct rules. For the training pool, 2,519 trivial `easy` instances (where all rollouts within a group tend to succeed simultaneously, causing group advantage collapse) are excluded after the initial split, yielding the final 1,110 non-trivial training samples. The fixed 400-sample hint-free evaluation split is stratified to contain 200 `hard`, 100 `medium`, and 100 `hard_compile` samples, with zero easy instances. Consequently, the base model zero-shot accuracy of 0.385 represents performance on a difficulty-controlled target benchmark rather than an unselected natural distribution, ensuring that all evaluated methods are assessed on an informative, non-deflated evaluation set.

C.2 Semantic Judge Validation: 400-Sample Human Audit

Both layered rewards and step-level process scores rely on the Kimi-K2.5 semantic judge. We treat it as a reward proxy and audit it offline; the audit does not change the training loop.

Protocol. The audit covers four checkpoints: GRPO-compile-only, GRPO-binary, GRPO-early (post-fork tail), and GRPO-full (final). This characterizes judge reliability *across* methods and training stages, not only on the final main-method checkpoint. From each checkpoint's 400-sample evaluation split, we stratify-sample 50 judge-`consistent` and 50 judge-`inconsistent` compile-success patches, for 100 patches per checkpoint and 400 patches in total. Compile-failure cases are excluded since the judge is not invoked. Two annotators familiar with the C/C++ build system independently label each patch as consistent or inconsistent with the GT minimal sufficient repair, blind to the judge's verdict and to the source checkpoint; a third annotator adjudicates disagreements.

Results. Table 5 combines the pooled confusion matrix and aggregate reliability metrics. The two annotators agree on $372/400 = 0.930$ cases, and the judge agrees with the adjudicated human label on $360/400 = 0.900$ cases; precision/recall/F_1 on the `consistent` class are $0.880/0.917/0.898$. The slight precision–recall gap reflects a mild over-acceptance tendency typical of reference-conditioned

Table 5: Confusion matrix and reliability metrics for the 400-sample human audit. Confidence intervals use Wilson for proportions and bootstrap (10k resamples) for κ.

Confusion matrix pooled over four checkpoints			
	Human: consistent	**Human: inconsistent**	**Row total**
Judge: consistent	176	24	200
Judge: inconsistent	16	184	200
Column total	192	208	400
Aggregate metrics			
Metric	**Value**	95% **CI**	**Note**
Inter-annotator agreement	$372/400 = 0.930$	$[0.901, 0.951]$	–
Inter-annotator κ	0.86	$[0.81, 0.91]$	–
Judge–human agreement	$360/400 = 0.900$	$[0.867, 0.927]$	–
Judge–human κ	0.80	$[0.74, 0.86]$	–
Precision / Recall / F_1 on `consistent`	$0.880/0.917/0.898$	–	–

LLM-as-judge on code. Per-checkpoint agreement is tightly clustered in $[89\%, 91\%]$ (GRPO-compile-only: $89/100$, GRPO-binary: $90/100$, GRPO-early: $90/100$, GRPO-full: $91/100$), with no clear method-specific bias.

Disagreement categories. Of the 40 mismatches: **FP** $= 24$ (judge says consistent, human says inconsistent), comprising symptom-level fixes that suppress the diagnostic (11), unrelated behavior changes such as different fallback or extra logging (7), and coarser-granularity workarounds such as disabling a warning (6); **FN** $= 16$ (judge says inconsistent, human says consistent), comprising functionally equivalent repairs at different locations or surface forms (10), and alternate correct repair paths over-weighted as lexically divergent from GT (6). FP outnumbers FN by a moderate margin (24 vs. 16), consistent with the well-documented over-acceptance bias of LLM-as-judge on code; the bias is distributed similarly across the four source checkpoints (χ^2 test, $p = 0.74$).

Supplementary note on the post-fork comparison. The judge is a bounded-noise reward proxy, not an oracle. The audit shows mild over-acceptance (FP 24 vs. FN 16), corresponding to an approximately 2.0pp net offset in absolute success rate. Because GRPO-full and GRPO-early use the same judge, this offset mainly affects absolute levels; per-checkpoint judge–human agreement clusters in $[89\%, 91\%]$, with no visible leniency toward GRPO-full over GRPO-early. Thus, the audit is only a supplementary check on the post-fork 5.0pp increment, reducing the likelihood that it is entirely due to differential judge bias.

Paired significance and bias-corrected human-equivalent accuracy of the post-fork $+5.0$pp. Since GRPO-full and GRPO-early are evaluated on the same 400-sample hint-free split, we run a paired McNemar test on `compile_and_semantic`: the contingency (both correct 176, only-full 36, only-early 16, both wrong 172) gives $\chi^2 = (36 - 16)^2/(36 + 16) \approx 7.69$ ($p \approx 0.006$), with a Wilson 95% CI of $[+0.018, +0.082]$ for the paired difference, excluding zero. Applying Rogan–Gladen judge-bias correction with the audit-derived sensitivity 0.917 and specificity 0.885 ($\hat{p}_{\text{true}} = (p_{\text{obs}} - (1 - \text{spec}))/(\text{sens} + \text{spec} - 1)$, denominator 0.802), the human-equivalent rates are 0.455 (GRPO-early) and 0.517 (GRPO-full), a corrected gap of $+6.2$pp, slightly larger than the raw $+5.0$pp; FP rates on `consistent`-judged patches for GRPO-full vs. GRPO-early ($5/50$ vs. $6/50$, $\chi^2 = 0.10$, $p = 0.75$) further rule out method-specific judge leniency. The post-fork increment therefore survives both paired statistical inference and reasonable judge-bias correction.

Independent outcome judge evaluation (GLM-5.2). To eliminate circularity concerns regarding the training judge (Kimi-K2.5), we re-evaluated the full 400-problem evaluation split at iteration 399 using an independent model from a different family, GLM-5.2, blind to method names and previous verdicts. Paired contingency cells: Full-only correct 55, Early-only correct 32, both correct 192, neither correct 121. The $+5.75$pp gain confirms that the improvement does not rely on the judge that supplied training rewards (Table 6).

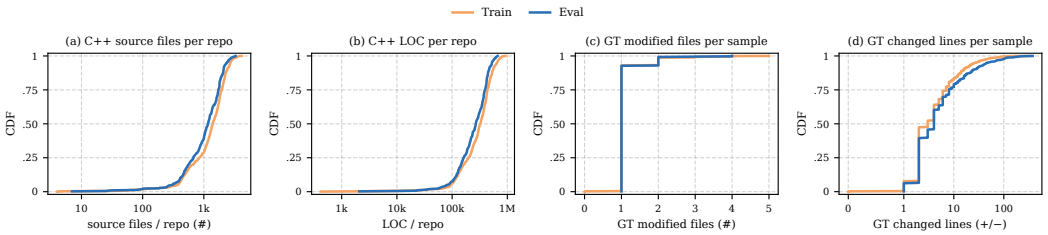


Figure 4: Distributions of repository scale and GT-modification scale across the 1,510 samples (train 1,110 + eval 400). Each curve is the empirical CDF of one split; the x-axis is logarithmic in (a)(b)(d) and linear in (c). Source-file statistics include only C/C++ files and the companion `.proto` IDL, excluding build artifacts, logs, and configuration files.

