BEYOND FULL-TRAJECTORY IMITATION: LOCALIZED EXPERT SUPERVISION FOR ISSUE REPRODUCTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent imitation learning methods improve code agents by learning from complete expert rollouts and expert continuations collected during student execution. However, effectively using such supervision requires identifying both where the student needs expert intervention and which expert behaviors are useful for correcting it. In this work, we present LOEC, a three-stage post-training framework that selectively allocates expert supervision for repository-level issue reproduction. LOEC first shapes the initial strategy through plan-level on-policy distillation, and then uses issue-reproduction progress signals to localize stalled or invalid states that require expert intervention. For successful expert recoveries, it further narrows supervision through intervention-level plan shaping and masked SFT over selected segments, including effective test edits retained in the final test. On SWT-Bench Verified, LOEC improves Pass@1 from 21.11% to 32.24%, and outperforms two expert-takeover imitation baselines. Compared with full-trajectory BC for initial alignment and full-suffix SFT for correction, LOEC achieves better performance while reducing supervised-token usage by over 91% and over 18%.

1 INTRODUCTION

Code agents have shown promising capabilities in software-related tasks, such as resolving real-world software issues, generating tests, and repairing software vulnerabilities (Yang et al., 2024; Pan et al., 2025; Liu et al., 2025). Beyond issue resolution, code agents are also applied to repository-level issue reproduction, an important task for issue localization and fix validation that requires generating tests that fail on the buggy repository and pass after the corresponding fix is applied (Mündler et al., 2024; Kitsios et al., 2025). However, reliable issue reproduction based on code agents remains challenging (Soni et al., 2026; Ahmed et al., 2026). Code agents may generate tests that pass on the buggy repository, become trapped in loops after producing inapplicable edits, or introduce syntax errors (Mündler et al., 2024). Even when a test fails on the buggy version, it may encode an incorrect assertion that fails both before and after the issue is fixed (Nashid et al., 2026; Soni et al., 2026). These failures motivate post-training to improve the reliability of code agents in issue reproduction.

Imitation learning has gained increasing attention for improving language-model agents. Recent methods construct supervision from successful expert-guided behavior, including complete expert trajectories used to initialize the student and successful expert continuations used to correct student behavior (Lauffer et al., 2026; Li et al., 2026a; Hao et al., 2026; Lyu et al., 2026). However, two challenges remain in determining which student states should trigger intervention and which parts of expert trajectories should receive supervision. ❶ *Intervention-state selection.* Existing methods select intervention states using randomly sampled switch points (Lauffer et al., 2026) or error states identified by an expert (Lyu et al., 2026). These criteria do not explicitly account for task-specific progress, leading to corrections that may not target the states where the student actually struggles. ❷ *Supervision scope.* Existing methods often apply supervision to all turns in complete expert rollouts for student initialization (Lauffer et al., 2026; Lyu et al., 2026) or successful expert continuations for later correction (Hao et al., 2026). However, the success of a rollout does not imply that every supervised turn is necessary for effective task solving. Consequently, full-sequence imitation may allocate supervision to turns that contribute little to the final issue reproduction success.

To address these challenges, we propose **Localized Expert Correction (LOEC)**, a three-stage post-training framework for issue reproduction. Accordingly, LOEC narrows the *supervision scope* dur-

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.

ing both initial strategy shaping and later correction, while selecting *intervention states* based on task-specific progress. To give the student an expert-aligned starting strategy without requiring it to imitate the entire trajectory, LOEC first performs **initial plan shaping**, which applies plan-level on-policy distillation (OPD) (Agarwal et al., 2024) using expert scores on student-generated initial plans. For later correction, the goal is to localize where expert intervention is needed and identify which recovery behavior should be retained for supervision. **Progress-guided stall localization** traces task-specific progress failures back to intervention states near the onset of stalled or invalid behavior. For each localized state, the expert produces a complete recovery trajectory. After verifying the recovery, LOEC uses the resulting trajectory as evidence for two complementary forms of **localized expert correction**. Intervention-level plan shaping distills the recovery strategy on student-generated plans, while masked SFT retains selected corrective segments, such as test edits preserved in the final verified patch and subsequent task-closure behavior. Together, these stages enable targeted expert supervision throughout the training process.

We collect training trajectories on issue-reproduction tasks adapted from SWE-Gym (Pan et al., 2025), using Qwen3.5-4B as the student and Qwen3.5-27B as the expert, and use the collected supervision to train the student with LOEC. We then evaluate the resulting student on SWT-Bench Verified (Mündler et al., 2024). LOEC improves Pass@1 from 21.11% to 32.24%, corresponding to a 52.72% relative improvement over the base student, and outperforms the SOTA baselines OEC (20.74%) (Lauffer et al., 2026) and SCORE (29.65%) (Lyu et al., 2026). We further find that selective supervision outperforms full-trajectory BC and full-suffix SFT while reducing supervised-token usage by 91.50% during initial alignment and 18.20% during expert correction. Our ablation evaluation shows that both stages of our localized expert supervision contribute to performance. Finally, our case studies show that LOEC more reliably targets the reported bug, avoids hidden errors in generated tests, and carries valid reproductions through to final submission.

Our contributions are summarized as follows:

- We introduce LOEC, which uses task-specific progress to localize intervention states, applies on-policy plan distillation at both initial and recovery states, and extracts corrective segments from verified expert takeovers using evidence such as test edits retained in the final patch.
- We validate LOEC for repository-level issue reproduction on SWT-bench, improving the student average Pass@1 from 21.11% to 32.24%.
- We empirically show that localized expert supervision outperforms full-trajectory imitation while requiring substantially fewer supervised tokens during both initial alignment and later correction.

2 Related Work

Issue reproduction with code agents. Issue reproduction requires exploring the repository and generating tests. SWT-Bench formalizes repository-level issue reproduction and provides a benchmark for evaluating code agents (Mündler et al., 2024). Prior approaches adapt general-purpose code agents or construct specialized workflows for test generation, localization, execution-feedback refinement, and candidate selection (Kang et al., 2023; Wang et al., 2025; Khatib et al., 2026; Ahmed et al., 2025; 2026). More recently, SWE-Tester trains open-source models on reproduction tests derived from historical pull requests (Soni et al., 2026). These methods primarily improve inference workflows or learn from offline examples. In contrast, we study post-training an issue-reproduction agent from states encountered along its own trajectories.

Trajectory-based training of software agents. Recent work increasingly post-trains software agents from interaction trajectories. SWE-Gym provides executable repositories for real-world software-engineering tasks, while SWE-Playground and Hybrid-Gym construct synthetic environments to improve the transferability of repository-level skills (Pan et al., 2025; Zhu et al., 2026; Xie et al., 2026). More broadly, agent-training methods learn from successful demonstrations and model-generated refinements to improve multi-step policies (Chen et al., 2023; Song et al., 2024). These methods demonstrate the value of trajectory supervision but often treat complete trajectories as training targets. In contrast, LOEC focuses supervision on critical steps for efficient correction.

Expert correction on student trajectories. Existing expert-intervention methods differ both in where the expert intervenes and in how the resulting correction data are used to train the student.

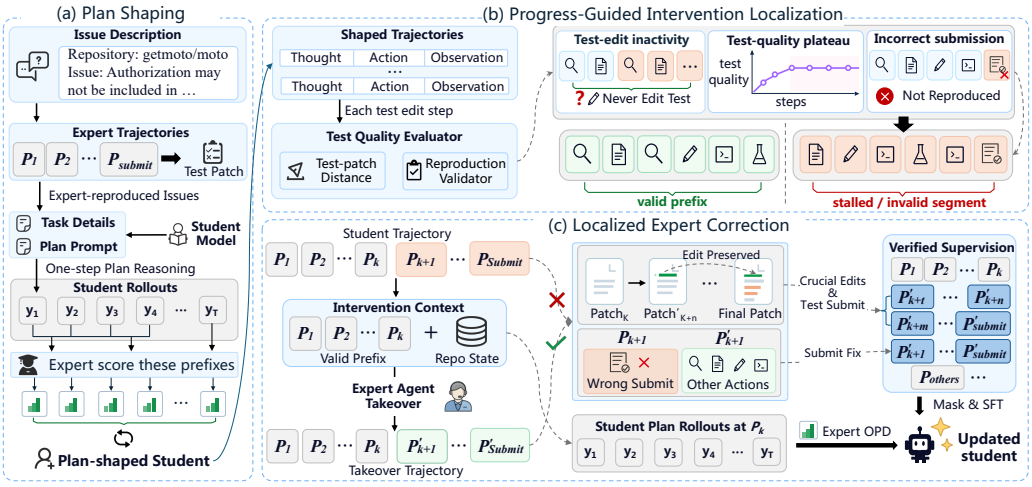

Figure 1: Overview of the LOEC.

