Steer, Don't Solve:
Training Small Critic Models for Large Code Agents

Anonymous ACL submission

Abstract

End-to-end code agent training is resource-intensive and plateaus on the strategy-level reasoning needed to resolve code issues, since jointly optimizing code-level execution and strategy-level reasoning leaves the latter underdeveloped. Instead, we freeze the agent and add a *critic model* to supply that signal. Prior code critics are *post-hoc*, scoring completed trajectories rather than steering the agent; we instead train a small critic that provides *intra-trajectory* feedback via Supervised Fine-Tuning. On SWE-bench Verified, a single critic trained on `CWM-32B` trajectories transfers to two unseen agents and improves resolve rate on all three by $+3.0$ to $+3.8$ points, at 40-92$\times$ lower critic cost than a strong teacher. On `Qwen3-Next-80B-A3B`, the critic-guided system is both more accurate (23.8% vs. 20.8%) and cheaper ($0.04 vs. $0.11) than the agent alone, because the critic also shortens trajectories. Our results show that a small, well-trained critic is a practical complement to scaling agent training.[1]

1 Introduction

Code agents have been widely used in real-world programming tasks (Jimenez et al., 2024; Yang et al., 2024), and post-training has proven effective for code agent training (Pan et al., 2025; Yang et al., 2026; Xie et al., 2026), where the model is optimized for the combination of multiple abilities, including high-level reasoning (e.g., troubleshooting, search, planning) and low-level implementation (executing tool calls correctly).

However, empirically, this training is resource-intensive and the performance plateaus quickly for base models with limited size. For instance, the issue-solving performance of a 14B model stops growing after training on only roughly 800 trajectories (Jain et al., 2025). Furthermore, during post-training, multiple studies have shown that the high-level reasoning ability of a code agent is still far from perfect after reaching the plateau on the training data (Xie et al., 2026; Wang et al., 2026a). For example, about 65% of code agent failures come from flawed reasoning, not wrong actions (Liu et al., 2025). A possible explanation is that integrating both kinds of knowledge makes learned patterns overly complex, which undercuts the ability to learn generalizable patterns, leading to a performance plateau. Separation of concerns enables learning simpler patterns that generalize better, in which case the interaction of more specialized models, one for code-level and one for strategy-level capability, leads to a performance increase. In this paper, we take code agents as the setting for investigating whether explicitly separating these two kinds of capability, without retraining the agent, can enable more efficient progress.

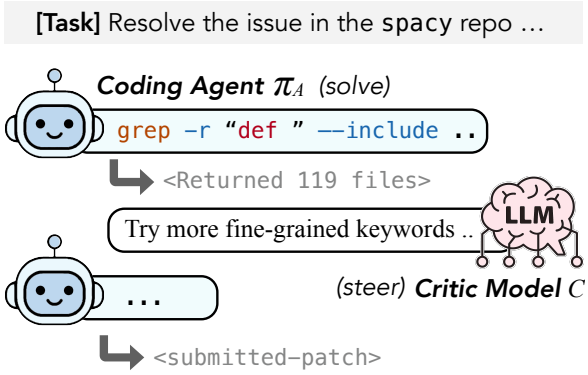

Figure 1: Inference setup. A trained critic C provides guidance to a frozen code agent π_A at every k agent steps. The agent is responsible for all concrete actions and the critic model inspects the partial trajectory and generates high-level feedback before its next step.

Inspired by research on other NLP domains such as question-answering (Gou et al., 2023), factuality (Xie et al., 2025), and mathematical problems (Lightman et al., 2024), we aim to improve

[1]We will release the code and training data.

the high-level reasoning abilities of code agents by training a **critic model**, which generates textual feedback during a code agent's run to guide its next actions. We highlight three advantages of training a critic model compared to training the code agent itself. First, separation of concerns simplifies the task for each of two specialized models. The agent continues to handle step implementation, and a separate critic is specialized for high-level reasoning. Second, critic model training is cost-efficient. As shown in §3, we can re-use existing code-agent environments for critic model training, without building new training tasks or environments. Results in §5.1 show that even a small finetuned critic model can provide useful guidance to a large code agent. Finally, the use of critic models is highly adaptive and flexible. We can apply critic models to proprietary API models without accessing their parameters. It can also be applied at any agent step with any frequency.

We design the setting as follows (Fig. 1): every k steps the critic inspects the agent's trajectory and writes a short critique that the agent reads before its next action; the agent stays frozen, and the critic is distilled from a frontier teacher (`Claude-Opus-4.6` (Anthropic, 2026)) as in Fig. 2. Two design choices about *what* the critic should say drive most of the gain (§3.3): we constrain the critic to strategy-level feedback rather than specific commands, and we switch it toward recommending submission once the agent nears its budget. On SWE-bench Verified, our 8B trained critic, trained on a single agent's trajectories (`CWM-32B` (team et al., 2025)), improves resolve rate by $+3.0$ to $+3.8$ points over the unguided agent on every agent model we test (`CWM-32B`, `Qwen3-Next-80B-A3B` (Yang et al., 2025), `Qwen3-32B` (Yang et al., 2025)), including the two it was never trained against, and sits on the cost vs. performance Pareto frontier for each (Fig. 3). On `Qwen3-Next-80B-A3B` the critic Pareto-dominates the unguided agent, raising resolve rate from 20.8% to 23.8% while cutting total cost from \$0.11 to \$0.04 per instance, since the same critic shortens trajectories enough to offset its own cost.

Contributions. (1) to our knowledge, the first study of training critic models for code agents, as a separation-of-concerns alternative to scaling the agent; (2) an analysis of which design choices about feedback form (concise vs. detailed prompts) transfer from teacher to student; and (3) empirical validation on SWE-bench Verified showing that a critic trained on a single agent's trajectories transfers to two unseen agents, with consistent gains of 3 to 3.8 points across all three.

2 Related Work and Design Rationale

Critic and advisor models for generation.. A growing body of work studies *critic* or *advisor* models that improve generation through evaluative feedback. Gou et al. (2023) established critique as an intermediate signal for improving model outputs. In code and math, McAleese et al. (2024); Gao et al. (2025) train critics that give natural-language feedback on model-written solutions, Yadavally et al. (2025) evaluates code edits without execution, and Wang et al. (2026a) shows that effective critics can be learned from sparse outcome signals using rubric-based supervision. Outside code, Xie et al. (2025) trains FenCE, a claim-level critic for factuality, while Asawa et al. (2026) learns small advisor models that steer stronger black-box models via instance-specific instructions. These critics operate on completed outputs or through the agent's initial prompt, without revisiting the agent once generation is underway. Issue resolution instead involves long, tool-using trajectories where many failures stem from flawed reasoning rather than incorrect low-level actions (Liu et al., 2025). We therefore train a compact critic that intervenes at intermediate steps to provide trajectory-aware, strategy-level feedback on *how* a frozen code agent should proceed, rather than judging only the final artifact. This trades additional inference-time compute (one critic call every k agent steps) for the ability to redirect the agent before it commits to a wrong path; we show in §5 that the trade is favorable on three agent models, and on `Qwen3-Next-80B-A3B` that the critic-guided system is in fact cheaper end-to-end than the unguided agent because the shorter trajectories save more agent compute than the critic itself spends.

Auxiliary guidance mechanisms for code agents.. Prior work improves code agents by altering the agent's context, architecture, or parameters. For example, Wang et al. (2026b) improves issue resolution via context selection and pruning. Modular systems instead decompose software-engineering agents into specialized roles such as planners and executors (Arora et al., 2024; Erdogan et al., 2025), and post-training methods directly optimize the code agent itself (Pan et al., 2025; Yang et al.,

2026; Xie et al., 2026; Jain et al., 2025). These approaches still couple strategy-level reasoning with the agent's implementation behavior (tool calls, code edits, prompt-specific action formats). We instead keep the agent frozen and train a separate critic whose role is deliberately narrower: it only supplies high-level feedback while the agent remains responsible for step implementation. This narrower role makes the critic easier to train from existing trajectories, more likely to capture transferable reasoning patterns, and applicable to agents without parameter access. It also motivates constraining the critic away from concrete commands and toward strategic feedback, such as diagnosing failure modes, suggesting next steps, and deciding when to submit.

3 Methodology

A frozen code agent π_A constructs its trajectory one step at a time. A **critic** C is interleaved with this step-by-step process: every k agent steps, C **intervenes** by inspecting the partial trajectory and writing textual feedback that the agent reads before its next step (Fig. 1); C never acts on the environment itself. We use C in three roles, with distinct names throughout the paper: an **untrained critic** (off-the-shelf), a **teacher critic** C_T (frontier model used to generate training data, Fig. 2), and a **trained critic** C_θ (a smaller model SFT'd on C_T critiques, used at inference). The remainder of §3 describes the agent scaffold (§3.1), the critic interface (§3.2), the critique design (§3.3), training data collection (§3.4), and how we obtain C_θ from C_T (§3.5).

3.1 Preliminary: Code-Agent Scaffold

The code agent π_A in Figs. 1 and 2 is the Mini-SWE-Agent scaffold (Yang et al., 2024)[2], which has access to the bash command-line tool. Given a coding task, the agent generates the solution with a trajectory $\tau_{1:t} = (a_1, o_1, \ldots, a_t, o_t)$, where a_i is the agent's action and o_i the corresponding environment observation. We limit the number of steps with a fixed budget $t \leq B$. In our setting, π_A is held frozen throughout and the only trained component is the critic model C_θ. Scaffold mechanics and task-instance specification are detailed in App. B.