Table 6: Independent outcome evaluation (GLM-5.2) on the 400-problem split at iteration 399.

Metric ($n = 400$)	**Early**	**Full**	Δ	**95% CI**	**McNemar**
Semantically correct	0.5600 (224/400)	**0.6175** (247/400)	**+5.75**pp	[+1.2, +10.3]pp	$p = 0.018$

Natural-sample blind human Full-vs-Early comparison. To verify that the gain holds under human judgment on an unstratified natural distribution, two human annotators independently evaluated all patches from the 400-problem split without judge stratification, blind to method identity and judge labels, with a third annotator adjudicating disagreements. Paired contingency cells: Full-only correct 53, Early-only correct 28, both correct 154, neither correct 165. Inter-annotator agreement is 0.858 ($\kappa = 0.70$); the paired outcome is reported in Table 7.

Table 7: Blind human Full-vs-Early comparison on the natural 400-problem evaluation split.

Metric ($n = 400$)	**Early**	**Full**	Δ	**95% CI**	**McNemar**
Semantically correct	0.4550 (182/400)	**0.5175** (207/400)	**+6.25**pp	[+1.9, +10.6]pp	$p = 0.008$

All three measurement channels converge on the same conclusion: training judge (+5.00pp), independent judge (+5.75pp), and human audit (+6.25pp). We note that this audit validates whole-patch outcomes; step-level process scores are validated separately in Appendix C.7.

C.3 Dataset and Repository Scale

To characterize both the workload of the *repair task* itself and the size of its *repair context*, we measure four scale metrics per sample: (i) the repository-side **source file count**, the number of files in the repository subtree under C/C++ extensions (`.c/.cc/.cpp/.cxx`, the corresponding header families, and `.inc/.inl/.ipp`) plus the companion `.proto` IDL files; (ii) the repository-side **source LOC**, the total line count of those files; (iii) the **number of GT-modified files**; and (iv) the **number of GT-changed lines**, i.e., the modified-file count and the combined added/deleted line count of the minimal sufficient repair patch. Figure 4 reports the empirical CDFs of these four metrics on the train and eval splits.

Strong asymmetry between read and write scale. Across the 1,510 samples, the per-repository C++ source-file count has median 1,339 and maximum 4,135, while source LOC has median $\sim 3.08 \times 10^5$ and maximum $\sim 9.18 \times 10^5$ (Figure 4(a)(b)). In contrast, the number of GT-modified files has median 1, $p_{75} = 1$, and maximum 5; the total GT-changed line count has median only 3, $p_{75} = 7$, and maximum 378 (Figure 4(c)(d)). The policy must therefore localize $O(1)$ files and $O(10)$ lines of precise edits within a context of 10^3-scale files and 10^5-scale LOC: the *read scale exceeds the write scale by roughly* 10^4 *to* 10^5 *times*.

Train/eval alignment. The train and eval curves in Figure 4 overlap on all four metrics: source-file count and LOC have similar medians (train: 1,393 files / 3.26×10^5 LOC; eval: 1,172 files / 2.68×10^5 LOC), and only the right tail of the train split extends slightly further toward larger repositories (max 4,135 files / 9.18×10^5 LOC vs. eval's 3,300 files / 6.68×10^5 LOC). The GT-modification scale is essentially identical between splits, supporting distributional comparability of fixed-split evaluation metrics.

Table 8: GRPO training and sampling hyperparameters used across experiments.

Hyperparameter	Value
Base model and training system	
Base model	Qwen3-Coder-30B-A3B-Instruct
Training backend	Fully Sharded Data Parallel (FSDP)
Attention	`flash_attention_3`
`rotary-base`	10^7
Sampling	
`rollout-temperature`	1
`top-p`	1
`rollout-max-context-len`	40k
`rollout-max-response-len`	8k
`n-samples-per-prompt`	8
`rollout-batch-size`	8 prompts
`global-batch-size`	64
GRPO loss	
`advantage-estimator`	`grpo`
`eps-clip` (low)	0.2
`eps-clip-high`	0.28
`use-kl-loss`	`low_var_kl`
`kl-loss-coef`	0.01
`entropy-coef`	0
Optimizer	
Optimizer	AdamW
`lr`	3×10^{-6} (constant)
β_1, β_2	0.9, 0.98
`weight-decay`	0.1
Evaluation	
`eval-interval`	20
`eval-max-response-len`	40k
`eval-top-p`	1
Evaluation set	fixed split of 400 samples
Trajectory control	
`max_steps`	50
Compile-timeout abort threshold	2

C.4 GRPO Training Hyperparameters and Sampling Budget

Table 8 summarizes the training, rollout, and optimizer configuration shared by the main experiments and ablations. The optimizer follows AdamW (Loshchilov & Hutter, 2019) with decoupled weight decay.

C.5 Rollout Governance Implementation Details

Three-way concurrency control used in training. The rollout runtime separately limits three resource classes: LLM inference, ordinary sandbox operations, and compilation. Sandbox-side I/O and reward computation are scheduled asynchronously, preventing long tool calls from blocking the rollout controller.

- *Resource mapping*: `compile_code` calls and the final compilation check at trajectory termination use the compilation resource pool; file editing, command execution, and environment reset use the sandbox resource pool.
- *Inference-slot release*: inference resources cover only a single model-generation request and do not cover subsequent tool execution, so tool I/O can overlap with other rollout generations.
- *Compilation concurrency constraint*: compilation tasks are queued under a conservative concurrency limit to prevent CPU, memory, and I/O contention from contaminating the execution opportunity of rollouts in the same group.
- *Cascading protection for compilation timeouts*: if a single trajectory experiences at least the compile-timeout abort threshold (default 2) consecutive compilation timeouts, it is proactively terminated at the next step and marked as `consecutive_compile_timeouts`. This pre-

vents a small number of toxic samples from continuously occupying compilation channels and slowing down the entire rollout batch.

Exception classification and handling. `abnormal_exit_type` is drawn from {`context_limit`, `abort`, `max_tokens`, `catastrophic_repetition`, `excessive_tool_calls`, `consecutive_compile_timeouts`}. The handling rules are:

- *Full-trajectory masking*: `context_limit`, `abort`, `max_tokens`, `consecutive_compile_timeouts`, and `environment_setup_failed`. Final compilation and semantic judgment are skipped to save evaluation budget.
- *Last-step retention*: `catastrophic_repetition` and `excessive_tool_calls`, implemented as `mask_except_last_step`.
- *Normal settlement*: `finish_called`, `max_steps`, and `no_tool_call_stop`.

The experiments use the following trigger rules: `context_limit` fires when prompt plus response tokens reach `rollout-max-context-len`; `max_tokens` fires when a single generation step hits `rollout-max-response-len`; `excessive_tool_calls` fires when one assistant turn contains more than 5 parsed tool calls, and only that turn's model tokens are retained as the penalty sample; `consecutive_compile_timeouts` fires after 2 consecutive `compile_code` timeouts in the same trajectory; and `max_steps` is treated as normal budget exhaustion with the default budget of 50 steps. `catastrophic_repetition` fires when a 15–50-token n-gram repeats more than 30 times; once it fires, the trajectory is terminated early and only the final assistant turn's tokens are retained as a penalty sample.