OEC switches from the student to the expert at a randomly sampled turn and applies SFT to successful expert-generated suffixes (Lauffer et al., 2026), whereas SCORE asks the expert to identify and correct errors in completed student trajectories before combining corrected-trajectory SFT with short-horizon reinforcement learning (Lyu et al., 2026). Related approaches train on bounded expert continuations or outcome-verified recovery trajectories (Ye et al., 2026; Xia et al., 2026). On-policy approaches instead provide supervision on student-induced distributions. OPD evaluates teacher feedback on student-generated outputs, while recent DAgger-style training collects learner-induced states and trains on expert labels for visited states (Agarwal et al., 2024; Li et al., 2026a) Recent analyses further show that on-policy distillation can become less effective when student and expert reasoning patterns diverge or when long-horizon student prefixes drift from the typical distribution of the expert (Li et al., 2026b; Fu et al., 2026). In contrast, LOEC localizes intervention states from issue-reproduction progress, uses verification evidence to retain selected corrective segments, and applies on-policy plan distillation both before execution and at localized intervention states to provide strategic supervision on student-generated plans in addition to action-level correction.

3 METHOD

Effective expert supervision must address three questions: how to shape the initial task-solving strategy, where to intervene when later progress stalls, and what parts of expert recovery should be retained for training. LOEC consists of three post-training stages. First, **plan shaping** (§3.1) aligns the student planning behavior with that of the expert through OPD. Second, **progress-guided intervention localization** (§3.2) analyzes student trajectories and identifies states associated with stalled progress or invalid termination. Third, **verified-takeover correction** (§3.3) retains intervention contexts from which the expert produces an effective continuation, applies localized recovery-plan OPD, and performs masked SFT on effective segments of the successful expert takeover.

3.1 INITIAL PLAN SHAPING

The initial plan specifies the student strategy for investigating the issue and organizing subsequent repository interactions toward reproducing it. Programming agents have been observed to follow problem-solving workflows internalized during training even in the absence of an explicit task plan (Liu et al., 2026). Motivated by this observation, we apply expert supervision only to the initial planning turn, rather than distilling every action along the trajectory.

Expert-solvable instance selection. Let $\mathcal{D}$ denote the set of SWE-Gym training instances and π_E the expert policy. We first generate expert rollouts and evaluate them using the issue-reproduction

verifier. We retain an instance if the expert produces at least one valid Fail-to-Pass test.

$$\mathcal{D}_{\mathrm{PS}} = \left\{ x \in \mathcal{D} \;\middle|\; \exists \tau^E \sim \pi_E(\cdot \mid x),\ \mathcal{V}(x, \tau^E) = 1 \right\}, \tag{1}$$

Here, $\mathcal{V}$ denotes the issue-reproduction verifier. Thus, $\mathcal{D}_{\mathrm{PS}}$ contains instances on which the expert has demonstrated task-specific competence. The successful expert rollouts are used only for instance selection and are not directly imitated as full-trajectory targets.

Plan-level on-policy distillation. For each instance $x \in \mathcal{D}_{\mathrm{PS}}$, the rollout policy π_{old} samples an initial plan $y = (y_1, \ldots, y_T)$ conditioned on the issue description and planning prompt. Rather than generating a separate reference plan, the expert evaluates the same student-sampled tokens under the corresponding prefixes. For each planning token, we compute the log-ratio cost

$$k_t = \log \pi_{\mathrm{old}}\left(y_t \mid x, y_{<t}\right) - \log \pi_E\left(y_t \mid x, y_{<t}\right). \tag{2}$$

Because y_t is sampled from π_{old}, the conditional expectation of k_t equals the reverse KL divergence.

$$\mathbb{E}_{y_t \sim \pi_{\mathrm{old}}(\cdot \mid x, y_{<t})}\left[k_t\right] = D_{\mathrm{KL}}\left(\pi_{\mathrm{old}}(\cdot \mid x, y_{<t}) \,\|\, \pi_E(\cdot \mid x, y_{<t})\right). \tag{3}$$

We define the token-level distillation advantage as

$$\hat{A}_t^{\mathrm{OPD}} = -\,\mathrm{sg}(k_t), \tag{4}$$

where $\mathrm{sg}(\cdot)$ denotes the stop-gradient operator. Using $\hat{A}_t^{\mathrm{OPD}}$ as the token-level learning signal, we optimize the on-policy distillation objective using a Proximal Policy Optimization (PPO)-style clipped policy update. The expert-induced log-ratio defines the supervision signal, whereas PPO constrains the update from the rollout policy π_{old} to the updated student policy π_θ. Full optimization details are provided in Appendix A.

Plan-shaped policy. Let θ_{PS} denote the student parameters obtained after plan-level on-policy distillation, and define $\pi_{\mathrm{PS}} \coloneqq \pi_{\theta_{\mathrm{PS}}}$. Aligning the student's planning behavior with that of the expert is intended to improve the strategy guiding its subsequent repository interactions. We use π_{PS} to generate the student trajectories analyzed in the next stage.

3.2 Progress-Guided Intervention Localization

The final binary reproduction result indicates whether a student trajectory succeeds, but does not identify where its issue-reproduction process stops making progress. Starting from the plan-shaped policy π_{PS}, we collect completed student trajectories and augment their final outcomes with task-specific progress signals to retrospectively localize suitable intervention states.

Intervention trigger identification. We derive the intervention triggers from the task structure of issue reproduction. A productive trajectory should translate repository investigation into concrete test edits, progressively refine those edits toward a valid Fail-to-Pass reproducer, and submit only after successful reproduction has been established. We identify prolonged ❶ *test-edit inactivity* and ❷ *test-quality plateaus* as two forms of temporal stall. We additionally treat ❸ *incorrect submission* as a terminal intervention trigger. Test-edit inactivity is triggered when no effective test-file modification occurs within a fixed action window. A test-quality plateau is triggered when repeated test executions fail to reduce the execution distance from the code covered by the test to the patch. An incorrect submission is triggered when the agent submits before establishing valid reproduction.

Retrospective intervention-state selection. Given a completed student trajectory, we first identify the segment associated with one of these intervention triggers. Localization is performed retrospectively. Rather than selecting the state at which enough evidence for a stall has already accumulated, we select an intervention state near the beginning of the identified behavior. Each selected student prefix and its associated repository state define an intervention context c for the following stage. The exact stall definitions and intervention-state selection rules are provided in Appendix B.

3.3 Localized Expert Correction

For each intervention context with a verified expert takeover, we provide two complementary forms of supervision. First, localized recovery-plan OPD aligns the recovery strategy of the student with that of the expert at the intervention point. Second, masked takeover SFT assigns token-level loss only to crucial assistant turns in the successful expert continuation.

Trajectory-level takeover verification. For each selected intervention context, the expert continues from the same interaction history and repository state to produce a complete takeover trajectory. We retain only takeovers whose final tests add at least one new Fail-to-Pass test. These verified takeovers establish that the localized state is recoverable and provide reference trajectories for the subsequent correction procedure.

Intervention-level plan shaping. After verifying a successful expert takeover, we return to the localized intervention state and reuse the plan-level OPD procedure introduced in Section 3.1. The student samples a recovery plan conditioned on the interaction history and repository state at the intervention point, and the expert evaluates the same student-sampled plan tokens under the corresponding prefixes. We apply the same token-level distillation strategies defined in Eqs. 2–4, replacing the initial planning context with the localized intervention context. This localized plan shaping provides strategic guidance for recovery before subsequent repository interaction.

Localized supervision extraction. Given the valid student prefix $P_1, \ldots, P_k$ and a verified expert takeover $P'_{k+1}, \ldots, P'_{\text{submit}}$, we selectively retain three types of expert segments for supervision.

(i) **Crucial edits.** For each test edit whose effective modification is retained in the final verified patch, we supervise the segment from the preceding test edit to the current edit, including the intermediate reasoning and repository interactions, as these edits make verified contributions to the final solution.

(ii) **Test submit.** Once the final tests are generated by the agent, we supervise the remaining verification, inspection, and submission steps, which capture the execution and termination behavior needed to complete a valid submission.

(iii) **Submit recovery.** If the original student action is an invalid submission while the expert takeover eventually succeeds, we supervise the expert continuation from P'_{k+1} to P'_{submit}, since the invalid submission directly exposes a failure of the student at this state, making the subsequent expert recovery particularly informative for correction.

Let $\mathcal{I}_{\text{edit}}$, $\mathcal{I}_{\text{close}}$, and $\mathcal{I}_{\text{sub}}$ denote the expert steps selected by the three criteria above. The final supervision set is $\mathcal{I} = \mathcal{I}_{\text{edit}} \cup \mathcal{I}_{\text{close}} \cup \mathcal{I}_{\text{sub}}$. All remaining expert steps are retained as trajectory context but masked from the training loss.