[2] https://github.com/SWE-agent/mini-swe-agent

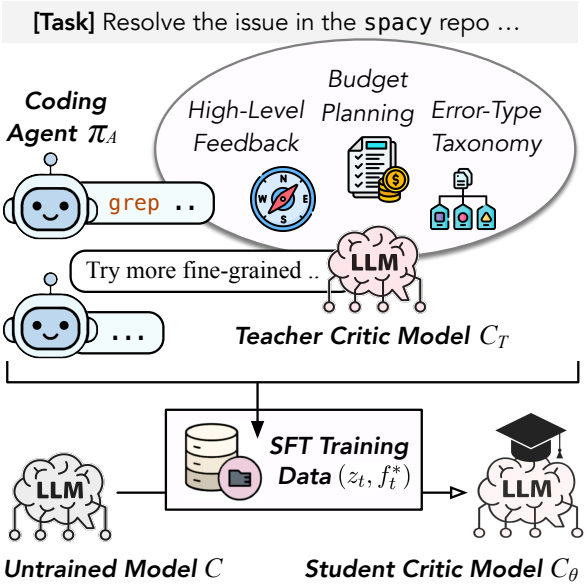

Figure 2: Critic training pipeline. Based on our analysis (§3.3), the critic contains error detection, budget planning, and only provides high-level feedback instead of concrete actions. We run the frozen code agent π_A on disjoint training tasks and query the teacher critic C_T every k steps. We collect (trajectory prefix, feedback) pairs (z_t, f_t^*) to train a critic C_θ by supervised fine-tuning. C_θ is paired with π_A at inference (Fig. 1).

3.2 Critic Models for Code Agents

Why a critic?. Long-horizon code agents drift: they over-commit to an early hypothesis, repeat failed commands, or run out of steps on the wrong subgoal. A second model that inspects the trajectory at runtime can flag such drift, but only if it is strong enough to catch what the agent model itself missed. Pairing our frozen agent model `CWM-32B` with `Claude-Opus-4.6` as a critic on SWE-bench Verified, using the trajectory-feedback prompt of Gandhi et al. (2025), lifts resolve rate from 29.2% (no critic) to 51.4% under the concise prompt and 65.0% under the detailed prompt (Table 1; App. D).

Why critic model training?. Although the detailed prompt performs better, it lets the teacher backseat-drive with explicit action-level instructions, which is exactly the supervision the trained critic should not inherit: a critic that overrides an agent whose understanding of the codebase is better than its own hurts more than it helps (§3.3). We therefore distill from the concise prompt and accept the smaller teacher gain as the price of the more conservative supervision source. A frontier critic at every intervention is also too expensive, which

further motivates training a compact critic (§3.5).

Formal Setup. Formally, we pair the code agent π_A with a critic C, using the teacher C_T as C when generating training data and the trained critic C_θ as C at inference. For every k agent steps (we use $k = 5, 10$ in our experiments), the critic inspects the task and the current trajectory prefix $\tau_{1:t}$. To preserve continuity across interventions, we also include the feedback the critic produced at earlier intervention points. Let $h_t = (f_k, f_{2k}, \ldots, f_{t-k})$ denote the critic's prior feedback history before step t. We form the critic input $z_t = \text{Serialize}(x, \tau_{1:t}, h_t)$ and sample feedback $f_t \sim C(\cdot \mid z_t)$, which is appended to the agent's context before its next step.

3.3 Designing the Critique

What should a critic say?. The "High-level only" constraint in Fig. 1 is enforced through the critique design. Following the observation in §3.2 that the trained critic should not inherit the teacher's action-level recipes, the critique we give to the agent has three components: (i) it organizes feedback over the 12-category trajectory-level error taxonomy of Gandhi et al. (2025) (e.g., *Step Repetition*, *Hallucinations*, *Task Derailment*); (ii) it gives *high-level feedback* rather than prescribing explicit actions; and (iii) it adapts as the agent's budget runs out. (i) is inherited from the upstream prompt; (ii) and (iii) are the design choices we study, and they are enforced at different stages: (ii) is a *supervision* property baked in at training time by collecting teacher feedback under the concise prompt, so the trained critic inherits it; (iii) is an *inference-time* mechanism, budget-aware instructions appended to the trained critic's own input late in the trajectory and never used during data collection. The three paragraphs below introduce each component and the mechanism behind it.

Trajectory-level error taxonomy. We adopt the prompt of Gandhi et al. (2025), which organizes critic feedback over 12 trajectory-level error categories grouped under *Specification*, *Reasoning*, and *Coordination*. For each category the critic reports whether it is detected with a one-sentence evidence/recovery field, and closes with a task-status verdict ("on track", "needs correction", or "critical intervention").

Providing only high-level feedback. The upstream prompt allows action-level prescriptions in the recovery field; we use it as our *detailed* setting. For our *concise* setting, we add four rules: (i) default to "On track" unless there is clear evidence of a problem; (ii) never suggest specific technical approaches or code-level changes; (iii) recommend immediate submission once the agent has verified a fix; and (iv) keep responses to 1-2 sentences per detected error. The concise prompt thereby restricts the teacher to high-level signals about whether the agent is making progress, looping, relying on an unsupported assumption, or pursuing the wrong subgoal (App. C).

Budget adherence. A high-level critic that ignores the budget will keep recommending exploration even when the agent is close to running out of steps, leaving useful in-progress work unsubmitted. *During inference*, once the agent passes step T (we use T=100 of the B=150-step budget), we append short budget-aware instructions that state the steps remaining and asks the critic to prioritize recommending submission if the agent has applied a fix or appears stuck (App. C.3).

3.4 Training Data Construction

We now describe how we collect the (trajectory prefix, feedback) training pairs (z_t, f_t^*) shown at the top of Fig. 2, which we will later use to train C_θ. Two design choices govern this collection. First, the supervision source is the *concise prompt* of §3.3, which constrains the teacher critic C_T to high-level feedback rather than action-level prescriptions; sampling $f_t^* \sim C_T(\cdot \mid z_t)$ under this prompt is what produces our training labels. Second, we collect feedback along trajectories that the agent has actually produced under critic guidance, rather than from off-policy rollouts: we run the agent π_A end-to-end with C_T intervening every k steps, so each prefix $\tau_{1:t}$ already reflects the agent's reaction to prior critic interventions h_t. This matches the distribution the trained critic will face at inference, where the prefix will be shaped by its own earlier feedback. We use training tasks that are disjoint at the instance level from the evaluation benchmark (R2E-Gym tasks in SWE-bench repositories; described in §4), so the critic is never exposed to test-time issues during training. As an ablation, we also collect a parallel corpus under the detailed prompt.

3.5 Critic Model Training with Distillation

Given these pairs, we distill C_θ from C_T by supervised fine-tuning (bottom of Fig. 2). Because z_t already contains the agent's intermediate actions and the critic's earlier feedback, an unmasked SFT

objective would teach the critic to imitate the agent as well as the critique; we therefore restrict the loss to the final critic turn, so C_θ is supervised only on producing f_t^* given the trajectory and feedback history. We filter out malformed teacher responses (empty completions or responses that drop out of the critique format) before training; filtering rules, truncation policy, and hyperparameters are described in §4. The trained C_θ is then paired with π_A at inference using the same every-k-step protocol of §3.2, with the budget-aware instructions of §3.3 appended past step T.

4 Experimental Setup

§4 isolates the factors that bear on whether a small trained critic helps a frozen agent model and transfers across agent models. We vary three factors: the **agent model** (§4.3), to test transfer across architecture and post-training recipe; the **critic configuration** (§4.4), which compares the trained critic against no-critic, an off-the-shelf untrained critic of the same size, and the teacher critic, so the gain can be attributed to training rather than to interleaving any 8B model; and the **supervision prompt format** (§4.4, concise vs. detailed, ablated on `CWM-32B`), which tests the design choice from §3.3. Intervention interval k and base-run fallback are reported in §4.5, App. D, and App. E.

4.1 Benchmark

We evaluate on SWE-bench Verified (Jimenez et al., 2024), a benchmark of 500 human-verified GitHub issues drawn from 12 open-source Python repositories. For each instance, the agent is given the issue's problem statement and access to the pre-issue repository snapshot, and must edit the codebase to resolve the issue. An instance is counted as resolved only if the resulting patch satisfies two hidden test sets: `FAIL_TO_PASS` tests that target the reported bug, and `PASS_TO_PASS` tests that guard against regressions in unrelated code.

4.2 Metrics

We report five metrics in three groups. *Outcome quality:* (i) **resolve rate**, the fraction of the 500 instances whose submitted patch passes both the `FAIL_TO_PASS` and `PASS_TO_PASS` test sets; (ii) **resolve-on-submitted rate**, resolve rate conditioned on submission, isolating patch quality from any change in submission rate; (iii) **localization rate**, the fraction of submitted patches that edit at least one file also touched by the gold patch (Xia et al., 2025). *Trajectory process:* (iv) **submission rate**, the fraction of runs that exit through the agent's submit command within the B=150 budget. *Cost:* (v) **cost per instance**, the mean cost in USD per instance, summed over agent and critic, computed from token usage and the rates in App. F. Trajectory-length and stuck-in-loop statistics are reported in App. E.

4.3 Models

Agent. We pair the critic with three open-weight agents spanning the two prevailing architectures for open code agents: dense transformers and mixture-of-experts. **`CWM-32B`** (team et al., 2025) is a 32B dense model post-trained for software engineering. **`Qwen3-32B`** (Yang et al., 2025) is a 32B dense general-purpose model. **`Qwen3-Next-80B-A3B-Instruct`** (Yang et al., 2025) is an 80B mixture-of-experts model with 3B active parameters per token. The agent is frozen throughout; we never update its weights.