Reward and process-score implementation. The main training reward emits layered outputs $\{0, 0.5, 1\}$, where 0 means no successful compilation, 0.5 means successful compilation with semantic inconsistency, skipped semantic checking, or semantic-judge failure, and 1 means successful compilation with semantic consistency. In addition, Kimi-K2.5 assigns a step-level process score $s_i \in [0, 1]$ to each tool-call step. This score is broadcast to all assistant tokens in the corresponding step and multiplied by GRPO's own loss mask to scale the update magnitude of model-generated tokens by step. Process scores are applied only when compilation succeeds and process scoring returns valid scores; ordinary non-compiling failures receive no step-level reweighting and therefore keep the original assistant-only mask (equivalent to $s_i \equiv 1$). Abnormal samples are still retained in the same-group reward normalization to keep the rollout group size K fixed: full-trajectory-masked samples receive $R = 0$ and a zero loss mask, so they affect only the group mean/variance and produce no gradient; last-step-retention samples also receive $R = 0$, but retain only the final assistant turn as the penalty sample. When process scores are disabled, $s_i \equiv 1$, yielding the layered baseline used to isolate the effect of process-level credit assignment. The binary reward ablation separately collapses the layered reward output into $\{0, 1\}$ to test outcome-level reward semantics loss.

Auditing the impact of fully-masked rollouts on group normalization. A potential concern is whether retaining fully masked rollouts at $R = 0$ distorts group-normalized advantages $\hat{A}_k = (R_k - \bar{R})/\sigma$. We audited 680 logged training groups (5,440 rollouts):

- 582 slots (10.7%) are fully masked across the run.
- 132 groups (19.4%) contain at least one masked slot: 47 groups are entirely masked (yielding 0 gradient, thus cannot bias updates), leaving 85 mixed groups (averaging $2.42/8$ masked slots).
- Comparing against valid-only normalization, advantage-sign flips occur in only $102/4{,}858$ valid rollouts (2.10% globally; 21.5% within mixed groups).
- **All** 102 **flips are negative → positive (**0 **positive → negative)**. Because masked slots enter at $R = 0 \leq \min R_{\text{valid}}$, $\bar{R}$ can only decrease and advantages can only increase.
- Flipped slots carry small advantages ($|A| = 0.355$ vs. valid mean 0.732), representing only 1.02% of total advantage mass.
- Pairwise rank order among valid rollouts is strictly invariant: 0 **rank inversions out of** 16,569 **pairs**, because $\hat{A}_k$ is an increasing affine transformation of R_k.

In contrast, valid-only normalization causes severe instability: in 5 of the 85 mixed groups with only one valid rollout, variance is undefined ($\sigma = 0$), completely discarding the valid update; in the remaining sparse groups, it contracts normalization scale ($\sigma_{\text{valid}}/\sigma_{\text{keep}}$ median 0.574), inflating variance based on 2–4 samples. Fixed-K normalization is therefore essential for estimator stability.

Process-scoring coverage and computational overhead. Process scoring is applied to all compile-successful trajectories: coverage is 88.6% in iterations 1–50, 89.7% in iterations 1–100, and 89%–92% across the full run. Rollout wall-clock overhead is $+21\%$ (969s for GRPO-early vs. 1177s for GRPO-full at matched iterations 240–439, $+208$s per iteration). Because scoring LLM API calls are executed concurrently across the batch after rollout generation, scoring remains off the environment critical path.

C.6 Process-as-Reweighting vs. Process-as-Additive-Reward Ablation

To isolate whether the $+5.0$pp gain is driven by the token reweighting interface or merely trajectory shortening, we evaluate an additive-reward ablation (AddRew) starting from the same fork point (iteration 240), with identical layered outcome rewards and identical step scores s_i, but folding s_i into the scalar reward: $R_{\text{add}} = R + \beta \,\text{mean}_i(s_i)$ with $\beta = 0.5$ and uniform token mask m_t. Table 9 reports the matched-iteration comparison.

Table 9: Matched evaluation of Process-as-Reweighting (GRPO-full) vs. Process-as-Additive-Reward (AddRew).

Iteration	AddRew CAS	Full CAS	Full − AddRew	AddRew Steps	Full Steps
259	0.4575	0.4850	+2.75pp	17.72	19.32
279	0.4925	0.4600	-3.25pp	16.35	17.33
299	0.4800	0.4900	+1.00pp	12.85	15.20
319	0.4775	0.4925	+1.50pp	12.34	15.51
339	0.4650	0.5025	+3.75pp	15.73	15.73
359	0.4925	0.5025	+1.00pp	12.88	15.41
379	0.5050	0.5350	+3.00pp	12.73	17.29
399	0.4450	0.5300	**+8.50pp**	12.13	17.02
Mean (8 pts)	0.4769	**0.4997**	**+2.28pp**	14.09	16.60

At iteration 399, AddRew scores 0.4450, underperforming Full (0.5300) by -8.50pp and falling below Early (0.4800). Furthermore, AddRew trajectories shorten even more than Full (12.13 vs. 17.02 steps), demonstrating that trajectory shortening does not by itself produce accuracy gains; the gain is specifically tied to the intra-trajectory credit reweighting interface.

C.7 Validation of Step-Level Process Scores and Loss-Mask Weights

Blind validation of step-level scores. We conducted an independent blind audit on 320 action steps sampled across 55 Early and Full trajectories. Annotators evaluated whether each action step advanced the repair toward fixing the compile error, blind to model identity and judge scores. Adjacent-band agreement between judge scores and human consensus is 92.2% ($295/320$, 95% CI $[88.7, 94.7]\%$), with Spearman rank correlation $\rho = 0.472$ (95% CI $[0.382, 0.553]$) and AUC 0.744 for identifying high-credit steps.

Empirical weight distribution and floor triggers. Across 991 scored steps from 55 compile-successful trajectories under Eq. (5):

- Positive-advantage weights have mean 1.00, std 0.387, and 80% fall within $[0.52, 1.41]$.
- Steps scored $s_i = 0$ result in $w_t = 0$ (sparse gating), occurring on 3.33% of steps ($33/991$).
- The floor $\alpha_i^- \geq 0.1$ is triggered on only 1.31% of scored steps ($13/991$, across 3 trajectories).

This confirms that the reweighting interface primarily performs moderate redistribution rather than extreme truncation.