Masked trajectory SFT. After intervention-level plan shaping, we perform LoRA SFT only on the expert steps selected by $\mathcal{I}$. Let $P'_i = (y_{i,1}, \ldots, y_{i,L_i})$ denote the assistant tokens at expert step P'_i. The training objective is

$$\mathcal{L}_{\text{SFT}}^{\text{corr}} = -\frac{1}{Z} \sum_{i \in \mathcal{I}} \sum_{j=1}^{L_i} \log \pi_\theta \left(y_{i,j} \mid P_1, \ldots, P_k, P'_{k+1}, \ldots, P'_{i-1}, y_{i,<j} \right), \tag{5}$$

where $Z = \sum_{i \in \mathcal{I}} L_i$ is the number of supervised expert tokens. Expert steps outside $\mathcal{I}$ remain available as trajectory context but are excluded from the training loss.

4 Experiments

Our experiments are designed to answer the following questions:

1. How does LoEC compare with representative expert-guided post-training methods? (§4.2)
2. How does selective expert supervision compare with full-trajectory imitation? (§4.3)
3. How do the key design choices of LoEC contribute to its effectiveness? (§4.4)
4. How does LoEC change student behaviors during issue reproduction? (§4.5)

4.1 Experiment Setup.

Student and expert models. We use Qwen3.5-4B as the student and Qwen3.5-27B as the expert (Qwen Team, 2026). We use Qwen3.5-4B as the student model because it balances post-training efficiency with strong agentic and tool-use capabilities. The expert model is selected

because its reported performance is substantially stronger than that of the 4B student on coding benchmarks (Qwen Team, 2026), making it a suitable source of supervision for multi-turn repository interaction generated by the student model. During evaluation, we deploy both models using SGLang (Zheng et al., 2024) with greedy decoding and enable deterministic inference[1] to reduce output variation caused by batch composition and improve reproducibility of our conclusion.

Agent scaffold. We use mini-SWE-agent (Yang et al., 2024), a lightweight software-engineering agent scaffold, to generate trajectories. Its agent loop restricts environment interaction to a shared Bash action space, reducing variation introduced by model-specific tool design. We cap each rollout at 100 interaction steps. The prompt detail is provided in Appendix C.

Training problem instances. We use SWE-Gym (Pan et al., 2025) as the source of all training instances and generate trajectories in its repository environments. We adapt these instances to issue reproduction, where an agent succeeds if its generated test suite contains at least one test that fails on the buggy repository and passes after the gold fix is applied. For the **plan shaping** stage, we evaluate 2,296 trajectories generated by Qwen3.5-27B and retain 1,034 unique problem instances for which at least one trajectory produces a valid fail-to-pass test. These successful trajectories demonstrate the expert ability to solve the corresponding instances, which we subsequently use for plan-level OPD. For the **stall takeover** stage, we roll out the plan-aligned student policy on SWE-Gym and generate expert continuations from 1,174 identified stall states over five rounds. We then localize the supervision segments within these trajectories and perform case-level deduplication. This process yields the recovery-plan OPD states and localized SFT trajectories for the subsequent correction. Detailed training configurations are provided in Appendix D.

Baselines. We compare LOEC with two SOTA expert-guided post-training methods for LLM agents that provide supervision on student-induced states, OEC (Lauffer et al., 2026) and SCORE (Lyu et al., 2026). For a controlled comparison, we instantiate all methods with the same Qwen3.5-4B student and Qwen3.5-27B expert used by LOEC. We adapt these baselines to issue-reproduction tasks in SWE-Gym and collect their training trajectories following the reported data-collection and intervention procedures. Detailed data-collection procedures, training-data composition, and optimization details for each baseline are provided in Appendix E.

Evaluation dataset. We evaluate all methods on SWT-Bench Verified (Mündler et al., 2024), which contains 433 human-verified issue-reproduction instances derived from real-world GitHub pull requests. Each instance consists of a natural-language issue description, the corresponding pre-fix repository snapshot, and a gold patch that resolves the issue. The gold patch is not exposed to the code agent and is used only for verification. A generated test is considered successful if it fails on the pre-fix repository and passes after the gold patch is applied. Moreover, SWT-Bench Verified has no repository-level overlap with the SWE-Gym. Dataset details are provided in Appendix F.

4.2 MAIN RESULTS

Table 1 compares LOEC with the base model and representative expert-guided baselines on SWT-Bench Verified. We evaluate each model in five separate runs using greedy decoding and deterministic model inference. Variation across runs arises from runtime environment interactions, which can produce different observations and subsequent agent actions. The repeated evaluations therefore measure reproduction performance and consistency under environment-induced execution variability. For the five independent runs, we report average Pass@1 to measure single-run reproduction performance, Pass@5 to measure the percentage of issues reproduced in at least one run, and Pass^5 to measure the percentage reproduced in all five runs. Following the benchmark protocol, we additionally report ΔCoverage and Applicability to evaluate test quality.

LOEC improves both issue-reproduction success and consistency across runs. LOEC achieves 32.24% Pass@1, improving over the base model by 11.13 percentage points. In comparison, the expert-guided baselines achieve between 20.74% and 29.65% average Pass@1, leaving LOEC ahead of the strongest baseline by 2.59 percentage points. The improvement remains consistent under five-run evaluation. Compared with the base model, LOEC increases Pass@5 from 42.26% to 50.35% and Pass^5 from 5.08% to 16.17%, while also outperforming all baselines with complete five-run

[1]SGLang deterministic inference documentation: https://docs.sglang.io/docs/advanced_features/deterministic_inference

Table 1: Issue-reproduction performance across methods on SWT-Bench Verified.

Method	Pass@1 (%) ↑	Pass@5 (%) ↑	Pass^5 (%) ↑	Δ Coverage (%) ↑	Applic. (%) ↑
Qwen3.5-4B	21.11 ± 0.52	42.26	5.08	27.73 ± 0.22	45.68 ± 0.69
+ OEC	20.74 ± 0.32	35.10	9.70	25.41 ± 0.46	41.85 ± 0.25
+ SCoRe-SFT	22.49 ± 0.52	40.65	7.16	27.06 ± 0.69	43.79 ± 0.69
+ SCoRe-RL	29.65 ± 0.65	47.11	15.01	33.81 ± 0.64	57.78 ± 0.61
+ LoEC	32.24 ± 0.39	50.35	16.17	34.90 ± 1.03	56.44 ± 1.03

Note. Values are reported as mean ± standard error of the mean (SEM) over five independent runs, except Pass@5 and Pass^5, which is computed from five attempts per instance.

results. The gains across these metrics show that LoEC improves not only average reproduction performance, but also the likelihood of reproducing an issue consistently across independent runs.

LoEC has data generation costs comparable to the supervised baselines and substantially lower than RL. Holding the initial expert rollouts on SWE-Gym fixed across methods, LoEC, OEC, and SCoRe-SFT require comparable amounts of additional takeover steps generation for trajectory correction. The corresponding takeover output-token counts are 17.15M, 23.13M, and 15.62M, respectively. The data-collection costs are broadly comparable. In contrast, SCoRe-RL performs an additional 27 RL iterations, each collecting 256 student rollouts, resulting in additional student trajectories, which makes its data generation cost much higher than that of other methods. LoEC achieves stronger reproduction performance than the supervised baselines with a comparable data generation budget, while avoiding the additional sampling required by SCoRe-RL.

LoEC improves both coverage and applicability over the base model. LoEC increases ΔCoverage from 27.73% to 34.90% and Applicability from 45.68% to 56.44%, showing that the improvement in reproduction success is accompanied by broader test coverage and a larger proportion of applicable tests.

4.3 Supervision Strategy

We investigate whether localized expert supervision can provide a more effective and efficient supervision over complete expert trajectory imitation. We compare LoEC with full-trajectory BC during initial trajectory shaping and full-suffix SFT during expert correction. Each stage-wise comparison uses the same starting checkpoint and training instances. Table 2 reports issue-reproduction performance and the number of supervised tokens.

Supervising only the initial plan outperforms full-trajectory imitation while using substantially fewer supervised tokens. Full-Trajectory BC supervises all assistant turns in successful expert rollouts, whereas Plan-Level OPD supervises only student-generated initial plans. We additionally evaluate Plan-Only SFT using fixed expert-generated planning targets and observe a substantial performance degradation compared with both the base model and Plan-Level OPD. One plausible explanation is that direct SFT on fixed targets can overfit to a narrow target distribution (Shenfeld et al., 2026), whereas OPD trains on student-generated plans and therefore reduces the distribution mismatch associated with fixed-target training (Agarwal et al., 2024). Accordingly, we use on-policy distillation for plan shaping in LOEC. Plan-Level OPD achieves 27.25% Pass@1, compared with 25.64% for Full-Trajectory BC, while using 91.50% fewer supervised tokens. This comparison shows that increasing the amount of expert supervision does not necessarily translate into better reproduction performance, as concentrating supervision on the initial planning step achieves higher Pass@1 with only a small fraction of the supervised tokens.