Critic. The teacher is **`Claude-Opus-4.6`**, a strong proprietary model queried directly at inference. The student is **`Qwen3-8B`** (Yang et al., 2025), an open-weight 8B dense model used either off-the-shelf or distilled from `Claude-Opus-4.6` critiques via SFT (§3.3).

4.4 Critic Configurations

We compare four critic configurations on each agent. Each prompted critic uses one of two prompt formats, **detailed** or **concise** (App. C).

(1) No critic. The unguided agent.

(2) Untrained critic. Off-the-shelf `Qwen3-8B`.

(3) Trained critic (ours). `Qwen3-8B` trained on `Claude-Opus-4.6`-generated critiques (§3.4).

(4) Teacher. `Claude-Opus-4.6`.

4.5 Implementation Details

Scaffold. All agent runs use the mini-SWE-agent scaffold described in §3.1, with a budget of 150 steps, $6.0 and 60-seconds per-command.

Sampling and serving. Open-weight models are served with vLLM[3] on 8×L40S; `Claude-Opus-4.6` is accessed via its hosted API. We sample at `temperature` 0, `top-p` 1, with a per-turn output cap of 2048 tokens.

Critic training. Training trajectories are collected by running `CWM-32B` on 500 instances randomly sampled from R2E-Gym tasks in SWE-Bench

[3] https://github.com/vllm-project/vllm

Critic	Res. (%)	Sub. (%)	Res. on Sub. (%)	Loc. (%)	Critic $	Total $
Agent: CWM-32B						
No critic	29.2	92.6	31.5	73.6	—	**0.12** (↓0.95)
Qwen3-8B	29.2 (0.0)	**95.6** (↑3.0)	30.5 (↓1.0)	72.2 (↓1.4)	0.01 (↓0.95)	0.15 (↓0.92)
Ours	**33.0** (↑3.8)	92.4 (↓0.2)	**35.7** (↑4.2)	**75.8** (↑2.2)	**0.01** (↓0.95)	0.27 (↓0.80)
Opus 4.6 (upper bound)	51.4 (↑22.2)	98.4 (↑5.8)	52.2 (↑20.7)	85.6 (↑12.0)	0.96	1.07
Agent: Qwen3-Next-80B-A3B						
No critic	20.8	97.0	21.4	72.6	—	0.11 (↓0.06)
Qwen3-8B	20.6 (↓0.2)	98.8 (↑1.8)	20.9 (↓0.5)	70.0 (↓2.6)	∼0.00 (↓0.15)	0.05 (↓0.12)
Ours	**23.8** (↑3.0)	**99.8** (↑2.8)	**23.8** (↑2.4)	**74.2** (↑1.6)	∼0.00 (↓0.15)	**0.04** (↓0.13)
Opus 4.6 (upper bound)	38.2 (↑17.4)	100.0 (↑3.0)	38.2 (↑16.8)	81.4 (↑8.8)	0.15	0.17
Agent: Qwen3-32B						
No critic	9.6	**88.8**	10.8	57.2	—	**0.05** (↓0.23)
Qwen3-8B	10.4 (↑0.8)	85.6 (↓3.2)	12.1 (↑1.3)	56.2 (↓1.0)	∼0.00 (↓0.24)	0.06 (↓0.22)
Ours	**13.0** (↑3.4)	86.6 (↓2.2)	**15.0** (↑4.2)	**59.4** (↑2.2)	0.01 (↓0.23)	0.06 (↓0.22)
Opus 4.6 (upper bound)	27.4 (↑17.8)	93.0 (↑4.2)	29.5 (↑18.7)	73.8 (↑16.6)	0.24	0.28

Table 1: SWE-bench Verified (500 tasks), all with fallback; per critic we report the best $k \in \{5, 10\}$. Ours = `Qwen3-8B` SFT'd on concise-prompt teacher critiques. Quality arrows vs. no-critic; cost arrows vs. Opus k=5. **Bold** = best non-Opus per agent; underlined = best overall. Steps, loops, and agent cost are in App. E; concise vs. detailed prompt training comparison on `CWM-32B` is in Table 2.

repositories (disjoint from SWE-bench at the instance level), querying `Claude-Opus-4.6` every $k = 5$ agent steps for trajectory feedback. The trained critic is evaluated on all three agents in §4.3, including the two unseen at training. We denoise the data by discarding teacher responses that lack the expected error-taxonomy markers (likely format failures or agent text mistakenly logged as feedback). To fit the 32,768-token context, we drop the oldest trajectory and feedback blocks first while preserving the system prompt, task description, final prompt, and teacher response in full. This yields 4,532 training samples under the concise prompt (headline) and 3,135 under the detailed prompt. We train `Qwen3-8B` for 3 epochs with AdamW (peak LR 5×10^{-6}, cosine schedule, 10% linear warmup, bfloat16); loss is computed only on teacher critique tokens, with agent actions, observations, and prior critiques kept in context but masked out. Global batch size is 8 across 8×L40S with gradient checkpointing.

Base-run fallback. If a critic-guided run does not submit within the B-step budget, we run the instance again with no critiques. Each critic-guided run thus uses two agent runs in expectation. To match this compute, our headline baseline is **no critic + fallback**: two independent base runs, taking run 1's patch if it submits, else run 2's.

Shared prefix. Every critic-guided run shares its first k agent steps with the base run that supplies its fallback. This isolates the critic's contribution from rollout variance: small differences in early actions otherwise cascade into divergent observations (e.g., reading lines 50-100 instead of 100-150 of a file).

Default settings. Our headline trained critic uses the **concise** prompt, $k = 10$, budget-aware instructions (§3.3), and base-run fallback.

5 Results and Analysis

5.1 Main Results

Table 1 reports our main results on SWE-bench Verified across three agent models (`CWM-32B`, `Qwen3-Next-80B-A3B`, `Qwen3-32B`). All rows use base-run fallback; for each critic we report the $k \in \{5, 10\}$ that maximizes resolve rate. The trained critic is always queried with the concise prompt at inference: it is smaller than the agent, and the concise prompt prevents it from backseat-driving with action-level prescriptions it is not qualified to give (§3.2). Our headline configuration ("Ours" in Table 1) is `Qwen3-8B` SFT'd on the concise-prompt teacher critiques. The full per-cell sweep over k and fallback is in App. E; App. D sweeps `Claude-Opus-4.6` across prompt format and k. A separate prompt-format ablation on `CWM-32B` (Table 2) compares concise- vs. detailed-prompt training.

Does a critic trained on one agent model transfer to others? Yes, and the gain is nearly the same

on the two unseen agent models as on the one it was trained against. Our critic sees trajectories from CWM-32B only at training (§4.5) but improves resolve rate on all three agent models at inference: +3.8 points on CWM-32B (29.2 → 33.0), +3.0 on Qwen3-Next-80B-A3B (20.8 → 23.8), and +3.4 on Qwen3-32B (9.6 → 13.0). The two off-distribution agent models differ from CWM-32B in architecture (80B MoE, 32B dense) and post-training recipe, yet the gain on each is within 0.8 points of the in-distribution gain. Other quality metrics move in the same direction on every agent model: localization rises by +1.6 to +2.2 points and resolve-on-submitted by +2.4 to +4.2 points, while submission rate shifts by at most ±2.8. The steering signal is not specific to the agent model it was trained against.

What does the gain cost relative to the teacher critic? A small fraction of it. The trained critic adds only \$0.004-\$0.010 of critic compute per instance, 40-92× cheaper than the Claude-Opus-4.6 teacher (\$0.15-\$0.96 across the three agent models; per-agent ratios in App. E), and recovers 17-19% of the resolve-rate gap between no-critic and Claude-Opus-4.6 on each. Total per-instance cost is 4-5× below Claude-Opus-4.6 on every agent model. On Qwen3-Next-80B-A3B the trained critic Pareto-dominates the unguided agent model: strictly better on both resolve rate (23.8 vs. 20.8) and total per-instance cost (\$0.04 vs. \$0.11), because the critic also shortens the agent's trajectories enough to offset its own cost (App. E).

Is the gain from training, or from interleaving any 8B model? From training. An off-the-shelf Qwen3-8B used as the critic with no SFT tracks the no-critic baseline within ±1 point on resolve rate on every agent model and shows no improvement in localization. The full +3.0 to +3.8 point resolve gain therefore comes from SFT, not from interjecting an 8B model into the agent's loop.

Which prompt format produces the better trained critic? The concise one, even though the detailed prompt wins at the teacher. On CWM-32B, the concise-trained critic reaches 33.0% resolve rate vs. 32.0% for the detailed-trained critic (Table 2). This reverses the prompt-format ranking at the teacher, where detailed outperforms concise by 13.6 points on CWM-32B (65.0 vs. 51.4 at k=5, App. D). The reversal is consistent with the hypothesis in §3.3: Claude-Opus-4.6 is a much stronger code agent than CWM-32B, so its detailed-prompt critiques carry low-level implementation knowledge that the agent could in principle exploit. However, the 8B SFT critic cannot reliably learn such implementation-level knowledge. The concise prompt restricts supervision to high-level signals and is therefore the form of feedback the trained critic can faithfully inherit.

Critic	Res. (%)	Total \$
No critic	29.2	0.12 (↓0.95)
Vanilla SFT (detailed-trained)	32.0 (↑2.8)	0.21 (↓0.86)
Ours (concise-trained)	**33.0** (↑3.8)	0.27 (↓0.80)
Teacher critic (upper bound)	51.4 (↑22.2)	1.07

Table 2: Prompt-format ablation on CWM-32B (best $k \in \{5, 10\}$, fallback enabled). Both trained critics are evaluated under the concise prompt at inference. Res. arrows vs. no-critic; Total \$ arrows vs. Opus teacher.