D Failure Analysis of Token-Level Policy Distillation

D.1 Research Position on Distilling Privileged Information

After obtaining a stably trainable GRPO baseline, we ask whether privileged hints I, visible during training but strictly unavailable during evaluation, can provide a token-level dense signal that replaces the process-credit signal from step-level process scores, without changing the test-time interface. We instantiate this path with token-level OPD as defined in Eq. (8)–(12) of Appendix D.3. Our answer is **no**. At the objective level, the token-level KL in Eq. (8)–(12) optimizes local distributional alignment

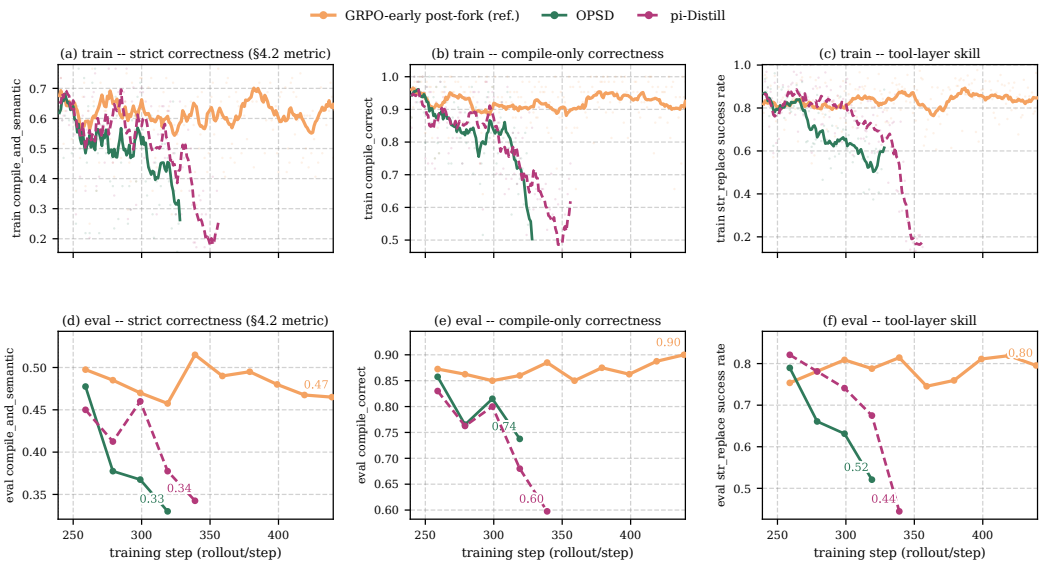


Figure 5: Outcome-level comparison of distillation over the shared `rollout/step` $\in (239, 399]$ window. The top and bottom rows show correctness, tool-level success, and exit-behavior metrics on training rollouts (on-policy) and strictly hint-free evaluation, respectively.

with the teacher, whereas GRPO in Eq. (2)–(3) optimizes outcome ranking among trajectories from the same prompt. In weak-feedback, long-trajectory code-agent tasks, success or failure is determined by a small number of tool calls, edit submissions, and termination decisions. Token-level KL therefore dilutes supervision over many non-critical tokens, violating one of the implicit assumptions discussed in Section 2.3; after KL is injected into R'_{env}, it also changes the within-group return scale used by GRPO. Consequently, token-level distillation cannot efficiently replace the process-credit signal provided by step-level process scores.

Based on this view, we evaluate OPSD and π-Distill under the same rollout pipeline, layered reward, and strictly hint-free evaluation protocol, using them as comparison methods for the token-level supervision path. This subsection states the research position and conclusion; the following sections analyze failure mechanisms and training diagnostics.

D.2 Negative Evidence: Token-Level Policy Distillation Does Not Replace Process Credit

Experimental setup. The loss forms and implementation details of the two methods are given in Eq. (8)–(12) of Appendix D.3. Both OPSD and π-Distill are *stacked on top of GRPO-early*: they are initialized from the GRPO-early fork checkpoint and compared with the GRPO-early post-fork baseline over the shared `rollout/step` $\in (239, 399]$ window; apart from the distillation-specific token-level KL term (written into $\widetilde{R}$ as in Eq. (9)/(11), with the teacher branch reading prompts augmented by the `<Privileged hint>` block), every other setting is identical to GRPO-early.

Outcome level: correctness lags across the board. The top row (train) and bottom row (eval) of Figure 5 show a central fact: regardless of KL direction, token-level distillation yields substantially lower strict outcome correctness than the GRPO-early post-fork baseline, and the gap shows no sign of closing throughout the shared window.

Specifically, the strict correctness rate on evaluation, `compile_and_semantic`, is only about 0.36 for OPSD and 0.35 for π-Distill, whereas GRPO-early post-fork stays around 0.48 in the same window. Compilation correctness, `compile_correct`, also lags behind, with OPSD at approximately 0.77, π-Distill at approximately 0.75, and the baseline at approximately 0.88. This indicates that the failure is not evaluation noise, but a real gap in policy quality.

The more informative divergence appears at the tool and behavior levels. The evaluation `str_replace` success rate of π-Distill, approximately 0.76, is close to GRPO-early, approximately 0.81, suggesting that its tool-use skill is largely preserved. However, its evaluation average

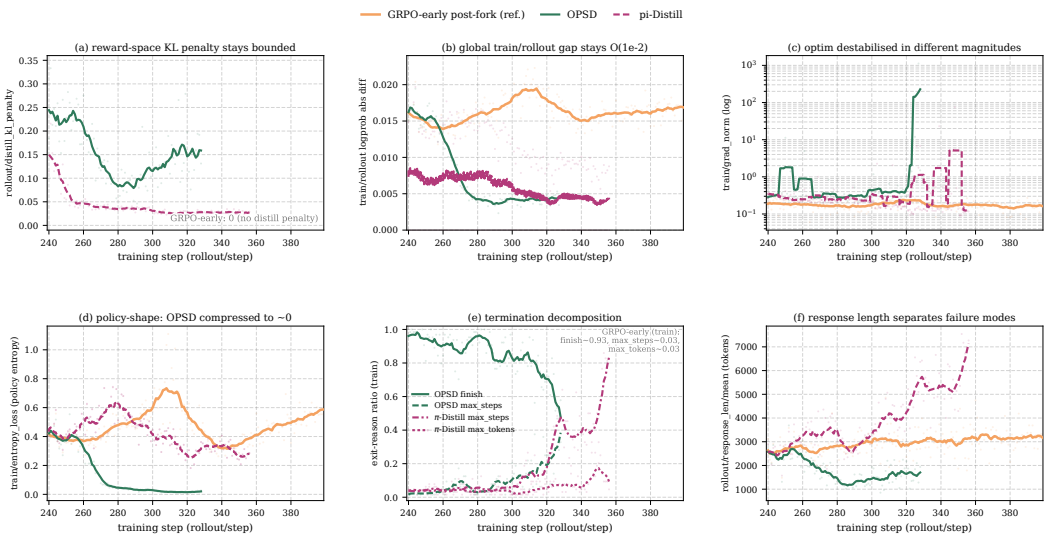


Figure 6: Mechanism-level diagnostics for distillation over the same shared window. The figure shows optimization and behavioral dynamics, including reward-space KL penalty, log-probability gap, gradient norm, policy entropy, and exit reasons.

steps are only about 12, much lower than the baseline, approximately 21, and OPSD, approximately 23. This means that the π-Distill agent learns to take actions and even to make plausible edits, but does not learn to terminate at the right time: it calls `finish` prematurely and submits incomplete repairs, causing both strict correctness and compilation correctness to drop. In contrast, OPSD also shows a clear degradation in `str_replace` success rate, approximately 0.60, indicating that reverse-KL-induced low entropy not only harms termination decisions but also erodes tool-level skill.

