

FLARE: A Full-Lifecycle Dense Supervision Paradigm for Long-Horizon Coding Agents via Generative Reward Model

Jingxuan Xu^{1,*} Gang Wu^{1,*} Yanan Wu^{3,*} Yutao Mou^{2,*}
 Songwei Yu¹ Tianzhuang He³ Zhengshuo Gong⁴ Zhao Liu¹
 Zihang Xu¹ Wenqiang Zhu¹ Xinpeng Lei³ Weihao Li¹
 Yuhui Bai¹ Zhongqiu Wang¹ Yan Wu¹ Ariel Deng^{1,†}

xujingxuan2002@163.com

¹Independent Researcher ²Peking University

³Nanjing University ⁴Beijing University of Posts and Telecommunications

Abstract

While test-time scaling enhances Large Language Model (LLM) agents in long-horizon software engineering (SWE), sparse binary rewards (Pass/Fail) create a severe credit assignment crisis and waste failed exploratory trajectories. Current trajectory optimization and scaling methods are costly and structurally limited, relying on heuristic state reuse without causal diagnosis or delayed scalar scoring without actionable online guidance. We propose FLARE (Full-Lifecycle Alignment and Reward Engine), a novel dense supervision paradigm driven by a lightweight Generative Reward Model (GRM). First, RADAR, an offline causal-aware diagnostic framework, extracts high-fidelity, hindsight-free supervision through causal-chain backtracking to distill a GRM providing real-time, step-level risk feedback. Second, FLARE uses this GRM to continuously optimize the agent across its entire lifecycle. During inference, FLARE acts as an Active Scaffold, autonomously intercepting high-risk generation steps for localized breakpoint re-execution, drastically reducing compute overhead. During post-training, the GRM’s structured signals serve as process-supervised reranking scores for Supervised Fine-Tuning (SFT) and step-level dense rewards for Reinforcement Learning (RL), mitigating policy collapse in sparse environments. Extensive evaluations show that FLARE establishes a new Pareto frontier across the agent lifecycle: FLARE (N=1) outperforms Global Rollout (N=5) with a 5x reduction in token consumption. Extending FLARE to training overcomes the sparse reward problem in long-horizon interactive tasks, delivering relative performance gains of 19.13% in SFT through process-aware data curation and a consistent 9.19% improvement in RL.

1 Introduction

Large Language Models (LLMs) have evolved significantly, progressing into sophisticated interactive agents capable of navigating repositories, executing tests, and autonomously resolving complex software engineering (SWE) issues. However, navigating these long-horizon tasks exposes a critical vulnerability: the extreme reliance on sparse, outcome-based binary rewards (Pass/Fail). When an agent fails after executing dozens of steps, a binary outcome offers no insight into the specific point of failure—whether it was a flawed plan at step 5 or a syntax error at step 40. This “blind trial-and-error” approach results in a severe credit assignment crisis, forcing the community to discard vast amounts of failed, yet highly informative, exploratory trajectories.

To enhance capabilities and mitigate these failures, a primary driver of recent progress has been test-time scaling (TTS), which increases inference-time computation to yield higher-quality solutions. Despite its promise, standard TTS is computationally expensive due to the cost of repeated sampling from scratch. Consequently, several recent methods have attempted to optimize this process, yet they face significant structural bottlenecks. Heuristic state reuse

*Equal contribution.

†Corresponding author.

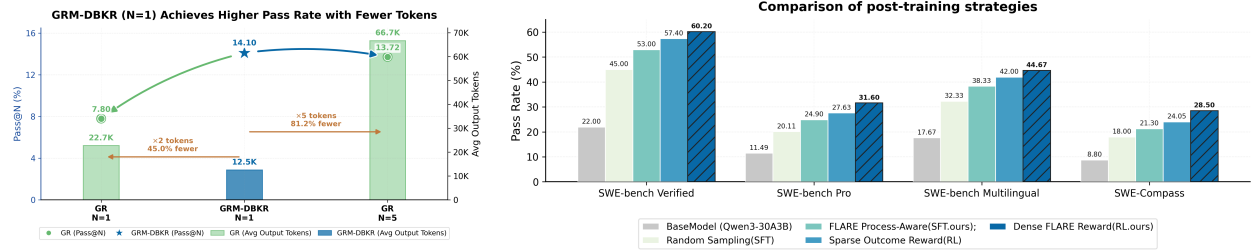


Figure 1 Comparison of test-time strategies (left) and post-training strategies (right).

methods, such as SWE-Replay (Ding and Zhang, 2026), attempt to recycle trajectories by branching at intermediate steps based on structural proxies like reasoning paragraph count. However, without true causal diagnosis, this branching remains a form of heuristic guessing rather than targeted repair. Conversely, offline evolutionary search methods (e.g., Satori-SWE (Zeng et al., 2025)) and multi-agent debate frameworks (e.g., SWE-Search (Antoniades et al., 2025)) rely on delayed scalar reward models or heavyweight negotiations. These approaches merely provide trajectory-level scores or incur exponential token overhead, completely failing to intervene dynamically when a trajectory first derails.

To fundamentally break this “high-cost compute vs. delayed feedback” deadlock, we introduce **FLARE (Full-Lifecycle Alignment and Reward Engine)**. FLARE shifts the paradigm from blind global retries and delayed scalar scoring to real-time, lightweight intervention. Our core insight is that true efficiency stems from equipping the agent with self-diagnostic capabilities via a **Generative Reward Model (GRM)**. To achieve this, we first develop **RADAR (Root-cause Attribution and Diagnostic Analysis Refiner)**, an offline causal-aware diagnostic framework. Through a dual-track data synthesis pipeline utilizing differential causal-chain backtracking, RADAR extracts precise, hindsight-free supervision signals from complex trajectories. This enables us to train a GRM that is uniquely capable of providing actionable, **step-level risk feedback**—outputting structured severity levels, error taxonomies, and causal analyses, rather than an uninterpretable scalar score (Li et al., 2026; Mou et al., 2026).

Crucially, FLARE is the first paradigm to close the loop across the agent’s entire lifecycle. During test-time inference, the GRM operates as an *Active Scaffold*. It asynchronously diagnoses the agent’s previously completed step and, when a critical risk is detected, redirects the continuation before the next proposed action is committed. This intervention does not undo an already executed tool operation. Beyond inference, this exact same step-level diagnostic signal is seamlessly internalized during training. FLARE utilizes the GRM’s output to curate high-quality demonstrations for Supervised Fine-Tuning (SFT) and constructs step-level dense rewards for Agentic Reinforcement Learning (RL), effectively resolving the sparse-reward bottleneck that plagues long-horizon SWE tasks.

In summary, we make the following contributions:

- **The RADAR Framework and GRM Distillation:** We develop an offline diagnostic framework RADAR that utilizes causal-chain backtracking to extract high-fidelity supervision signals, training a lightweight GRM capable of providing structured, step-level diagnostic feedback.
- **A Unified Full-Lifecycle Paradigm:** We introduce FLARE, which integrates the GRM to provide Active Scaffolding for real-time online intervention, while repurposing the same diagnostic signals for SFT data curation and as dense rewards for Agentic RL.
- **Comprehensive Empirical Validation:** We demonstrate that FLARE establishes a new Pareto frontier in test-time scaling: with a strict single-branch budget ($N = 1$), it achieves higher success rates than $N = 5$ global resampling while achieving a 5x reduction in token consumption. Furthermore, integrating FLARE into the training pipeline yields a 19.13% relative performance leap in the SFT stage, and a 9.19% average relative improvement during RL across four major SWE benchmarks, as illustrated in Figure 1.

2 Related Work

Test-Time Scaling and Trajectory Optimization in Long-Horizon Interactive Tasks. Repository-level coding agents such as SWE-agent, AutoCodeRover, and Agentless (Yang et al., 2024; Zhang et al., 2024; Xia et al., 2024) have

made test-time scaling (TTS) a central mechanism for improving long-horizon software engineering performance. Representative TTS optimizations include *heuristic state reuse*, exemplified by SWE-Replay (Ding and Zhang, 2026), which resumes exploration from intermediate states using lightweight heuristics, and *offline evolutionary search*, represented by Satori-SWE (Zeng et al., 2025), which refines complete trajectories using delayed feedback. These approaches improve sample efficiency through state reuse or trajectory refinement. FLARE instead couples branch selection with step-level diagnosis: an online Generative Reward Model (GRM) identifies critical deviations and provides targeted feedback for Active Scaffolding.

Trace Diagnosis and Failure Attribution. A complementary line of research focuses on diagnosing why agents fail in long-horizon environments. Frameworks like CodeTracer (Li et al., 2026) make agent states explicitly traceable, while TRAIL, AgenTracer, and TRACER (Deshpande et al., 2025; Zhang et al., 2025a; Tayebati et al., 2026) study error localization, failure attribution, and trajectory-level risk. Other work identifies the responsible agent or decisive step in multi-agent failures (Zhang et al., 2025b). CodeTracer further demonstrates that localized diagnostic evidence can support replay-based recovery. FLARE advances this diagnostic perspective through RADAR, which converts offline causal-chain analysis into prefix-grounded supervision for a lightweight online GRM. Its structured severity, error taxonomy, and repair rationale are directly reusable for both breakpoint intervention and policy training.

Process Supervision and Critical-Step Learning. To combat reward sparsity, Process Reward Models (PRMs) have been widely adopted to evaluate intermediate reasoning steps, traditionally in mathematical or multimodal reasoning (Lightman et al., 2024; Zhang et al., 2025c). In the agentic SWE domain, methods like STeCa and ATLaS (Wang et al., 2025; Chen et al., 2025) demonstrate that extracting critical steps can outperform full-trajectory behavior cloning. Scalar process scores summarize the quality of intermediate decisions, whereas textual diagnostics additionally expose the failure mechanism and corrective direction. FLARE adopts a conditional generative formulation: the GRM produces structured step-level feedback that supports actionable intervention and is subsequently converted into dense rewards for Train-Time Alignment.

Related approaches also reuse interpretable process feedback. SWE-PRM delivers taxonomy-guided feedback during execution (Gandhi et al., 2025); rubric-supervised critics support trajectory selection and data curation (Wang et al., 2026); and SWE-TRACE combines trajectory construction, process-guided RL, and inference-time guidance (Han et al., 2026). FLARE centers this lifecycle on a shared diagnostic representation learned from RADAR’s failed-trajectory backtracking and active error injection. The same representation specifies when to intervene, what to repair, and how to rank and reward trajectories, connecting causal-aware supervision construction to all three downstream uses.

3 Method

To address reward sparsity (Wei et al., 2025) and delayed post-hoc feedback (Li et al., 2026) in long-horizon SWE tasks, we propose **FLARE** (**F**ull-**L**ifecycle **A**lignment and **R**eward **E**ngine), a novel dense supervision paradigm driven by a lightweight *Generative Reward Model* (GRM) that provides real-time step-level supervision during agent execution (as shown in Figure 2). First, we propose **Root-cause Attribution and Diagnostic Analysis Refiner (RADAR)**, which extracts fine-grained, step-level supervision signals from agent trajectories for GRM optimization (Section 3.1). The GRM then provides real-time, step-wise supervision during SWE agent execution, enabling both test-time and training-time alignment (Section 3.2).

