OPERA: A VERBAL CRITIC FRAMEWORK FOR LONG-HORIZON CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon coding agents need timely corrections, yet feedback can be ineffective or even harmful when it misjudges ongoing work or fails to address the underlying problem. Existing critics focus on evaluating trajectories and generating feedback, but rarely track what happens after feedback is delivered. We present Opera, a verbal critic framework that treats each correction as a persistent note, followed until the diagnosed problem is resolved. Opera decides when to review through periodic and event-driven triggers, diagnoses issues with typed operators, audits feedback against visible evidence before delivery, and tracks the agent's subsequent actions to distinguish mere compliance from actual resolution. As a test-time critic, Opera improves the resolve rate of non-critic agents by up to 12.4, 15.0, and 8.9 percentage points on Terminal-Bench 2.1, a SWE-Bench Pro subset, and DeepSWE v1.1, respectively, across four policy models, and achieves the highest mean resolve rate among competitive critic baselines on all three benchmarks, and also improves policy models when the policy critiques itself. Beyond inference, Opera-guided rollouts provide approximately on-policy training data: fine-tuning Qwen3.5-9B on them improves its resolve rate on held-out SWE-Bench Pro repositories by 10.2 percentage points without a critic at inference time, matching fine-tuning on rollouts from a stronger model, while preserving its performance when switching harness, i.e., from Openhands to Terminus-2, which the latter substantially degrades.

1 INTRODUCTION

Coding agents tackle repository-level tasks through code exploration, editing, and testing (Yang et al., 2024a; Wang et al., 2024). Over long horizons, they often repeat unsuccessful actions or explore redundantly (Gandhi et al., 2025), and may report completion before the task is actually verified (Tang et al., 2026). A critic that diagnoses such problems during execution is a natural remedy, but it must judge unfinished work, where incomplete evidence blurs the line between an actual mistake and a reasonable intermediate step.

Prior work provides strong foundations for feedback and reflection, including Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and CRITIC (Gou et al., 2023), and for coding agents, SWE-PRM (Gandhi et al., 2025), SWE-Search (Antoniades et al., 2024), and Agentic Rubrics (Raghavendra et al., 2026). We move further to view critic feedback as an *intervention* whose value is revealed only afterwards, which raises three challenges: ① ***When should the critic intervene?*** Periodic feedback may arrive too late, while not every execution event requires correction. ② ***Is the feedback justified?*** A plausible diagnosis can misread partial evidence and derail work that would otherwise succeed (Vasudev et al., 2026), and LLM judges remain noisy even when verifying well-specified criteria (Peng et al., 2026). ③ ***Did the intervention work?*** An agent may follow a suggestion yet leave the underlying failure intact.

To address the above challenges, we introduce Opera, a verbal critic framework that manages each correction as a persistent note: opened when an issue is found, delivered only when justified, tracked as execution proceeds, and closed once evidence supports its resolution. Opera combines periodic and event-driven triggers with the option to stay silent (①); pairs an operator critic, which ties each issue to evidence, a corrective direction, and a resolution criterion, with an audit that screens feedback before delivery (②); and maintains a finding note that tracks adherence separately from

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

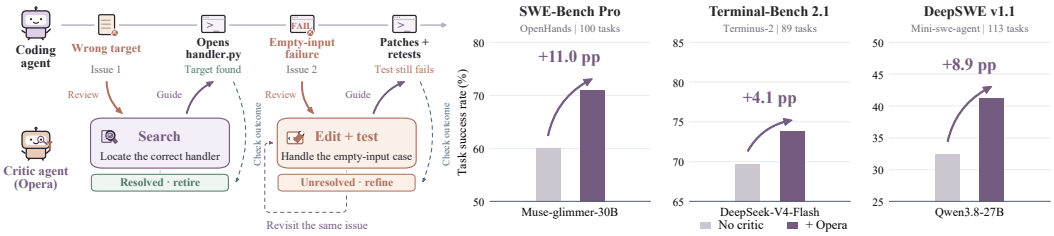


Figure 1: Illustration of Opera. **Left:** Opera discovers and tracks issues at different stages (search, edit, test, etc.) until it is resolved or refined in the long-horizon turns. **Right:** Opera improves task resolve rate across benchmarks, harnesses, and policy models.

resolution (③). Opera interacts with agents only through execution traces and natural-language feedback, so it plugs into existing harnesses without updating agent weights.

At test time, Opera acts as a teacher that guides the agent's self-reflection. Across Terminal-Bench 2.1, a 100-task SWE-Bench Pro subset, and DeepSWE v1.1, Opera outperforms four competitive critic baselines and shows consistent improvement on four policy models with different harnesses. Beyond inference, the same process yields training data that distillation from stronger models lacks: trajectories consisting of the student's own actions, together with targeted diagnoses of its errors.

Our contributions are threefold:

- **Framework.** We formulate verbal criticism as managing persistent corrective notes and introduce Opera, a critic framework that decides when to review, audits feedback before delivery, and follows each correction until it is resolved during long-horizon runs of coding agents.
- **Test-time critic.** Opera improves task resolve rate by up to 12.4, 15.0, and 8.9 pp on Terminal-Bench 2.1, SWE-Bench Pro, and DeepSWE v1.1 across four policy models, and outperforms competitive critic baselines. It improves most policy models with both weak and strong critic models, including the policy model critiquing itself.
- **Training recipe.** Opera-guided student rollouts provide approximately on-policy training data. Fine-tuning Qwen3.5-9B on them matches distillation from a stronger model on held-out SWE-Bench Pro repositories (+10.2 pp) while preserving its Terminal-Bench 2.1 performance.

2 Related Work

2.1 Critics and Verifiers for Coding Agents

Early approaches improve LLM outputs through iterative self-feedback (Madaan et al., 2023), verbal reflection across attempts (Shinn et al., 2023), and tool-grounded critique (Gou et al., 2023). For repository-level coding, SWE-PRM (Gandhi et al., 2025) periodically reviews agent trajectories and provides taxonomy-guided corrective feedback. Agentic Rubrics (Raghavendra et al., 2026) constructs repository-specific criteria for evaluating candidate patches, while SWE-Shepherd (Dihan & Khan, 2026) uses a trained process reward model to guide intermediate action selection. However, RuVerBench (Peng et al., 2026) identifies substantial noise in LLM-based rubric verification, motivating explicit quality control over critic feedback.

2.2 Test-Time Scaling for Coding Agents

Test-time scaling allocates additional inference computation to exploration, evaluation, and refinement. Tree of Thoughts (Yao et al., 2023) explores alternative reasoning states, while Language Agent Tree Search (Zhou et al., 2023) combines tree search with reflection and environment feedback. For coding, SWE-Search (Antoniades et al., 2024) applies Monte Carlo tree search to repository-level tasks, and S* (Li et al., 2025) combines parallel sampling, sequential refinement, and execution-grounded selection. Recent work uses rollout summaries for trajectory selection and reuse (Kim et al., 2026), differential testing for candidate selection (He et al., 2026), and an LLM orchestrator for adaptive solver allocation and answer synthesis (Qin et al., 2026). Opera allocates additional inference computation to diagnosing and correcting an ongoing coding trajectory. Its

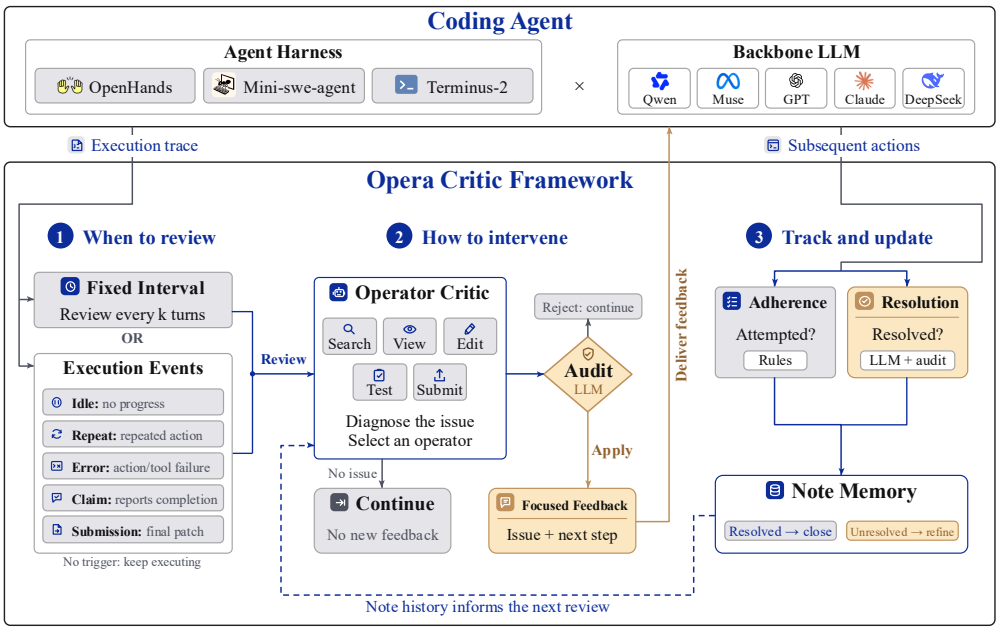


Figure 2: Overview of Opera. ❶ Reviews are triggered at fixed intervals or by execution events. ❷ The operator critic diagnoses one issue with a typed operator, and an audit decides whether the feedback is delivered. ❸ Adherence and resolution are tracked separately; resolved notes are closed, unresolved notes are refined, and note history conditions the next review.

hybrid review schedule and audited interventions connect compute allocation to when feedback is warranted and how it should guide subsequent execution.