Overall, the failure of token-level distillation is not a single-dimensional inability to learn. π-Distill preserves local skill while losing global pacing, manifested as premature termination; OPSD loses both tool precision and termination timing while compressing distributional entropy. Neither path yields a substantive improvement in strict outcome correctness.

Mechanism level: progressive evidence from six panels. Figure 6 provides mechanistic evidence along six dimensions: KL, global mismatch, optimizer behavior, policy shape, exit structure, and length drift. The evidence can be summarized in three layers.

The first layer rules out simple explanations. In panel (a), the reward-space distillation penalty is bounded for both methods and generally decreases, so the failure cannot be attributed to an exploding KL penalty. In panel (b), the estimator-agnostic train-rollout log-probability gap remains at the $O(10^{-2})$ scale for all three runs. π-Distill does not clearly separate from the baseline on this global mismatch scalar, so a single global policy drift cannot explain the failure either.

The second layer shows divergence in optimization and policy shape. In panel (c), `grad_norm` on a log scale reaches approximately 10^2 for OPSD, 10^0 for π-Distill, and 10^{-1} for the baseline. Both distillation methods perturb optimization stability, but through different entry points and magnitudes. Panel (d) shows that OPSD compresses policy entropy nearly to zero, whereas π-Distill maintains substantially higher entropy. This indicates that OPSD anchors the policy to a narrow teacher-induced distribution, while π-Distill does not undergo comparable compression.

The third layer identifies behavioral divergence. Panel (e) shows that OPSD has decreasing `finish_called`, increasing `max_steps`, and nearly unchanged `max_tokens`; π-Distill exhibits a different pattern, with `finish_called` as high as approximately 0.98 and both `max_steps` and `max_tokens` compressed close to zero. Panel (f) shows that response length remains relatively short for OPSD, while π-Distill contracts sharply to approximately 952 tokens, even shorter than the GRPO-early baseline. Combining panels (d), (e), and (f), OPSD corresponds to a low-entropy form of failure in which the policy becomes reluctant or unable to terminate appropriately, with entropy near zero, unchanged `max_tokens`, and no length expansion. π-Distill instead corresponds to premature termination: length is controlled, entropy remains open, and gradient norm is moderate, but evaluation strict correctness still does not improve.

These three layers show that both methods inject token-level supervision along long trajectories. OPSD drives the policy toward low entropy and degraded termination behavior, whereas π-Distill removes length expansion and gradient instability at the mechanism level but moves toward a subtler local optimum: premature termination.

Root-cause explanation. Combining outcome-level and mechanism-level diagnostics, the failure can be attributed to one common root cause and two method-specific paths.

(A) Common root cause: mismatch between supervision granularity and decision granularity. GRPO optimizes the relative outcome ranking of complete trajectories under the same prompt, while token-level distillation constrains local distributional alignment at each token. For long-trajectory, weak-feedback tasks such as compile-fix, success is often determined by a small number of action-level branching points, especially whether to continue using tools or call `finish`. In contrast, many schema, path, log, and NL tokens are nearly neutral with respect to the final reward. Token-level supervision is therefore easily diluted by many low-information positions. Moreover, the implementation uses `masked sum` to aggregate KL, so distillation pressure accumulates with sequence length and couples with statistics of non-critical tokens. The teacher signal is not concentrated on key decisions; instead, it first reshapes local distributions and behavioral rhythm over long trajectories.

The direct consequence of this mismatch is degraded policy quality: both distillation methods maintain lower `str_replace` success and `compile_and_semantic` rates than the GRPO baseline. On top of this shared failure, the different KL directions further separate the degradation paths. OPSD's reverse-KL penalty favors narrow-band alignment; after masked-sum aggregation, it changes trajectory-level rewards and amplifies the corresponding policy-gradient updates, manifested as low entropy and degraded termination behavior. π-Distill's forward-KL penalty is more tolerant of distributional support; panel (a) shows that its KL penalty is bounded and even lower than OPSD's. However, token-level supervision is diluted by many neutral tokens, and the policy drifts in an open-entropy space toward a surface shortcut: very high `finish_called` with low outcome correctness. We detail the two paths below.

(B) Method-specific path of π-Distill: mode covering under forward KL and supervision dilution lead to premature termination. π-Distill uses $\mathrm{KL}(\pi_T \| \pi_S)$, whose mode-covering property requires the student to cover the full probability support of the teacher and therefore does not compress the distribution into a single narrow band. Panel (a) shows that its distillation penalty is bounded and even below OPSD's; entropy remains open, gradient norm is moderate, and training remains stable. However, the granularity mismatch of token-level supervision remains. As a reward penalty, the forward-KL form encourages coverage of teacher support rather than concentrating trajectory-level advantages on the few high-quality repair decisions. The policy therefore drifts in an open-entropy space toward a surface shortcut. The teacher hint contains many language patterns of summarization and submission; by aligning to these patterns, the policy learns the rhythm of finishing early, but not the causal logic of submitting only after the repair is complete. As a result, `finish_called` reaches approximately 0.98, while evaluation strict correctness remains only about 0.35, substantially below GRPO-early's approximately 0.48.

(C) Method-specific path of OPSD: mode seeking under reverse KL and long-trajectory gradient accumulation cause low entropy and gradient amplification. OPSD uses $\mathrm{KL}(\pi_S \| \pi_T)$. Its mode-seeking property directly pushes the policy toward the narrow band of the teacher distribution, thereby suppressing exploration. In weak-feedback, long-trajectory tasks, this rigid alignment does not necessarily prioritize the few key edits needed for final correctness; instead, it first compresses overall distributional entropy and termination choices. The evidence in Figure 6 shows three aspects of OPSD's failure: panel (d) shows that policy entropy is compressed nearly to zero, meaning that the distribution is anchored to a narrow teacher-induced band; panel (c) shows that `grad_norm` grows to approximately 10^2, indicating that the reverse-KL penalty, after being written into trajectory rewards through masked-sum aggregation, changes within-group advantages and substantially amplifies policy-gradient updates over long trajectories. The mechanism here is not direct backpropagation through the KL term itself; it is the effect of KL-shaped rewards on GRPO's relative reward scale. Compared with π-Distill, the narrow-band pressure of reverse KL more readily directs these advantages toward entropy collapse. Panels (e) and (f) show that `max_tokens` barely changes and response length does not expand, indicating that pressure is not expressed as length growth but as low entropy and termination degradation. Behaviorally, this appears as degraded `finish` decisions, making trajectories harder to end at the appropriate time.