Localized correction combining recovery-plan shaping and selective takeover supervision outperforms full-suffix SFT. Full-Suffix SFT supervises all assistant turns in successful expert takeovers, whereas Localized Correction combines recovery-plan OPD at the intervention state with masked SFT over only the crucial steps identified from the verified takeover. Localized Correction achieves 32.24% Pass@1, compared with 28.41% for Full-Suffix SFT, while using 18.20% fewer supervised tokens. This result shows that even within successful expert takeovers, supervising the complete continuation is not necessary for effective correction.

Table 2: Supervised-token usage under the configured training budget for each strategy. The initial-alignment comparison starts from the base student (21.11% Pass@1), while the expert-correction comparison starts from the plan-shaped student (27.25% Pass@1).

Stage	Method	Supervision	Supervised Tokens ↓	Pass@1 ↑
Alignment	Full-Trajectory BC	All expert turns	3,952,834	25.64%
	Initial Plan OPD (Ours)	Initial plan only	**335,813** (↓ 91.50%)	**27.25%** (↑ 1.61 pp)
Correction	Full-Suffix SFT	All takeover steps	975,296	28.41%
	Localized Correction (Ours)	Plan & crucial steps	**797,824*** (↓ 18.20%)	**32.24%** (↑ 3.83 pp)

Note. * Comprises 284,073 tokens for recovery-plan OPD and 513,751 tokens for localized SFT.

Table 3: Ablation study of LOEC.

Phase	Design	Pass@1 ↑	Δ Cov. ↑	Applic. ↑	Turns ↓
—	Base Model	21.11%	27.73%	45.68%	46.10
Plan Shaping	Plan-Level OPD Only	27.25%	28.61%	51.96%	35.70
Localization & Training	w/o Stall Localization	27.94%	31.03%	49.65%	41.76
	w/o Localized Correction	28.41%	28.48%	46.19%	39.47
	Full Design	32.24%	34.90%	56.44%	42.55

Across stages, targeted supervision is more effective than exhaustive expert imitation. Both stage-wise comparisons exhibit the same pattern that LOEC achieves higher Pass@1 while using fewer supervised tokens than full-trajectory or full-suffix imitation. This cross-stage consistency supports the central design principle of LOEC that expert supervision should be concentrated on critical decision points rather than uniformly applied over complete expert rollouts.

4.4 Ablation Study

Table 3 presents ablations of the key components of LOEC. **Plan-Level OPD Only** applies plan shaping without subsequent correction training. **w/o Stall Localization** replaces progress-guided intervention localization with intervention points selected by the teacher after examining the full student trajectory. **w/o Localized Correction** replaces the combination of intervention plan shaping and SFT over selected steps with SFT on the complete successful expert takeover suffixes.

Initial plan shaping alone improves both reproduction effectiveness and interaction efficiency. Compared with the base model, Plan-Level OPD Only increases Pass@1 from 21.11% to 27.25% and test applicability from 45.68% to 51.96%, while reducing the average number of interaction turns from 46.10 to 35.70. This improvement is achieved without any subsequent correction training, indicating that supervision at the initial planning stage already leads to more effective issue reproduction with substantially shorter trajectories.

Localized expert correction benefits from both progress-guided intervention and selective supervision. Replacing progress-guided localization with teacher-selected intervention points reduces Pass@1 from 32.24% to 27.94%, while ΔCoverage and test applicability decrease by 3.87 and 6.79 percentage points, respectively. Similarly, applying SFT to complete successful expert takeover suffixes instead of the selected corrective steps reduces Pass@1 to 28.41%. Δ Coverage decreases from 34.90% to 28.48%, while test applicability falls from 56.44% to 46.19%. These results show that effective expert correction depends on both localizing appropriate intervention states and restricting supervision to the corrective steps retained by LOEC.

More extensive preparation accompanies more reliable initial test construction. The full design uses an average of 6.85 more interaction turns than Plan-Level OPD Only. To better understand this difference, we examine how successful trajectories allocate interaction turns across different stages of issue reproduction. Within this subset, LOEC spends an average of 16.00 turns on preparation before initial test construction, compared with 9.69 turns for Plan-Level OPD Only. We further ex-

amine whether this additional preparation is associated with more reliable initial tests. For LoEC, the first complete version of the ultimately submitted test already reproduces the issue without further edits in 58.45% of successful trajectories, compared with 45.76% for Plan-Level OPD Only.

4.5 Qualitative Analysis

We conduct an exploratory qualitative comparison of generated trajectories from the base student and LoEC. We observe three recurring failure patterns of the base student that are addressed by the resulting model of LoEC. We illustrate each pattern with a representative example below. Detailed trajectories and tests generated by both models are provided in Appendix G.

LoEC focuses reproduction on the reported bug. The base student sometimes focuses on behavior unrelated to the target defect, producing failing tests that do not actually reproduce the reported bug. For example, on `matplotlib-24177`, scaling the input exposes a bug where the y-axis does not expand enough to display the full step-style histogram. The base student misinterprets this setup and instead checks whether the histogram bin edges remain unchanged after scaling, which fails regardless of whether the bug is fixed. In contrast, LoEC compares the histogram height with the y-axis upper bound and directly captures the clipping behavior.

LoEC avoids hidden errors in reproduction tests. Even after triggering the reported bug, the base student sometimes accepts the test too early without checking whether later inputs or assertions are also correct. For example, on `sklearn-14496`, the base student tests several `min_samples` values in sequence. The first input already triggers the target `TypeError`, causing execution to stop before reaching a later invalid value of 2.5, which is outside the allowed range for a floating-point `min_samples` value. The student therefore regards the test as successful without exposing this hidden test error. After the target bug is fixed, the hidden error becomes visible.

LoEC carries valid reproductions through to submission. The base student sometimes continues exploring even after it has already constructed a valid reproduction, preventing the completed test from being formally submitted. For example, on `matplotlib-25287`, the base student creates valid tests but then repeatedly reruns the same test command. Such repetitive, non-progressing action loops have been observed as a recurring failure pattern in software-engineering agents (Yang et al., 2026; Bouzenia & Pradel, 2025). In contrast, LoEC checks the target failures, inspects the final test diff, and successfully submits the reproduction. Across our evaluation, LoEC reduces the frequency of more than 10 repeated command execution by 16.79% relative to the base student.

5 Conclusion

We presented LoEC, a three-stage post-training framework for repository-level issue reproduction that concentrates expert supervision on task-relevant decisions and recovery behaviors. LoEC first shapes the initial strategy through plan-level on-policy distillation, then localizes intervention states using issue-reproduction progress signals, and finally corrects recoverable failures through recovery-plan distillation and selective supervision of verified expert continuations. On SWT-Bench Verified, LoEC improves the Pass@1 of the base student from 21.11% to 32.24%. Our stage-wise comparisons further show that LoEC outperforms imitating all assistant steps. Ablation results confirm that all our designs contribute to the final performance. Overall, these results suggest that effective expert supervision that focuses on strategic decisions and verified recovery behaviors can provide a more effective and supervision-efficient training signal.

Limitations

Our study has several limitations. First, our experiments use a single student–expert model pair and evaluate on SWT-Bench Verified. The effectiveness of LoEC across other models remains to be evaluated. Second, stall localization is performed retrospectively on completed student trajectories, and expert correction relies on task-specific progress signals and execution-based reproduction verification. We have not investigated whether using an LoEC-trained student as the initialization for subsequent RL can achieve further gains compared with applying RL directly to the base student. Finally, we measure supervision efficiency through the number of tokens included in the training objective, rather than end-to-end computational cost.

Reproducibility Statement

Our experiments use publicly available models and benchmarks: Qwen3.5-4B as the student, Qwen3.5-27B as the expert, SWE-Gym as the source of training instances, and SWT-Bench Verified for evaluation. Our source code and trained model are publicly available at https://anonymous.4open.science/status/replication-BD53. The code includes trajectory collection in SWE-Gym, test-quality evaluation, and the source code of LoEC. To improve reproducibility, we enable deterministic inference and use greedy decoding during evaluation.

AI Use Statement

In this work, we used generative AI tools to assist with polishing the manuscript and improving its readability and linguistic presentation. We carefully reviewed all AI-assisted revisions and take full responsibility for the final content of the manuscript. Generative AI tools were not used for any tasks categorized as **Required** for disclosure in the Author Guide.[2]

References

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

Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate SWE patches. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=b0jYs6JOZu.

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Heterogeneous prompting and execution feedback for swe issue test generation and selection. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, ICSE '26, pp. 1250–1261, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400720253. doi: 10.1145/3744916.3787837. URL https://doi.org/10.1145/3744916.3787837.

Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 2846–2857. IEEE Press, 2025. doi: 10.1109/ASE63991.2025.00234. URL https://doi.org/10.1109/ASE63991.2025.00234.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning, 2023. URL https://arxiv.org/abs/2310.05915.

Yuqian Fu, Haohuan Huang, Kaiwen Jiang, Jiacai Liu, Zhuo Jiang, Yuanheng Zhu, and Dongbin Zhao. Revisiting on-policy distillation: Empirical failure modes and simple fixes, 2026. URL https://arxiv.org/abs/2603.25562.