3 Opera: Verbal Critic Framework through Persistent Notes

An effective critic must decide when to review, deliver feedback the agent can act on, and verify afterwards that the problem is resolved. Opera organizes supervision around a persistent *note*: one diagnosed issue, its current guidance, and a fixed resolution criterion. Persistence lets the critic follow up on its own verbal feedback across reviews; the fixed criterion ensures that resolution is judged against the original problem rather than the latest guidance. Figure 2 shows how Opera addresses the three challenges from Section 1: when to review (§3.1), how to intervene (§3.2), and how to track and update notes (§3.3).

3.1 Hybrid Review Scheduling

Fixed-interval review reacts slowly: an agent may repeat a failing command several times before the next review. Event-triggered review reacts quickly but misses silent failures, such as confidently editing the wrong file. Opera combines both. Unlike SWE-PRM (Gandhi et al., 2025), which reviews at fixed intervals, Opera performs a periodic review every k turns plus immediate reviews on five events: *Idle* (no visible progress), *Repeat* (repeated actions), *Error* (failed actions or tool calls), *Claim* (reported completion), and *Submission* (final patch). Simultaneous signals produce a single review, and an event-triggered review resets the periodic timer. A review does not imply an intervention: the critic may let the agent continue without feedback.

3.2 Operator-Typed Review

Free-form critique is often generic (e.g., "consider edge cases") or mixes several concerns, leaving the agent little to act on. Opera instead requires each diagnosis to use a typed *operator*. The operators follow the stages of code repair, namely localization, inspection, editing, and verification (Xia et al., 2024; Bouzenia et al., 2024), and cover common failure types such as misunderstood requirements, incorrect edit scope, and faulty logic (Tang et al., 2026). Nine operators span five stages (Figure 3a):

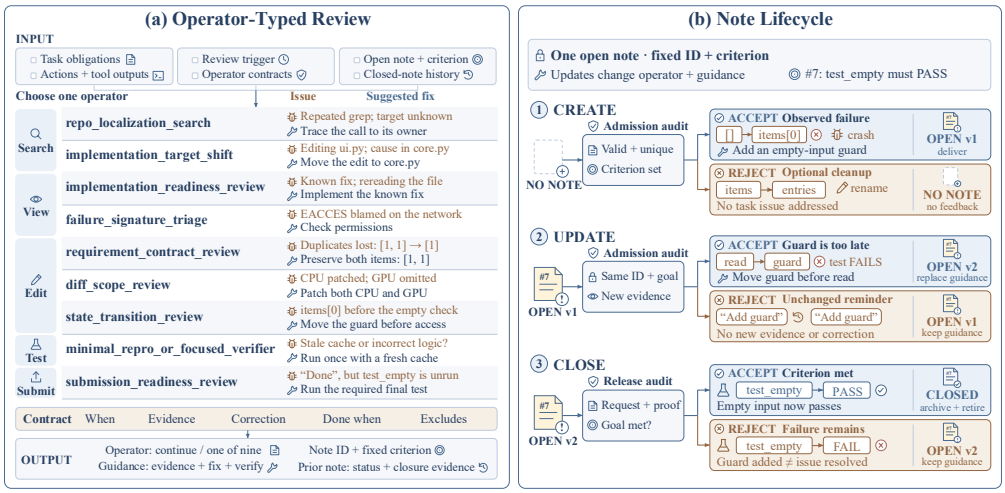


Figure 3: Core mechanisms of Opera. (a) Nine operators, grouped by repair stage and specified by contracts, map an observed issue to a targeted fix. (b) Admission audits gate note creation and updates, rejecting unsupported or redundant feedback; the release audit closes a note only when its fixed criterion is met.

- **Search**: *repo_localization_search* (target not yet identified); *implementation_target_shift* (editing the wrong target).
- **View**: *implementation_readiness_review* (re-inspecting code when the fix is already clear); *failure_signature_triage* (misread tool output).
- **Edit**: *requirement_contract_review* (requirement violated); *diff_scope_review* (missing or unrelated changes); *state_transition_review* (faulty execution logic).
- **Test**: *minimal_repro_or_focused_verifier* (missing evidence needed for the next decision).
- **Submit**: *submission_readiness_review* (completion claimed without sufficient evidence).

All operators remain available throughout execution, since agents often revisit earlier stages. Following Design by Contract (Meyer, 1992), each operator is specified by a contract stating when it applies, the evidence it requires, the correction it may request, when the issue counts as resolved, and which problems it excludes. The exclusion clause keeps operators from overlapping, and the resolution clause supplies the criterion against which the note is later checked.

Selection and output. Given the task, execution history, review trigger, and current note, the critic either returns *continue* when the agent is making useful progress, since unnecessary interventions can disrupt trajectories that would otherwise succeed (Vasudev et al., 2026), or selects one supported issue and the smallest correction for it. Restricting each review to one issue avoids competing instructions and keeps later outcomes attributable to a specific correction. Each proposal names an operator, cites evidence, and specifies the location, the current behavior, the required change, and how to verify it. Independently, the critic reports whether the open note remains unresolved, so follow-up continues even without new guidance.

3.3 Audited Note Management

Inspired by reflection memory and persistent goals (Shinn et al., 2023; Cohen & Levesque, 1990), each note is either *open* or *closed*, with at most one open note at a time (Figure 3b). This keeps the agent focused on a single correction and makes outcomes attributable. Because the critic is itself an LLM whose diagnoses may be plausible but wrong, two audits govern the note lifecycle. Both only accept or reject proposals without rewriting them, acting as verifiers rather than co-authors.

Opening and updating. The critic may open a note when none is open, or update the open note when new evidence changes the diagnosis or suggests a clearer next step. Updates may change the operator and guidance but preserve the note's identity and criterion, so the goal stays fixed while the advice improves. After validity and duplicate checks, the audit verifies that the evidence supports the diagnosis, the correction is necessary, and the proposed check can establish resolution (Vasudev

et al., 2026). It rejects, for example, optional cleanups unrelated to the task and reminders that repeat guidance without new evidence. Rejected proposals leave the current note unchanged.

Tracking and closing. Opera tracks two distinct questions after delivery. *Adherence*, checked by rules, asks whether the agent attempted the correction. *Resolution*, judged against the fixed criterion, asks whether the original problem is gone. They are separated because adherence does not imply resolution: in Figure 3b, the agent adds the suggested guard, yet the required test still fails. Closing a note requires a *release audit* with supporting evidence, namely execution results for behavioral requirements or visible edits for static ones. Accepted closure archives the note and removes its guidance from the agent's context. Closure is processed before admission, so one review can close an issue and open the next.

4 Evaluation

4.1 Benchmarks and Setup

We evaluate on Terminal-Bench 2.1 (89 tasks) (Terminal-Bench Team, 2026) using Terminus-2, SWE-Bench Pro (100 tasks) (Deng et al., 2025) using OpenHands (Wang et al., 2024), and DeepSWE v1.1 (113 tasks) (Huang et al., 2026) using mini-swe-agent Yang et al. (2024a). For SWE-Bench Pro, we apply task corrections informed by OpenAI's benchmark audit (OpenAI, 2026a) and sample 100 tasks from the corrected set and cover all of the repos included in the raw swebench-pro. Across all three benchmarks, we employ Harbor (Harbor Framework Team, 2026) to manages tasks, and Pier (Datacurve, 2026) to provides network isolation for anti-cheating. For policy model, we use Qwen3.5-9B (thinking) (Qwen Team, 2026a), Muse-glimmer-30B (high) (Meta Superintelligence Lab, 2026), Qwen3.8-27B (xhigh) (Qwen Team, 2026b) and Deepseek-V4-Flash-0731 (max) (DeepSeek-AI, 2026) for our evaluations. We use GPT-5.6-Sol (high) (OpenAI, 2026b) as the default critic model for all of our experiments if not specifically mentioned, results of using other models are also reported in Section 5.3. For timeouts, we employ the task-specific timeouts configured in Harbor's official dataset cards [1].

4.2 Overall Task Performance

Comparison with critic baselines. We compare Opera with the non-critic agent and four representative critics, all using Qwen3.8-27B as the policy model (Table 1). They fall into two groups by what they judge and what signal they return. *Process critics* evaluate the ongoing trajectory: SWE-PRM (Gandhi et al., 2025) periodically reviews recent steps and returns taxonomy-guided verbal feedback, and SWE-Search (Antoniades et al., 2024) scores the last action with a numerical value and a natural-language assessment, originally used to guide tree search. *Outcome verifiers* score the agent's solution: LLM-as-verifier (Kwok et al., 2026) assigns fine-grained scalar scores, and Agentic Rubrics (Raghavendra et al., 2026) scores patches against a repository-grounded rubric checklist without executing tests. For *Process Critics*, we adapt them to use the same critic schedule as Opera to ensure a fair comparison, more implementation details are included in Appendix A. Table 1 shows that Opera achieves the highest mean resolve rate on all three benchmarks. Its margin over the strongest baseline is 1.9 pp on Terminal-Bench 2.1 and 1.8 pp on DeepSWE, but only 0.6 pp on SWE-Bench Pro, where the non-critic agent is already strong and all critics yield similar, modest gains; This improvement suggests the the advantages of Opera's design on the two aspects: it inter-

Table 1: Comparison with critic baselines (policy: Qwen3.8-27B). Resolve rate (%, mean ± std) on Terminal-Bench 2.1 (TB-2.1, 89 tasks), a 100-task subset of SWE-Bench Pro (SWE-Pro), and DeepSWE v1.1 (113 tasks). Green bold / blue underline: best / second-best.