3.1 RADAR for GRM Optimization: From Trajectory to Step-wise Supervision

The key challenge in training a Generative Reward Model (GRM) lies in obtaining high-quality, causally consistent step-level supervision from agent trajectories, where only a small subset of steps truly leads to the final failure. To address this, we propose a unified framework in which **RADAR** serves as a data synthesis engine. Given a trajectory $\tau = (s_1, \dots, s_T)$ for a task q , RADAR identifies causally relevant steps and assigns structured step-wise annotations, thereby producing fine-grained supervision signals. These step-level annotations are then used to train GRM as a conditional generative model that produces structured reward-aligned diagnostic outputs at each step.

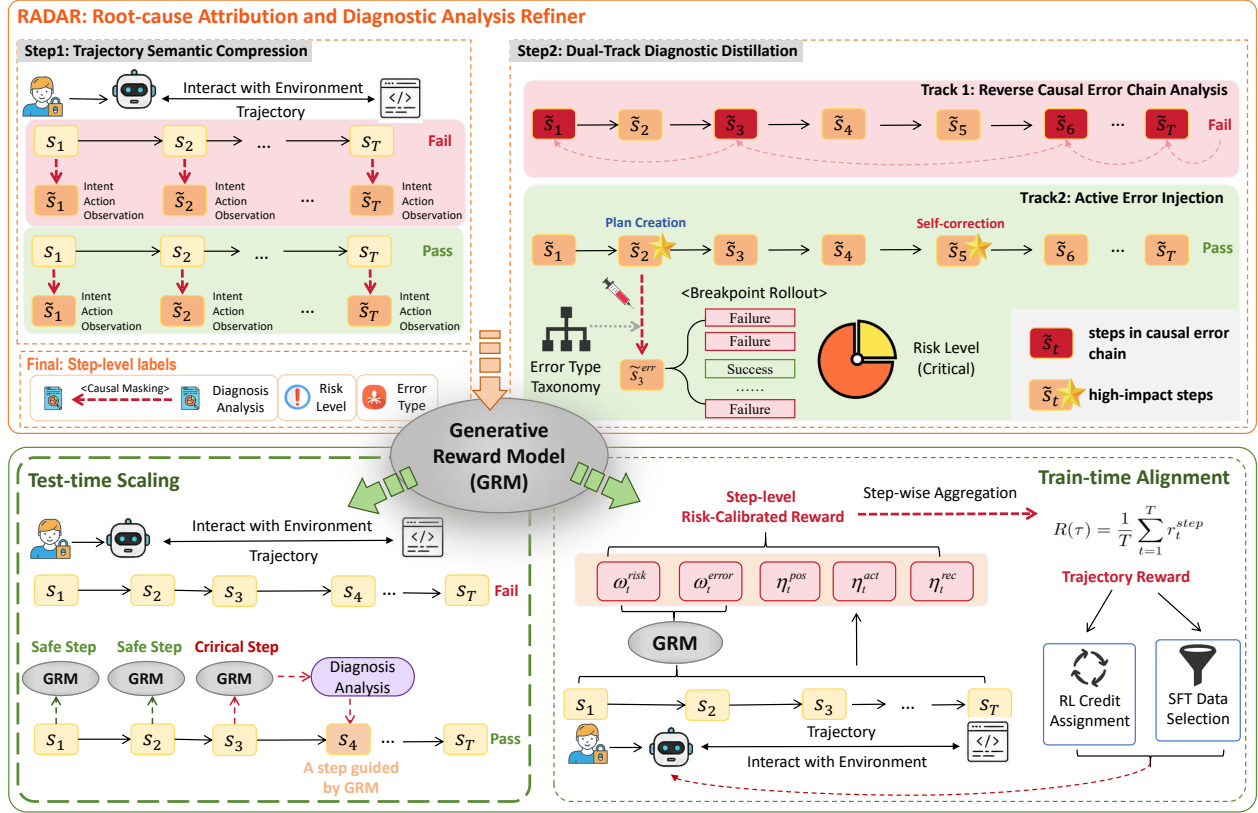


Figure 2 FLARE overview. RADAR extracts step-level supervision signals from agent trajectories for training a lightweight Generative Reward Model (GRM), which provides real-time step-wise feedback during execution, enabling both test-time scaling and training-time alignment.

3.1.1 RADAR as a Step-level Supervision Engine

RADAR extracts step-level supervision signals from agent trajectories to support GRM optimization. It first performs trajectory semantic compression by summarizing each step and retaining only the key information. It then applies a dual-track diagnostic distillation process: for failed trajectories, RADAR conducts reverse causal error-chain analysis to identify the risk level, error type, and diagnostic analysis of each step; for successful trajectories, it introduces active error injection to augment long-tail error-type samples and improve supervision diversity.

Step1: Trajectory Semantic Compression. Each step s_t is first compressed into a structured semantic representation $\tilde{s}_t = (I_t, A_t, O_t) = \phi(s_t)$, where I_t , A_t , and O_t denote the intent, action, and observation, respectively, all represented as natural language summaries. This abstraction provides a unified basis for subsequent dual-track diagnostic distillation.

Step2: Dual-track Diagnostic Distillation

- **Track 1: Reverse Causal Error Chain Analysis.** For failed trajectories, RADAR identifies a root-prioritized set of causally relevant steps via backward tracing:

$$\mathcal{C} = \mathcal{B}(\tau; \mathcal{G}) = (\tilde{s}_{t_1}, \dots, \tilde{s}_{t_k}) \quad (1)$$

where $\mathcal{G}$ contains global signals (e.g., patch differences and failed test cases), and each $\tilde{s}_{t_i}$ is retained through an evidence-supported dependency on the observed failure. RADAR first anchors a behavioral or contract discrepancy, then traces backward through steps that introduce, propagate, or reinforce it. The earliest supported source is prioritized for replay; unrelated exploration is excluded, while independent defects may retain separate roots. This causal-aware procedure targets explanatory error chains rather than a formally identified minimal causal set. Each

step in the causal chain is assigned a structured label:

$$y_{t_i} = (z_{t_i}, c_{t_i}, d_{t_i}) = \mathcal{A}(\tilde{s}_{t_i}; \mathcal{C}, \mathcal{G}) \quad (2)$$

where $\mathcal{A}$ denotes the annotation function conditioned on the full causal chain $\mathcal{C}$ and global signals $\mathcal{G}$. Here, z_{t_i} denotes the risk level, c_{t_i} denotes the error type under a predefined taxonomy (see Appendix B for details), and d_{t_i} is an offline diagnostic analysis derived from the global signals.

- **Track 2: Active Error Injection.** To improve error diversity and balance, RADAR perturbs successful trajectories by injecting errors into predefined high-impact steps, including *Plan Creation*, *Critical Observation*, *Critical Action*, and *Self-Correction*. Key steps are selected as:

$$\mathcal{K} = \{\tilde{s}_t \mid s_t \in \tau^+, \kappa(\tilde{s}_t) \in \mathcal{H}\}, \quad (3)$$

where $\kappa(\cdot)$ denotes a step-type classifier and $\mathcal{H}$ is the predefined high-impact category set. Unlike semantic compression ϕ , κ assigns a functional role to each compressed step. For each selected step, RADAR samples an error label $\ell \sim p(\ell)$ and applies a label-specific perturbation policy: $\tilde{s}_t^{\text{err}} = \mathcal{I}(\tilde{s}_t \mid \ell, \pi_\ell)$. The modified trajectory is then re-executed via breakpoint rollout, producing stochastic rollouts $\{\mathcal{T}_t^{(n)}\}_{n=1}^N$ with empirical recovery rate: $\hat{p}_t = \frac{1}{N} \sum_{n=1}^N \mathbb{I}(\mathcal{T}_t^{(n)} \text{ solves task})$. Injected errors are finally categorized using a recovery threshold $\theta \in [0, 1]$:

$$z_t = \begin{cases} \text{critical}, & \hat{p}_t \leq \theta, \\ \text{trivial}, & \hat{p}_t > \theta. \end{cases} \quad (4)$$

The two tracks provide complementary supervision: natural failures supply observed error chains, while perturbations of successful prefixes expose underrepresented failure modes. Track 2 severity is grounded in observed recoverability under the continuation policy and rollout budget, rather than assigned solely from the injected error type. The augmentation targets coverage, not the natural frequency of errors.

Final: Step-level Labels Through the above process, each trajectory $\tau^{(i)}$ is converted into a sequence of annotated tuples:

$$\{(\tilde{s}_t^{(i)}, y_t^{(i)})\}_{t=1}^T \quad (5)$$

forming a large-scale dataset with fine-grained step-wise supervision signals. Causal-prefix distillation separates the evidence used to construct labels from the evidence available to an online explanation:

$$\tilde{d}_t = \mathcal{M}(d_t; q, s_t, \mathcal{H}_t), \quad \mathcal{H}_t = (s_1, \dots, s_{t-1}).$$

Here, $\mathcal{M}$ rewrites the offline analysis using only the task, current step, and preceding history. Reference patches and final outcomes may inform offline attribution, but they are not available to the runtime GRM and must not appear as prefix-invisible evidence in $\tilde{d}_t$. A semantic compliance gate checks taxonomy consistency, unsupported anchors, and patch-, future-, or test-oracle leakage, routing violations for regeneration. The resulting target is $\tilde{y}_t = (z_t, c_t, \tilde{d}_t)$; the full offline analysis is not used directly. This defines the operational information boundary of hindsight-free supervision without asserting a formal zero-leakage guarantee. The annotation, distillation, and compliance prompts are provided in Appendix C.

3.1.2 GRM Optimization with Step-wise Supervision

We train GRM as a conditional generative model:

$$p_\psi(\tilde{y}_t \mid q, s_t, \mathcal{H}_t), \quad (6)$$

where ψ denotes the GRM parameters and $\mathcal{H}_t$ is the preceding interaction history. Given the RADAR-constructed dataset, GRM is optimized to generate structured outputs including risk level, error type, and causal-consistent diagnostic explanations.

In practice, we perform supervised fine-tuning on Qwen3.5-4B, enabling a lightweight GRM to produce step-wise risk assessment and interpretable error analysis conditioned on trajectory context. The actor and critic solve different

problems: the actor searches for and implements a repair, whereas the GRM diagnoses a completed step against the task, available evidence, and a fixed error taxonomy. Structured supervision concentrates the smaller model on this diagnostic role without requiring it to solve the entire issue independently. The cross-model and cross-scaffold results in Section 4.2 support this complementary use.

3.2 GRM for Test-Time Intervention and Training-Time Alignment

Our GRM provides real-time step-level supervision during SWE agent execution, enabling both test-time intervention and train-time alignment. Its shared diagnostic representation serves three complementary functions: severity determines when to intervene, the diagnostic rationale specifies what to repair, and severity together with taxonomy information defines the process score used for SFT selection and dense reward construction. The following mechanisms reuse one diagnostic model rather than requiring a separate evaluator for each stage.