Longkun Hao, Hongyu Lin, Hao Li, Zhuowen Liu, Zhichao Yang, Haojie Hao, Dongshuo Huang, Haitao Yang, Hongyu Ge, Mingjie Xie, Yanjun Wu, Zihao Yin, Yan Bai, and Yihang Lou. Speculative rollback correction for quality-diverse web agent imitation. *arXiv preprint arXiv:2606.12485*, 2026. URL https://arxiv.org/abs/2606.12485.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pp. 2312–2323. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00194. URL https://doi.org/10.1109/ICSE48619.2023.00194.

[2] https://iclr.cc/Conferences/2027/AIPolicyForAuthors

Lara Khatib, Noble Saji Mathews, and Meiyappan Nagappan. Assertflip: Reproducing bugs via inversion of llm-generated passing tests, 2026. URL https://arxiv.org/abs/2507.17542.

Konstantinos Kitsios, Marco Castelluccio, and Alberto Bacchelli. Automated generation of issue-reproducing tests by combining llms and search-based testing. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1982–1994. IEEE Press, 2025. doi: 10.1109/ASE63991.2025.00165. URL https://doi.org/10.1109/ASE63991.2025.00165.

Niklas Lauffer, Xiang Deng, Srivatsa R Kundurthy, Brad Kenstler, and Jeff Da. Imitation learning for multi-turn LM agents via on-policy expert corrections. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=l0LA4SUA50.

Changhao Li, Rushi Qiang, Jiawei Huang, Chenxiao Gao, Chao Zhang, Niao He, and Bo Dai. Revisiting DAgger in the era of LLM-agents. *arXiv preprint arXiv:2605.12913*, 2026a. URL https://arxiv.org/abs/2605.12913.

Yaxuan Li, Yuxin Zuo, Bingxiang He, Jinqian Zhang, Chaojun Xiao, Cheng Qian, Tianyu Yu, Huan ang Gao, Wenkai Yang, Zhiyuan Liu, and Ning Ding. Rethinking on-policy distillation of large language models: Phenomenology, mechanism, and recipe, 2026b. URL https://arxiv.org/abs/2604.13016.

Shuyang Liu, Saman Dehghan, Jatin Ganhotra, Martin Hirzel, and Reyhaneh Jabbarvand. From plan to action: How well do agents follow the plan?, 2026. URL https://arxiv.org/abs/2604.12147.

Zhengyao Liu, Yunlong Ma, Jingxuan Xu, Junchen Ai, Xiang Gao, Hailong Sun, and Abhik Roychoudhury. Agent that debugs: Dynamic state-guided vulnerability repair, 2025. URL https://arxiv.org/abs/2504.07634.

Yuanjie Lyu, Chengyu Wang, Jun Huang, and Tong Xu. Student-centered distillation narrows the agentic gap between small and large LLMs. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=KaTYG9LGJv.

Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. SWT-bench: Testing and validating real-world bug-fixes with code agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=9Y8zUO11EQ.

Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. Issue2test: Generating reproducing test cases from issue reports, 2026. URL https://arxiv.org/abs/2503.16320.

OpenAI. Introducing SWE-bench Verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 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*, pp. 47717–47737, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

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

Idan Shenfeld, Mehul Damani, Jonas Hübotter, and Pulkit Agrawal. Self-distillation enables continual learning. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=qA6FgH0nnZ.

Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. Trial and error: Exploration-based trajectory optimization of LLM agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7584–7600, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.409. URL https://aclanthology.org/2024.acl-long.409/.

Aditya Bharat Soni, Rajat Ghosh, Vaishnavi Bhargava, Valerie Chen, and Debojyoti Dutta. Swetester: Training open-source llms for issue reproduction in real-world repositories, 2026. URL https://arxiv.org/abs/2601.13713.

Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. Aegis: An agent-based framework for bug reproduction from issue descriptions. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, pp. 331–342, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712760. doi: 10.1145/3696630.3728557. URL https://doi.org/10.1145/3696630.3728557.

Xu Xia, Jinghua Piao, Min Yang, Xiaochong Lan, Jiaju Chen, and Yong Li. From scoring to acting: Outcome-verified comparative self-distillation for llm agents, 2026. URL https://arxiv.org/abs/2607.27937.

Yiqing Xie, Emmy Liu, Gaokai Zhang, Nachiket Kotalwar, Shubham Gandhi, Sathwik Acharya, Xingyao Wang, Carolyn Rose, Graham Neubig, and Daniel Fried. Hybrid-gym: Training coding agents to generalize across tasks. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=2TVSJxxz9u.

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

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=63iVrXc8cC.

Junze Ye, Jiayi Cheng, Miao Lu, Michal Mankowski, Jose Blanchet, and Mohsen Bayati. A few teacher steps go a long way: Cost-efficient on-policy data augmentation for agent post-training, 2026. URL https://arxiv.org/abs/2607.04574.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Yiqi Zhu, Apurva Gandhi, and Graham Neubig. Training versatile coding agents in synthetic environments, 2026. URL https://arxiv.org/abs/2512.12216.

A Plan-Shaping Optimization Details

We provide additional optimization details for the plan-level OPD described in Section 3.1. For each rollout batch, the student generates one initial plan per training instance, and the expert only scores the student-sampled tokens rather than generating a separate response.

Student sampling and expert scoring. Let π_{old} denote the frozen student policy used to collect the current batch of plans and π_θ the student policy being updated. For each instance, we sample one planning response from π_{old}. Both π_{old} and π_E then evaluate the same sampled response token by token. Only tokens belonging to the planning response are included during optimization.

Token-level OPD signal. For each sampled planning token y_t, we compute the student–expert log-probability ratio

$$k_t = \log \pi_{\text{old}}(y_t \mid x, y_{<t}) - \log \pi_E(y_t \mid x, y_{<t}). \tag{6}$$

To limit the influence of extreme student–expert log-probability differences, we clip k_t to $[-10, 10]$ before constructing the token-level distillation advantage:

$$\hat{A}_t^{\text{OPD}} = -\,\text{sg}\left[\text{clip}(k_t, -10, 10)\right], \tag{7}$$

where $\text{sg}(\cdot)$ denotes the stop-gradient operator. Intuitively, a sampled token receives a positive learning signal when the expert assigns it a higher probability than the student, and a negative signal in the opposite case.

PPO-clipped update. During optimization, we compute the importance ratio between the updated student policy and the rollout policy:

$$r_t(\theta) = \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{old}}(y_t \mid x, y_{<t})}. \tag{8}$$

We optimize the student using a PPO-style clipped surrogate objective:

$$\mathcal{L}_{\text{OPD}}(\theta) = -\frac{1}{\sum_{b,t} m_{b,t}} \sum_{b,t} m_{b,t} \min\left(r_{b,t}(\theta)\hat{A}_{b,t}^{\text{OPD}}, \text{clip}\left(r_{b,t}(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_{b,t}^{\text{OPD}}\right), \tag{9}$$

where $m_{b,t}$ indicates valid planning-response tokens and ϵ denotes the PPO clipping coefficient. The clipped policy ratio constrains each update relative to the rollout policy, preventing excessively large policy shifts.

Loss aggregation. We use one sampled plan per instance and aggregate the objective with a token mean over all valid response tokens in the batch. Thus, each planning token receives equal weight, while longer plans contribute proportionally more tokens to the batch objective. We do not use an environment reward, task-level reward, or an additional KL penalty against a separate reference policy during this stage. The OPD estimator requires only the student and expert log probability of each sampled token and does not require computing a full-vocabulary KL divergence.

B Stall Definitions and Intervention-State Selection

We identify intervention points retrospectively after a student trajectory has completed. Since a stall may only become evident after several later actions or test revisions, we select an earlier state near the onset of the stalled behavior for expert takeover.

B.1 Candidate-Test Evaluation

Let s_t denote the repository state after the first t agent actions, and let $P(s_t)$ denote the candidate test patch present at that state. We evaluate the test items introduced or modified by $P(s_t)$, which we refer to as *candidate-owned test items*.

Four-state evaluation. For each candidate test patch, the evaluator compares the behavior of candidate-owned test items across four repository states. Let B_0 and G_0 denote the buggy and gold-fixed repositories without the candidate test patch, respectively, and let B_1 and G_1 denote the corresponding repositories after applying the candidate test patch.

Verification procedure. The evaluator progressively checks whether the candidate test provides evidence for reproducing the target issue. It first verifies that the candidate-owned test items can be collected and executed on both B_1 and G_1.

It then examines the execution trace on B_1 to determine whether the candidate test reaches code affected by the gold patch. The reachability check becomes progressively more specific: whether execution reaches a file modified by the gold patch, enters a function containing a modified region, and finally reaches a modified line of the gold patch.

The evaluator next compares the outcome of the same candidate-owned test item on B_1 and G_1. A stable difference between the two indicates that the gold fix changes the behavior exercised by the candidate test. At the highest verification levels, we further determine whether this behavior constitutes a newly introduced Fail-to-Pass reproduction.