Method	TB-2.1	SWE-Pro	DeepSWE
Non-critic	65.9 ± 1.7	75.3 ± 1.2	32.4 ± 1.0
SWE-PRM	68.9 ± 3.2	77.7 ± 2.5	36.6 ± 3.1
SWE-Search	70.0 ± 4.7	77.3 ± 3.8	38.6 ± 4.4
LLM-as-verifier	71.2 ± 2.6	<u>78.7</u> ± 2.3	38.9 ± 2.3
Agentic Rubrics	<u>71.9</u> ± 4.5	78.3 ± 3.2	<u>39.5</u> ± 2.8
Opera (ours)	**73.8** ± 2.3	**79.3** ± 3.5	**41.3** ± 3.1
Gain (pp)	+7.9	+4.0	+8.9

[1] https://hub.harborframework.com/datasets

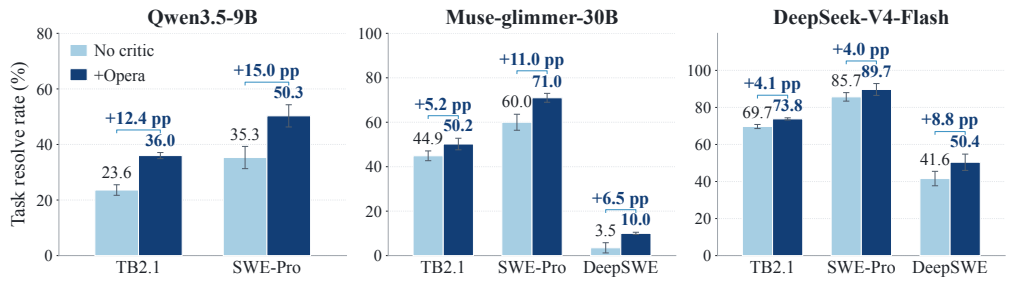


Figure 4: Task resolve rate of Opera across policy models (critic: GPT-5.6-Sol). Gains (pp) are generally larger for weaker backbones and on benchmarks with more headroom. Qwen3.5-9B resolves no DeepSWE task with or without the critic. Error bars denote sample standard deviation.

Table 2: Task distribution of the SFT training split and the held-out OOD evaluation split of SWE-Bench Pro. TS/JS: TypeScript/JavaScript.

Split	Repositories	Py	Go	TS	JS	Total
Train	qutebrowser, ansible, teleport, flipt, vuls, webclients, NodeBB	99	115	32	24	270
Held-out	openlibrary, navidrome, element-web, tutanota	47	30	31	0	108

Table 3: Resolve rate (%) of Qwen3.5-9B trained on different trajectory sources, evaluated without a critic on held-out SWE-Bench Pro tasks and Terminal-Bench 2.1. Mean $\pm$ std over three runs.

Training data	SWE-Pro	TB2.1
None (base)	28.7 ± 1.9	23.6 ± 1.9
Qwen3.8-27B rollouts	$\mathbf{39.5 \pm 1.4}$	7.9 ± 1.9
Opera-guided (ours)	38.9 ± 2.4	$\mathbf{25.8 \pm 2.2}$

venes during execution like a process critic, while each note carries an explicit resolution criterion like the output verifier before closure.

Performance on different policy models. Figure 4 shows that Opera improves every policy model on every benchmark, with one exception: Qwen3.5-9B resolves no DeepSWE task, with or without the critic. The gains are shaped by how much room the agent leaves for correction. Weaker agents generally benefit more: Qwen3.5-9B gains 12.4 and 15.0 pp on Terminal-Bench 2.1 and SWE-Bench Pro, and gains tend to shrink as the non-critic resolve rate rises. Headroom matters more than model strength alone: DeepSeek-V4-Flash gains little on benchmarks where it is already strong, but substantially on the harder DeepSWE. Yet a critic cannot substitute for missing capability. DeepSWE consists of original, long-horizon engineering tasks, and Qwen3.5-9B resolves no DeepSWE task in the evaluated runs, with or without the tested critics. Muse Glimmer-30B on DeepSWE shows a milder version of this effect: starting from a low base, feedback nearly triples its resolve rate, but the absolute gain remains limited. A critic is therefore most valuable when the agent is capable enough to act on feedback but still makes errors it cannot correct on its own.

4.3 Recipes for Training Open-weight Coding Agents

Beyond inference-time assistance, Opera can generate training data in which the agent receives targeted diagnoses and takes corrective actions. We ask whether such data can be used to train Qwen3.5-9B to learn from error correction without a critic at inference time.

Setup. We split SWE-Bench Pro by repository into 270 training tasks and 108 held-out tasks (Table 2), and additionally evaluate on Terminal-Bench 2.1, which differs from the training data in task format and harness. We compare two trajectory sources: *Opera-guided student rollouts*, with Qwen3.5-9B as the policy and Qwen3.8-27B as the critic, and *teacher rollouts*, with Qwen3.8-27B as the policy. The same model thus supplies the expertise in both settings, either by critiquing or by acting. We select 67 training tasks on which the base Qwen3.5-9B rarely succeeds but both sources contain at least one successful trajectory, so that both sources supervise the same tasks (selection criteria and more training details are in Appendix B.1).

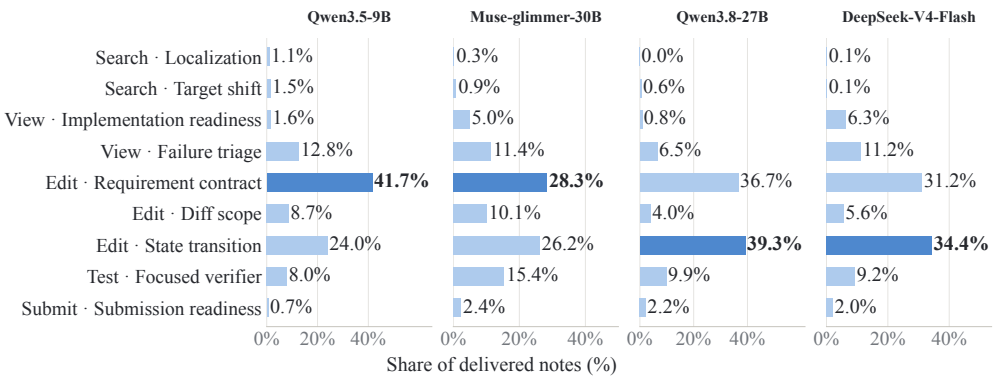


Figure 5: Distribution of admitted operators. Edit-stage operators dominate while the leading one shifts from requirement contract to state transition as model capability increases.

Results and analysis. Both sources improve the held-out SWE-Bench Pro resolve rate by about 10 pp, with no meaningful difference between them (Table 3). On Terminal-Bench 2.1, however, fine-tuning on teacher rollouts reduces the resolve rate from 23.6% to 7.9%, whereas Opera-guided rollouts preserve it (25.8%). Both sources thus improve the target task similarly, but only the Opera-guided data does so without degrading performance on a benchmark with a different task format and harness. We attribute this mainly to whose actions the data contain. Teacher rollouts are off-policy for the student, and fitting them shifts the student toward the teacher's distribution; forgetting is known to grow with such shift (Shenfeld et al., 2026), and fine-tuning on self-generated data mitigates it (Yang et al., 2024b). Opera-guided rollouts are approximately on-policy: every action comes from the student, and the critic's influence enters only through notes in its context. Such data has been shown to underlie RL's robustness to forgetting (Chen et al., 2025). The recipe also follows the principle of DAgger (Ross et al., 2011) and on-policy distillation (Agarwal et al., 2024): the student receives expert corrections at the states it actually visits, which teacher trajectories rarely contain, since the teacher seldom makes the student's mistakes.

Section 4.3 Takeaways

- A critic can turn a stronger model's expertise into approximately on-policy self-reflection data for a weaker student.
- Fine-tuning on critic-guided student trajectories matches distillation from the stronger model on held-out repositories, while preserving generalization capabilities better.

5 Analysis

5.1 Critic Behavior and Utility

We analyze which operators the critic admits for each policy model, how agents respond to them, and how these responses affect task outcomes. Operator frequencies reflect the critic's admitted diagnoses rather than all agent errors.

The critic mostly diagnoses implementation errors, and the dominant type is model-specific. Across all four models, Edit-stage operators account for most delivered notes, while Search and Submit operators are rare (Figure 5). This suggests that, in the trajectories the critic flags. Qwen3.5-9B mostly receives requirement-contract notes, suggesting that it often misreads what the task requires. The stronger Qwen3.8-27B and DeepSeek-V4-Flash shift toward state-transition notes, suggesting that they understand the goal but get the execution logic wrong. Muse Glimmer-30B shows a flatter distribution with more focused-verifier notes, pointing to weaker verification habits.