3.2.1 Test-Time Scaling.

At each step, the agent proposes a tentative action $\hat{s}_t$ while GRM asynchronously diagnoses the previously completed step s_{t-1} , producing a diagnostic triplet $y_{t-1} = (z_{t-1}, c_{t-1}, d_{t-1}^{\text{diag}})$. The intervention therefore acts at the boundary before the next action is committed; it does not assume that an already executed tool operation can be undone. If the diagnosis indicates a non-critical state, the agent proceeds with $\hat{s}_t$; otherwise, it regenerates the next step conditioned on the diagnostic feedback:

$$s_t = \begin{cases} \hat{s}_t, & z_{t-1} \in \{\text{safe}, \text{trivial}\}, \\ \pi(d_{t-1}^{\text{diag}}; \mathcal{H}_t), & z_{t-1} = \text{critical}, \end{cases} \quad (7)$$

where $\pi(\cdot)$ denotes regeneration conditioned on diagnostic feedback and interaction history.

3.2.2 Train-Time Alignment.

To optimize the agent with fine-grained supervision, we formulate a step-level risk-calibrated reward function that captures diagnostic severity and recoverability, based on empirical statistics derived from breakpoint rollouts on offline GRM-annotated trajectories. This connects reward construction to executable continuation outcomes rather than assigning fixed penalties solely from taxonomy labels.

First, let $\bar{P}_{\text{succ}}$ and $\bar{P}_{\text{fail}}$ denote the global baseline success and failure rates, respectively, obtained by randomly sampling steps from offline trajectories for breakpoint rollout. For the t -th step in a trajectory, let $z_t \in \{\text{safe}, \text{trivial}, \text{critical}\}$ denote its risk severity level, and c_t denote the specific error type triggered. We define the **Risk Severity Factor** ω_t^{risk} and the **Error Importance Factor** ω_t^{err} :

$$\omega_t^{\text{risk}} = \begin{cases} \max\left(0, \frac{P_{\text{succ}}(z_t) - \bar{P}_{\text{succ}}}{\bar{P}_{\text{succ}}}\right), & \text{if } z_t = \text{safe}, \\ \max\left(0, \frac{P_{\text{fail}}(z_t) - \bar{P}_{\text{fail}}}{\bar{P}_{\text{fail}}}\right), & \text{if } z_t \in \{\text{trivial}, \text{critical}\}, \end{cases} \quad (8)$$

$$\omega_t^{\text{err}} = \max\left(0, \frac{\bar{P}_{\text{succ}} - P_{\text{succ}}(c_t)}{\bar{P}_{\text{succ}}}\right), \quad (9)$$

where $P_{\text{succ}}(\cdot)$ and $P_{\text{fail}}(\cdot)$ denote the empirically estimated success and failure rates, respectively, precomputed from offline breakpoint rollouts conditioned on a specific risk severity level or error type. These factors effectively translate offline statistical patterns into reward or penalty weights for critical steps.

Combining these statistical priors with the dynamic characteristics of the trajectory, we formalize the step-level risk-calibrated reward r_t^{step} for step t as:

$$r_t^{\text{step}} = \begin{cases} \omega_t^{\text{risk}} \cdot \eta_t^{\text{pos}} \cdot \eta_t^{\text{act}}, & \text{if } z_t = \text{safe}, \\ -\omega_t^{\text{risk}} \cdot \omega_t^{\text{err}} \cdot \eta_t^{\text{pos}} \cdot \eta_t^{\text{act}} \cdot \eta_t^{\text{rec}}, & \text{if } z_t \in \{\text{trivial}, \text{critical}\}. \end{cases} \quad (10)$$

where three dynamic adjustment coefficients tailored to the current trajectory state are defined as follows:

- **Position-aware weight** $\eta_t^{\text{pos}} = \frac{1 + \log t}{1 + \log T}$: Emphasizes proximity to the submitted solution while retaining positive weight for early decisions. Logarithmic growth limits concentration on the final steps, and normalization sets the terminal weight to one (T is the trajectory length).

- **Action rarity weight** $\eta_t^{\text{act}} = 1 - \frac{N(a_t)}{T}$: Downweights reward contributions from high-frequency actions, where $N(a_t)$ is the occurrence count of action a_t . Because this factor multiplies both positive and negative rewards, it attenuates their magnitudes rather than imposing an additional negative penalty.
- **Recurrence penalty** $\eta_t^{\text{rec}} = 1 + \frac{n_t^{(c)}(n_t^{(c)} - 1)}{2}$, where $n_t^{(c)}$ counts occurrences of error type c_t up to step t . The pair-count term increases the penalty as an error recurs, targeting repeated failures rather than repetition of an action alone.

This step-level formulation yields a dense reward that jointly captures diagnostic severity, temporal position, action frequency, and error recurrence. The modifiers encode explicit preferences for trajectory quality rather than estimates of individual causal effects. Their utility is supported by the component ablations in Section 4.3, where severity and error importance provide the largest gains and the remaining terms refine the score. We then define the trajectory-level reward as the normalized aggregation of step-level signals:

$$R(\tau) = \frac{1}{T} \sum_{t=1}^T r_t^{\text{step}}, \quad (11)$$

where T is the trajectory length and r_t^{step} is the GRM-derived step reward. This trajectory-level score provides a unified criterion for evaluating the overall quality of generated rollouts, and is further used to guide both supervised and reinforcement learning stages.

SFT Data Selection. To improve SFT quality, we move away from random trajectory sampling and instead select high-quality demonstrations based on the trajectory-level score $R(\tau)$. Specifically, we retain the top-ranked subset of trajectories with the highest $R(\tau)$, enabling the model to better internalize stable reasoning patterns and self-correction behaviors.

Dense Reward Integration. Prior work optimizes a sparse trajectory-level signal, assigning each trajectory $\tau^{(i)}$ a binary pass/fail reward $R^{\text{out}}(\tau^{(i)}) \in \{0, 1\}$. We extend this by incorporating a reranking reward that aggregates step-level supervision signals, yielding a unified trajectory reward $r^{(i)} = R(\tau^{(i)}) + R^{\text{out}}(\tau^{(i)})$.

Within each group, we compute relative advantages via z-score normalization:

$$\hat{A}_i = \frac{r^{(i)} - \mu_G}{\sigma_G + \epsilon}, \quad \mu_G = \frac{1}{G} \sum_j r^{(j)}, \quad \sigma_G = \sqrt{\frac{1}{G} \sum_j (r^{(j)} - \mu_G)^2}.$$

All tokens in $\tau^{(i)}$ share the same $\hat{A}_i$. We apply standard GRPO-based optimization (Shao et al., 2024), using a trajectory-level advantage to integrate the dense step rewards. Here, *dense* refers to supervision constructed throughout the trajectory, not to a separate advantage for every step or token. Trajectories with the same terminal Pass/Fail outcome can receive different rewards through $R(\tau)$, allowing policy optimization to distinguish their process quality even after aggregation.

4 Experiments

We evaluate FLARE along three research questions, each corresponding to one use of the GRM across the agent lifecycle:

- **RQ1: Test-Time Intervention and Inference Efficiency.** Can the GRM halt and repair failing trajectories online more effectively than global resampling or post-hoc guided replay?
- **RQ2: Process-Aware Trajectory Scoring and SFT Data Curation.** Does the GRM-derived trajectory score identify higher-quality demonstrations for SFT than random or rubric-critic selection?
- **RQ3: Dense Reward for Agentic Reinforcement Learning.** Can GRM-derived dense rewards improve long-horizon SWE policy optimization over outcome-only sparse rewards?

4.1 Experimental Setup

Benchmarks and agents. We evaluate FLARE on four repository-level SWE benchmarks: SWE-bench Verified, SWE-bench Pro (Deng et al., 2025), SWE-bench Multilingual (Khandpur et al., 2025), and SWE-Compass (Xu et al., 2025). These benchmarks cover issue-resolution tasks with different difficulty levels, repository structures, and programming languages. RQ1 covers GLM-5, Kimi K2.5, Qwen3.5-Plus, and DeepSeek V3.2 (Zhipu AI, 2026; Moonshot AI, 2026; Qwen Team, 2026; DeepSeek-AI, 2025), deployed with Claude Code and Cline-style scaffolds (Anthropic, 2025; Cline Team, 2024); aggregate and disaggregated results are distinguished below. RQ2 fine-tunes Qwen3-30B-A3B (Qwen Team, 2025). Unless otherwise specified, comparisons within each experiment use the same repository-level test harness, task prompts, branch budgets, and base-agent configurations.

Test-time intervention baselines. For RQ1, we compare four repair strategies that isolate the value of full reruns, checkpoint reuse, offline diagnosis, and online GRM intervention:

- **GR** (*Global Rollout*) is the standard Pass@ N baseline: for each task, the agent samples N complete trajectories from the initial task state, and the task is counted as solved if any trajectory passes. GR does not use checkpoints, branching, or reward-model feedback.
- **Blind-BKR** (*Blind Breakpoint Rollout*) isolates the effect of state reuse. After an initial failure, the environment is restored to the first high-risk step, and the agent continues from that checkpoint without diagnostic guidance.
- **RADAR-DBKR** (*RADAR diagnostic-guided breakpoint rollout*) is a post-hoc diagnostic replay baseline: it waits for a complete failed trajectory, truncates the trajectory at the root-cause step identified by RADAR, appends RADAR diagnostic feedback as user content, and performs a localized breakpoint rollout.
- **GRM-DBKR** (*Online GRM diagnostic-guided breakpoint rollout*) is the online FLARE intervention method. The GRM diagnoses completed steps and, when critical risk is detected, provides localized diagnostic feedback to regenerate the next action before it is committed.

Budget protocol. For a repair strategy, N denotes the number of repair branches allowed for each task. For Global Rollout, these branches are full trajectories sampled from the initial state. For checkpoint-based methods, they start from the selected or detected checkpoint. A task is counted as repaired if at least one branch produces a patch that passes the repository tests. Costs include all budgeted branches, not only the successful branch. Under the reported conditional protocol, GR, Blind-BKR, and RADAR-DBKR include the initial baseline rollout, while GRM-DBKR counts its online-guided rollouts. Thus, matching N controls branch count, not the number of full trajectories or model calls.

Trajectory data and training comparisons. The SFT and RL data pools are built from *multi-swe-rl*, *rebench*, and supplementary GitHub instances. For trajectory scoring and SFT selection, we generate 18,000 complete inference trajectories via instance distillation, including 6,379 passing trajectories and 11,621 failing trajectories. For RQ2, all SFT data-selection methods choose 3K trajectories from this pool and fine-tune Qwen3-30B-A3B with the same optimization recipe; only the selection signal changes. We compare **Random Sampling**, **Rubric-Supervised Critic Selection**, and **FLARE Process-Aware Selection**. The rubric-critic baseline follows Wang et al. (2026), which learns a critic from sparse real-world outcomes using rubric-based supervision. It is distinct from the heuristic **Critical Model** baseline in the ROC-AUC comparison. For RQ3, all RL variants start from the same initialization and use the same training data source, rollout budget, optimizer, learning-rate schedule, and evaluation harness; only the reward signal differs.

Metrics. For RQ1, we report fail-to-pass (F2P) rate on initially failed tasks and pass-to-pass (P2P) rate on initially successful tasks. F2P uses Pass@ N , where a task succeeds if any of the N branches passes. P2P uses Avg@ N , the mean pass rate across branches, to assess preservation of successful behavior. These task/trajectory-level metrics are distinct from the F2P/P2P test-case labels used by the SWE-bench harness. We also report average base-agent output tokens and agent turns under the budget protocol; token counts exclude agent inputs and GRM inputs/outputs. For RQ2 and RQ3, the downstream metric is benchmark pass rate, also referred to as resolve rate. The main tables report point estimates.