Outcome transitions. For aligned test items, let $\mathcal{T}_X^{(k)}$ denote the set of items with outcome transition X under $B_k \to G_k$, where $k \in \{0, 1\}$ and $X \in \{\mathrm{F2P}, \mathrm{P2F}, \mathrm{F2F}\}$. Here, F2P, P2F, and F2F denote fail-to-pass, pass-to-fail, and fail-to-fail, respectively.

The comparison $B_0 \to G_0$ represents the baseline behavior without the candidate patch, while $B_1 \to G_1$ represents the behavior after the candidate patch is applied. We define transitions newly introduced by the candidate patch as

$$\Delta\mathcal{T}_X = \mathcal{T}_X^{(1)} \setminus \mathcal{T}_X^{(0)}. \tag{10}$$

Thus, an F2P transition that already exists in the baseline does not count as a new reproduction introduced by the candidate patch.

Verification levels and test-to-patch distance. We summarize reproduction progress using ordered verification levels L0–L7. Higher levels indicate stronger evidence that the candidate test reaches and reproduces the target issue. In particular, L6 establishes a baseline-relative F2P reproduction, while L7 corresponds to the final strict-clean reproduction criterion.

Let $L(s) \in \{0, \ldots, 7\}$ denote the verification level at state s. We define the verification-aware test-to-patch distance as

$$d_{\mathrm{tp}}(s) = 7 - L(s). \tag{11}$$

A smaller value therefore indicates stronger reproduction progress, with $d_{\mathrm{tp}}(s) = 0$ corresponding to a successful L7 reproduction.

Table 4: Verification levels used to measure issue-reproduction progress. Higher levels indicate stronger evidence of a valid reproducer, with L7 representing successful strict-clean reproduction.

Level	d_{tp}	**Operational criterion**
L0	7	No executable candidate test items are obtained; the candidate patch cannot be applied; or the candidate tests cannot be validly executed on both B_1 and G_1.
L1	6	Candidate-owned test items can be collected and executed on both B_1 and G_1.
L2	5	On B_1, execution of candidate-owned tests reaches a file modified by the gold patch.
L3	4	On B_1, execution enters a function containing a region modified by the gold patch.
L4	3	On B_1, execution reaches a modified line of the gold patch.
L5	2	The same candidate-owned test item exhibits a stable but different outcome on B_1 and G_1. The difference need not yet be Fail-to-Pass.
L6	1	The candidate patch introduces at least one new F2P transition, $\Delta\mathcal{T}_{\mathrm{F2P}} \neq \varnothing$, introduces no new P2F transition, $\Delta\mathcal{T}_{\mathrm{P2F}} = \varnothing$, and preserves all baseline F2P transitions, $\mathcal{T}_{\mathrm{F2P}}^{(0)} \subseteq \mathcal{T}_{\mathrm{F2P}}^{(1)}$.
L7	0	The L6 criterion holds and the candidate patch introduces no new F2F transitions, $\Delta\mathcal{T}_{\mathrm{F2F}} = \varnothing$. We regard L7 as a successful strict-clean reproduction.

B.2 Stall Detection

Using the verification progress defined above, we identify two forms of temporal stall (test-edit inactivity and test-quality plateau) and one terminal trigger, incorrect submission. We define $\phi(s_t)$ as a representation of the effective test workspace at state s_t.

Test-edit inactivity. We identify a test-edit inactivity stall when the effective test workspace remains unchanged over $H = 20$ consecutive non-submission actions. A segment beginning at state s_u satisfies

$$\phi(s_u) = \phi(s_{u+1}) = \cdots = \phi(s_{u+H}), \qquad H = 20. \tag{12}$$

Any effective test edit resets the inactivity counter. A submission terminates the scan and is not counted as one of the H actions. We mark the result as indeterminate when the required repository state is missing or when multiple commands in a single response cannot be uniquely aligned with state transitions.

Test-quality plateau. Let

$$r_1 < r_2 < \cdots < r_M$$

denote the interaction steps at which the student produces distinct, non-empty candidate test revisions with valid evaluator results. For revision i, define

$$d_i = d_{\mathrm{tp}}(s_{r_i}), \tag{13}$$

and let

$$d_i^\star = \min_{j \le i} d_j \tag{14}$$

denote the best, i.e., smallest, verification-aware distance achieved up to revision i.

We detect a test-quality plateau when $K = 3$ consecutive revisions fail to improve upon the previously achieved best distance. A plateau ending at revision m therefore satisfies

$$d_i \ge d_{i-1}^\star, \qquad i = m - K + 1, \ldots, m, \qquad K = 3. \tag{15}$$

A decrease in d_i establishes a new historical best and resets the plateau counter. Repeated evaluation of an unchanged candidate test patch does not constitute a new revision. An invalid or unreadable evaluation interrupts the consecutive sequence and is not assigned a numerical distance. In contrast, a valid L0 evaluation has $d_{\mathrm{tp}} = 7$ and remains part of the sequence.

Incorrect submission. We additionally identify an incorrect submission when the student submits a candidate patch without establishing a valid L7 certificate for that submitted revision, or when the submitted patch receives a valid evaluation below L7.

If verification of the submitted patch is unavailable because the evaluator result is invalid or unreadable, we mark the verification status as indeterminate rather than converting it to L0. Incorrect submission is treated as a terminal intervention trigger rather than a temporal stall.

B.3 Retrospective Intervention-State Selection

After identifying a trigger, we trace the corresponding behavior back to an earlier state and use that state as the expert takeover point. Table 5 summarizes the selection rule for each trigger.

Table 5: Retrospective intervention-state selection for each trigger.

Trigger	Selected intervention state
Test-edit inactivity	After the first action in the identified inactive segment and before its second action.
Test-quality plateau	Immediately after the first valid test edit associated with the localized plateau and before the following action.
Incorrect submission	Immediately before the student's submission action.

C EVALUATION PROMPT

We use the following task prompt for issue-reproduction evaluation, based on the SWE-Agent prompt in the publicly released SWT-Bench trajectories (Mündler et al., 2024), which is used by the SWE-Agent baseline with GPT-4. The detailed issue description is inserted at the placeholder.

Evaluation Prompt

We have received the following issue within our repository. Here's the issue text: [ISSUE TEXT]

INSTRUCTIONS:
Now, you're going to create unit tests that cover the issue. In other words, you should write unit tests that fail in the current state of the repository
but will pass when the issue has been resolved. Essentially, you'll want to write a unit test that reproduces the described issue.
Your terminal session has started and you're in the repository's root directory. Use standard non−interactive bash commands to inspect the repository, modify files, and run checks or tests.
Remember, YOU CAN ONLY ENTER ONE COMMAND AT A TIME. You should always wait for feedback after every command.
When you're satisfied with all of the changes you've made, you can submit your changes to the code base by simply running the submit command.
Note however that you cannot use any interactive session commands (e.g. python, vim) in this environment, but you can write scripts and run them. E.g. you can write a python script and then run it with 'python <script_name>.py'.

When writing a multi−line source file from the shell, preserve real newlines and indentation; do not accidentally collapse the whole file into one line containing escaped '\n' separators. Inspect the resulting file after every modification.

IMPORTANT TIPS:
1. Always start by trying to replicate the bug that the issue discusses.
If the issue includes code for reproducing the bug, we recommend that you re−implement that in your environment, and run it to make sure you can reproduce the bug.
Then write a focused unit test for that behavior.

2. If you run a command and it doesn't work, try a different command. A command that did not work once will not work the second time unless you modify it.

3. If the bug reproduction requires a file, search the repository with standard bash tools such as 'find' or 'rg' to locate existing examples and the correct test layout.

4. Run the repository's actual test entry point and confirm that the new test is discovered. A zero exit status from an unrelated command or a command that discovers no tests is not validation.

5. Before submitting, inspect both the modified file and 'git diff' to confirm that the patch contains valid source code and only the intended unit−test changes.

For this SWT−Bench run, submit only the unit−test diff. Do not include source−code fixes in the submitted patch.

In this mini−swe−agent environment, the submit command is:

```
mswea_bash_command
cd /testbed && git diff −− . ':(exclude)patch.txt' ':(exclude)*.pyc' ':(exclude)__pycache__' > patch.txt
&& echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
```

(Current directory: /testbed)

D TRAINING CONFIGURATION

Models and checkpoint initialization. We use Qwen3.5-4B as the student and Qwen3.5-27B as the teacher. Training proceeds through initial-plan on-policy distillation (OPD), recovery-plan OPD at selected takeover states, and localized LoRA SFT. Initial-plan OPD starts from the original Qwen3.5-4B checkpoint. Its checkpoint at optimizer update 259 initializes recovery-plan OPD, whose checkpoint at update 57 then initializes localized LoRA SFT. Both OPD procedures update all student parameters. Localized SFT updates only a newly initialized LoRA adapter, while keeping the underlying model parameters frozen. We create a new optimizer at each transition. **Optimization settings.** Table 6 summarizes the training data, batch sizes, optimization settings, and sequence-length limits for the three training procedures. Epochs denote configured training budgets, while optimizer updates refer to the checkpoints used in the training sequence described above. We use a one-epoch budget for initial-plan OPD as training diagnostics show substantial early alignment improvement but only marginal changes in the student–teacher log-probability discrepancy over the final 100 updates.