Stronger models gain more rescues but also more regressions. On trials receiving notes, Opera improves over the expected no-critic success for all models (Figure 6a). The rescue–regression breakdown, however, reveals a trade-off (Figure 6b), consistent with the recovery–disruption trade-

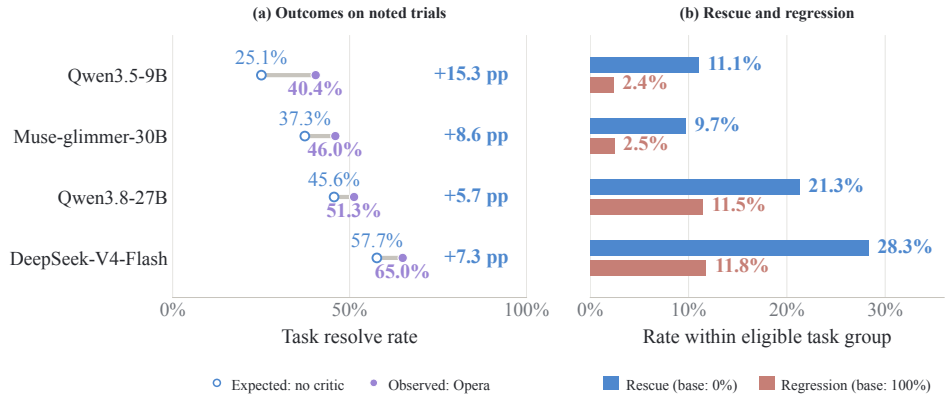


Figure 6: Task outcomes on trials receiving notes. (a) Observed Opera success versus expected no-critic success on the same tasks. (b) Rescue rate among tasks never solved without the critic and regression rate among tasks always solved without it; the two rates use different denominators.

off reported by Vasudev et al. (2026). Stronger models convert feedback into more rescues of previously unsolved tasks, but also regress more often on tasks they would otherwise solve, while weaker models rarely regress but are rescued less often. One possible explanation is that stronger models act on feedback more readily (Figure 9), which amplifies both correct and unnecessary interventions; for these models, the cost of an unneeded correction is therefore higher.

Section 5.1 Takeaways

- The critic mostly flags how agents change code rather than where they look.
- As policy capability increases, the most frequent diagnosis shifts from misread requirements to faulty execution logic.
- A critic's benefit is a rescue–regression trade-off, and the stronger policy models we tested rank high on both.

5.2 Operator Ablation

To further examine the role of operators, we note that Edit-stage operators account for most delivered notes, e.g., about 80% for Qwen3.8-27B (Figure 5). We compare two restricted variants that keep all other components of Opera unchanged: *Edit-only*, which retains only the Edit-stage operators, and *Non-edit*, which retains only the Search, View, Test, and Submit operators (Table 4). The full operator set achieves the highest mean on all three benchmarks, although its margin over Edit-only is small (e.g., 0.3 pp on SWE-Bench Pro) and within run-to-run variance. Both restricted variants retain most of the full gain. Notably, Non-edit operators account for only about one fifth of delivered notes but recover over half of the full gain on Terminal-Bench 2.1, three quarters on SWE-Bench Pro, and about four fifths on DeepSWE. The gains of the two variants are also not additive, as each alone recovers much of the full gain. One possible explanation is that the critic can often address the same underlying problem through different operators, so that the review and follow-up process compensates for a restricted operator set.

Table 4: Operator ablation with Qwen3.8-27B. Resolve rate (%), mean ± std over three runs; parentheses show gains over non-critic (pp).

Variant	TB2.1	SWE-Pro	DeepSWE
Non-critic	65.9 ± 1.7	75.3 ± 1.2	32.4 ± 1.0
Opera (full)	**73.8 ± 2.3** (+7.9)	**79.3 ± 3.5** (+4.0)	**41.3 ± 3.1** (+8.9)
Edit-only	71.9 ± 1.6 (+6.0)	79.0 ± 3.5 (+3.7)	40.1 ± 1.8 (+7.7)
Non-edit	70.2 ± 0.8 (+4.3)	78.3 ± 0.6 (+3.0)	39.5 ± 2.7 (+7.1)

Section 5.2 Takeaways

- Both restricted operator sets retain substantial gains, while the full set achieves the best.

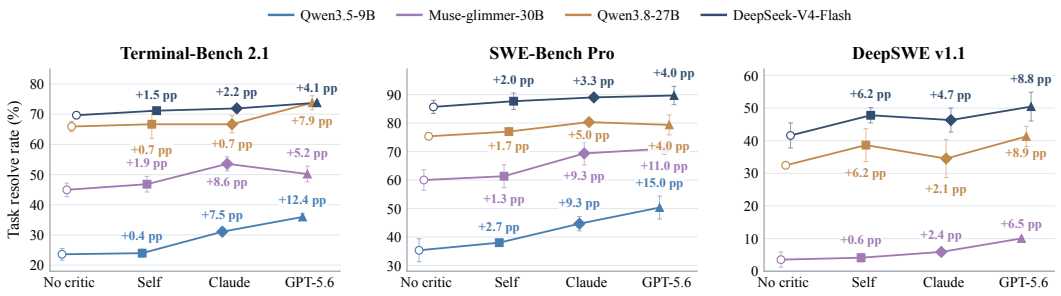


Figure 7: Task resolve rate with different critic models: the policy model itself (Self), Claude, and GPT. Opera has consistent improvement regardless of the critic. Self-critique is effective where the policy is competent, while stronger critics bring large gains where the policy struggles.

5.3 Sensitivity to Critic Model Choice

We vary the critic model while keeping the policy fixed, using the policy model itself (Self), Claude, or GPT as the critic (Figure 7). We exclude Qwen3.5-9B on DeepSWE, where the policy resolves no task under any critic (Section 4.1). Opera is robust to the choice of critic: all three critics improve over the no-critic agent in all but one of the remaining 33 policy–benchmark–critic combinations. Stronger critics generally yield larger gains, but no single critic is best in every setting. Notably, a policy model can serve as an effective critic of itself. On DeepSWE, self-critique improves Qwen3.8-27B and DeepSeek-V4-Flash by 6.2 pp each, matching or exceeding the gains from Claude. Since the self-critic has no knowledge beyond the policy's own, these gains suggest that much of the benefit comes from the framework: timely review, typed diagnosis, and follow-up help the model apply knowledge it already has but fails to use during execution. Self-critique does add inference compute, however, which this comparison does not match. Self-critique is less effective when the policy lacks the capability to solve the task, as for Muse Glimmer-30B on DeepSWE or Qwen3.5-9B on Terminal-Bench 2.1, likely because the critic shares the policy's blind spots. In these low-capability settings, a stronger critic brings substantial gains. With GPT as the critic, Qwen3.5-9B improves at least 12.4 pp, whereas self-critique yields at most 2.7 pp. Similarly, Muse Glimmer-30B gains 11.0 pp on SWE-Bench Pro and nearly triples its resolve rate on DeepSWE.

Section 5.3 Takeaways

- Opera shows consistent improvement when critic model varies.
- Use self-critique when the policy is competent; use a stronger critic when it struggles; no critic helps a policy that never reaches a viable solution.

5.4 Role of Audit

Due to the space limit, we report the effects of removing audit in Appendix C.

6 Conclusion

We presented Opera, a verbal critic framework for long-horizon coding agents that treats each correction as a persistent note and follows it until the diagnosed problem is resolved. As a test-time critic, Opera improves the resolve rate of non-critic agents by up to 15.0 percentage points across three benchmarks and four policy models, and achieves the highest mean resolve rate among competitive critic baselines with Qwen3.8-27B as the policy. Beyond inference, Opera-guided rollouts provide approximately on-policy training data: fine-tuning Qwen3.5-9B on them matches distillation from a stronger model on held-out repositories (+10.2 pp) while preserving its Terminal-Bench 2.1 performance. Our training results use supervised fine-tuning on a single model with one out-of-domain benchmark; scaling this recipe to larger datasets, iterating it with the trained model as the new policy, or combining critic feedback with reinforcement learning to internalize self-correction could be the future work.

7 AI USE STATEMENT

In this work, we used generative AI tools to assist with code implementation, literature review, and editing of the manuscript text. All research ideas, method design, experimental analysis, and final claims are the authors' own, and all AI-assisted content was reviewed and verified by the authors.

ETHICS STATEMENT

All datasets used in this work are publicly available and contain no sensitive information. Our method applies LLM in coding scenarios. While stronger reasoning ability may be misused, we emphasize that this work is intended for beneficial research and responsible applications.

REPRODUCIBILITY STATEMENT

We will release our code and data once the paper is published. The appendix includes detailed experimental setups and hyperparameters so that others can reproduce our results. We also encourage the community to follow good research practices when using our code and data, to help maintain the reliability and transparency of future work.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations (ICLR)*, 2024.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-Search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*, 2024.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.

Howard Chen, Noam Razin, Karthik Narasimhan, and Danqi Chen. Retaining by doing: The role of on-policy data in mitigating forgetting. *arXiv preprint arXiv:2510.18874*, 2025.

Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2–3):213–261, 1990.

Datacurve. Pier: A Harbor-compatible framework for evaluating coding agents, 2026. URL `https://github.com/datacurve-ai/pier`.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL `https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash`.

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.

Mahir Labib Dihan and Md Ashrafur Rahman Khan. SWE-Shepherd: Advancing PRMs for reinforcing code agents. *arXiv preprint arXiv:2604.10493*, 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.

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

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026.

Yifeng He, Ethan Wang, Jicheng Wang, Xuanxin Ouyang, and Hao Chen. Code generation by differential test time scaling. *arXiv preprint arXiv:2605.20473*, 2026.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

Joongwon Kim, Wannan Yang, Kelvin Niu, Hongming Zhang, Yun Zhu, Eryk Helenowski, Ruan Silva, Zhengxing Chen, Srinivasan Iyer, Manzil Zaheer, Daniel Fried, Hannaneh Hajishirzi, Sanjeev Arora, Gabriel Synnaeve, Ruslan Salakhutdinov, and Anirudh Goyal. Scaling test-time compute for agentic coding. *arXiv preprint arXiv:2604.16529*, 2026.

Jacky Kwok, Shulu Li, Pranav Atreya, Yuejiang Liu, Yixing Jiang, Chelsea Finn, Marco Pavone, Ion Stoica, and Azalia Mirhoseini. Llm-as-a-verifier: A general-purpose verification framework. *arXiv preprint arXiv:2607.05391*, 2026.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382*, 2025.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Meta Superintelligence Lab. Muse Glimmer model card, August 2026. URL `https://huggingface.co/meta-models/Muse-Glimmer-30B`.

Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

OpenAI. Separating signal from noise in coding evaluations, 2026a. URL `https://openai.com/index/separating-signal-from-noise-coding-evaluations/`.

OpenAI. GPT-5.6 Sol model, 2026b. URL `https://developers.openai.com/api/docs/models/gpt-5.6-sol`.

Yangda Peng, Yunjia Qi, Haotian Xia, Guanzhong He, Xintong Shi, Richeng Xuan, Songyuanyi Lu, Yixian Liu, Zhichao Hu, Yuhong Liu, and Hao Peng. Can LLM-as-a-Judge reliably verify rubrics in agentic scenarios? *arXiv preprint arXiv:2606.29920*, 2026.

Peijia Qin, Qi Cao, and Pengtao Xie. ATLAS: Agentic test-time learning-to-allocate scaling. *arXiv preprint arXiv:2606.01667*, 2026.

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

Qwen Team. Qwen3.8-27b, August 2026b. URL `https://huggingface.co/Qwen/Qwen3.8-27B`.

Mohit Raghavendra, Anisha Gunjal, Bing Liu, and Yunzhong He. Agentic rubrics as contextual verifiers for SWE agents. *arXiv preprint arXiv:2601.04171*, 2026.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.

Idan Shenfeld, Jyothish Pari, and Pulkit Agrawal. RL's razor: Why online reinforcement learning forgets less. In *International Conference on Learning Representations (ICLR)*, 2026.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

Ningzhi Tang, Chaoran Chen, Gelei Xu, Yiyu Shi, Yu Huang, Collin McMillan, Tao Dong, and Toby Jia-Jun Li. How coding agents fail their users: A large-scale analysis of developer-agent misalignment in 20,574 real-world sessions. *arXiv preprint arXiv:2605.29442*, 2026. URL https://arxiv.org/abs/2605.29442.

Terminal-Bench Team. Terminal-Bench 2.1, 2026. URL https://www.tbench.ai/news/terminal-bench-2-1.

Rakshith Vasudev, Melisa Russak, Dan Bikel, and Waseem Alshikh. Accurate failure prediction in agents does not imply effective failure prevention. *arXiv preprint arXiv:2602.03338*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

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. *arXiv preprint arXiv:2405.15793*, 2024a.

Zhaorui Yang, Tianyu Pang, Haozhe Feng, Han Wang, Wei Chen, Minfeng Zhu, and Qian Liu. Self-distillation bridges distribution gap in language model fine-tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 1028–1043, 2024b.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

CONTENTS

1 Introduction **1**

2 Related Work **2**

2.1 Critics and Verifiers for Coding Agents 2

2.2 Test-Time Scaling for Coding Agents 2

3 Opera: Verbal Critic Framework through Persistent Notes **3**

3.1 Hybrid Review Scheduling 3

3.2 Operator-Typed Review 3

3.3 Audited Note Management 4

4 Evaluation **5**

4.1 Benchmarks and Setup 5

4.2 Overall Task Performance 5

4.3 Recipes for Training Open-weight Coding Agents 6

5 Analysis **7**

5.1 Critic Behavior and Utility 7

5.2 Operator ablation 8

5.3 Sensitivity to Critic Model Choice 9

5.4 Role of Audit 9

6 Conclusion **9**

7 AI Use Statement **10**

A Implementation details of Opera **14**

A.1 Baselines and Adaptations 14

B More details of training recipe **15**

B.1 Data Preprocessing 15

B.2 Training details 15

C Role of Audit **15**

D Additional analysis of operator behavior and utility **16**

D.1 Behavior responses to the different operators 16

D.2 Critic behavior of using different critic models 16

E Case studies **18**

A IMPLEMENTATION DETAILS OF OPERA

Opera is deployed as a proxy between the agent and its model endpoint. From the agent's perspective, nothing changes: the harness sends each request as it would to a model server, and the proxy forwards it. This design lets Opera work with any harness without modification, and confines the critic to a single channel of influence, namely inserting at most one message into a forwarded request. The critic itself never executes commands, accesses files, or runs tests. It sees only what the agent has sent, and if it fails for any reason, such as a timeout or an unparseable reply, the request is forwarded unchanged, so the critic can never block the agent.

Hybrid Review: Most requests pass through without review. A review is triggered when the agent attempts to finish, when it appears stuck after three repeated or three consecutive failed commands, or when it has taken six work actions without a successful edit; otherwise, the critic checks in every 5 agent turns (10 on the longer DeepSWE tasks), starting from turn 3 and at least 2 turns apart. At each review, the critic reads the task and the rollout so far, presented as untrusted data and bounded to its context window: the 30 newest messages are kept verbatim, long messages are shortened to their beginning and end, and older messages are elided. Unlike a stateless judge, the critic also sees the currently open note, its resolution criterion, and the history of earlier notes, which allows it to follow up on its own feedback rather than start afresh.

Audited Note Management: A proposed note must pass several checks before it reaches the agent. While a note is open, the critic may only update it or let the agent continue, so that corrections do not pile up. Notes that refer to hidden tests or evaluation material, or that repeat the same suggestion within 6 turns, are discarded. A readiness note is also withheld while the agent is still exploring unfamiliar code, measured as more than half of its recent observations covering new ground. A candidate that survives these checks goes to the admission audit, a separate call to the critic model that either accepts it or turns it into *continue*; the audit never rewrites the proposal, and no confidence threshold is involved. Once accepted, a note is delivered as a single message stating the operator, the guidance, and the resolution criterion, and asking the agent to fold the correction into its plan rather than reset its work. The message is inserted before the agent's next turn and stays at that position in later requests, so the agent sees one piece of advice it received a few turns ago instead of a fresh instruction at every step. If the note is delivered when the agent tries to finish, it takes the place of the finish attempt. As the agent continues, later reviews may refine the guidance while keeping the criterion fixed, or propose closing the note; closure takes effect only if the release audit finds the criterion met on the visible evidence, at which point the message disappears from the agent's context. Every review, together with its trigger, decision, filter and audit verdicts, and delivery outcome.

A.1 BASELINES AND ADAPTATIONS

All baselines run in the same harness as Opera: the critic sits behind the agent's model endpoint, sees the same bounded rollout and task statement. For process critic baselines, they share the same hybrid review with Opera.

SWE-PRM (Gandhi et al., 2025). We reproduce it as: at each review, the critic sees the task and the eight most recent steps, checks them against the paper's taxonomy of twelve trajectory inefficiencies, and delivers its overall guidance unless it judges the task on track. Unlike the original, which is invoked only periodically, it also reviews at the shared event triggers.

SWE-Search (Antoniades et al., 2024). We reproduce its value function, including the prompt, the -100 to 100 reward scale, and the evaluation of the last executed action. Since a single trajectory has no search tree, the feedback that SWE-Search writes for an alternative branch is delivered to the current trajectory when the reward is negative.

LLM-as-verifier (Kwok et al., 2026) and Agentic Rubrics (Raghavendra et al., 2026). Both methods were designed to score finished candidates for test-time selection. We run them as in-flight outcome checks at readiness events, i.e., when the agent attempts to finish or idles after its last edit, and deliver their own statement of what is missing when the candidate falls below the acceptance threshold. LLM-as-verifier scores the candidate on a 1–20 scale and accepts scores of at least 14. The original method takes the expectation over the score token's log-probabilities; we do so when the serving API exposes them (vLLM-served critics) and otherwise use the sampled score, which is

Table 5: Training configuration. Shared settings apply to both trajectory sources; per-source statistics are listed at the bottom.

Shared settings		
Student model	Qwen3.5-9B, full-parameter, bf16	
Framework	ms-swift, DeepSpeed ZeRO-3	
Maximum sequence length	128k tokens	
Loss	Target turn only	
Optimizer	AdamW, β_2=0.95, wd 0.1, clip 1.0	
Learning rate	2×10^{-6}, cosine, 3% warm-up	
Per-device batch / accumulation	1 / 8	
Epochs / seed	3 / 42	
Per source	Opera-guided	Qwen3.8-27B
Tasks	67	67
Training samples	8,378	3,476
Supervised tokens	2.69M	2.25M
GPUs (H100)	8	4
Effective batch size	64	32
Optimizer steps	393	326

the case for the GPT-5.6 critic in Table 1. Agentic Rubrics writes up to eight repository-grounded rubric items once per task, scores the candidate item by item, and delivers failed required items.

B More details of training recipe

B.1 Data Preprocessing

Training tasks are drawn from the 270 training-split tasks of SWE-Bench Pro, and no repo from the 108 held-out tasks is used. We select tasks where error recovery matters and both sources can supply successful demonstrations: the base Qwen3.5-9B resolves the task at most once in three runs without a critic, at least one critic-guided run of Qwen3.5-9B resolves it, and Qwen3.8-27B resolves it without a critic. This yields 67 tasks shared by both sources. For the Opera-guided source, we keep every resolved trajectory of Qwen3.5-9B on these tasks from critic runs, with medium reasoning effort and for the teacher source, we run Qwen3.8-27B at medium reasoning effort without a critic. Each agent turn becomes one sample: the context contains the system prompt, the task, and all earlier actions and observations, and the target is the turn's reasoning, visible text, and tool call. A turn with multiple tool calls is split into single-call samples. We remove turns with invalid or rejected tool calls when the agent immediately retried, and drop samples longer than 128k tokens rather than truncating them. Since the trained model runs without a critic, critic notes are removed from all contexts. On the 4.9% of turns where an audited note was delivered, its diagnosis is instead rewritten as the model's own reasoning and prepended to that turn's thought, so the student learns to perform the diagnosis itself rather than wait for it. Sentences in which the policy refers to the critic are removed from both targets and histories, and samples whose commands or observations still mention it are dropped.