For training-free trajectory scoring in RQ2, we report ROC-AUC. Given a trajectory score $R(\tau)$ and binary outcome label $y \in \{0, 1\}$, where $y = 1$ denotes a passing trajectory, let $\mathcal{P} = \{i : y_i = 1\}$ and $\mathcal{N} = \{j : y_j = 0\}$. ROC-AUC is

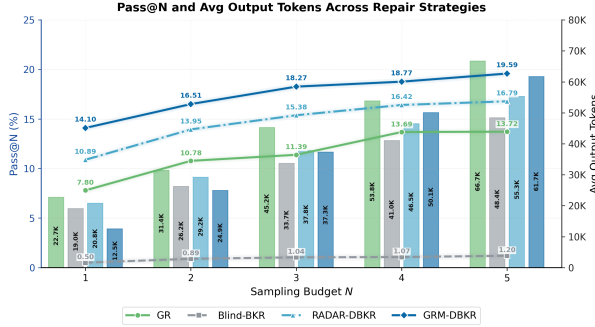


Figure 3 F2P repair curves under increasing branch budgets, aggregated across RQ1 model–scaffold settings.

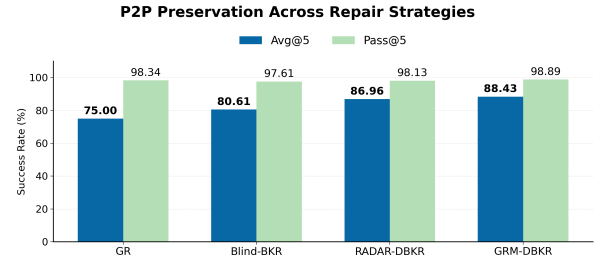


Figure 4 P2P preservation across RQ1 settings. Avg@5 measures mean continuation success; Pass@5 measures success in at least one of five continuations.

computed as:

$$\text{AUC} = \frac{1}{|\mathcal{P}||\mathcal{N}|} \sum_{i \in \mathcal{P}} \sum_{j \in \mathcal{N}} \left[\mathbb{I}(R(\tau_i) > R(\tau_j)) + \frac{1}{2} \mathbb{I}(R(\tau_i) = R(\tau_j)) \right]. \quad (12)$$

This metric measures the probability that a randomly sampled passing trajectory is ranked above a randomly sampled failing one, with ties receiving half credit. It is appropriate because the GRM score is used as a ranking signal for top- K data selection rather than as a calibrated success probability.

GRM output format. The GRM is trained and used with a structured XML output contract so that its signal can be reused for online repair, trajectory scoring, and dense reward construction. For risky steps, the model outputs a causal analysis, one or more taxonomy-backed issue tags, and a severity label:

GRM output example

```
<risk>
  <analysis>
    [Detailed causal analysis and actionable repair advice]
  </analysis>
  <issue rubric="E1.1" label="Misunderstood Requirements"/>
  <issue rubric="E1.4" label="Wrong Tool Selection"/>
  <severity>critical</severity>
</risk>
```

This structured output prevents the GRM from acting as a monolithic scalar reranker and instead exposes severity, taxonomy, and repair guidance for downstream use.

4.2 RQ1: Test-Time Repair and Inference Efficiency

Evaluation protocol. We split tasks by an initial baseline rollout, uniformly distributed across four models and two agent frameworks. Tasks that fail form the F2P subset, where success is measured by Pass@ N over repair branches. Tasks that pass form the P2P subset, where we report Avg@ N to measure whether interventions preserve successful behavior. We use $N = 5$ as the maximum repair budget. This decomposition tests two complementary requirements for an intervention: rescuing failures and maintaining successful continuations. The resulting conditional rates diagnose intervention behavior rather than directly reporting full-benchmark solve rates. Table 1 and Figures 3–4 summarize the evaluated model–scaffold settings; Figure 5 provides the corresponding slices.

F2P repair and efficiency. Table 1 shows that GRM-DBKR achieves the best repair rate at both budgets. With a single branch, it reaches 14.10% F2P, nearly doubling GR at $N = 1$ and exceeding the 13.72% obtained by GR at $N = 5$. The corresponding agent-output ratio is $66,711/12,517 \approx 5.33$, supporting the approximately $5\times$ reduction in base-agent output under this protocol. At $N = 5$, GRM-DBKR further improves to 19.59%, exceeding GR by 5.87 percentage points. Figure 3 shows that GRM-DBKR continues to benefit from additional branches over the evaluated range, while offline and blind replay methods saturate earlier. These results improve the observed repair-rate–agent-output trade-off.

Table 1 Combined test-time intervention performance across the RQ1 model–scaffold settings. N denotes the repair-branch budget. Costs follow the accounting in Section 4.1 and measure base-agent output, excluding agent inputs, GRM inputs/outputs, and latency. Subscripts denote relative changes from GR at the same N ; green is favorable and red unfavorable.

Method	Budget	Score (%)	Agent Output Tokens	Avg. Turns
Fail-to-Pass Subset			[Score: F2P Pass@N]	
GR	$N = 1$	7.80	22,738	103.9
	$N = 5$	13.72	66,711	307.8
Blind-BKR	$N = 1$	0.50 _{-93.59}	19,046	86.6
	$N = 5$	1.20 _{-91.25}	48,393	221.8
RADAR-DBKR (Ours)	$N = 1$	10.89 _{+39.62}	20,814	92.8
	$N = 5$	16.79 _{+22.38}	55,284	252.6
GRM-DBKR (Ours)	$N = 1$	14.10 _{+80.77}	12,517	57.6
	$N = 5$	19.59 _{+42.78}	61,718	284.2
Pass-to-Pass Subset			[Score: P2P Avg@N]	
GR	$N = 5$	75.00	10,861	50.76
Blind-BKR	$N = 5$	80.61 _{+7.48}	7,836	37.6
RADAR-DBKR (Ours)	$N = 5$	86.96 _{+15.95}	7,624	35.94
GRM-DBKR (Ours)	$N = 5$	88.43 _{+17.91}	9,400	48.15

P2P preservation. On initially successful trajectories, GRM-DBKR obtains the highest P2P rate, 88.43%, while reducing agent output relative to GR. Figure 4 further shows that GRM-DBKR achieves the highest Avg@5 and Pass@5 among the evaluated strategies. This supports the stability of guided continuations under the evaluated protocol. A failed continuation is not itself evidence of GRM-induced harm, since P2P also reflects the stochasticity of the continuation policy.

Why online intervention changes the cost-performance frontier. The headline numbers in Table 1 reflect a structural contrast in the timing of feedback. RADAR-DBKR waits for a completed trajectory to perform post-hoc attribution; GRM-DBKR can diagnose and redirect the first live trajectory. Global Rollout samples fresh trajectories without diagnostic feedback and does not intrinsically require a prior failure. The initial-rollout cost charged to GR in this experiment follows the conditional evaluation protocol, rather than an algorithmic requirement of global sampling.

At $N = 1$, the reported protocol charges GR, Blind-BKR, and RADAR-DBKR for the baseline plus one repair branch, while GRM-DBKR uses its online-guided rollout. GRM-DBKR requires 12,517 agent-output tokens and 57.6 turns, compared with 22,738 tokens and 103.9 turns for GR, while increasing F2P from 7.80% to 14.10%. At $N = 5$, GRM-DBKR uses 61,718 agent-output tokens versus GR’s 66,711, with the 5.87-point F2P gain noted above. These counts reflect the actor’s guided continuations, not tokens generated by the GRM. Online intervention changes when corrective computation is spent and reduces redundant actor generation in the evaluated setting.

Robustness across settings. Figure 5 further evaluates whether the RQ1 gains persist across model, scaffold, language, task-difficulty, and error-taxonomy slices. Gains are consistent across both Cline and Claude Code scaffolds and across GLM-5, Kimi K2.5, Qwen3.5-Plus, and DeepSeek V3.2, supporting the GRM’s use as a plug-and-play intervention module across the evaluated settings. Across the nine evaluated language slices, FLARE yields positive absolute gains, including large lifts in TypeScript (+7.91 points) and Python (+6.39 points), with a positive gain also observed in C (+1.46 points).

The largest gains appear at the capability boundary. Using baseline rerun performance as a difficulty proxy, GRM-DBKR provides larger gains in the harder task buckets: in the lowest-baseline-success bucket, FLARE raises success from 1.0% to 10.6%. Taxonomy-level slicing also clarifies the method’s operational scope. GRM-guided repair is strongest for process-level and validation failures, such as E4.1 Missing Validation (30.2% repair) and E2.1 Missed Retrieval

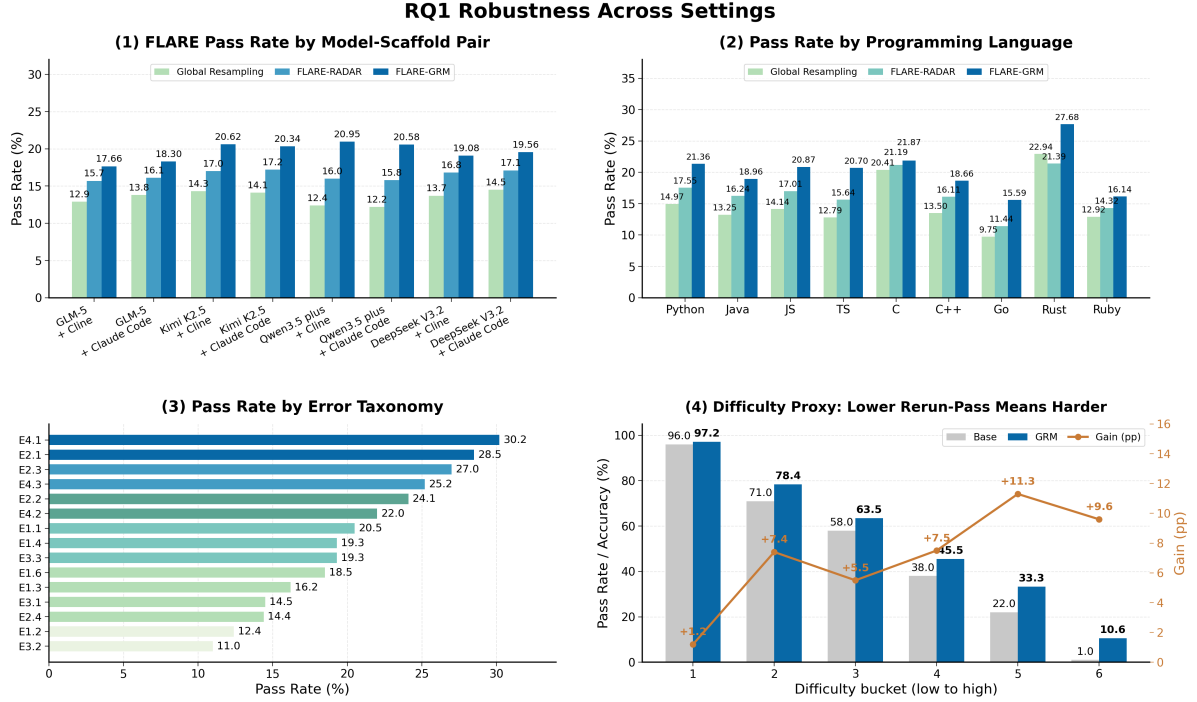


Figure 5 RQ1 robustness across settings. We report FLARE performance across model–scaffold combinations, programming languages, error taxonomy categories, and task-difficulty buckets. Across these slices, FLARE maintains positive gains over global resampling and RADAR-guided replay, with the largest improvements appearing in harder tasks where ordinary reruns rarely succeed.