Table 6: Optimization settings for LOEC.

Setting	Initial-plan OPD	Recovery-plan OPD	Localized LoRA SFT
Training inputs	1,034 prompts	306 states	269 trajectories
Global batch size	4	16	16
Configured epochs	1	3	3
Optimizer updates	259	57	16
Optimizer	AdamW	AdamW	AdamW
Learning rate	1×10^{-6}	1×10^{-6}	5×10^{-6}
Weight decay	0.01	0.01	0.1
Adam β_1	0.9	0.9	0.9
Adam β_2	0.999	0.999	0.999
Learning-rate schedule	Constant	Constant	Cosine
Warmup steps	0	0	2
Maximum gradient norm	1.0	1.0	1.0
Precision	BF16	BF16	BF16
Maximum prompt length	32,768	32,768	65,536
Maximum sequence length	33,793	33,793	65,536

Plan-level on-policy distillation. In both OPD procedures, the current student samples one plan per input online. The teacher computes log probabilities for the same student-generated tokens under their corresponding prefixes. We use K1 policy-gradient distillation, with the distillation signal clipped to $[-10, 10]$ and a PPO clipping coefficient of 0.2. Only generated plan tokens contribute to the loss, which uses token-mean aggregation. Table 7 reports the shared sampling and loss settings.

For recovery-plan OPD, the student and teacher receive the same pre-intervention history and planning instruction. Successful teacher continuations are used to select takeover states, but are not supplied to the teacher as additional reference answers when scoring student plans.

Localized LoRA supervision. Localized SFT uses 269 recorded trajectories and offline teacher actions. Each trajectory retains its complete causal history as context, with a maximum sequence length of $65{,}536$ tokens. Only tokens belonging to selected teacher actions contribute to the loss. Student history, tool outputs, unselected teacher actions, and padding are masked from the loss. We first average the negative log-likelihood over the selected target tokens within each trajectory, and then average these per-trajectory losses across the batch. Thus, the 269 trajectories provide context for localized supervision; they are not supervised in full. The LoRA adapter uses rank 8, scaling parameter $\alpha = 16$, and dropout 0.

Data ordering. For initial-plan OPD, prompts are grouped into batches by length, and the batch order is shuffled once and fixed before training. Recovery-plan OPD samples states without replacement and reshuffles them each epoch. Localized LoRA SFT samples trajectories without replacement. Both OPD procedures additionally use seed 42 for rollout and FSDP execution.

Table 7: Sampling and loss settings shared by both OPD procedures.

Setting	Value
Student plans sampled per input	1
Sampling temperature	1.0
Top-p	1.0
Top-k truncation	Disabled
Thinking mode	Disabled
Distillation objective	K1 policy gradient
Distillation-signal clipping	$[-10, 10]$
PPO clipping coefficient	0.2
Supervised tokens	Generated plan tokens only
Loss aggregation	Token mean
Task reward	Disabled
Additional reference-policy KL penalty	Disabled
Entropy bonus	Disabled
Rollout/FSDP seed	42
Gradient checkpointing	Enabled

Distributed execution. Each OPD procedure uses four GPUs: two for the student with FSDP2, and two for the teacher with tensor parallelism of degree two. Gradient checkpointing is enabled for OPD. Localized LoRA SFT uses four GPUs with DeepSpeed ZeRO-3.

E Baseline Details

E.1 OEC

Trajectory collection. We adapt OEC (Lauffer et al., 2026) to issue reproduction on SWE-Gym. Following its 7B-scale setup, we perform three independent collection passes over SWE-Gym. All three passes use Qwen3.5-4B student and Qwen3.5-27B expert. For each rollout, we sample a single switch step $s \sim \mathcal{U}\{0, \ldots, 30\}$. The student generates the trajectory prefix up to step s, after which the expert continues from the same interaction history and repository state. Each combined rollout follows the shared limit of 100 interaction steps.

Trajectory filtering. We adapt the reject criterion of OEC to issue reproduction and retain only trajectories with valid final tests. A retained patch must introduce at least one new Fail-to-Pass test, preserve existing Fail-to-Pass behavior, and introduce no new Pass-to-Fail or Fail-to-Fail tests. Following the repetition filters of OEC, we additionally remove trajectories that contain either three consecutive identical actions or at least 20 consecutive file-reading actions. After filtering and deduplication, the three collection passes yield **2,412** OEC trajectories.

Behavioral-cloning mixture. Following the 7B experiments of Lauffer et al. (2026), we supplement the OEC trajectories with behavioral-cloning (BC) data. BC trajectories are generated entirely by the Qwen3.5-27B expert from the initial state of each training instance and are subjected to the same issue-reproduction verification and repetition filtering. We combine the retained data using a BC-to-OEC ratio of 1:3, resulting in **804** BC trajectories.

Optimization. During supervised fine-tuning, the student-generated prefix of each OEC trajectory is retained as context and excluded from the loss. Only expert-generated suffix tokens receive supervision. For BC trajectories, the loss is applied to all expert-generated assistant turns. The three collection passes are merged into a single training set, and the student is fine-tuned once from the original Qwen3.5-4B checkpoint. Because our 4B student is closest in scale to the 7B configuration, we adopt its supervised fine-tuning hyperparameters, summarized in Table 8.

Table 8: Hyperparameters used to train OEC baseline (7B configuration).

Hyperparameter	Value
Batch size	64
Training epochs	3
Learning rate	5×10^{-5}
Warmup steps	5
Maximum sequence length	32,768
Maximum gradient norm	1.0
Optimizer	Adam
Adam β_1	0.9
Adam β_2	0.999
Learning-rate schedule	Cosine

E.2 SCoRe

After removing cases for the BC cold-start stage and student-independently solved cases without any modification, we partitioned the remaining 1,870 tasks eligible for teacher correction into two disjoint subsets of 935 tasks each, designated for SFT and RL, respectively.

E.2.1 SFT Stage.

On the SFT subset, the Qwen3.5-27B teacher helped the student successfully complete 383 tasks within at most five rounds of correction. We used these 383 final successful trajectories for SFT. Following the SFT hyperparameters of SCoRe, we fine-tuned Qwen3.5-4B for three epochs. The training hyperparameters are reported in Table 9.

Table 9: Hyperparameters used for SFT of the SCoRe baseline (4B configuration).

Hyperparameter	Value
Global batch size	128
Training epochs	3
Learning rate	7×10^{-6}
Warmup ratio	0.1
Warmup steps	1
Maximum sequence length	65,536
Maximum gradient norm	1.0
Optimizer	AdamW
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	10^{-8}
Weight decay	0.1
Learning-rate schedule	Cosine
Precision	BF16
DeepSpeed stage	ZeRO-3

E.2.2 RL Stage.

For the RL stage, among the 935 tasks processed by the teacher-guided correction pipeline, 361 tasks are solved after takeover. Following Lyu et al. (2026), we supplemented the corrected tasks with easy and hard examples. For the easy subset, applying an independent Bernoulli retention probability of 0.1 selected 28 easy tasks. For the hard subset, the sample size was chosen so that hard tasks constituted approximately 10% of the final RL mixture ($43/432 = 9.95\%$). Consequently, 361 tasks entered the RL set directly, whereas 71 tasks entered through the two sampling procedures.

We initialized the actor from the Qwen3.5-4B checkpoint obtained after three epochs of SFT and optimized it using GRPO. At each RL iteration, we sampled 16 tasks and generated 16 independent

rollouts per task, yielding 256 on-policy trajectories. With 432 training tasks, this results in 27 RL iterations per epoch.

We largely followed the original SCoRE RL configuration, with two modifications to accommodate the longer interaction histories and multi-step nature of our tasks. Specifically, we increased the maximum model sequence length from 8K to 81,920 tokens and the maximum number of policy calls per trajectory from 8 to 32, thereby reducing premature truncation of long-horizon trajectories. To offset the additional computational cost introduced by these larger trajectory budgets, we reduced training from three epochs to one. Even under this reduced training schedule, the resulting training-update FLOPs remain 7.59× those of LoEC. Unless otherwise noted, all remaining optimization hyperparameters follow the original SCoRE. The configuration is summarized in Table 10.

Table 10: Hyperparameters used for RL optimization of the SCoRE baseline (4B configuration).