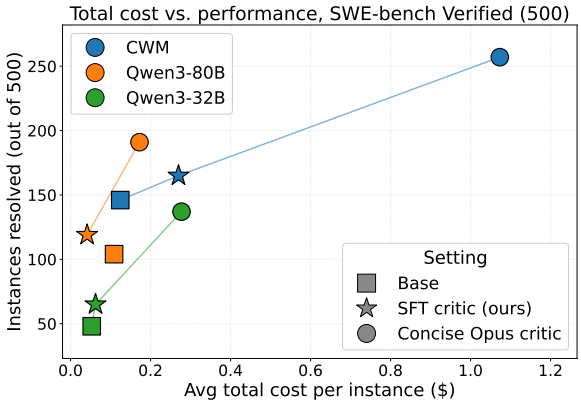

Figure 3: Cost-performance Pareto on SWE-bench Verified across three agent models, all with fallback. Each agent shows base run (■), trained critic (★), and Opus teacher (●). Cost is per-instance agent+critic.

5.2 Qualitative Analysis

§5.1 leaves two follow-ups open. *What does SFT actually change in the critic's outputs*, given that an off-the-shelf 8B critic produces no resolve gain? And *why does the detailed prompt win at the Claude-Opus-4.6 teacher but lose at the 8B trained critic*, reversing the prompt-format ranking? All case studies are on CWM-32B, drawn from base, untrained-critic, and trained-critic runs that diverge in outcome, plus a 50-instance Claude-Opus-4.6 teacher subsample run under both prompt formats.

What SFT removes, and what it adds. The untrained Qwen3-8B fails in two ways: only 43.4%

of its outputs are well-formed (vs. 85.8% after SFT) and 19.3% contain hallucinated bash command actions (vs. 0% after SFT). When it does raise an error it almost always flags only Verification Failures (330/354 firings) with a "run the test script" recovery, pulling the agent off self-review even on instances the base agent would otherwise have salvaged (App. G.1). Beyond removing this noise, the trained critic resolves 41 `CWM-32B` instances the base agent does not. The common pattern is a strategy-level redirect: the critic identifies the failure category the agent is in and points to a different class of approach, without dictating the specific code. Two case studies illustrate the pattern. On `django__django-9296`, `sed` append commands silently duplicate a method each time the pattern matches; the critic jointly flags Step Repetition and Tool Selection Errors and redirects the agent to a heredoc rewrite, which resolves the task (App. G.1). On `sympy__sympy-22714`, a `ModuleNotFoundError: mpmath` blocks execution-based verification; the critic names the import as the root cause and tells the agent to bypass it and read the source directly (App. G.2).

Backseat driving in the detailed teacher prompt. Let ρ denote the fraction of the agent's final-submission lines copied verbatim from the critic. On a 50-instance `CWM-32B` subsample with the `Claude-Opus-4.6` teacher, the detailed prompt emits 6.7 code lines per firing and yields $\rho \geq 0.30$ on 23/50 instances at k=5; the concise prompt cuts these to 1.3 and 1/50 (Table 3). The detailed prompt's higher resolve rate (32/50 vs. 22/50) is driven by this copying rather than agent reasoning: when the teacher dictates a single patch, the agent transcribes it and resolves; when the teacher offers multiple candidate fixes without committing, the agent picks wrong and writes a confidently incorrect patch (App. G.3). The 8B trained critic cannot reproduce code-level dictation under SFT, so the prompt that wins at the teacher does not transfer.

Prompt	k	Lines/fire	Mean share	≥ 0.30
Detailed	5	6.7	0.32	23/50
Concise	5	1.3	0.01	1/50
Detailed	10	7.1	0.27	10/49
Concise	10	1.9	0.00	0/49

Table 3: `Claude-Opus-4.6` teacher backseat driving on the 50-instance `CWM-32B` subsample. *Lines/fire*: mean code lines per firing. *Mean share*: mean ρ, fraction of submission lines copied from the critic. ≥ 0.30: instances above threshold.

6 Discussion

End-to-end agent training plateaus on strategy-level reasoning (§1), and our small trained critic recovers part of that gap without touching the agent. Adjacent work does not address this gap: post-hoc code critics that score completed trajectories (McAleese et al., 2024; Gao et al., 2025; Yadavally et al., 2025; Wang et al., 2026a) cannot redirect a wrong run, and continued agent post-training (Pan et al., 2025; Jain et al., 2025) couples strategy to execution and saturates. The supervision is agent-agnostic: the error categories of Gandhi et al. (2025) describe failure modes shared across models, and the concise prompt avoids agent-specific action vocabulary. A critic trained on `CWM-32B` therefore transfers to `Qwen3-Next-80B-A3B` and `Qwen3-32B`, and on `Qwen3-Next-80B-A3B` shortens trajectories enough to stay on the Pareto frontier (Fig. 3). The trained critic also has failure modes: on some agents it induces repeated commands or budget exhaustion (App. G.4), because the SFT loss matches teacher tokens at each prefix and never penalizes critiques the agent reacts to this way. It often diagnoses a loop correctly but reissues a near-identical `OVERALL_GUIDANCE` every k steps until the agent stops acting on it (App. G.5). Optimizing critic trainig for agent-side outcomes with DPO over critique pairs (Rafailov et al., 2024) could be a promising direction for future work.

7 Conclusion

We study whether a small, trained critic can supply the strategy-level signal that a code agent's end-to-end training leaves underdeveloped. Our recipe distills `Qwen3-8B` from `Claude-Opus-4.6` trajectory critiques on one agent's rollouts, restricts it to high-level feedback, and intervenes every k steps with budget-aware instructions and base-run fallback. On SWE-bench Verified, the same critic improves resolve rate on three agents by +3.0 to +3.8 points at 40 to 92× lower critic cost than the teacher, and on `Qwen3-Next-80B-A3B` is cheaper than the unguided agent. Transfer to two unseen agents, alongside a same-size untrained critic that does not move resolve rate, points to the supervision rather than a second model in the loop as the source of gain. Our results show that training critic models is a practical complement to scaling agent training.

Limitations

Our findings are based on Python issue resolution in SWE-bench Verified, evaluated under the mini-SWE-Agent scaffold with a fixed step budget and every-k-step intervention; the trajectory-level error taxonomy that drives the critique was developed on a similar distribution, so the same recipe may need slight modification before it transfers to non-issue tasks (feature implementation, refactoring) or to other languages and toolsets. The SWE-bench Verified instances overlap by repository with the publicly released agent and teacher pretraining corpora, so absolute resolve rates may be inflated by familiarity; the relative comparison across critics holds the agent and prompt fixed and is unaffected, but cross-paper absolute comparisons should be made cautiously. Second, the trained critic is a single SFT run on critiques from `Claude-Opus-4.6`, and its quality is conditioned on the teacher, so a substantially weaker teacher could shift both the absolute gain and the prompt-format ranking we report. Due to compute limitations, we do not show results over multiple runs per configuration.

References

Anthropic. 2026. Claude opus 4.6.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. Masai: Modular architecture for software-engineering ai agents. *Preprint*, arXiv:2406.11638.

Parth Asawa, Alan Zhu, Abby O'Neill, Matei Zaharia, Alexandros G. Dimakis, and Joseph E. Gonzalez. 2026. How to train your advisor: Steering black-box llms with advisor models. *Preprint*, arXiv:2510.02453.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-act: improving planning of agents for long-horizon tasks. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org.

Shubham Gandhi, Jason Tsay, Jatin Ganhotra, Kiran Kate, and Yara Rizk. 2025. When agents go astray: Course-correcting swe agents with prms. *Preprint*, arXiv:2509.02360.

Bofei Gao, Zefan Cai, Runxin Xu, Peiyi Wang, Ce Zheng, Runji Lin, Keming Lu, Dayiheng Liu, Chang Zhou, Wen Xiao, and 1 others. 2025. Llm critics help catch bugs in mathematics: Towards a better mathematical verifier with natural language feedback. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 14588–14604.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pages 39578–39601.

Simiao Liu, Fang Liu, Liehao Li, Xin Tan, Yinghao Zhu, Xiaoli Lian, and Li Zhang. 2025. An empirical study on failures in automated issue solving. *Preprint*, arXiv:2509.13941.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. 2024. Llm critics help catch llm bugs. *arXiv preprint arXiv:2407.00215*.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with SWE-gym. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 47717–47737. PMLR.

Deepak Pandita, Tharindu Cyril Weerasooriya, Ankit Parag Shah, Isabelle Diana May-Xin Ng, Christopher M Homan, and Wei Wei. 2025. Prorefine: Inference-time prompt refinement with textual feedback. *arXiv preprint arXiv:2506.05305*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Preprint*, arXiv:2305.18290.

Ming Shen, Raphael Shu, Anurag Pratik, James Gung, Yubin Ge, Monica Sunkara, and Yi Zhang. 2025. Optimizing llm-based multi-agent system with textual feedback: A case study on software development. *arXiv preprint arXiv:2505.16086*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement

learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pages 8634–8652.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, and 32 others. 2025. Cwm: An open-weights llm for research on code generation with world models. *Preprint*, arXiv:2510.02387.

Xingyao Wang, Valerie Chen, Heng Ji, and Graham Neubig. 2026a. A rubric-supervised critic from sparse real-world outcomes. *arXiv preprint arXiv:2603.03800*.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. 2026b. Swe-pruner: Self-adaptive context pruning for coding agents. *arXiv preprint arXiv:2601.16746*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE).

Yiqing Xie, Emmy Liu, Gaokai Zhang, Nachiket Kotalwar, Shubham Gandhi, Sathwik Acharya, Xingyao Wang, Carolyn Rose, Graham Neubig, and Daniel Fried. 2026. Hybrid-gym: Training coding agents to generalize across tasks. *Preprint*, arXiv:2602.16819.