B.2 Training details

Table 5 shows the hyperparameters we use during the SFT training and Figure 8 shows the training dynamics (loss and gradient) in the two settings.

C Role of Audit

To isolate the role of the audits, we remove both the admission and release audits while keeping all other components of Opera unchanged, so that every note proposed by the critic is delivered and every proposed closure is accepted (Table 6). Removing the audits reduces the gain on all three

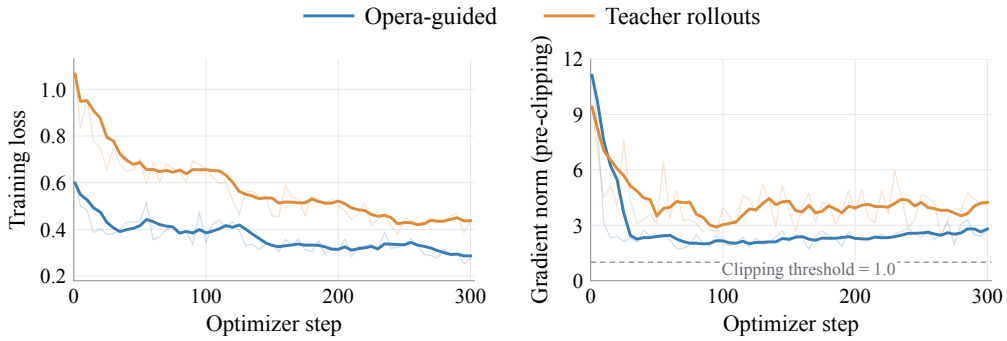


Figure 8: Training dynamics in SFT using Opera guided self-reflection trajectories and Qwen3.8-27B trajectories.

Table 6: Audit ablation with Qwen3.8-27B as the policy and GPT-5.6-Sol as the critic. Resolve rate (%), mean ± std over three runs; parentheses show gains over the non-critic agent (pp). Bold: best mean per benchmark.

Variant	TB2.1	SWE-Pro	DeepSWE
Non-critic	65.9 ± 1.7	75.3 ± 1.2	32.4 ± 1.0
Opera (w/ audit)	$\mathbf{73.8 \pm 2.3}$ (+7.9)	$\mathbf{79.3 \pm 3.5}$ (+4.0)	$\mathbf{41.3 \pm 3.1}$ (+8.9)
Opera (w/o audit)	68.5 ± 2.2 (+2.6)	78.3 ± 2.1 (+3.0)	36.9 ± 1.8 (+4.5)

benchmarks. The drop is largest on Terminal-Bench 2.1, where the gain over the non-critic agent falls from 7.9 to 2.6 pp, losing about two thirds of the improvement, and on DeepSWE, where it falls from 8.9 to 4.5 pp, about half. The results suggest that the audit acts as a filter against these harmful interventions, which matters much on benchmarks where intermediate evidence is easy to misread and misguided corrections are costly.

D Additional analysis of operator behavior and utility

D.1 Behavior responses to the different operators

Following feedback is not the same as resolving the issue. Stronger models act on feedback almost immediately and close most notes (Figure 9). Qwen3.5-9B edits after feedback as often as Muse-glimmer-30B but resolves only about half of its requirement-contract and state-transition notes, whereas Muse-glimmer-30B rarely edits right away yet eventually resolves most issues. Immediate adherence is thus a poor proxy for correction. Across models, agents seldom test unprompted, with the focused-verifier operator being the main trigger for testing, and failure-triage notes have the lowest edit and closure rates.

Diagnoses that name a code change pay off; reinterpreting an error rarely does. Grouped by the first admitted operator, all three Edit-stage operators yield positive gains for every model (Figure 10). Failure triage helps only Qwen3.5-9B and yields slightly negative gains for the other models, mirroring its low closure rates: correcting how an agent reads an error does not by itself tell it what to change.

D.2 Critic behavior of using different critic models

The critic intervenes selectively, and more often for weaker policies. Most reviews end in `continue`: in nearly all settings, fewer than a third of reviews produce a candidate (Figure 13). Intervention frequency follows the policy's needs: Qwen3.5-9B draws the most interventions on Terminal-Bench 2.1 and SWE-Bench Pro, the stronger Qwen3.8-27B and DeepSeek-V4-Flash far fewer, and all policies receive more on the harder DeepSWE. Among critics, GPT-5.6 generally delivers the most notes, consistent with its larger gains (Section 5.3).

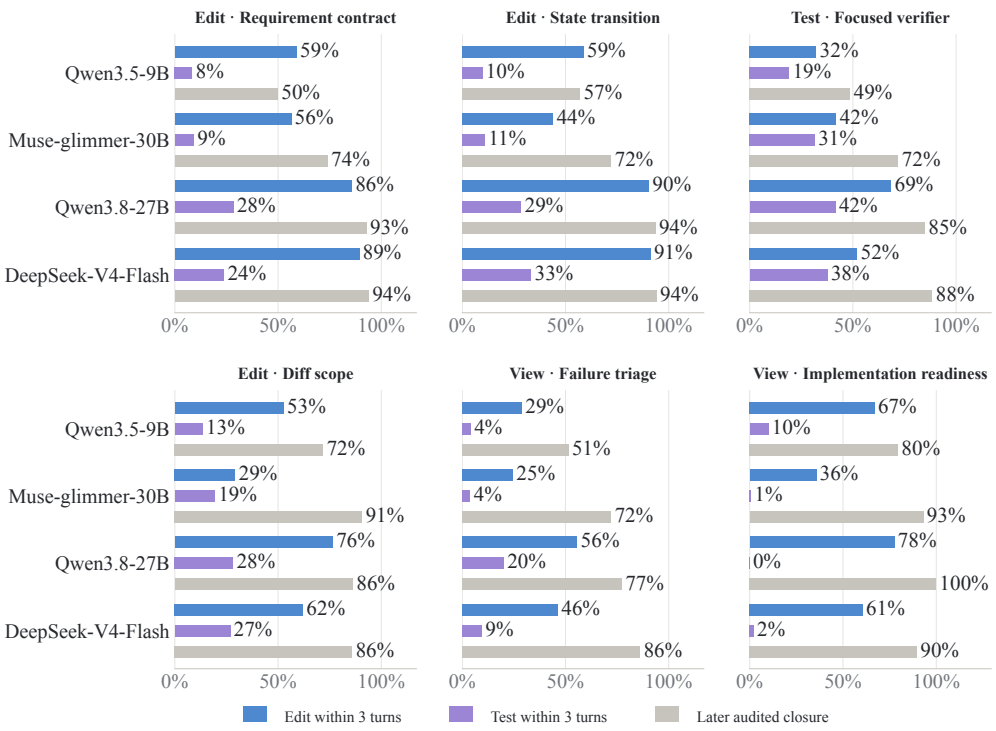


Figure 9: Agent responses to common operators. Bars show the percentages of delivered notes followed by an edit or a test within three turns, and the percentage eventually closed under audit.

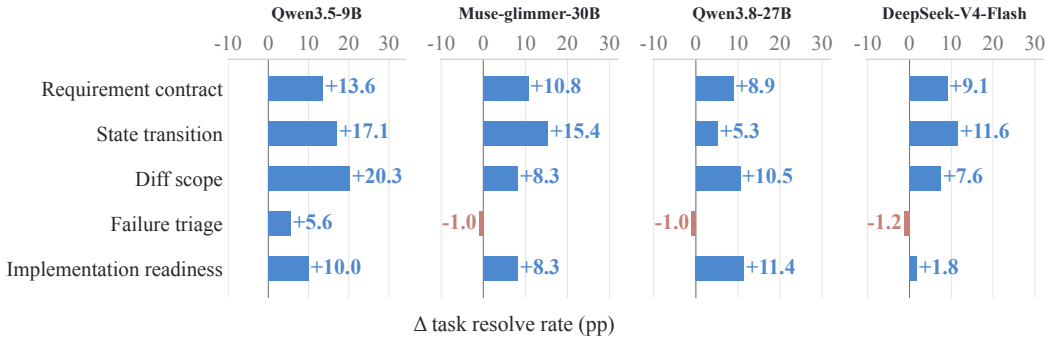


Figure 10: Task-success gains relative to baseline expectations, grouped by the first admitted operator. Common operator groups are shown. Gains reflect subsequent execution as a whole, not the isolated effect of the first operator.

Weaker critics rely more on the framework's filters. With GPT-5.6, most candidates are delivered, whereas self-critique delivers only a third, with nearly half filtered before the audit (Figure 12). The dominant filter is the open-note constraint: the self-critic keeps proposing new issues instead of following up on its open note. For Qwen3.5-9B on Terminal-Bench 2.1, it proposes candidates at more than twice Claude's rate yet delivers notes at a similar rate. Filtering thus brings a weak critic's intervention frequency close to a stronger one's, which helps explain why self-critique remains effective. Stronger critics fail differently: Claude is filtered mainly for referring to evaluation material the agent cannot access.