(28.5% repair), but is less effective in the reported E3.2 and E1.2 slices. This pattern supports the intended interpretation: localized diagnosis is most useful when it redirects an otherwise viable repair process, while the final repair remains bounded by the base agent’s capabilities.

4.3 RQ2: Process-Aware Trajectory Scoring and SFT Data Curation

Training-free score validation. Before using the FLARE process score for fine-tuning data selection, we test whether it aligns with ground-truth trajectory outcomes while keeping both the trained GRM and actor fixed. Here, *training-free* refers to score evaluation, not to the construction of the GRM. Table 2 reports ROC-AUC over the 18,000-trajectory pool described in Section 4.1. The full FLARE Process-Aware score reaches 75.74% ROC-AUC, outperforming chance ranking (50.00%) and the heuristic Critical Model baseline (69.34%). Aggregating step-level diagnostics therefore provides a useful ranking signal before downstream fine-tuning.

The leave-one-out ablation confirms that all five components contribute to predictive power. Consistent with the downstream SFT ablation below, the **Risk Severity Factor** (ω_t^{risk}) and **Error Importance Factor** (ω_t^{err}) are the most important intrinsic components: removing them causes the steepest absolute ROC-AUC drops, 5.62 and 4.68 points, respectively. The behavioral and temporal modifiers further refine the ranking among trajectories with similar diagnostic severity.

SFT data selection. We select the top 3,000 trajectories from the distillation pool under each selection rule and fine-tune the same Qwen3-30B-A3B base model. As visualized in Figure 1, and shown in the top block of Table 3, FLARE Process-Aware Selection consistently outperforms random sampling and Rubric-Supervised Critic Selection across all four benchmarks. It improves the average pass rate by 19.13% relative to random sampling, with especially large gains on SWE-bench Pro and SWE-Compass.

Component ablation. The bottom block of Table 3 shows that each score component contributes to downstream data quality. Removing the risk severity factor ω_t^{risk} or error importance factor ω_t^{err} causes the largest average SFT drops, 8.52% and 7.18%, respectively. The action-rarity, recurrence, and position-aware terms provide smaller but

Table 2 Training-free leave-one-out ablation of the FLARE trajectory score. ROC-AUC is computed over the 18,000-trajectory pool with Pass/Fail outcomes, without updating the trained GRM or actor. Critical Model is a separate heuristic ranking baseline, not the rubric critic used in the SFT comparison. Subscripts denote absolute drops in percentage points from the full score.

Scoring Rule	ω_t^{risk}	ω_t^{err}	η_t^{act}	η_t^{rec}	η_t^{pos}	ROC-AUC (%)
Random Sampling	—	—	—	—	—	50.00
Critical Model	—	—	—	—	—	69.34
FLARE Process-Aware (Ours)	✓	✓	✓	✓	✓	75.74
w/o Risk Severity Factor (ω_t^{risk})	×	✓	✓	✓	✓	70.12 _{-5.62}
w/o Error Importance Factor (ω_t^{err})	✓	×	✓	✓	✓	71.06 _{-4.68}
w/o Action rarity weight (η_t^{act})	✓	✓	×	✓	✓	72.95 _{-2.79}
w/o Recurrence penalty (η_t^{rec})	✓	✓	✓	×	✓	73.20 _{-2.54}
w/o Position-aware weight (η_t^{pos})	✓	✓	✓	✓	×	73.55 _{-2.19}

Table 3 Combined SFT data-selection performance and scoring-component ablation. All methods fine-tune the Qwen3-30B-A3B base model with 3K selected trajectories.

Selection Strategy	Verified	Pro	Multilingual	Compass	Avg.
SFT Data Selection Strategies					
Random Sampling	45.00	20.11	32.33	18.00	28.86
Rubric-Supervised Critic Selection	47.00 _{+4.44}	21.20 _{+5.42}	33.67 _{+4.14}	18.90 _{+5.00}	30.19 _{+4.61}
FLARE Process-Aware (Ours)	53.00 _{+17.78}	24.90 _{+23.82}	38.33 _{+18.56}	21.30 _{+18.33}	34.38 _{+19.13}
Ablation of Process Score Components					
FLARE Process-Aware (Ours)	53.00	24.90	38.33	21.30	34.38
w/o Risk Severity Factor (ω_t^{risk})	49.20 _{-7.17}	22.16 _{-11.00}	35.00 _{-8.69}	19.45 _{-8.69}	31.45 _{-8.52}
w/o Error Importance Factor (ω_t^{err})	49.60 _{-6.42}	21.89 _{-12.09}	36.33 _{-5.22}	19.80 _{-7.04}	31.91 _{-7.18}
w/o Action rarity weight (η_t^{act})	50.00 _{-5.66}	23.26 _{-6.59}	36.00 _{-6.08}	19.95 _{-6.34}	32.30 _{-6.05}
w/o Recurrence penalty (η_t^{rec})	51.60 _{-2.64}	23.80 _{-4.42}	37.00 _{-3.47}	20.85 _{-2.11}	33.31 _{-3.11}
w/o Position-aware weight (η_t^{pos})	50.80 _{-4.15}	23.39 _{-6.06}	37.33 _{-2.61}	20.45 _{-3.99}	32.99 _{-4.04}

consistent gains by refining the ranking among trajectories with similar diagnostic severity. This mirrors the training-free ROC-AUC evidence in Table 2: the same two diagnostic factors are also the most important for ranking passing trajectories above failing ones before any fine-tuning occurs.

Takeaway for RQ2: Structured process-aware scoring selects stronger SFT demonstrations than the evaluated random and rubric-critic baselines, with consistent gains across four benchmarks and supporting component ablations.

4.4 RQ3: Dense Reward for Agentic Reinforcement Learning

Protocol. RQ3 tests whether the same GRM signal remains useful when used as an RL reward. We compare sparse-reward RL, which uses only the final repository-level Pass/Fail outcome, with FLARE-RL, which augments policy optimization with GRM-derived dense rewards. Both variants use the same model initialization, data source, rollout budget, optimizer, learning-rate schedule, and evaluation harness; only the reward construction differs. Step rewards are aggregated before computing the shared trajectory advantage, following Section 3.2.

Results. Table 4 shows that FLARE-RL improves over sparse-reward RL on all four benchmarks: +2.80 points on SWE-bench Verified, +3.97 on SWE-bench Pro, +2.67 on SWE-bench Multilingual, and +4.45 on SWE-Compass. Since the two RL variants differ only in the reward signal, this consistent improvement supports dense process supervision as a useful complement to terminal correctness, even when optimization uses a shared trajectory-level advantage.

Takeaway for RQ3: The GRM signal transfers from offline trajectory ranking to online policy optimization, completing

Table 4 RQ3: RL comparison between outcome-only sparse rewards and GRM-derived dense rewards under the shared initialization and training protocol in Section 4.1. Scores are pass rates on the four evaluation benchmarks; Avg. is their unweighted mean.

Reward Signal	Verified	Pro	Multilingual	Compass	Avg.
Sparse outcome reward	57.40	27.63	42.00	24.05	37.77
Dense FLARE reward (Ours)	60.20 _{+4.88}	31.60 _{+14.37}	44.67 _{+6.36}	28.50 _{+18.50}	41.24 _{+9.19}

the lifecycle use of FLARE across inference, SFT, and RL.

5 Conclusion

FLARE unifies test-time inference and train-time alignment through structured GRM diagnostics, connecting causal-aware supervision construction, online intervention, and dense reward learning in long-horizon SWE tasks. Under the conditional repair protocol, one GRM-guided branch achieves higher F2P than five global reruns while reducing base-agent output by approximately $5\times$, improving the observed repair-rate-agent-output frontier. Reusing the same diagnostic signal for SFT selection and dense RL rewards delivers relative performance gains of 19.13% and 9.19%, respectively. Together, these results establish a unified framework for turning trajectory diagnosis into actionable supervision across the agent lifecycle.

References

- Anthropic. Claude 3.7 sonnet and claude code. <https://www.anthropic.com/news/claude-3-7-sonnet>, 2025.
- Antonis Antoniadis, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing software agents with monte carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations*, 2025.
- Zhixun Chen, Ming Li, Yuxuan Huang, Yali Du, Meng Fang, and Tianyi Zhou. ATLaS: Agent tuning via learning critical steps. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 25334–25349, 2025.
- Cline Team. Cline: Autonomous coding agent for VS Code. <https://github.com/cline/cline>, 2024.
- DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models. <https://arxiv.org/abs/2512.02556>, 2025.
- Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.
- Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. TRAIL: Trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638*, 2025.
- Yifeng Ding and Lingming Zhang. SWE-Replay: Efficient test-time scaling for software engineering agents. *arXiv preprint arXiv:2601.22129*, 2026.
- Shubham Gandhi, Jason Tsay, Jatin Ganhotra, Kiran Kate, and Yara Rizk. When agents go astray: Course-correcting SWE agents with PRMs. *arXiv preprint arXiv:2509.02360*, 2025.
- Hao Han, Jin Xie, Xuehao Ma, Weiquan Zhu, Ziyao Zhang, Zhiliang Long, Hongkai Chen, and Qingwen Ye. SWE-TRACE: Optimizing long-horizon SWE agents through rubric process reward models and heuristic test-time scaling. *arXiv preprint arXiv:2604.14820*, 2026.
- Kabir Khandpur, Kilian Lieret, Carlos E. Jimenez, Ofir Press, and John Yang. SWE-bench Multilingual. <https://www.swebench.com/multilingual.html>, 2025. Official benchmark documentation. Accessed September 16, 2026.
- Han Li, Yifan Yao, Letian Zhu, Rili Feng, Hongyi Ye, Jiaming Wang, Yancheng He, Pengyu Zou, Lehan Zhang, Xinpeng Lei, Haoyang Huang, Ken Deng, Ming Sun, Zhaoxiang Zhang, He Ye, and Jiaheng Liu. CodeTracer: Towards traceable agent states. *arXiv preprint arXiv:2604.11641*, 2026.
- Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024.