Yiqing Xie, Wenxuan Zhou, Pradyot Prakash, Di Jin, Yuning Mao, Quintin Fettes, Arya Talebzadeh, Sinong Wang, Han Fang, Carolyn Rose, Daniel Fried, and Hejia Zhang. 2025. Improving model factuality with fine-grained critique-based evaluator. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics.

Aashish Yadavally, Hoan Nguyen, Laurent Callot, and Gauthier Guinet. 2025. Large language model critics for execution-free evaluation of code changes. *arXiv preprint arXiv:2501.16655*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2026. SWE-smith: Scaling data for software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

A Additional Related Work

Textual feedback for prompt optimization. A related line of work uses textual feedback not as the output of a standalone critic model, but as a signal for improving prompts, roles, or coordination strategies within a larger agent system. Shen et al. (2025) is a representative example, showing that failure feedback can be used to improve a multi-agent coding system. Likewise, Pandita et al. (2025) studies refinement of prompts using feedback signals at inference time. Closely related, Shinn et al. (2023) proposes to improve agents via natural language reflections stored in memory, enabling agents to learn from prior failures without parameter updates. These approaches demonstrate that textual feedback can serve as a powerful learning signal. However, their primary optimization target is the prompting strategy or in-context behavior of the agent, rather than a compact critic model that can serve as an independent advisor during issue resolution.

B Method Preliminaries

Division of labor. The code agent π_A handles concrete repository interaction: inspecting files, running commands, editing code, and deciding when to submit. The critic instead provides high-level judgments about the *trajectory*: whether the agent is making progress, pursuing an unsupported assumption, repeating an unproductive strategy, or needs to reconsider its overall direction.

Why concise feedback. We do not want the critic to become a second actor that "backseat drives" π_A with commands, file-level instructions, or step-by-step recovery plans. The code agent is already the more capable model for concrete tool use and code editing; asking a smaller critic to prescribe specific actions risks replacing its reasoning with weaker instructions. We therefore train the critic to provide *concise strategy-level feedback*, not action-level control.

Scaffold details. The scaffold gives the code agent a simple command-line interface rather than a large set of specialized tools. Keeping the scaffold minimal lets us study the effect of the critic without confounding it with scaffold-specific machinery. Each task instance x consists of a natural-language issue description and a repository snapshot. At each step, the code agent π_A may reason about the task, execute shell commands, inspect files, edit code, or submit a final patch. A run succeeds if the submitted patch passes the benchmark's evaluation harness.

C Critic Prompts

This appendix gives the two critic prompts compared in §3.3 and §5.1: the *concise* prompt that constrains the critic to high-level feedback, and the *detailed* prompt that permits action-level prescriptions. Both prompts adapt the trajectory-feedback prompt of Gandhi et al. (2025) and share its 12-category trajectory-level error taxonomy. The detailed prompt uses the upstream version unchanged; the concise prompt adds the four framing rules listed in §3.3, removes the per-category recovery hints, and tightens the example response. Both critics receive the same agent-side context (task description, trajectory prefix, and prior critic feedback history) at every k steps. Once the agent passes step 100 of its 150-step budget, the budget-aware instructions in §C.3 are appended to whichever prompt is in use; they prioritize recommending submission as the budget approaches.

C.1 Concise Prompt

```
You are a supervisor monitoring an LLM-based code agent. Your role is to detect
↪ trajectory-level errors and provide corrective guidance to prevent task failure.
Analyze the provided agent trace for the following error categories. For each
↪ detected error, provide specific evidence and a recovery action.

The agent works in an isolated repo; assume no hidden external edits. There is
↪ EXACTLY ONE final submission command; it is ALWAYS essential.

IMPORTANT RULES:
- Respond ONLY with analytical feedback in the format below. Never generate bash
↪ commands, code blocks, or simulated agent/environment output.
- Be CONSERVATIVE. Default to "On track" unless you see clear evidence of a problem.
↪  The agent is competent - let it work. Unnecessary interventions cause the agent
↪ to second-guess itself and loop, which is worse than a slow but steady trajectory.
↪
- Use "Critical intervention required" ONLY for: (1) source file is corrupted/broken
↪  by a bad edit, or (2) agent has repeated the exact same action 3+ times with no
↪ progress. Most other issues are "Needs correction" at worst.
- NEVER suggest specific technical approaches, code changes, or where in the code to
↪  look. The agent understands the codebase better than you. If the agent's
↪ approach seems wrong, say so, but do NOT suggest an alternative - let the agent
↪ decide.
- If the agent has applied a fix and verified it works, tell it to submit
↪ immediately.
- Keep your response SHORT. 1-2 sentences per detected error. 1-2 sentences for
↪ OVERALL_GUIDANCE. Long feedback is counterproductive.

SPECIFICATION ERRORS:

1. Task Specification Violations
Definition: Agent fails to adhere to task constraints or requirements

2. Role Specification Violations
Definition: Agent behaves outside its defined role/responsibilities

3. Step Repetition
Definition: Unnecessary repetition of completed steps or actions

4. Termination Condition Unawareness
Definition: Agent continues working when task completion criteria are met

REASONING ERRORS:

5. Problem Misidentification
Definition: Agent misunderstands the core problem or current subtask

6. Tool Selection Errors
Definition: Agent uses inappropriate tools for the current task

7. Hallucinations
Definition: Agent generates false information or fabricates tool outputs

8. Information Processing Failures
```

```
Definition: Poor retrieval of relevant information or misinterpretation

COORDINATION ERRORS:

9. Task Derailment
Definition: Agent deviates from intended objective or loses focus

10. Goal Deviation
Definition: Agent pursues goals that don't serve the main objective

11. Context Handling Failures
Definition: Agent loses important context or forgets previous findings

12. Verification Failures
Definition: Inadequate checking of work quality or correctness

=====================================================
RESPONSE FORMAT
=====================================================

For each error category, respond with:
DETECTED: Yes/No
EVIDENCE: (if detected) One sentence.
RECOVERY_ACTION: (if detected) One sentence. No code, no commands.

Then provide:
TASK_STATUS: Exactly one of: "On track" / "Needs correction" / "Critical
↪ intervention required"
OVERALL_GUIDANCE: 1-2 sentences only.

EXAMPLE:

SPECIFICATION ERRORS:
1. Task Specification Violations: DETECTED: No
2. Role Specification Violations: DETECTED: No
3. Step Repetition: DETECTED: Yes
EVIDENCE: Agent ran the same test command three times with identical results.
RECOVERY_ACTION: Stop repeating and analyze the existing test output.
4. Termination Condition Unawareness: DETECTED: No

REASONING ERRORS:
5. Problem Misidentification: DETECTED: No
6. Tool Selection Errors: DETECTED: No
7. Hallucinations: DETECTED: No
8. Information Processing Failures: DETECTED: No

COORDINATION ERRORS:
9. Task Derailment: DETECTED: No
10. Goal Deviation: DETECTED: No
11. Context Handling Failures: DETECTED: No
12. Verification Failures: DETECTED: No

TASK_STATUS: Needs correction
OVERALL_GUIDANCE: You are repeating the same test. Analyze the output you already
↪ have and proceed to the next step.

Now review the agent's trajectory and provide your supervisor feedback.
```