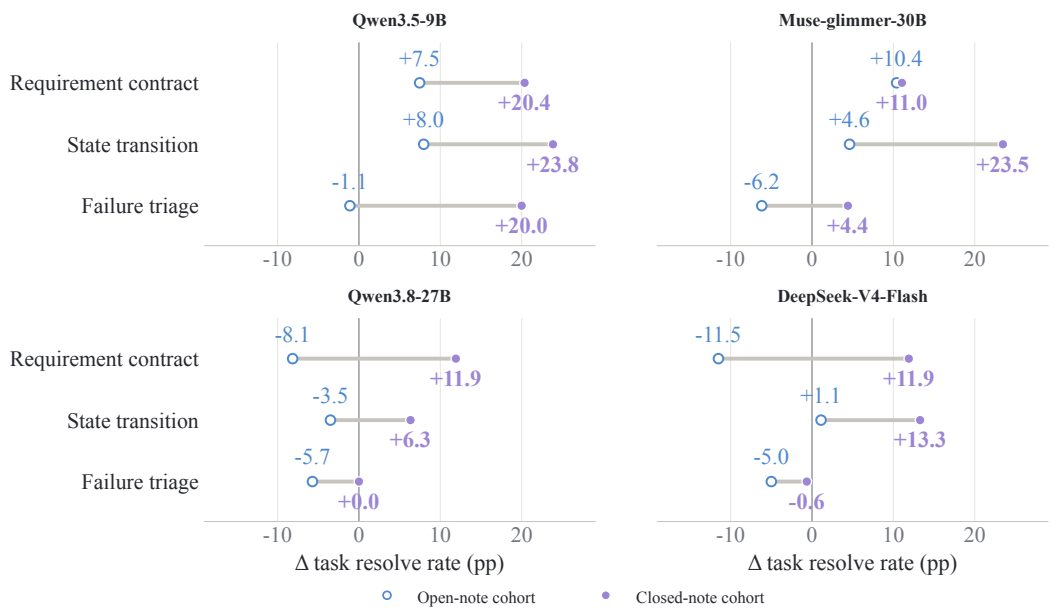


Figure 11: Task resolve-rate gains (pp) relative to no-critic expectations, grouped by policy model, first admitted operator, and note-closure status. Differences between open-note and closed-note cohorts describe associations rather than the causal effect of closure.

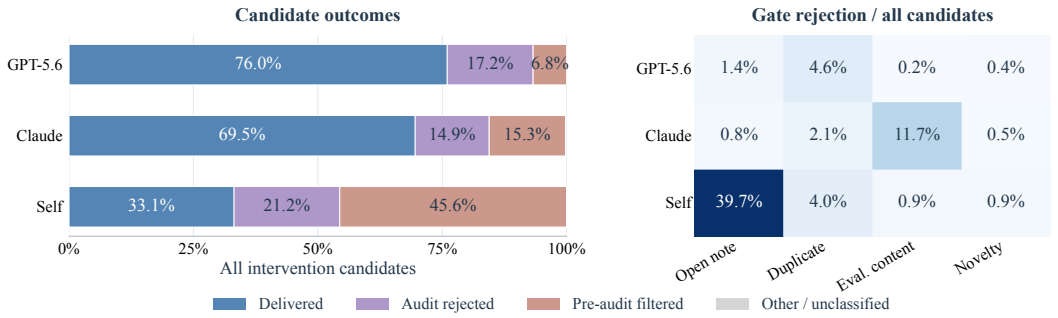


Figure 12: Intervention candidate outcomes across critic models. Left: delivered, audit-rejected, pre-audit-filtered, and unclassified candidates. Right: rejection rates for individual pre-audit gates. All percentages use all intervention candidates for the corresponding critic as the denominator.

E CASE STUDIES

Case titles use the operator stages defined in Section 3.2; these need not match the action the agent was performing when the issue was detected.

(1) Successful correction (search stage) — `gravitational/teleport-b1bcd8b9`

Task. The ingress reporter counts every HTTP connection as authenticated. Only TLS connections that present a client certificate may be counted; the fix has to live in the HTTP connection-state tracker (Active / Closed / Hijacked events).

Observation. A dead search loop: **eight consecutive `grep -rn ... -include='*.go' 2>/dev/null` calls returned nothing** because BusyBox grep does not support `-include` (it had printed `unrecognized option: include=*.go`

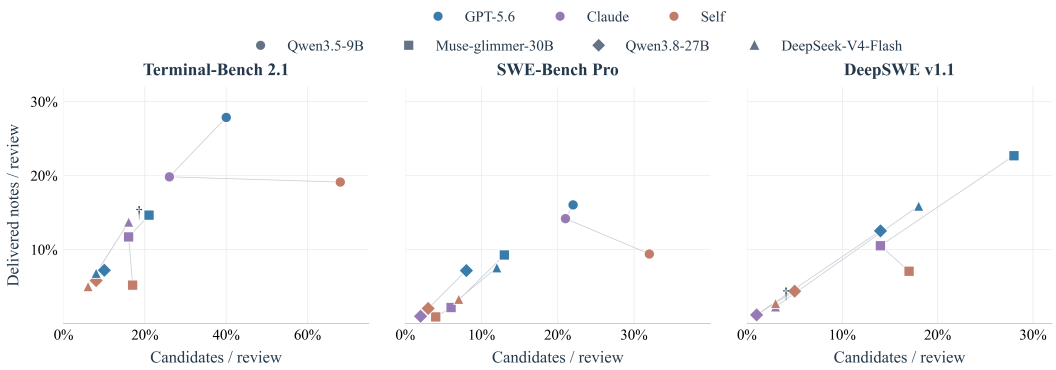


Figure 13: Proposal and delivery rates across critic and policy models. Each point represents a policy–critic setting within a benchmark: candidates per review on the x-axis and delivered notes per review on the y-axis. Colors denote critics, shapes denote policies, and lines connect the same policy across critics. Delivery rates are estimated from rounded aggregate statistics.

once), and the agent hid that error with `2>/dev/null` and kept re-issuing the same form. After 20 turns the connection-state callback was still unlocated.

Operator. *repo_localization_search* for investigation that repeats a search form already shown to be unsupported.

Resolution criterion. The Go code that registers the `http.Server.ConnState` callback is located, with visible handling of the Active, Closed and Hijacked events.

T1 **Agent.** Turns 13–21: **empty-output searches for `metric`, `StateActive`, ... with the same unsupported flag**; in between it reads unrelated client code (`api/client/webclient/webclient.go`).

T2 **Critic.** Evidence: the empty results are an artefact of the suppressed error, not of the repository; the governing tracker remains unidentified. **Correction: run `git grep -nE 'ConnState|StateActive|StateClosed|StateHijacked' -- '*.go'` (pathspec-aware) and follow the callback to its tracker/reporter.** Admission audit: accepted ("repeated unsupported searches about eight times without progress"); note `F-aaa5aa2a` opened.

T3 **Agent.** Turn 22: runs the suggested `git grep` verbatim, which returns `lib/srv/ingress/reporter.go:89 HTTPConnStateReporter`; turns 24–27: views `reporter.go` and its test file and records that `StateNew` calls `ConnectionAuthenticated` without checking for TLS client certificates.

T4 **Follow-up.** Audited closure at turn 33: **the `git grep` output identified `HTTPConnStateReporter`, and the viewed implementation shows its `http.ConnState` callback handling `StateClosed` and `StateHijacked`** — the exact site where `StateActive` handling and the TLS check must be added. Two later *state_transition_review* notes (an uninitialised `tracker` in `NewReporter`; a close path that re-checked TLS instead of decrementing) were also closed, and **`go test ./lib/srv/ingress -run TestIngressReporter` passes at turn 66**.

Outcome. Note: closed. **Task:** success (hidden tests pass; 0/3 without the critic).

Takeaway. The agent was not wrong about *what* to look for, only stuck in a tool form that silently returned nothing. One search-stage note replaced the loop with a single working query; localisation then unlocked the edit/test phase that the no-critic runs never reached.

(2) Successful correction (view stage) — `protonmail/webclients-d3e51304`

Task. Add `@proton/metrics` as a dependency and implement metric helpers: `getLabelID` must return `"custom"` for user-defined labels and the original ID for built-in mailboxes; `getPageSizeString` must normalise the page-size setting.

Observation. A mis-read of the definition it had just viewed: **after reading the `MAILBOX_LABEL_IDS` enum the agent wrote `import type { MAILBOX_LABEL_IDS }` and the predicate `labelID in MAILBOX_LABEL_IDS`.** The `in` operator tests enum *keys* (`INBOX`) while the mailbox IDs are the enum *values* (`"0"`), and a type-only import leaves no runtime object to test against — so the built-in-label branch can never be taken. The agent then treated the helper as finished and moved on to `package.json`.

Operator. *state_transition_review* for a branch that the required inputs can never reach.

Resolution criterion. `getLabelID("0")` returns `"0"` while a user-defined ID returns `"custom"`.

T1 **Agent.** Turns 22–26: creates `mailMetricsHelper.ts` with **the type-only import and the `in` predicate**, then edits `package.json`; nothing has exercised the helper.

T2 **Critic.** Evidence: `getLabelID("0")` falls into the `custom` branch (or fails at runtime because the enum is imported as a type). **Correction: import the enum as a value and classify by membership among `Object.values(MAILBOX_LABEL_IDS)`; verify by calling the helper and expecting `0` `custom`.** Admission audit: accepted; note `F-415d225d` opened.

T3 **Agent.** Turn 27 (thought): "the critic is correct"; turn 28: switches to a runtime import; turn 29: rewrites the predicate as value membership; turn 31: checks the logic with a **separately built `VALUES` array instead of the exported function**. The critic updates the note (turn 33, *minimal_repro_or_focused_verifier*): **call the exported `getLabelID` itself**; turn 33: the agent does so for `"0"` and `"custom-id"`.

T4 **Follow-up.** **The direct call prints `0` and `custom` with exit code 0**; audited closure at turn 37 cites exactly that output. (A third note on the same file is case (5).)