Conclusion. In this setting, token-level distillation does not replace the process-credit signal provided by step-level process scores: it does not translate into better outcome correctness in evaluation, nor does it solve the attribution of credit to key steps. It also does not itself repair outcome-level reward semantics or within-group comparability, and introduces additional optimization issues. π-Distill further shows that even when KL strength is comparable to OPSD and mechanism-level indicators appear normal, with no length expansion or gradient failure, evaluation strict correctness still does not match the GRPO baseline. The limitation of token-level alignment is therefore not determined by an overly strong KL alone, but by the mismatch between supervision granularity and decision granularity. This result is consistent with the main conclusions from the previous sections: layered rewards and step-level process-score weighting are more directly matched to the task than token-level alignment. This conclusion is also consistent with recent reports on the failure of token-level OPD under distribution shift (Fu et al., 2026), and suggests that future work should explore tool-call-level supervision, trajectory-level counterfactual correction (Lyu et al., 2025), or teacher-consistency gating (Stein et al., 2026).

D.3 On-Policy Distillation Implementation Details

π-Distill. Let $\pi_\theta^T(\cdot|s, I)$ denote the teacher with privileged information, and let $\pi_\theta^S(\cdot|s)$ denote the student without privileged information; the two share parameters. The standard π-Distill objective contains both teacher and student terms:

$$J_{\text{Teacher}}(\theta) = \mathbb{E}_{o\sim\pi_\theta^T(\cdot|s,I),\, s\sim P}\big[R(o,s)\big] - \beta D_{\text{KL}}\big(\pi_\theta^T(o|s,I) \,\|\, \text{sg}\big(\pi_\theta^S(o|s)\big)\big)\,, \tag{6}$$

$$J_{\text{Student}}(\theta) = \mathbb{E}_{o\sim\pi_\theta^T(\cdot|s,I),\, s\sim P}\left[\frac{\pi_\theta^S(o|s)}{\text{sg}(\pi_\theta^T(o|s,I))}R(o,s)\right] - \beta D_{\text{KL}}\big(\text{sg}(\pi_\theta^T(o|s,I)) \,\|\, \pi_\theta^S(o|s)\big)\,, \tag{7}$$

$$J_{\pi\text{-Distill}}(\theta) = \alpha J_{\text{Teacher}}(\theta) + (1-\alpha) J_{\text{Student}}(\theta). \tag{8}$$

In the compile-fix implementation, the KL regularizers in the joint objective above are merged into a single term and applied as a reward-shaping penalty entering the GRPO group advantage. The teacher samples trajectories on the hint-augmented prompt, and the reference policy is taken to be the stop-gradient distribution of the current hint-free student branch:

$$\widetilde{R}_{\pi\text{-Distill}}(o_k, s) = R_{\text{env}}(o_k, s) - \beta\, D_{\text{KL}}^{\text{masked-sum}}\big(\pi_\theta^T(\cdot|s,I) \,\|\, \text{sg}\big(\pi_\theta^S(\cdot|s)\big)\big)\,. \tag{9}$$

Within-group advantages are then computed from $\widetilde{R}_{\pi\text{-Distill}}$. The implementation uses *batch expansion* to retain both a student-copy and a teacher-copy, which share the same $\widetilde{R}_{\pi\text{-Distill}}$ and the same group advantage and differ only in the importance ratio: the student-copy uses hint-free tokens with $\rho_k^S = \pi_\theta^S(o_k|s)/\pi_{\text{old}}^T(o_k|s,I)$ for off-policy GRPO, while the teacher-copy uses hint-augmented tokens with $\rho_k^T = \pi_\theta^T(o_k|s,I)/\pi_{\text{old}}^T(o_k|s,I)$; the two copies have loss weights $2(1-\alpha)$ and 2α so that the average over the doubled batch reproduces $\alpha J_{\text{Teacher}} + (1-\alpha)J_{\text{Student}}$, with default $\alpha = 0.5$. The KL is estimated with `masked-sum` / RB top-k over assistant-generation tokens only; defaults are $\beta = 0.01$, RB top-$k = 32$, and C3 hints.

OPSD. OPSD uses the student policy as the sampling policy: the hint-free student performs on-policy rollout, while the teacher is the same shared-parameter model conditioned on a hint-augmented prompt and scores the same trajectory. Its objective is

$$J_{\text{OPSD}}(\theta) = \mathbb{E}_{o\sim\pi_\theta^S(\cdot|s),\, s\sim P}\big[R(o,s)\big] - \beta D_{\text{KL}}\big(\pi_\theta^S(o|s) \,\|\, \text{sg}\big(\pi_\theta^T(o|s,I)\big)\big)\,, \tag{10}$$

where the reverse KL can be viewed as a dense token-level reward measuring how closely the hint-free student matches the hint-conditioned teacher. In the implementation, this KL term is first written into the raw reward:

$$\widetilde{R}_{\text{OPSD}}(o_{i,k}, s_i) = R_{\text{env}}(o_{i,k}, s_i) - \beta\, D_{\text{KL}}^{\text{masked-sum}}\big(\pi_\theta^S(\cdot|s_i) \,\|\, \text{sg}\big(\pi_\theta^T(\cdot|s_i, I_i)\big)\big)\,, \tag{11}$$

and then used to compute group-normalized advantages:

$$\hat{A}_{i,k} = \frac{\widetilde{R}_{\text{OPSD}}(o_{i,k}, s_i) - \frac{1}{K}\sum_{j=1}^K \widetilde{R}_{\text{OPSD}}(o_{i,j}, s_i)}{\text{std}(\{\widetilde{R}_{\text{OPSD}}(o_{i,j}, s_i)\}_{j=1}^K)}, \qquad \rho_{i,k} = \frac{\pi_\theta^S(o_{i,k}|s_i)}{\pi_{\text{old}}^S(o_{i,k}|s_i)}. \tag{12}$$

In practice, this group-normalized advantage is implemented by slime's GRPO reward normalization. Teacher log-probabilities are precomputed during rollout and used with stop-gradient; KL is estimated

with `masked-sum` / RB top-k over assistant-generation tokens only. The defaults are $\beta = 0.02$, RB top-$k = 32$, and C3 hints.

Implementation details. Both π-Distill and OPSD subtract a KL penalty from the raw reward before computing GRPO within-group advantages, with KL directions $D_{\mathrm{KL}}(\pi^T \| \pi^S)$ and $D_{\mathrm{KL}}(\pi^S \| \pi^T)$, respectively. To adapt to the long trajectories and tool interactions in compile-fix, we estimate sequence-level KL only on assistant-generation tokens covered by the loss mask and use a `masked-sum` / RB top-$k = 32$ approximation; β is scaled to $\{0.01, 0.02\}$; privileged information is fixed to C3 hints abstracted from GT diffs.