C.2 Detailed Prompt

The detailed prompt below is reproduced verbatim from Gandhi et al. (2025) for reproducibility. The concise prompt above is our modification of it; see §3.3 for the four added framing rules.

```
You are a supervisor monitoring an LLM-based code agent. Your role is to detect
↪ trajectory-level errors and provide corrective guidance to prevent task failure.
Analyze the provided agent trace for the following error categories. For each
↪ detected error, provide specific evidence and a recovery action.
```

```
The agent works in an isolated repo; assume no hidden external edits. There is
↪ EXACTLY ONE final submission command; it is ALWAYS essential.

SPECIFICATION ERRORS (System Design Issues)

1. Task Specification Violations
Definition: Agent fails to adhere to task constraints or requirements
Recovery: Redirect agent to original task requirements

2. Role Specification Violations
Definition: Agent behaves outside its defined role/responsibilities
Recovery: Remind agent of its specific role and boundaries

3. Step Repetition
Definition: Unnecessary repetition of completed steps or actions
Recovery: Acknowledge completed work and guide to next logical step

4. Termination Condition Unawareness
Definition: Agent continues working when task completion criteria are met
Recovery: Signal completion criteria and instruct to finalize

REASONING ERRORS (Decision Making Issues)

5. Problem Misidentification
Definition: Agent misunderstands the core problem or current subtask
Recovery: Clarify the actual problem and expected approach

6. Tool Selection Errors
Definition: Agent uses inappropriate tools for the current task
Recovery: Suggest correct tools and explain their appropriate usage

7. Hallucinations
Definition: Agent generates false information or fabricates tool outputs
Recovery: Request verification of claims against actual evidence

8. Information Processing Failures
Definition: Poor retrieval of relevant information or misinterpretation
Recovery: Guide agent to correct information sources and interpretation

COORDINATION ERRORS (Process Management Issues)

9. Task Derailment
Definition: Agent deviates from intended objective or loses focus
Recovery: Realign agent with original objectives and priorities

10. Goal Deviation
Definition: Agent pursues goals that don't serve the main objective
Recovery: Refocus on primary goals and expected outcomes

11. Context Handling Failures
Definition: Agent loses important context or forgets previous findings
Recovery: Provide context summary and key information recap

12. Verification Failures
Definition: Inadequate checking of work quality or correctness
Recovery: Instruct specific verification steps and quality checks

=====================================================
RESPONSE FORMAT
=====================================================

For each error category, respond with:
DETECTED: Yes/No
EVIDENCE: Specific quote or observation from trace (if detected)
RECOVERY_ACTION: Specific instruction to correct the error (if detected)

Then provide:
TASK_STATUS: On track / Needs correction / Critical intervention required
OVERALL_GUIDANCE: Detailed and specific guidance for the agent
```

```
Example Response Structure

SPECIFICATION ERRORS:
1. Task Specification Violations: DETECTED: No
2. Role Specification Violations: DETECTED: No
3. Step Repetition: DETECTED: Yes
EVIDENCE: "Agent ran the same test command three times: 'pytest test_file.py'"
RECOVERY_ACTION: "The test has already been executed successfully. Proceed to
↪ analyze the results and move to the next development step."
4. Termination Condition Unawareness: DETECTED: No

REASONING ERRORS:
5. Problem Misidentification: DETECTED: No
6. Tool Selection Errors: DETECTED: Yes
EVIDENCE: "Agent used text editor to run Python code instead of using the Python
↪ interpreter"
RECOVERY_ACTION: "Use the Python interpreter tool for code execution. The text
↪ editor is for viewing and modifying files only."
7. Hallucinations: DETECTED: No
8. Information Processing Failures: DETECTED: No

COORDINATION ERRORS:
9. Task Derailment: DETECTED: No
10. Goal Deviation: DETECTED: No
11. Context Handling Failures: DETECTED: No
12. Verification Failures: DETECTED: No

TASK_STATUS: Needs correction
OVERALL_GUIDANCE: You are repeating actions unnecessarily and using incorrect tools.
↪  Specifically:
1. Stop running the same test command repeatedly - the test 'pytest test_file.py'
↪ has already been executed successfully three times with the same result
2. Use the Python interpreter tool for executing Python code, not the text editor
↪ which is only for viewing and modifying files
3. Now focus on analyzing the test results you already obtained to determine what
↪ the next development step should be
4. Review the test output to identify any failing tests or areas that need
↪ improvement
5. If all tests are passing, proceed to verify your implementation meets the
↪ original requirements before considering the task complete

=====================================================
INSTRUCTIONS
=====================================================

1. Focus on errors that can be corrected through guidance
2. Provide specific, actionable recovery instructions
3. Be concise but precise in evidence citations
4. Only mark "DETECTED: Yes" if you have clear evidence
5. Prioritize errors that most threaten task completion

Now review the agent's trajectory and provide your supervisor feedback.
```

C.3 Budget-Aware Instructions

After the agent passes step 100 of its 150-step budget, the following budget-aware instructions are appended to whichever critic prompt is in use:

```
NOTE: The agent has used {{current_step}} of its {{step_limit}} total steps. Work
↪ that is not submitted before the step limit will be permanently lost. If the
↪ agent has made code changes that address the issue, prioritize recommending
↪ submission. If the agent appears stuck in a loop, recommend that it submit its
↪ current changes immediately rather than continuing to iterate.
```

Critic	k	FB	Res. (%)	Sub. (%)	Res. on Sub. (%)	Steps	Loc. (%)	Loop (%)	Agent $	Critic $	Total $
No critic	—	✗	28.8	89.0	32.4	34.5	71.2	11.6	0.08	0.00	0.08
	—	✓	29.2	92.6	31.5	42.1	73.6	13.6	0.12	0.00	0.12
Opus 4.6 (C)	5	✗	51.0 (↑22.2)	91.4 (↑2.4)	55.8 (↑23.4)	38.6 (↑4.1)	80.8 (↑9.6)	25.2 (↑13.6)	0.10 (↑0.02)	0.96 (↑0.96)	1.06 (↑0.98)
	5	✓	51.4 (↑22.2)	98.4 (↑5.8)	52.2 (↑20.7)	42.3 (↑0.2)	85.6 (↑12.0)	25.2 (↑11.6)	0.12 (0.0)	0.96 (↑0.96)	1.07 (↑0.95)
	10	✗	47.2 (↑18.4)	93.4 (↑4.4)	50.5 (↑18.1)	39.0 (↑4.5)	81.0 (↑9.8)	23.2 (↑11.6)	0.09 (↑0.01)	0.97 (↑0.97)	1.07 (↑0.98)
	10	✓	47.6 (↑18.4)	98.8 (↑6.2)	48.2 (↑16.7)	42.0 (↓0.1)	84.8 (↑11.2)	23.2 (↑9.6)	0.10 (↓0.02)	0.97 (↑0.97)	1.08 (↑0.95)
Opus 4.6 (D)	5	✗	**65.0** (↑36.2)	99.8 (↑10.8)	**65.1** (↑32.7)	**30.2** (↓4.3)	**91.6** (↑20.4)	9.2 (↓2.4)	**0.04** (↓0.04)	0.59 (↑0.59)	0.63 (↑0.55)
	5	✓	**65.0** (↑35.8)	**100.0** (↑7.4)	65.0 (↑33.5)	**30.2** (↓11.9)	**91.6** (↑18.0)	**9.2** (↓4.4)	**0.04** (↓0.08)	0.59 (↑0.59)	0.63 (↑0.51)
	10	✗	55.8 (↑27.0)	99.0 (↑10.0)	56.4 (↑24.0)	32.7 (↓1.8)	89.4 (↑18.2)	14.2 (↑2.6)	0.06 (↓0.02)	**0.49** (↑0.49)	**0.55** (↑0.46)
	10	✓	55.8 (↑26.6)	99.6 (↑7.0)	56.0 (↑24.5)	33.3 (↓8.8)	90.0 (↑16.4)	14.4 (↑0.8)	0.06 (↓0.06)	**0.49** (↑0.49)	**0.55** (↑0.42)

Table 4: Opus 4.6 teacher on CWM-32B, swept across prompt (C/D), $k \in \{5, 10\}$, and fallback. Arrows vs. the matching no-critic row at the same fallback state. **Bold** = best per metric across Opus rows. ✓ = fallback to base run 0 (no-critic + ✓ uses run 0 → run 1).

D Opus Teacher Critic Ablation

Table 4 sweeps the `Claude-Opus-4.6` teacher critic across both prompt formats (concise / detailed), both intervention intervals ($k \in \{5, 10\}$), and both fallback states on the `CWM-32B` agent. The other critic conditions (no-critic baselines, the untrained `Qwen3-8B` critic, and our trained critic) are reported across all three agent models in App. E; here we hold the agent fixed and isolate the teacher's prompt and budget axes. Arrows are vs. the matching no-critic row at the same fallback state.

Opus prompt and budget effects. The detailed prompt (D) dominates the concise prompt (C) on every metric at both k values: under fallback at k=5, D reaches 65.0% resolve vs. C's 51.4%, with shorter trajectories (30.2 vs. 42.3 steps), higher localization (91.6 vs. 85.6%), lower stuck-in-loop rate (9.2 vs. 25.2%), and lower agent cost (\$0.04 vs. \$0.12). The detailed prompt's per-step instructions appear to give the agent enough scaffolding to converge faster, which more than offsets the longer critic prompt: total cost is \$0.63 (D) vs. \$1.07 (C), a $\sim 1.7\times$ saving despite higher resolve rate. Within each prompt, k=5 outperforms k=10 on resolve (51.4 vs. 47.6 for C; 65.0 vs. 55.8 for D), so more frequent intervention is worth the additional critic calls when the critic itself is strong. Fallback adds only a small bump for Opus (which already submits on $\geq 91\%$ of tasks) compared to the much larger bump it provides for the trained critic in App. E.

E Cross-Agent-Model Fallback Ablation

Table 5 reports the cross-agent-model fallback ablation across three agent models with nine diagnostic metrics; App. D complements it by sweeping the `Claude-Opus-4.6` teacher across prompt format and k on `CWM-32B`. The agent models are `CWM-32B` (the main-paper agent), `Qwen3-Next-80B-A3B`, and `Qwen3-32B`. For each we report the no-critic baseline, the untrained `Qwen3-8B` critic, our trained critic ("Ours" = `Qwen3-8B` SFT'd on concise-prompt teacher critiques), and the `Claude-Opus-4.6` teacher critic, all with and without base-run fallback. "Sub." is the submission rate; "Res. on Sub." is resolve rate computed only over the submitted runs (resolved divided by submitted); "Steps" is the average trajectory length in agent steps; "Loc." is the localization rate (the fraction of tasks where the agent visits one of the gold patch's files); "Loop" is the fraction of trajectories flagged stuck-in-loop by the trajectory analyzer; "Agent $" and "Critic $" are the average per-instance costs of the agent and the critic respectively, in US dollars; "Total $" is their sum. Arrows show the direction of change vs. the matching no-critic baseline at the same fallback state.