Outcome. **Note:** closed. **Task:** success (0/3 without the critic).

Takeaway. The agent had looked at the right definition but mis-read what the language construct tests. The note named the concrete input that breaks (`"0"`) and the observable to produce; the update then refused a proxy verification and insisted on the public interface, which is what the hidden tests call.

(3) Successful correction (edit stage) — `qutebrowser-0b621cb0`

Task. When a process cannot start, `GUIProcess._on_error` must report the capitalised process name, the exact command in single quotes and the error kind ("failed to start:", "crashed:", ...), and on non-Windows platforms append a hint that the command must exist and be executable.

Observation. A wrong-branch edit: **the first edit attached the new "failed to start" wording to `if error == QProcess.WriteError:`**, whereas a nonexistent command is reported by Qt as `QProcess.FailedToStart` and would still reach the old generic message. The task's own reproduction (start a nonexistent command) would therefore show none of the required wording; no test had been run yet.

Operator. *requirement_contract_review* for an edit that satisfies the wording but not the stated trigger.

Resolution criterion. A nonexistent command reported as `FailedToStart` yields `Name 'cmd' failed to start: <detail>` followed by the non-Windows hint.

T1 **Agent.** Turn 5: views `_on_error`; turn 6: **rewrites it so that the `WriteError` case produces the failed-to-start message, leaving `FailedToStart` to the generic branch**.

T2 **Critic.** Evidence: line 87 keys the message on `WriteError`; a real `FailedToStart` reaches lines 110–111. **Correction: move the formatting and the conditional non-Windows hint to the `FailedToStart` case and keep `WriteError` as its own message; verify with `pytest tests/unit/misc/test_guiprocess.py`.** Admission audit: accepted ("the agent is mistakenly linking it to WriteError"); note `F-8349cefc` opened.

T3 **Agent.** Turn 7 (thought): "the critic is correct"; turns 8–15: enumerates `QProcess.ProcessError` members (`FailedToStart = 0`); turn 16: re-edits the handler to key on `FailedToStart`; turns 20–22: finds and runs the unit test, whose log shows the message **still lacks the hint**. Turn 28 update: **the hint predicate compares the whole detail to `"no such file or directory"` while Qt returns `execvp: No such file or directory` — match the OS error text within the detail**; turn 30: the agent switches to a substring match; turn 31: reruns the test file.

T4 **Follow-up.** Audited closure at turn 35: **the visible pytest run started a real nonexistent command and logged `Testprocess 'this_does_not_exist_either' failed to start: execvp: No such file or directory (Hint: Make sure '...' exists and is executable)`**, satisfying the criterion even though the pre-existing test still asserts the obsolete format.

Outcome. Note: closed. **Task:** success (0/3 without the critic).

Takeaway. An edit-stage contract check caught a wrong-branch edit before any test had run, and the note's update handled the second-order mismatch (exact vs. substring) that the agent's own reasoning had missed; the no-critic runs ended with the same first edit in place.

(4) Successful correction (test stage) — `qutebrowser-ef5ba1a0`

Task. Locale override for QtWebEngine 5.15.3 on Linux: `_get_lang_override(webengine_version, locale_name)` must return no override when the setting is off, the platform/version does not match, or the original `.pak` exists; a mapped fallback locale when its `.pak` exists; otherwise `en-US`.

Observation. A green test run that proved nothing about the missing branch: **the implementation built `locales_dir` and went straight to the two `.pak` checks, so an unavailable directory falls through to `return 'en-US'` instead of returning `None`**; the agent then ran `tests/unit/config/test_qtargs.py`, which does not cover that state, and read the result as confirmation.

Operator. *state_transition_review* for a final-default transition taken where an early no-override return is required.

Resolution criterion. When the resolved `qtwebengine_locales` directory is unavailable, `_get_lang_override` returns `None`, not `en-US`.

T1 **Agent.** Turns 30–31: reads the existing test file and runs it; **the unavailable-directory path is neither guarded in the code nor exercised by the tests**.

T2 **Critic.** Evidence: lines 377–398 construct the directory, check two `.pak` files and return `'en-US'`, with no availability guard. **Correction: return `None` immediately after resolving `locales_dir` when it is unavailable; verify with a self-contained snippet that points `QLibraryInfo` at an empty temp dir and asserts `None`.** Admission audit: accepted; note `F-feda8c3f` opened.

T3 **Agent.** Turn 32 (thought) restates the requirement; turn 33: inserts `if not locales_dir.exists(): return None` before the `.pak` checks; turn 34: runs the critic's snippet.

T4 **Follow-up.** **The snippet prints `Result: None / unavailable directory: no override`**; audited closure at turn 36. Two later notes on the same trial were also closed: *diff_scope_review* (the `qt.workarounds.locale` option was read but never registered in `configdata.yml`; closed at turn 52 after **`registered`**) and *requirement_contract_review* (the function took a `WebEngineVersions` object instead of the version value; closed at turn 63 after **`version accepted`**).

Outcome. **Note:** closed. **Task:** success (0/3 without the critic).

Takeaway. The agent's own test run was green for the branches it exercised; the note supplied the missing state (directory absent) together with a runnable check, turning an untested transition into an observable one.

(5) Successful correction (test stage, failure triage) — `protonmail/webclients-d3e51304`

Task. As in case (2): `getLabelID(labelID: string)` must accept any string and type-check.

Observation. Half of a failure signature acted on, the other half abandoned: **`tsc` reported TS2322 and TS2345 in the helper; the agent fixed the return statement (TS2322), then drifted into `grep` searches of the constants file while the second error, `TS2345: Argument of type 'string' is not assignable to parameter of type 'MAILBOX_LABEL_IDS' at includes(labelID)`, was still printed by the re-run**.

Operator. *failure_signature_triage* for a visible compiler error that the latest actions no longer address.

Resolution criterion. `mailMetricsHelper.ts` type-checks with `getLabelID` accepting arbitrary strings for the enum-value membership test.

T1 **Agent.** Turn 37: `tsc -noEmit` prints both errors; turn 38: edits the return statement; turns 43–45: **greps the enum definition and re-runs `tsc`, which still prints TS2345 at line 36**.

T2 **Critic.** Evidence: `Object.values(MAILBOX_LABEL_IDS)` infers an enum-typed array, so `.includes(labelID)` rejects a plain `string`. **Correction: type the collected values as `readonly string[]` while keeping the declared `MAILBOX_LABEL_IDS | 'custom'` return; re-run `tsc` and expect no diagnostic for the helper file.** Admission audit: accepted; note `F-2cd990db` opened.

T3 **Agent.** Turn 46: declares `const MAILBOX_LABEL_ID_VALUES: readonly string[] = Object.values(...);` turn 47: re-runs `tsc` filtered to the helper file.

T4 **Follow-up.** **The filtered `tsc` output is empty (the 31 remaining errors are all in other files)**; audited closure at turn 51. Turn 50 then exercises `getPageSizeString` and `getLabelID` end to end before the agent finishes.

Outcome. **Note:** closed. **Task:** success.

Takeaway. The failure signature was already on screen; the agent had acted on half of it and then started searching. The triage note re-attached the remaining error line to a one-line fix and a re-run, instead of letting the search continue.

(6) Successful correction (submit stage) — `qutebrowser-8cd06741`

Task. Expose the QtWebEngine 6.6 dark-mode image classifier: `colors.webpage.darkmode.policy.images` gains the value `smart-simple`; on Qt 6.6+ `smart` must emit `ImagePolicy=2, ImageClassifierPolicy=0` and

`smart-simple` must emit `ImageClassifierPolicy=1`; older Qt versions must be unaffected.

Observation. A completion without the central check: **the one direct `darkmode.settings()` call had crashed with `AttributeError: 'NoneType' object has no attribute 'get'` (no config instance) and was never retried**; instead the agent inspected `_MultiMappingSetting.tuples()` and the internal mapping tables, marked all five task-tracker items done and was about to call `finish`.

Operator. *submission_readiness_review* for verification that bypasses the interface the task is about.

Resolution criterion. A direct `darkmode.settings()` call shows Qt 6.5 emitting only `ImagePolicy=2` for both values and Qt 6.6 additionally emitting `ImageClassifierPolicy` 0 or 1.

T1 **Agent.** Turns 102–106: **updates the task tracker to five "done" items and prints `_IMAGE_POLICIES` and the classifier mapping as its final check**; the failed end-to-end attempt (turn 101) is not retried.

T2 **Critic.** Evidence: every successful check reads internals; none shows configured values flowing through `settings()` into the emitted switch data. **Correction: run one end-to-end script that mocks `config.instance.get` and asserts the four required `settings()` outputs for Qt 6.5 and 6.6; expect `settings() end-to-end OK`.** Admission audit: accepted ("bypasses the necessary interface"); note `F-b81e67aa` opened.

T3 **Agent.** Turn 108: runs the script verbatim.

T4 **Follow-up.** **Output `settings() end-to-end OK`** — the close audit accepts on that output and the agent finishes on the next action (turn 109). Earlier in the same trial a *requirement_contract_review* note (opened turn 43, three updates, closed turn 86) and a *state_transition_review* note (a shallow `copy.copy` in `copy_replace_setting` mutated the shared `_Setting` objects so building the Qt 6.4 definition changed Qt 5.15.3's keys; closed at turn 101 after **`Settings are independent!`**) had been closed.

Outcome. Note: closed. **Task:** success (0/3 without the critic).

Takeaway. The submit-stage note did not change the patch; it forced the one verification the agent had skipped after an unrelated setup error, and in this trial that verification passed. The same operator is what blocks premature `finish` calls when it does not.