- Moonshot AI. Kimi K2.5. <https://github.com/MoonshotAI/Kimi-K2.5>, 2026.
- Yutao Mou, Zhangchi Xue, Lijun Li, Peiyang Liu, Shikun Zhang, Wei Ye, and Jing Shao. ToolSafe: Enhancing tool invocation safety of LLM-based agents via proactive step-level guardrail and feedback. *arXiv preprint arXiv:2601.10156*, 2026.
- Qwen Team. Qwen3 technical report. <https://qwenlm.github.io/blog/qwen3/>, 2025.
- Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents. <https://qwen.ai/blog?id=qwen3.5>, 2026.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- Sina Tayebati, Divake Kumar, Nastaran Darabi, Davide Etti, Ranganath Krishnan, and Amit Ranjan Trivedi. TRACER: Trajectory risk aggregation for critical episodes in agentic reasoning. *arXiv preprint arXiv:2602.11409*, 2026.
- Hanlin Wang, Jian Wang, Chak Tou Leong, and Wenjie Li. STeCa: Step-level trajectory calibration for LLM agent learning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 11597–11614, 2025.
- Xingyao Wang, Valerie Chen, Heng Ji, and Graham Neubig. A rubric-supervised critic from sparse real-world outcomes. *arXiv preprint arXiv:2603.03800*, 2026.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.
- Jingxuan Xu, Ken Deng, Weihao Li, Songwei Yu, Huaixi Tang, Haoyang Huang, Zhiyi Lai, Zizheng Zhan, Yanan Wu, and Chenchen Zhang. SWE-Compass: Towards unified evaluation of agentic coding abilities for large language models. *arXiv preprint arXiv:2511.05459*, 2025.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37: 50528–50652, 2024.
- Guangtao Zeng, Maohao Shen, Delin Chen, Zhenting Qi, Subhro Das, Dan Gutfreund, David Cox, Gregory Wornell, Wei Lu, and Zhang-Wei Hong. Satori-SWE: Evolutionary test-time scaling for sample-efficient software engineering. *arXiv preprint arXiv:2505.23604*, 2025.
- Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. AgenTracer: Who is inducing failure in the LLM agentic systems? *arXiv preprint arXiv:2509.03312*, 2025a.
- Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, and Yiran Chen. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Forty-second International Conference on Machine Learning*, 2025b.
- Xiuhui Zhang, Ning Gao, Xingyu Jiang, Yihui Chen, Yuheng Pan, Mohan Zhang, and Yue Deng. Progress reward model for reinforcement learning via large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025c.
- Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.
- Zhipu AI. GLM-5: From vibe coding to agentic engineering. <https://github.com/zai-org/GLM-5>, 2026.

A Limitations

While FLARE establishes a highly effective dense supervision paradigm for long-horizon SWE tasks, we acknowledge several limitations that provide directions for future research:

Bounded by Base Agent Semantic Capabilities. FLARE functions as an alignment and intervention scaffold rather than a substitute for fundamental reasoning. As observed during test-time intervention, while the Generative Reward Model (GRM) excels at localizing process-level missteps and validation failures, its effectiveness diminishes when failures require deep semantic architectural shifts or when the agent experiences severe goal drift. If the base model lacks the inherent coding capability to implement a complex repair, simply pinpointing the error step is insufficient to rescue the trajectory.

Overhead of Step-Level Generative Diagnosis. Although GRM-DBKR substantially reduces base-agent output under the reported repair protocol, diagnostic inference incurs additional computation and potential latency. The reported token counts measure agent outputs only; end-to-end resource use also depends on agent inputs, GRM inference, environment execution, and the serving configuration. The one-time costs of offline annotation and GRM training are also outside this accounting.

Dependency on a Predefined Error Taxonomy. RADAR uses a fixed taxonomy to make diagnostic supervision interpretable and reusable, but this design may miss unfamiliar errors outside the annotated categories. We audited diagnostic severity, evidence grounding, and leakage through independent review by human SWE experts and a stronger language model, with disagreements adjudicated against the visible prefix (Appendix C). This audit measures residual annotation errors in the sampled diagnostics rather than guaranteeing error-free supervision.

B Error Taxonomy

This appendix provides the fixed error taxonomy used throughout the RADAR–GRM pipeline, including offline RADAR annotation, causal-prefix distillation, runtime GRM diagnosis, GRM-guided checkpoint repair, and taxonomy-level experimental analysis. The taxonomy is designed to characterize actionable process-level failures in SWE-agent trajectories, rather than merely describe the final Pass/Fail outcome. All non-safe RADAR annotations and GRM risk diagnoses are constrained to use only rubric codes from this taxonomy.

Table 5 Error taxonomy used for RADAR annotation and GRM step-level diagnosis.

Code	Error Type	Definition / Evidence Cue
C1 Task Understanding & Planning		
E1.1	Misunderstood Requirements	The agent misinterprets the task goal or ignores explicit constraints; modifies the wrong target or bypasses the required interface.
E1.2	Goal Drift	The agent drifts from the original goal to irrelevant objectives such as unrelated refactoring, formatting, or premature completion.
E1.3	Incorrect Step Decomposition	The agent omits necessary inspection, setup, dependency, or validation steps, causing failures from skipped prerequisites.
E1.4	Wrong Tool Selection	The agent uses an inappropriate or fragile tool when a direct, reliable, or project-native method is available.
E1.5	Ordering / Dependency Error	The agent executes steps in an invalid order, such as using files before generation, running before setup, or committing before validation.
E1.6	Incomplete Implementation / Residual TODOs	The agent claims completion while leaving placeholders, TODOs, missing logic, or partial call-chain updates.

Code	Error Type	Definition / Evidence Cue
C2 Context & Evidence Handling		
E2.1	Irrelevant or Missed Retrieval	The agent searches the wrong scope or misses relevant files, symbols, implementation locations, or evidence needed for the task.
E2.2	Wrong Evidence Selected	Relevant evidence is available, but the agent selects the wrong file, snippet, candidate, or code path.
E2.3	Misinterpreted Tool Output	The agent misreads compiler, test, linter, shell, or diagnostic output and takes an ineffective action.
E2.4	Lost Context / Memory Mismatch	The agent forgets or mismatches earlier requirements, constraints, or established conclusions in a long trajectory.
C3 Code Generation & Modification		
E3.1	Syntax / Type Error	The generated or modified code violates syntax, compilation, interpretation, or static type-checking constraints.
E3.2	API / Library Misuse or Version Mismatch	The agent misuses an API, library, framework convention, or version-specific behavior.
E3.3	Semantic / Logic Error	The code runs or appears plausible but violates the required behavior due to incorrect algorithmic, boundary, business-logic, or state reasoning.
C4 Validation & Self-Check		
E4.1	Missing Validation / Insufficient Coverage	The agent skips necessary tests or validates too narrowly after non-trivial changes.
E4.2	Shallow Check	The agent only checks that execution succeeds, without verifying outputs, side effects, or edge cases.
E4.3	Ignored Failure Signal / Misread Test Report	The agent ignores or dismisses clear failure signals from tools, tests, logs, or CI.

C RADAR Annotation and Quality-Control Prompts

This appendix provides the prompts used by RADAR to construct causally grounded step-level supervision before GRM training. These offline prompts are distinct from the runtime GRM prompt in Appendix D. The pipeline contains differential causal backtracking, causal-prefix distillation, and semantic consistency checking. Full-trajectory evidence supports offline attribution, while online explanations are constrained to the task and the prefix through the current step. The checks enforce this information boundary at the prompt level rather than establishing a formal guarantee of zero leakage.

We complemented the automated gate with an independent audit of 608 diagnostics by 3 human SWE experts and a stronger language model, GPT-5 (gpt-5-2025-08-07). Diagnostics were sampled according to stratified random sampling by repository and predicted severity, with one expert assigned to each diagnostic. Human and model reviewers independently assessed each diagnostic using only the task and current prefix, marking severity errors, unsupported repair advice, patch leakage, and future- or test-oracle information. Disagreements were adjudicated against the visible evidence. Table 6 reports adjudicated error counts and rates separately from downstream repair performance. Pre-adjudication agreement on severity labels was 92.60%, with unweighted Cohen’s $\kappa = 0.887$ between the assigned expert and the model. The audit provides an empirical check of the information boundary in addition to automated prompt compliance.

Table 6 Independent diagnostic audit. Entries report error count / eligible diagnostics (error rate in percent) after adjudication. False-critical errors use diagnostics labeled critical as the denominator; the other categories use all audited diagnostics. Categories may overlap.

Audit criterion	Adjudicated errors
Incorrect severity assignment	37 / 608 (6.09%)
False critical prediction	14 / 163 (8.59%)
Unsupported repair advice	23 / 608 (3.78%)
Patch leakage	3 / 608 (0.49%)
Future- or test-oracle information	7 / 608 (1.15%)

C.1 Offline Differential Causal Backtracking and Joint Step-level Attribution Prompt

Error attribution prompt

```
# RADAR Offline Differential Causal Backtracking and Joint Step-level Attribution Prompt

You are the offline diagnostic module of RADAR. Your task is to perform post-hoc causal analysis over SWE-agent trajectories and
construct step-level supervision data for GRM training.

Read all inputs from the predefined directory:

- problem_statement.txt: original task requirement.
- traj_summary.txt: final result claimed by the agent.
- failing_tests.txt: final failure signals.
- golden_patch.patch: reference patch, used only for offline diagnosis.
- model_patch.patch: agent-produced patch.
- abstract_llm.json: compressed per-turn trajectory with intent, action, and observation.
- trajectory.json: raw trajectory, used only when abstract_llm.json is insufficient.
- rubrics.jsonl: fixed error taxonomy. Do not invent or modify rubric categories.

=====
Stage A: Differential Causal Backtracking
=====

Construct a causal error chain explaining how final failure, patch discrepancy, or requirement violation arose from earlier
trajectory decisions.

The chain must be:

- plausible: every node helps explain the final failure or deviation;
- complete: major causally relevant decisions are captured;
- precise: unrelated, merely redundant, stylistic, or inefficient turns are excluded.

Procedure:

1. Difference anchoring.
   Compare golden_patch.patch and model_patch.patch semantically. Identify affected files, symbols, interfaces, behaviors,
   contracts, and how each mismatch relates to failing_tests.txt or the task requirement. Patch and failing-test information is
   allowed only in offline_analysis and must not appear in later online_analysis.

2. Backward traversal.
   Traverse abstract_llm.json backward. Mark a turn as a causal-chain node only if concrete evidence shows that it:
   - directly created, modified, validated, or preserved a behavior in the final deviation; or
   - introduced or committed to an assumption, contract interpretation, scope decision, or evidence selection required for the
     final deviation.
   Discard incidental, exploratory, redundant, or unrelated turns.

3. Root-cause priority.
   Identify the earliest observable turn where the erroneous assumption, wrong contract, wrong evidence selection, or invalid
   implementation direction first appears. This is the preferred root-cause breakpoint. Later propagation or amplification
   turns may be non-safe, but they are not preferred over the root cause for breakpoint replay.

4. Independent roots.
   If multiple failures arise from independent assumptions or modifications, output separate root nodes.

Each causal-chain node must contain:

{
  "turn_id": <int>,
  "role": "root" | "propagation" | "amplification",
  "divergence_anchor": "<file/symbol/interface/contract/behavior>",
  "dependency": <turn_id | null>
```

```
}

```

where root is the earliest source, propagation carries forward a prior error, amplification expands or solidifies a prior error, and dependency points to the earlier causal node or null for an independent root.

Stage B: Joint Step-level Attribution

Annotate every turn in abstract_llm.json with exactly one record containing:

1. risk_level
2. rubric_hits
3. offline_analysis

risk_level must be one of:

- safe: the turn is correct, neutral, or constructively aligned with the task.
- trivial: the turn has a real defect, but it is unlikely to independently cause final failure or major derailment.
- critical: the turn can independently cause task failure, substantially derail the trajectory, validate a wrong assumption, or make recovery significantly harder.

Severity must be determined by the current turn's causal role and evidence, not by a fixed rubric whitelist. Any rubric category may be critical if the causal role is critical. Do not force any target label distribution.