Cross-agent-model observations. The trained critic improves resolve rate over the no-critic + fallback baseline on all three agent models: `CWM-32B` 33.0 vs. 29.2 ($+3.8$, Ours at k=10), `Qwen3-Next-80B-A3B` 23.8 vs. 20.8 ($+3.0$, Ours at k=5), and `Qwen3-32B` 13.0 vs. 9.6 ($+3.4$, Ours at k=5). The `Claude-Opus-4.6` teacher at k=5 is the upper bound on each agent model (51.4 / 38.2 / 27.4) and remains substantially above the trained critic in every case, mirroring the gap reported in Table 1. The untrained `Qwen3-8B` critic under the detailed prompt ties or trails the no-critic baseline on every agent model where it has been measured: on `CWM-32B` it drops to 27.0 / 29.2 (vs. 28.8 / 29.2), on `Qwen3-Next-80B-A3B` it is essentially flat (20.4 to 20.6 vs. 20.0 to 20.8), and on `Qwen3-32B` it trails the no-critic baseline at the matching fallback state (9.0 / 10.4 vs. 8.8 / 9.6, with the k=5 + fb row notably below baseline). The Agent $ / Critic $ split shows that the trained critic adds only ∼$0.01 of critic compute per instance on every agent model, while the `Claude-Opus-4.6` teacher adds $0.15 to $0.96 depending on the agent model; the resolve-rate gap to `Claude-Opus-4.6` therefore comes at a $40\times$ to $92\times$ critic-cost premium (Critic $ ratios from Table 5: `CWM-32B` $0.9576/0.0104{=}92\times$, `Qwen3-Next-80B-A3B` $0.1539/0.0038{=}40\times$, `Qwen3-32B` $0.2408/0.0050{=}48\times$; comparing the headline Ours row to Opus k=5+fb on each agent model). On `Qwen3-Next-80B-A3B` the trained critic also reduces stuck-in-loop rate, average trajectory length, and per-instance agent cost relative to the no-critic baseline; on `CWM-32B` the trained critic increases trajectory length and stuck-in-loop rate even as resolve rate improves, indicating that on this agent model the additional steps the critic induces are productive.

Critic	k	FB	Res. (%)	Sub. (%)	Res. on Sub. (%)	Steps	Loc. (%)	Loop (%)	Agent $	Critic $	Total $
Agent: CWM-32B											
No critic	—	✗	28.8	89.0	32.4	**34.5**	71.2	**11.6**	**0.08** (↓0.02)	—	**0.08** (↓0.98)
	—	✓	29.2	92.6	31.5	42.1	73.6	13.6	0.12 (0.0)	—	0.12 (↓0.95)
Qwen3-8B	5	✗	—	—	—	—	—	—	—	—	—
	5	✓	—	—	—	—	—	—	—	—	—
	10	✗	27.0 (↓1.8)	84.8 (↓4.2)	31.8 (↓0.6)	39.5 (↑5.0)	64.8 (↓6.4)	21.2 (↑9.6)	0.12 (↑0.02)	0.01 (↓0.95)	0.12 (↓0.94)
	10	✓	29.2 (0.0)	95.6 (↑3.0)	30.5 (↓1.0)	47.4 (↑5.3)	72.2 (↓1.4)	21.6 (↑8.0)	0.14 (↑0.02)	0.01 (↓0.95)	0.15 (↓0.92)
Ours	5	✗	—	—	—	—	—	—	—	—	—
	5	✓	—	—	—	—	—	—	—	—	—
	10	✗	29.2 (↑0.4)	71.6 (↓17.4)	**40.8** (↑8.4)	57.5 (↑23.0)	60.2 (↓11.0)	30.6 (↑19.0)	0.21 (↑0.11)	0.01 (↓0.95)	0.22 (↓0.84)
	10	✓	**33.0** (↑3.8)	92.4 (↓0.2)	35.7 (↑4.2)	71.7 (↑29.6)	75.8 (↑2.2)	31.0 (↑17.4)	0.26 (↑0.14)	**0.01** (↓0.95)	0.27 (↓0.80)
Opus 4.6	5	✗	51.0 (↑22.2)	91.4 (↑2.4)	55.8 (↑23.4)	38.6 (↑4.1)	80.8 (↑9.6)	25.2 (↑13.6)	0.10	0.96	1.06
	5	✓	51.4 (↑22.2)	98.4 (↑5.8)	52.2 (↑20.7)	42.3 (↑0.2)	85.6 (↑12.0)	25.2 (↑11.6)	0.12	0.96	1.07
Agent: Qwen3-Next-80B-A3B											
No critic	—	✗	20.0	92.6	21.6	22.8	69.8	5.0	0.08 (↑0.06)	—	0.08 (↓0.09)
	—	✓	20.8	97.0	21.4	27.0	72.6	5.6	0.11 (↑0.09)	—	0.11 (↓0.06)
Qwen3-8B	5	✗	20.4 (↑0.4)	96.6 (↑4.0)	21.1 (↓0.5)	21.6 (↓1.2)	67.6 (↓2.2)	4.0 (↓1.0)	0.04 (↑0.02)	∼0.00 (↓0.15)	0.05 (↓0.12)
	5	✓	20.4 (↓0.4)	98.8 (↑1.8)	20.6 (↓0.8)	23.6 (↓3.4)	69.8 (↓2.8)	4.2 (↓1.4)	0.06 (↑0.04)	∼0.00 (↓0.15)	0.06 (↓0.11)
	10	✗	20.6 (↑0.6)	97.6 (↑5.0)	21.1 (↓0.5)	19.3 (↓3.5)	69.4 (↓0.4)	5.2 (↑0.2)	0.03 (↑0.01)	∼0.00 (↓0.15)	0.04 (↓0.13)
	10	✓	20.6 (↓0.2)	98.8 (↑1.8)	20.9 (↓0.5)	21.3 (↓5.7)	70.0 (↓2.6)	5.6 (0.0)	0.04 (↑0.02)	∼0.00 (↓0.15)	0.05 (↓0.12)
Ours	5	✗	23.6 (↑3.6)	99.6 (↑7.0)	23.7 (↑2.1)	21.5 (↓1.3)	74.0 (↑4.2)	2.6 (↓2.4)	0.04 (↑0.02)	∼0.00 (↓0.15)	0.04 (↓0.13)
	5	✓	23.8 (↑3.0)	**99.8** (↑2.8)	23.8 (↑2.4)	21.8 (↓5.2)	74.2 (↑1.6)	2.6 (↓3.0)	0.04 (↑0.02)	∼0.00 (↓0.15)	0.04 (↓0.13)
	10	✗	23.6 (↑3.6)	99.4 (↑6.8)	23.7 (↑2.1)	**18.5** (↓4.3)	74.6 (↑4.8)	2.2 (↓2.8)	0.03 (↑0.01)	∼0.00 (↓0.15)	0.03 (↓0.14)
	10	✓	23.6 (↑2.8)	99.6 (↑2.6)	23.7 (↑2.3)	19.1 (↓7.9)	74.8 (↑2.2)	**2.2** (↓3.4)	**0.03** (↑0.01)	∼0.00 (↓0.15)	**0.03** (↓0.14)
Opus 4.6	5	✗	38.2 (↑18.2)	100.0 (↑7.4)	38.2 (↑16.6)	17.0 (↓5.8)	81.4 (↑11.6)	0.6 (↓4.4)	0.02	0.15	0.17
	5	✓	38.2 (↑17.4)	100.0 (↑3.0)	38.2 (↑16.8)	17.0 (↓10.0)	81.4 (↑8.8)	0.6 (↓5.0)	0.02	0.15	0.17
Agent: Qwen3-32B											
No critic	—	✗	8.8	74.8	11.8	33.0	49.8	8.0	0.04 (↑0.01)	—	0.04 (↓0.23)
	—	✓	9.6	**88.8**	10.8	43.9	57.2	9.6	0.05 (↑0.01)	—	0.05 (↓0.23)
Qwen3-8B	5	✗	7.2 (↓1.6)	62.0 (↓12.8)	11.6 (↓0.2)	38.2 (↑5.2)	37.4 (↓12.4)	11.8 (↑3.8)	0.05 (↑0.02)	0.01 (↓0.23)	0.05 (↓0.22)
	5	✓	9.0 (↓0.6)	84.6 (↓4.2)	10.6 (↓0.2)	55.1 (↑11.2)	52.4 (↓4.8)	14.4 (↑4.8)	0.07 (↑0.03)	0.01 (↓0.23)	0.08 (↓0.20)
	10	✗	10.0 (↑1.2)	73.0 (↓1.8)	13.7 (↑1.9)	32.5 (↓0.5)	49.4 (↓0.4)	7.0 (↓1.0)	0.04 (↑0.01)	∼0.00 (↓0.24)	0.04 (↓0.23)
	10	✓	10.4 (↑0.8)	85.6 (↓3.2)	12.1 (↑1.3)	47.5 (↑3.6)	56.2 (↓1.0)	9.8 (↑0.2)	0.06 (↑0.02)	∼0.00 (↓0.24)	0.06 (↓0.22)
Ours	5	✗	11.4 (↑2.6)	65.0 (↓9.8)	**17.5** (↑5.7)	30.9 (↓2.1)	45.8 (↓4.0)	7.4 (↓0.6)	0.04 (↑0.01)	0.01 (↓0.23)	0.04 (↓0.23)
	5	✓	**13.0** (↑3.4)	86.6 (↓2.2)	15.0 (↑4.2)	46.6 (↑2.7)	59.4 (↑2.2)	11.6 (↑2.0)	0.06 (↑0.02)	0.01 (↓0.23)	0.06 (↓0.22)
	10	✗	11.2 (↑2.4)	76.8 (↑2.0)	14.6 (↑2.8)	27.8 (↓5.2)	54.8 (↑5.0)	**6.2** (↓1.8)	0.03 (0.0)	∼0.00 (↓0.24)	0.03 (↓0.24)
	10	✓	11.6 (↑2.0)	87.6 (↓1.2)	13.2 (↑2.4)	40.5 (↓3.4)	**60.4** (↑3.2)	8.6 (↓1.0)	0.05 (↑0.01)	∼0.00 (↓0.24)	0.05 (↓0.23)
Opus 4.6	5	✗	26.8 (↑18.0)	82.4 (↑7.6)	32.5 (↑20.7)	25.0 (↓8.0)	67.8 (↑18.0)	7.6 (↓0.4)	0.03	0.24	0.27
	5	✓	27.4 (↑17.8)	93.0 (↑4.2)	29.5 (↑18.7)	32.7 (↓11.2)	73.8 (↑16.6)	9.0 (↓0.6)	0.04	0.24	0.28

Table 5: Cross-agent-model fallback ablation on SWE-bench Verified (500 tasks). Trained critic "Ours = `Qwen3-8B` SFT'd on concise-prompt teacher critiques (training data collected at k=5). Untrained Qwen3-8B uses the detailed prompt at inference; Ours and Opus use the concise prompt. Quality, trajectory, and step arrows are vs. matching no-critic at the same fallback state; cost arrows vs. Opus k=5 at the same fallback state. Critic $ "—" for no-critic rows. ∼0.00: positive Critic $ rounding to 0.00. **Bold** = best non-Opus per metric per agent; underline = best overall; ties broken by table order. ✓ = fallback to base run 0 (no-critic + ✓ uses run 0 → run 1).