D.4 Full OPSD Sweep and Additional Diagnostics

Following the presentation in Appendix D.2, the main paper keeps only one representative OPSD run that enters long-horizon comparison, while early-stopped experiments and other sweep configurations are reported uniformly in the appendix. This section summarizes the OPSD training configurations. The goal is not to compare every setting in the main text, but to show common patterns under different KL injection forms and strengths. The diagnostics are as follows:

- per-token KL in advantage ($\beta = 1$): the gradient norm increases rapidly at the beginning of training, and the loss soon diverges.
- masked mean KL in reward ($\beta = 0.25$): the KL term remains very small for a long period and has limited effect on group-normalized relative returns; the training trajectory nearly overlaps with the pure GRPO baseline, with no stable gain observed.
- masked sum KL in reward ($\beta = 0.25$): the KL contribution fluctuates substantially with trajectory length, so the same β corresponds to different penalty strengths for short and long samples; training becomes clearly unstable and terminates early.
- masked sum KL in reward ($\beta = 0.05$): this setting also exhibits length-coupled optimization instability. Although it is slightly milder than the larger coefficient, it still does not enter a comparable long-horizon interval.
- masked sum KL in reward ($\beta = 0.02$): this is the only masked-sum configuration that enters long-horizon comparison. Its explicit KL can be continuously reduced, but correctness evaluation does not outperform GRPO-early post-fork, and the run gradually shows termination-behavior degradation and low entropy. It therefore serves as a runnable but non-beneficial counterexample, rather than positive evidence that OPSD can replace GRPO's signal conditions.
- masked mean KL in reward ($\beta = 50$): training can formally continue, but learning is clearly suppressed, eventually producing long-term reward degradation and interruption in the later stage.

These additional diagnostics are consistent with the failure-mechanism analysis in the main text. On the one hand, token-level KL over long trajectories easily couples with length and non-critical token statistics. On the other hand, even when the explicit KL is reduced to a low range, we do not identify an OPSD operating regime that simultaneously maintains training stability, normal behavior, and improved outcome correctness. Larger KL coefficients more directly induce instability, while smaller coefficients may allow training to continue for longer but still mainly produce termination-behavior degradation or suppressed learning rather than correctness improvement.

E Core Prompt Design

This section lists the main prompt templates used in the experiments. All prompts are iteratively tuned to ensure stable output format and consistent judgment criteria. The templates below preserve placeholders, control tags, and JavaScript Object Notation (JSON) fields.

E.1 System Prompt: Role Definition and Tool Instructions

The system prompt is injected through the tokenizer chat-template API as the top-level role constraint for each interaction round of the agent. The template is shown below.

```
You are an experienced C++ engineer specializing in fixing compilation
↪  errors.

## Task Description
Locate and fix code issues based on compiler error messages.
```

```
## Repair Requirements
1. Fix only the code that causes compilation ERRORs; do not fix WARNINGs.
2. Ignore all compilation warnings and focus only on compilation errors.
3. Do not break the original logic or functionality of the code; remain
↪  consistent
   with the original intent.
4. Make only necessary changes and do not introduce new issues.
5. Ensure that the repaired code compiles successfully. Warnings are
↪  allowed, but
   errors are not.

## Workflow
1. **Analyze errors**: Carefully read the compiler error messages and
↪  understand
   the specific cause of each error.
2. **Make a plan**: Use the plan command of task_tracker to create a task
↪  list.
3. **Locate the issue**: Use the view command of str_replace_editor to
↪  inspect
   the reported file.
4. **Understand context**: When necessary, use bash to search for
↪  relevant
   definitions, declarations, or usages.
5. **Make a precise fix**: Use str_replace in str_replace_editor to make
↪  a
   minimal modification.
6. **Update progress**: After completing each repair, update the task
↪  status with
   task_tracker.
7. **Verify the result**: Call compile_code to recompile.
8. **Iterate**: If errors remain, repeat steps 3--7.
9. **Submit**: After confirming successful compilation, call finish to
↪  submit.

## Available Tools
- task_tracker: task management (plan / view)
- str_replace_editor: file operations (view / str_replace / create /
↪  insert)
- bash: execute shell commands
- compile_code: compile and verify
- finish: submit after the repair is complete

## Notes
- Make precise changes and avoid introducing new issues.
- Call compile_code and confirm success before calling finish.
- When calling finish, briefly explain the cause of the issue and the
↪  solution.
```

E.2 User Task Prompt: Initial Observation

During data preprocessing, the raw compilation error log is wrapped as the following user message, which serves as the initial observation of the trajectory. If a teacher hint is enabled, a `<Privileged hint>` block in the format of Section E.7 is inserted at the beginning of the user message.

```
# Compilation Error Repair Task

## Problem Overview
The code fails to compile. Analyze the error log and fix the issue.

## Compilation Error Log
```
{error_log}
```

## Task Requirements
```

```
1. Analyze the compilation errors above and identify the root cause.
2. Propose a minimal code modification.
3. Ensure that the repaired code compiles successfully.

Please start analyzing and fixing the compilation error.
```

E.3 SEMANTIC-CONSISTENCY JUDGMENT PROMPT: TERMINAL REWARD r_{SEMANTIC}

Both training and evaluation use Kimi-K2.5 for outcome semantic judgment with the following prompt. The output is strict JSON containing `intent_consistency`, `confidence_score`, and `detailed_analysis`.

```
Please analyze the semantic consistency between the code repair generated
↪  by the
model and the ground-truth (GT) repair.

**Compilation error information:**
{error_info}

**Model-generated repair:**
{model_patches_str}

**GT repair:**
{gt_patches_str}

**Semantic-consistency evaluation criteria:**
1. **Semantic equivalence**: The repair generated by the model is
↪  semantically
   equivalent to the GT repair.
2. **Functional consistency**: The repaired program produces the same
↪  output
   under the same input.
3. **Behavior preservation**: The repair does not change the logical flow
↪  or
   computation process of the program.

**Evaluation requirements:**
- Compare only the modifications related to the target compilation error.
- The GT may contain changes unrelated to the target error; these
  should not affect the judgment.
- Focus on the semantic intent and runtime behavior of the repair.

**Return the result strictly in the following JSON format, with no extra
↪  text:**

{
    "is_semantic_consistent": true,
    "intent_consistency": "consistent",
    "confidence_score": 0.8,
    "detailed_analysis": "Detailed analysis and rationale",
    "key_differences": ["difference 1", "difference 2"]
}
```

E.4 PROCESS-LEVEL SEMANTIC SCORING PROMPT: STEP-LEVEL WEIGHTS IN GRPO-FULL

GRPO-full uses Kimi-K2.5 to score the complete tool trajectory step by step, producing a `score` $\in [0, 1]$ and a brief `reason` for each step. These scores are used to reweight the loss mask by step.

```
You are a reward model for code repair. Please evaluate the directional
correctness and information gain of the entire tool trajectory based on
↪  semantic
reasoning.

[Compilation error information]
{error_info}
```

```
[Model tool trajectory, step by step]
{tool_calls_str}

[Ground-truth (GT) repair used for semantic comparison]
{gt_patches_str}

Semantic-consistency evaluation criteria. The comparison with GT should use
semantic judgment rather than exact matching:
1. Semantic equivalence: a model-generated repair is correct if it is
   semantically equivalent to the GT.
2. Functional consistency: the repaired program should produce the same output
   under the same input.
3. Behavior preservation: the repair should not change the logical flow or
   computation process of the program.

Evaluation requirements:
- Compare only the modifications related to the target compilation error.
- The GT may contain modifications unrelated to the target error; these
  should not affect the judgment.
- Focus on the semantic intent and runtime behavior of the repair.
- Repeatedly viewing files or regions that have already been inspected should be
  penalized as low information gain.
- compile_code is used to verify the repair and is part of the normal workflow;
  it should receive a relatively high score.

Output JSON only, with no extra text:
{
  "process_score": 0.0,
  "step_process_scores": [
    {"step_index": 0, "score": 0.0, "reason": "Brief reason"}
  ],
  "global_comment": "Overall process assessment"
}
```

E.5 Hint-Type Construction and Selection

Design motivation. The GT for the compile-fix task is the final code state, represented as a diff, rather than an action sequence, and the compiler already provides rich error-localization information. Our privileged information is structurally extracted from GT diffs in the annotated data and requires no additional inference cost. Data analysis shows that nearly half of the repairs occur in files not directly reported by the compiler, while error logs contain many noisy files. File-localization information therefore provides substantial incremental value.