For every non-safe turn, assign one or more rubric hits from rubrics.jsonl:

```
{
  "rule_id": "<rubric code>",
  "label": "<canonical label>",
  "confidence": "high" | "low",
  "reason": "<specific evidence from the current turn and relevant context>"
}
```

Rules:

- safe turns must have rubric_hits = [].
- non-safe turns must have at least one rubric hit.
- every rubric code and label must exist in rubrics.jsonl.
- do not assign rubrics merely because the final outcome failed.

offline_analysis must be one concise English paragraph per turn.

For safe turns, explain what the turn did and why it is aligned or neutral.

For non-safe turns, explain:

1. what the turn did;
2. what went wrong, with evidence;
3. how it relates to final failure, patch discrepancy, or requirement violation;
4. why the severity is appropriate;
5. the minimal corrective direction.

offline_analysis may use golden_patch.patch, model_patch.patch, failing_tests.txt, and future turns because it is post-hoc. Such hindsight information must later be removed by causal-prefix distillation before GRM training or online repair.

Consistency Requirements

- Every turn in abstract_llm.json must have exactly one annotation.
- Every causal-chain node must have risk_level in {"trivial", "critical"}.
- A turn outside the causal chain may be non-safe only for a local defect unrelated to the final failure; offline_analysis must state this.
- If a later critical turn is merely downstream of an earlier safe/trivial turn, revisit and update the earlier turn before finalizing.
- Do not label a turn non-safe solely for verbosity, redundancy, or inefficiency unless it creates concrete task risk.
- Do not force any safe/trivial/critical ratio.

Output

Return exactly one parseable JSON object and nothing else:

```
{
  "causal_chain": [
    {
      "turn_id": <int>,
      "role": "root" | "propagation" | "amplification",
```

```

    "divergence_anchor": "<string>",
    "dependency": <int | null>
  }
],
"annotations": [
  {
    "turn_id": <int>,
    "risk_level": "safe" | "trivial" | "critical",
    "rubric_hits": [
      {
        "rule_id": "<string>",
        "label": "<string>",
        "confidence": "high" | "low",
        "reason": "<string>"
      }
    ],
    "offline_analysis": "<English paragraph>"
  }
]
}

```

Do not output Markdown, comments, explanations, or code fences.

C.2 Causal-prefix Distillation Prompt

Error attribution prompt

RADAR Causal-prefix Distillation Prompt

You are the causal-prefix distillation module of RADAR. Your task is to rewrite post-hoc offline_analysis into online_analysis using only information visible up to and including the current turn. The resulting online_analysis is used for GRM training and may be used as checkpoint-repair feedback.

This prompt constructs training annotations. It is distinct from the runtime GRM prompt, which may use compact parseable outputs such as <safe/>.

Read all inputs from the predefined directory:

- annotated_trajectory.json: output of offline RADAR attribution, including turn_id, risk_level, rubric_hits, and offline_analysis.
- problem_statement.txt: original task requirement, always visible to the downstream agent.
- abstract_llm.json: compressed trajectory. For each turn, only the prefix up to and including that turn is visible.
- trajectory.json: raw trajectory, used only to verify prefix-visible anchors.
- rubrics.jsonl: fixed taxonomy, used only to preserve rubric meaning. Do not invent categories.

=====

Hard Causal Masking Constraints

=====

Every online_analysis must satisfy all constraints below:

1. No patch leakage.
Do not mention or reveal golden_patch.patch, model_patch.patch, diff hunks, reference/expected/correct patch content, or patch-only paths, symbols, constants, or implementation details not visible in the prefix.
2. No final-evaluation leakage.
Do not mention final failing_tests.txt, final evaluation results, final patch outcome, future turns, or facts visible only after the current turn. Mentions of test failures are allowed only if the failure message is directly visible in the current or earlier trajectory prefix.
3. No test-oracle leakage.
Do not mention specific test function names, hidden expected values, assertions, fixtures, or symbols that appear only inside test files. Test directories or test file paths may be mentioned only if already prefix-visible and used solely as validation targets.
4. Prefix-only evidence.
Use only problem_statement.txt, the current turn, and earlier turns from abstract_llm.json or trajectory.json.
5. English only.
6. No unsupported anchors.
Do not invent symbols, paths, line numbers, APIs, constants, commands, outputs, or error messages. Every concrete anchor must be prefix-visible or stated in problem_statement.txt.
7. No unsupported runtime/library claims.
Do not assert runtime, standard-library, framework, or dependency semantics as decisive unless directly visible in the prefix. Otherwise, phrase the issue using observed code, command output, or task requirements.

8. No speculative wording for confirmed non-safe issues.
 For trivial or critical turns, avoid hedges such as may, might, possibly, could potentially, perhaps, seems to, appears to. If evidence is insufficient, weaken the severity instead.

=====
 Content Requirements
 =====

Append exactly one field to each annotation:

"online_analysis": "<English paragraph or compact structured English text>"

Preserve the input causal_chain unchanged in the output.

For safe turns:

- write 2--3 concise English sentences;
- describe what the turn did;
- explain why it is aligned, neutral, or useful;
- end with a natural sentence stating the risk level.

For trivial or critical turns, include all five components:

1. Current action:
 State what the turn did and cite at least one current-turn prefix-visible anchor, such as a path, symbol, branch condition, command, error message, or tool output.
2. What remains on track:
 Acknowledge useful or correct partial progress so replay does not discard valid context.
3. Decisive issue:
 State 1--3 concrete issues. Each must cite a prefix-visible anchor, explain conflict with the problem statement or visible code /output, avoid patch/future/test-oracle information, and use direct non-speculative wording.
4. Remaining:
 Include an explicit field:

Remaining:
 - <requirement>

or:

Remaining: none

Items must come from problem_statement.txt and prefix-visible implementation status, not hidden tests or patches.

5. Suggested next step:
 Give the minimal concrete next action: which existing file/function/symbol/module to inspect or modify, what direction to take, and how to keep the change within scope. If a relevant test path is prefix-visible, mention only that path as a validation target. End with a natural sentence stating the risk level.

=====
 Output
 =====

Return exactly one parseable JSON object and nothing else:

```
{
  "causal_chain": [...],
  "annotations": [
    {
      "turn_id": <int>,
      "risk_level": "safe" | "trivial" | "critical",
      "rubric_hits": [...],
      "offline_analysis": "<original offline analysis>",
      "online_analysis": "<causally masked English analysis>"
    }
  ]
}
```

Do not output Markdown, comments, explanations, or code fences.

C.3 Semantic Consistency and Causal Compliance Gate Prompt

Error attribution prompt

RADAR Semantic Consistency and Causal Compliance Gate Prompt

You are the quality-control gate for RADAR annotations. Audit a fully annotated trajectory before it is used for GRM training, checkpoint repair, trajectory scoring, or reward construction. Check logical consistency, causal consistency, taxonomy consistency, prefix-visibility compliance, and output completeness. Do not create new annotations unless explicitly requested.

Read all inputs from the predefined directory:

- annotated_trajectory.json: contains causal_chain and annotations with turn_id, risk_level, rubric_hits, offline_analysis, and online_analysis.
- problem_statement.txt: original task requirement.
- abstract_llm.json: compressed trajectory.
- trajectory.json: raw trajectory, used only to verify prefix-visible evidence.
- rubrics.jsonl: fixed taxonomy.

=====

D1: Logical and Taxonomy Consistency

=====

Check that:

- safe turns have rubric_hits = [].
- trivial/critical turns have non-empty rubric_hits.
- every rubric_hit.rule_id exists in rubrics.jsonl.
- every rubric_hit.label matches rubrics.jsonl.
- every rubric_hit.reason cites concrete evidence from the corresponding turn or necessary prefix context.
- risk_level is justified by causal role, not by a rubric whitelist.
- any rubric category may be critical if the step can cause failure, derail the trajectory, validate a wrong assumption, or make recovery harder.
- trivial does not describe a clearly decisive failure unless offline_analysis explains why it is recoverable or non-decisive.
- critical does not rely only on final failure; it must identify a current-turn defect or decision.

=====

D2: Causal-chain Consistency

=====

Check that:

- every causal_chain node maps to exactly one annotation.
- every causal_chain node has risk_level in {"trivial", "critical"}.
- every divergence_anchor is concrete and evidence-supported.
- root nodes have dependency = null.
- propagation/amplification nodes have valid earlier dependencies unless justified as independent roots.
- no later critical turn is merely the downstream consequence of an earlier safe turn.
- if a later critical turn depends on an earlier trivial turn, offline_analysis must justify why the earlier turn is recoverable or less severe.
- non-safe turns outside the causal chain must be explicitly described as local non-causal defects in offline_analysis.

=====

D3: Online-analysis Causal Masking Compliance

=====

For every online_analysis, check:

- no patch leakage: no golden/reference/expected/correct patch, diff hunk, or prefix-invisible patch-only details.
- no future-outcome leakage: no future turns, final evaluation, final failing_tests.txt, final patch outcome, or facts visible only after the current turn.
- test-failure mentions are allowed only if directly visible in the current or earlier prefix.
- no test-oracle leakage: no specific test function names, hidden expected values, assertions, fixtures, or test-only symbols.
- test directories/files are allowed only if prefix-visible and used only as validation targets.
- English only.
- every non-safe online_analysis cites at least one prefix-visible or problem-statement anchor.
- no invented paths, line numbers, symbols, APIs, constants, commands, outputs, or error messages.
- trivial/critical issues avoid speculative wording such as may, might, possibly, could potentially, perhaps, seems to, appears to.
- every trivial/critical online_analysis contains "Remaining:" with items or "Remaining: none".
- every safe online_analysis ends with a natural sentence stating the risk level.

=====

D4: Coverage and Format Completeness

=====

Check that:

```

- every turn in abstract_llm.json has exactly one annotation.
- annotations contain no duplicate turn_id.
- every annotation includes turn_id, risk_level, rubric_hits, offline_analysis, and online_analysis.
- causal_chain and annotations fields both exist.
- the artifact is valid parseable JSON.

=====
Violation Severity and Routing
=====

Use "block" if the artifact must be regenerated before downstream use, including patch leakage, future leakage, invalid JSON,
missing/duplicate annotation, invalid rubric code, safe with rubric_hits, non-safe without rubric_hits, causal-chain node
labeled safe, unsupported critical label, or non-English online_analysis.

Use "warn" for non-blocking issues such as mild vagueness, redundancy, overly verbose safe analysis without leakage, or weak but
parseable validation reminders.

Routing:

- causal_chain, risk_level, rubric_hits, or offline_analysis issues route to "RADAR_attribution".
- online_analysis or causal masking issues route to "causal_prefix_distillation".
- include both routes if both issue types occur.

=====
Output
=====

Return exactly one parseable JSON object and nothing else:

{
  "passed": true | false,
  "violations": [
    {
      "turn_id": <int | null>,
      "dimension": "D1" | "D2" | "D3" | "D4",
      "rule": "<short rule identifier>",
      "evidence": "<quoted or summarized evidence>",
      "severity": "block" | "warn",
      "route_back_to": "RADAR_attribution" | "causal_prefix_distillation"
    }
  ],
  "route_back_to": [
    "RADAR_attribution",
    "causal_prefix_distillation"
  ]
}

If there are no violations, return:

{
  "passed": true,
  "violations": [],
  "route_back_to": []
}

Do not output Markdown, comments, explanations, or code fences.