F Inference Pricing

All per-instance dollar costs reported in this paper are computed from input/output token counts using the public hosted-inference rates in Table 6. Qwen-family rates are taken from PricePerToken. `CWM-32B` is not publicly hosted, so we use `Qwen3-32B` as a proxy on the basis that both are dense 32B models. The same `Qwen3-8B` rate is applied to the untrained `Qwen3-8B` critic and to all `Qwen3-8B` SFT variants distilled from it.

Model	Input ($/Mtok)	Output ($/Mtok)	Source
`CWM-32B`	0.080	0.280	Proxy: `Qwen3-32B` (same-size 32B dense)
`Qwen3-32B`	0.080	0.280	PricePerToken[4]
`Qwen3-Next-80B-A3B-Instruct`	0.090	0.780	PricePerToken[5]
`Qwen3-8B` and SFT variants	0.050	0.200	PricePerToken[6]
`Claude-Opus-4.6`	5.500	27.500	API pricing

Table 6: Hosted-inference rates used to convert token counts to the dollar costs in this paper. Opus critic costs include prompt-caching: cache-write tokens billed at $1.25\times$ input ($6.875/Mtok) and cache-read at $0.10\times$ input ($0.55/Mtok); the reported Opus cost combines these.

G Qualitative Case Studies

Verbatim critic excerpts cited in §5.2; ellipses mark omitted rubric boilerplate (§C).

G.1 Untrained vs. trained critic on the same loop

On `django__django-16595` the untrained critic flags only Verification Failures and prescribes a test the agent has not set up; the agent abandons self-correction and submits a corrupted file (Listing 1). On `django__django-9296`, under matched surface conditions (`sed` edits to a Django source file), the trained critic flags Step Repetition and Tool Selection Errors jointly and prescribes a heredoc rewrite (Listing 2); the agent submits a correct patch ten steps later.

Listing 1: Untrained critic at agent step 10 on `django__django-16595`. Only Verification Failures is flagged, with a recovery prescribing a test the agent has not set up.

```
... 12. Verification Failures: DETECTED: Yes
EVIDENCE: "The agent made changes to the reduce method in
django/db/migrations/operations/fields.py without verifying
if the fix works"
RECOVERY_ACTION: "Run the test script to verify that the fix
resolves the issue with multiple AlterField operations not
being reduced"

TASK_STATUS: Needs correction
OVERALL_GUIDANCE: You have modified the reduce method in the
AlterField class ... Run the test script to ensure that the
optimizer now correctly reduces multiple AlterField operations
... If the tests pass, you can submit your changes.
```

Listing 2: Trained critic at agent step 10 on `django__django-9296`. Flags Step Repetition and Tool Selection Errors jointly and tells the agent to switch off `sed` append.

```
3. Step Repetition: DETECTED: Yes
EVIDENCE: Agent has repeatedly applied the same sed append
command, each time creating duplicate `__iter__` methods
inside the `Paginator` class.
RECOVERY_ACTION: Stop using sed append commands and instead
rewrite the file using a heredoc or a different approach ...
6. Tool Selection Errors: DETECTED: Yes
EVIDENCE: Agent is using `sed -i '/def page_range(self):/a'`
... but this creates multiple duplicate method definitions
each time it's run because the pattern matches multiple
times after the first insertion.
RECOVERY_ACTION: Use a different editing strategy: write the
entire file using a heredoc, or use a Python script ...

TASK_STATUS: Critical intervention required
OVERALL_GUIDANCE: The file is now corrupted with multiple
duplicate `__iter__` method definitions. You must view the
file ... and then use a Python script (not sed append) to
properly insert a single `__iter__` method ...
```

G.2 Routing around a dependency failure

On `sympy__sympy-22714` the unguided agent loops on `ModuleNotFoundError: mpmath`; the trained critic at step 20 identifies the import as the root cause and tells the agent to bypass it (Listing 3), and the agent submits a correct patch at step 25.

Listing 3: Trained critic at agent step 20 on `sympy__sympy-22714`.

```
TASK_STATUS: Needs correction
OVERALL_GUIDANCE: The mpmath import in sympy/__init__.py is
preventing any import of sympy. You need to bypass this
import to test, or focus on reading the source code directly
to understand the issue without running code. The key issue
is in `sympy/core/sympify.py` line 472 where `sympify` calls
`parse_expr` with `evaluate=evaluate` ...
```

G.3 Backseat driving in the detailed teacher prompt

The detailed prompt at the `Claude-Opus-4.6` teacher mixes patch dictation and architectural prescription in the same firing slot. Listing 4 is the dictation case on `django__django-12193` (ρ=1.00, resolved): `Claude-Opus-4.6` supplies the before/after blocks for `CheckboxInput.get_context` and the agent transcribes them into a patch over its next three messages. Listing 5 is the wrong-fork case on `sphinx-doc__sphinx-7590` (ρ=0.70, gold overlap 0.12, fails): `Claude-Opus-4.6` hedges between modifying the existing parser path and introducing a new AST class, the agent commits to the latter and writes 46 lines around an `ASTUserDefinedLiteral` class that the gold patch never introduces.

Listing 4: `Claude-Opus-4.6` teacher (detailed prompt) at agent step 6 on `django__django-12193`. The agent's submitted patch reproduces both blocks verbatim.

```
OVERALL_GUIDANCE: The agent has correctly identified the
root cause in `CheckboxInput.get_context()` ... The fix
should be to make a copy of `attrs` before modifying it.
Specifically, change the `get_context` method of
`CheckboxInput` from:

```python
def get_context(self, name, value, attrs):
    if self.check_test(value):
        if attrs is None:
            attrs = {}
        attrs['checked'] = True
    return super().get_context(name, value, attrs)
```

to:

```python
def get_context(self, name, value, attrs):
    if self.check_test(value):
        attrs = {**(attrs or {}), 'checked': True}
    return super().get_context(name, value, attrs)
```
```

Listing 5: `Claude-Opus-4.6` teacher (detailed prompt) at agent step 6 on `sphinx-doc__sphinx-7590`. Bullet 4's "or if a new ... class is needed" is the wrong fork; the agent takes it.

```
3. Implementation approach: After matching a numeric
literal and consuming any standard suffixes (`uUlLfF`),
also try to match an identifier suffix (the ud-suffix)...

4. Check the AST classes - look at `ASTNumberLiteral`,
`ASTStringLiteral`, `ASTCharLiteral` to see if they need
modification, or if a new `ASTUserDefinedLiteral` class
wrapping them is needed.
```

G.4 Where the trained critic still falls short

The resolve gain is uniform across agent models, but on the trajectory-process side stuck-in-loop rate rises by 17.4 points on `CWM-32B` and on `Qwen3-32B` submission rate drops by 2.2 points with stuck-in-loop up 2.0; `Qwen3-Next-80B-A3B` is the exception, with stuck-in-loop falling by 3.0. The `CWM-32B` case is the most pronounced. Stuck-in-loop rises from 13.6% to 31.0% and trajectories grow from 42.1 to 71.7 steps. Without fallback the trained critic produces a non-empty patch on only 350/500 tasks vs. 433/500 for the base agent, and 126 of the 150 non-completing tasks are resolved by the base agent: the trajectories are tractable, the critic derails them. LLM-assisted classification attributes 78% of these stuck runs to critic-caused failures, dominated by *repeated demand* (28.3%): the critic identifies the loop and prescribes the right fix but reissues a near-identical `OVERALL_GUIDANCE` every k steps, which the agent stops acting on until the budget is exhausted (App. G.5). The SFT loss matches teacher tokens at each prefix without penalizing the agent-side outcome, which we discuss further in §6.

G.5 Repeated demand on `django__django-13012`

Listing 6 shows two of the 13 firings on `django__django-13012` to illustrate the structural similarity of the `OVERALL_GUIDANCE` that the agent stops acting on.

Listing 6: Trained critic firings 6 (msg 127) and 10 (msg 211) on `django__django-13012`; both prescribe the same fix path.

```
[firing 6, agent step 60]
TASK_STATUS: Critical intervention required
OVERALL_GUIDANCE: You are stuck in a loop viewing the same
code. The `ExpressionWrapper.get_group_by_cols` calls
`self.get_source_expressions()` which returns
`[self.expression]`, and then calls `source.get_group_by_cols()`
on that expression. When `self.expression` is a `Value`,
`Value.get_group_by_cols` returns `[]`, so the constant
should NOT be in GROUP BY. The real issue is likely in
`query.py`'s `set_group_by` method ...

[firing 10, agent step 100]
TASK_STATUS: Critical intervention required
OVERALL_GUIDANCE: You are stuck in a loop viewing the same
code. Create a minimal reproduction script that only imports
`Value`, `IntegerField`, and `Sum` from `django.db.models`
(no `functions` or `asgiref`), then look at `set_group_by`
in `query.py` (line 1942) to understand how the GROUP BY
clause is populated ...
```