Three basic hint types. Table 10 summarizes the construction form, information density, and intended use of the three basic hint types. All types are generated from the same metadata, including GT patches, error types, and error logs, and differ only in their level of abstraction.

Table 10: Construction forms and information density of the three basic hint types.

Type	Construction form	Information density	Design intent
Type A (GT Diff)	Inject the GT diff as a teacher-side upper-bound hint in the system prompt	Highest: upper-bound hint	Validate the performance upper bound; excluded from no-hint evaluation
Type B (File localization)	Extract file paths to modify, line ranges, error-type categories, and repair directions	Medium: WHERE	Help localization without directly giving the answer, balancing information content and generalization
Type C (NL strategy)	Use glm-5 to abstract an NL repair strategy of approximately 150 words from the GT diff	Low: HOW	Minimize KL cost while encouraging error-analysis ability

Type A example: upper-bound hint.

````
## Teacher-side upper-bound hint
The following information is used only to construct a hinted teacher for
diagnostic upper-bound analysis and is not used in no-hint evaluation:

### Modification 1: condition_matcher.h
```diff
@@ -108,6 +108,7 @@ class ConditionMatcher {
      MMERR("cond.values.size < 1");
      return result;
    }
+
    switch (field.type) {
      case FieldType::UINT64: {
@@ -153,6 +154,9 @@
        break;
      }
+      default:
+        MMERR("invalid field type");
+        break;
    }
```
````

Type B example: file localization.

```
## Repair hint
- Error type: warning_as_error
- Files that need modification:
  - compute/matcher/condition_matcher.h (around line 111)
- Repair direction: eliminate the compilation warning, which has been
↪  promoted
  to an error by -Werror.
```

Type C example: NL strategy.

```
## Repair strategy hint
This compilation error occurs because the switch statement does not
↪  handle all
enumeration values. The compiler enables -Werror=switch, which promotes
↪  this
warning to an error. The repair is to add a default branch at the end of
↪  the
switch statement to handle unlisted enum values.
```

Multi-level NL hint granularity experiments: C1–C5. Based on the Phase 0 result that Type C is safe and effective, we further design five granularity levels. All are produced by LLM calls that abstract a general repair strategy from the GT diff, and they strictly avoid leaking any file names, function names, variable names, line numbers, or code snippets. Table 11 shows the construction goal and abstraction level of each hint level.

Table 11: Construction goals and abstraction levels of five NL hint granularities.

Level	Construction goal	Target length	Prompt guidance
C1	Error-pattern recognition	$\sim$60 words	One-sentence summary of the error pattern and general repair direction
C2	Repair-strategy direction	$\sim$120 words	General diagnostic procedure and repair-strategy direction
C3	Repair reasoning chain	$\sim$180 words	Reasoning chain from observing the error to selecting the repair, including key judgment points
C4	Step-by-step methodology	$\sim$300 words	A 3–5 step general methodology and judgment principle for each step
C5	Complete repair guide	$\sim$500 words	Complete guide covering diagnosis, strategy, verification, and pitfalls

C1–C5 inference results. We inject hints during inference on 400 evaluation samples at the step-239 checkpoint, using temperature 0.7 and pass@1. The results are shown in Table 12. C3, an NL strategy summary of approximately 180 words, obtains the highest pass@1 of 53.5%, improving over the no-hint baseline of 44.5% by +9.0pp. C1 provides only +2.5pp because it contains too little

information, while C4 and C5 cause more fallback cases because of information overload, yielding lower net gains than C3.

Table 12: Inference results for five NL hint granularities at step 239 on 400 samples.

Hint type	Compile success	Semantic consistency	pass@1	Δ pass@1	Net gain
Baseline (no hint)	88.8%	45.5%	44.5%	—	—
C1 (∼60 words)	91.8%	48.0%	47.0%	+2.5	10
C2 (∼120 words)	91.2%	50.5%	49.5%	+5.0	20
C3 (∼180 words)	**92.2%**	**53.5%**	**53.5%**	**+9.0**	**36**
C4 (∼300 words)	88.8%	50.0%	48.2%	+3.7	15
C5 (∼500 words)	90.0%	50.5%	49.8%	+5.2	21

Rationale for choosing C3. The experiments show a clear inverted-U curve: C1 is under-informative; C2 provides the right direction but lacks a reasoning chain; C3 strikes the best balance among error-pattern recognition, diagnostic procedure, and key judgment criteria; and C4/C5 are too long, causing attention dispersion and more fallback cases, with 41 fallback cases for C4 versus 34 for C3. In addition, C3 has the highest `str_replace` success rate among the five levels, 87.5%, and a slightly lower average number of steps, 17.9, than the baseline, 19.1, indicating that it guides the repair direction without adding extra operational burden. Therefore, the teachers in both π-Distill and OPSD use the C3 hint type.

E.6 C3 Repair-Strategy Hint Generation Prompt

The teachers in both π-Distill and OPSD use C3 hints, approximately 180 words. The following prompt calls glm-5 to abstract a general repair reasoning chain from the GT diff, requiring that no concrete file names, function names, or code snippets be included.

```
You are an expert in fixing C++ compilation errors. Below is a
↪  compilation error
and the corresponding correct repair patch.

Your task: abstract from this patch the reasoning process for fixing this
↪  type
of error, describing the chain of thought from "seeing the error" to
↪  "choosing
the repair".

Strict requirements:
- Describe a general reasoning process: what information to inspect
↪  first, what
  judgment to make, what conclusion to draw, and what repair strategy to
  ↪  adopt.
- Focus on key judgment points and decision criteria, such as: "when an
↪  error
  reports an undefined symbol, determine whether the declaration is
  ↪  missing or
  whether the declaration is in the wrong scope".
- The strategy must be general and transferable to all errors of the same
↪  type.
- Do not include any file names, function names, variable names, line
↪  numbers, or
  concrete code snippets.
- Keep the total length strictly within 180 words.

Compilation error type: {error_types}
Compilation error log:
{errors}

Repair patch:
{diffs}
```

E.7 PRIVILEGED-HINT INJECTION FORMAT

π-Distill and OPSD wrap the teacher hint in `<Privileged hint>` tags and insert it at the beginning of the user message. This allows the teacher to receive strategy guidance before reading the compilation error log. The hint is used only for training-time distillation comparisons and hinted upper-bound analysis, and is strictly excluded from no-hint evaluation.

```
<Privileged hint>
The following is a teacher-side strategy hint abstracted from the
↪  annotated
repair.
{hint_text}
</Privileged hint>
```