```

D Runtime GRM Prompt

This appendix gives the exact runtime prompting format used by the GRM to evaluate a single step during SWE-agent execution. Unlike the RADAR prompts in Appendix C, which construct and validate training annotations under offline or distilled supervision settings, this prompt is used at inference time. The GRM receives the problem statement, prefix trajectory, current step, rubric taxonomy, and tool descriptions, and returns either a safe judgment or a rubric-backed risk judgment in a parseable XML format. This output format enables the same diagnostic signal to be reused for checkpoint repair, trajectory scoring, and dense reward construction.

D.1 System Prompt

Error attribution prompt

You are a Generative Reward Model (GRM) that evaluates exactly one current step in a SWE agent trajectory.

Input Structure

The user message will always use these exact XML tags:

- `<problem_statement>`: the task description the agent is solving.
- `<history_trajectory>`: all prior steps before the current one, in raw JSON format. Each step is an assistant message (which may contain reasoning and tool calls) followed by the corresponding tool result messages.
- `<current_step_to_evaluate>`: the single current step to judge, in the same raw JSON format.
- `<rubrics>`: the error taxonomy. Each entry contains a code, definition, criteria, and examples.
- `<tools>`: descriptions of all tools available to the agent in its framework.

Your Task

Judge whether `<current_step_to_evaluate>` contains one or more rubric-backed errors.

- Use `<problem_statement>`, `<history_trajectory>`, and `<tools>` only as context.
- Evaluate only the current step itself.
- Only mention a prior-step mistake if the current step explicitly repeats it or builds on it.
- Use only rubric codes that appear in `<rubrics>`.

Hard Output Contract

Return exactly one top-level XML result and nothing else.

Option A: safe

`<safe/>`

Option B: risk

```
<risk>
  <analysis>
    {what went wrong, why it matters, and what concrete evidence in the current step supports the judgment}
  </analysis>
  <issue rubric="{code}" label="{level_1_category}/{level_2_label}"/>
  <severity>{critical|trivial}</severity>
</risk>
```

If there are multiple rubric-backed errors, output multiple `<issue .../>` lines inside the same `<risk>` block.

Required Rules

- The first character of your response must be `<`.
- The last character of your response must be `>` from the final XML tag.
- Do not output any natural language before the XML block.
- Do not output any natural language after the XML block.
- Do not output Markdown, code fences, JSON, comments, XML declarations, or any `<think>` / `</think>` tags.
- Never output both `<safe/>` and `<risk>` in the same response.
- If the step is safe, output exactly `<safe/>` and stop.
- If the step is risky, output exactly one top-level `<risk>` block.
- A `<risk>` response must contain:
 - exactly one `<analysis> ... </analysis>` block,
 - at least one `<issue .../>` line,
 - exactly one `<severity>` whose value is either `'critical'` or `'trivial'`.
- Do not emit any other XML tags.
- Escape reserved XML characters in text and attribute values (for example, `&` as `&` and `<` as `<`).

Decision Rules

- `'critical'`: the issue is likely to cause task failure, a wrong final result, or a major derailment.
- `'trivial'`: the issue exists, but is unlikely to affect final correctness or overall progress.
- If any listed issue is critical, the final severity must be `'critical'`.
- If you cannot support an error with concrete evidence from the current step plus context, prefer `<safe/>`.

Correct / Incorrect Output Examples

Correct:

```
<safe/>
```

Incorrect:

```
<risk>
  <analysis>
    ...
```

```

</analysis>
<issue rubric="E1.2" label="Task Understanding & Planning Errors/Goal Drift"/>
<severity>trivial</severity>
</risk>

```

D.2 User Prompt Template

Error attribution prompt

```

<problem_statement>
{problem_statement}
</problem_statement>

<history_trajectory>
{history_trajectory}
</history_trajectory>

<current_step_to_evaluate>
{current_step_to_evaluate}
</current_step_to_evaluate>

<rubrics>
{rubrics}
</rubrics>

<tools>
{tools}
</tools>

```

E Case Study: GRM-Guided Checkpoint Rollout

We provide a qualitative case study to illustrate how FLARE uses GRM feedback to recover a long-horizon SWE trajectory through localized checkpoint repair. This case is intended as an interpretability example rather than additional quantitative evidence. Unlike aggregate F2P metrics, the goal here is to expose the concrete mechanism by which a GRM intervention changes the subsequent implementation path.

Task. The task is from alibaba/fescar#644. The original issue reports a serialization failure in the distributed transaction rollback path: `java.io.NotSerializableException:com.alibaba.fescar.rm.datasource.sql.struct.Null`. The failing object is `com.alibaba.fescar.rm.datasource.sql.struct.Null`, a sentinel object used to represent SQL NULL values in prepared-statement parameters. Therefore, the task is not merely to add a marker interface somewhere in the codebase. A correct repair must make `Null` serializable while preserving its intended sentinel-object semantics.

Why this case is informative. This trajectory contains a particularly subtle failure mode. Before the GRM interruption, the agent had already found the right file and had made a superficially plausible edit: it modified `Null.java` so that `Null` implements `java.io.Serializable`. However, this only addresses the surface exception. The class is also exposed through `Null.get()`, indicating that it is used as a singleton-style sentinel object. Under that design, serialization must not create semantically distinct `Null` objects after deserialization. The GRM intervention is useful because it prevents the agent from treating “implements `Serializable`” as a complete solution.

E.1 Breakpoint Context

At the interruption point, the agent had narrowed the problem to the correct class and stated the following implementation intent:

```

The 'Null' class is used to represent SQL NULL values in prepared statements. When Fescar performs distributed
transaction rollbacks, it needs to serialize these objects, but 'Null' doesn't implement 'Serializable'.

The fix is to make the 'Null' class implement 'java.io.Serializable':

```

The immediately following tool output shows that the actual edit only added the serialization marker interface and a `serialVersionUID` field:

```
package com.alibaba.fescar.rm.datasource.sql.struct;

import java.io.Serializable;

/**
 * The type Null.
 */
public class Null implements Serializable {

    private static final long serialVersionUID = 748997787189430077L;
```

This is the critical breakpoint. The agent is on the right file and has addressed the exception type, but the current patch has not yet handled the semantic interaction between Java serialization and the singleton sentinel pattern.

E.2 GRM Diagnosis

Given the current step and its prefix context, the GRM produced the following critical-risk diagnosis:

```
<risk>
  <analysis>
    The agent edits Null.java to add import java.io.Serializable, implements Serializable, and
    serialVersionUID field. The current action modifies the Null.java file. What is on track: The agent
    correctly identifies that Null class needs to be serializable for distributed transaction rollback
    scenarios. Decisive issue: The agent implements Serializable but does not consider the class's singleton
    pattern (instance field) and toString() method. The existing code shows Null.get() returns a singleton
    instance; making it serializable may not align with the singleton design. The agent also does not examine
    how Null is used in prepared statements (AbstractPreparedStatementProxy.java) to ensure serialization works
    correctly. The implementation is incomplete for the task requirements. Remaining: 1) Ensure Null class is
    serializable for distributed transaction rollback scenarios, 2) Verify Null.get() singleton pattern works
    with serialization, 3) Check prepared statement usage of Null in AbstractPreparedStatementProxy.java for
    serialization compatibility. Suggested next step: Examine AbstractPreparedStatementProxy.java to see how
    Null.get() is used in setParamByIndex, then adjust Null.java to maintain singleton pattern while making it
    serializable. Run tests in rm-datasource module before declaring done. The turn is critical.
  </analysis>
  <issue rubric="E3.3" label="Code Generation & Modification Errors/Semantic / Logic Error (algorithm /
  boundary condition / state)"/>
  <issue rubric="E1.6" label="Task Understanding & Planning Errors/Incomplete Implementation / Residual
  TODOs"/>
  <severity>critical</severity>
</risk>
```

This diagnosis is important for two reasons. First, it does not discard the useful part of the agent's work: the GRM explicitly recognizes that making Null serializable is directionally correct for the rollback failure. Second, it identifies the missing semantic condition: the fix must preserve the Null.get() singleton pattern across serialization boundaries. The assigned taxonomy labels also match the failure mode: the current edit is a semantic logic error (E3.3) and an incomplete implementation (E1.6), rather than a syntax or tool-use failure.

E.3 Checkpoint Replay and Behavioral Change

After checkpoint replay, the agent's next steps directly followed the GRM's repair direction. Instead of immediately declaring the task complete, the agent first re-examined the current implementation and explicitly reframed the problem around the singleton pattern:

```
Let me examine the current state of the file and understand the singleton pattern issue:
```

The replayed branch then articulated the missing failure mechanism:

```
Now I understand the usage. The 'Null.get()' singleton is used to represent SQL NULL values in prepared statement
parameters. The issue is that when serializable objects containing the 'Null' instance are serialized and
then deserialized, Java's default serialization will create a new object, breaking the singleton pattern.
```

To fix this properly, I need to add a ‘readResolve()’ method to ensure the singleton pattern is preserved during deserialization:

The resulting code-level repair adds a readResolve() method to Null.java:

```
private Object readResolve() {
    return instance;
}
```

This small change is the key evidence that the replayed branch internalized the GRM feedback. The original interrupted branch treated serializability as a marker-interface requirement. The repaired branch instead implements singleton-safe serialization: Null can now participate in serialized rollback state, while deserialization resolves back to the canonical singleton instance used by Null.get().

E.4 Effect of the Intervention

Table 7 summarizes the causal role of this single GRM interruption.

Table 7 GRM-guided checkpoint repair in the alibaba/fescar#644 case. The GRM prevents a shallow serializability patch from being treated as complete and redirects the replay toward singleton-safe serialization.

Stage	Observed behavior	Interpretation
Before interruption	The agent edits Null.java to add import java.io.Serializable, implements Serializable, and serialVersionUID.	The agent has found the right class and addressed the surface exception, but the implementation is incomplete because it does not preserve the singleton sentinel semantics of Null.get().
GRM diagnosis	The GRM marks the step as critical, citing E3.3 and E1.6. It states that the implementation must verify the singleton pattern and inspect prepared-statement usage through AbstractPreparedStatementProxy.java.	The feedback is localized and actionable: it keeps the correct target file, preserves the useful partial edit, and identifies the missing semantic contract rather than requesting a full restart.
After checkpoint replay	The agent re-examines the singleton pattern, reasons that default deserialization would create a new object, and adds readResolve() returning instance.	The repaired branch changes the completion criterion from “Null implements Serializable” to “Null remains the canonical SQL NULL sentinel across serialization and deserialization.”

Discussion. This case illustrates the advantage of GRM-guided repair over Blind-BKR. The diagnosed step is not obviously catastrophic: it has already modified the correct file and implemented the interface named by the exception; the intervention redirects the subsequent continuation. However, it is still a high-risk process step because it creates a plausible but shallow patch that can prematurely terminate the trajectory. The GRM feedback exposes the hidden semantic requirement, namely preserving the singleton-style Null sentinel across serialization. Checkpoint replay then uses this diagnosis to produce a more precise repair in Null.java. In this way, FLARE does not merely resample a new continuation; it converts a step-level diagnosis into a concrete implementation constraint that changes the subsequent behavior of the agent.