Hyperparameter	**Value**
RL algorithm	GRPO
Number of training tasks	432
Training epochs	1
Task batch size per RL iteration	16
Rollouts per task	16
Total RL iterations	27
PPO epochs per iteration	1
Learning rate	7×10^{-7}
Learning-rate schedule	Constant
Warmup steps	0
Optimizer	AdamW
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	10^{-8}
Weight decay	0.01
Maximum gradient norm	1.0
PPO clipping ratio	0.2
Entropy coefficient	0
KL reward penalty	Disabled
KL loss	Disabled
Advantage normalization	Group-standard-deviation normalization
Policy-loss aggregation	Token mean
Maximum prompt length	65,536
Maximum response length	16,384
Maximum model sequence length	81,920
Maximum policy calls per trajectory	32
Rollout temperature	1.0

A rollout receives reward 1.0 if it normally submits a patch that reproduces the issue. For an unsuccessful corrected-task rollout, following the settings of SCoRE, we compare its first newly generated action with the teacher-chosen action and then with the rejected action using an external Qwen3.5-Flash verifier. Functional equivalence to the chosen action receives 0.5, equivalence to the rejected action receives 0, and equivalence to neither receives 0.1. Normal unsuccessful easy and hard rollouts also receive 0.1.

F Evaluation Dataset Details

Dataset composition and training-set separation. We use SWT-Bench Verified (Mündler et al., 2024), which adapts instances from the human-validated SWE-bench Verified subset to issue reproduction (OpenAI, 2024). The released split contains 433 instances from 12 Python repositories. Table 11 summarizes their distribution, with counts computed from the released instance metadata. The repositories cover areas such as web development, scientific computing, visualization, and developer tools. None of these repositories appear in the SWE-Gym training instances used in our experiments.

Table 11: Repository distribution of SWT-Bench Verified.

Repository	Application area	Instances	Share
`django/django`	Web framework	216	49.88%
`sympy/sympy`	Symbolic mathematics	73	16.86%
`matplotlib/matplotlib`	Visualization	32	7.39%
`sphinx-doc/sphinx`	Documentation generation	28	6.47%
`scikit-learn/scikit-learn`	Machine learning	24	5.54%
`astropy/astropy`	Astronomical computing	17	3.93%
`pydata/xarray`	Labeled multidimensional arrays	15	3.46%
`pytest-dev/pytest`	Testing framework	15	3.46%
`pylint-dev/pylint`	Static analysis	6	1.39%
`psf/requests`	HTTP client	4	0.92%
`mwaskom/seaborn`	Statistical visualization	2	0.46%
`pallets/flask`	Web framework	1	0.23%
Total		**433**	**100.00%**

Original repair difficulty. The dataset includes difficulty labels inherited from SWE-bench Verified. These labels estimate the time an experienced developer would need to understand the issue and implement a fix, after a few hours of familiarization with the codebase (OpenAI, 2024). There are 172 instances labeled as requiring less than 15 minutes (39.72%), 225 requiring 15 minutes to 1 hour (51.96%), and 36 requiring 1 to 4 hours (8.31%).

Evaluation protocol. For each instance, the agent receives the natural-language issue description and accesses the repository at the specified pre-fix commit. It can inspect the code and existing tests, write tests, and execute commands in the task environment. The gold patch is withheld from the agent during generation and is used only for evaluation. We evaluate submitted test patches using the SWT-Bench harness. A successful submission must add at least one test that fails before the gold fix and passes afterward, without introducing additional test failures after the fix.

G Qualitative Case Studies

We present three examples of issue-reproduction trajectories and model-generated tests. These examples are selected from cases where LoEC addresses specific failure patterns of the base student.

G.1 Target-aligned assertions: `matplotlib-24177`

> **Issue report:** *ax.hist density not auto-scaled when using histtype='step'*
> I need to plot a histogram of some data (generated by `numpy.save` in binary format) from my work using the `matplotlib.axes.Axes.hist` function. I noted that the histogram's density axis (when setting `density=True`) is not automatically adjusted to fit the whole histogram.
> I played with different combinations of parameters, and noted that the densities changes if you rescale the whole data array, which is counterintuitive as rescaling the data should only affect the x-axis values. I noted that if you set `histtype="step"`, the issue will occur, but is otherwise okay for other `histtypes`.
> ...

Reported behavior. With `density=True` and `histtype="step"`, the y-axis may not expand enough to show the full histogram.

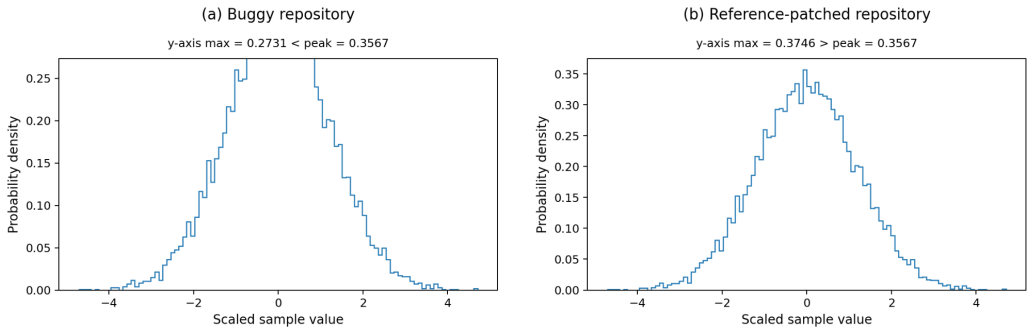

Figure 2: Illustration of the reported clipping issue. In the left panel, the peak is above the y-axis limit, while in the right panel it is fully visible.

Trajectory comparison. We compare the trajectories generated by the base student and LoEC on this issue. The base student shifts its focus from the reported clipping issue to an unrelated scaling property, and therefore constructs tests that do not directly check the reported behavior. In contrast, LoEC compares the histogram height with the y-axis limit to test whether the histogram is clipped.

Base student: tests an unrelated property

Action 13. Construct the tests.
The student assumes: "*rescaling data should only affect the x-axis values, not the density.*" Its tests compare entry `[1]` of the histogram outputs before and after scaling; this entry contains bin edges.

Action 15. Interpret the failure.
"*The test confirms the bug exists.*"

Action 30, 32. Run the tests.
The bar-style variant also fails; running both new tests yields two failures.

Action 36. Submit.
The student submits both tests, predicting that they will pass after the fix.

LoEC: checks the reported clipping symptom

Action 16. Inspect height and axis limits.
Observed: diagnostic output:

```
Max density value:  0.356750
Y-axis limits:  0.000000 to 0.273137
```

Action 26. Select the test condition.
"*the y-axis limits are not automatically adjusted to fit the histogram data.*"

Action 32. Run the repository test.
The height-versus-axis assertion fails on the buggy repository.

Action 36. Submit.
The student submits the generated test.

Figure 3: Selected trajectory excerpts for `matplotlib-24177`.

Generated tests. Both models modify `lib/matplotlib/tests/test_axes.py`.

Base student: test with `histtype="step"`

```python
def test_hist_density_step_scaled_data():
    np.random.seed(42)
    data = np.random.randn(10000)
    fig, ax = plt.subplots(1, 2, figsize=(10, 5))
    hist_original = ax[0].hist(data, bins=100, density=True, histtype="
    step")
    scale = 1.2
    data_scaled = data * scale
    hist_scaled = ax[1].hist(data_scaled, bins=100, density=True, histtype
    ="step")
    density_original = hist_original[1]
    density_scaled = hist_scaled[1]
    assert_allclose(density_original, density_scaled)
    plt.close(fig)
```

LOEC: submitted clipping test

```python
def test_hist_step_density_auto_scale():
    np.random.seed(42)
    base_data = np.random.randn(10000)
    scale = 1.2
    scaled_data = base_data * scale
    fig, ax = plt.subplots()
    ax.hist(scaled_data, bins=100, density=True, histtype="step")
    y_min, y_max = ax.get_ylim()
    _, _, step_y = plt.hist(scaled_data, bins=100, density=True, histtype=
    "step")
    if isinstance(step_y, list):
        step_y = step_y[0]
    max_density = step_y.get_path().vertices[:, 1].max()
    assert max_density <= y_max * 1.1, f"Max density {max_density:.6f} is
    not within y-axis limits {y_min:.6f} to {y_max:.6f}"
```

Figure 4: Generated tests. The base student test still fails after the reference fix, while the LOEC test passes.

Why the base-student test does not reproduce the reported bug. The base student compares the entry `[1]` returned by `plt.hist` before and after rescaling the input. However, this entry contains the bin edges, which naturally change when the input data is rescaled. Therefore, the failure does not test whether the histogram is clipped by the y-axis.

How the resulting model of LOEC checks the reported behavior. LOEC directly checks the symptom described in the issue. The final generated test observes a histogram height of `0.356750`, while the y-axis extends only to `0.300000`:

Generation-time observation: LOEC, Action 32

```
AssertionError: Max density 0.356750 is not within y-axis limits 0.300000
```

Model	Outcome	Result
Base student	FAIL → FAIL	Unresolved
LOEC	FAIL → PASS	Reproduced

What this case shows. The base test failure comes from checking a property unrelated to the reported behavior. In contrast, LOEC constructs an assertion that targets the reported symptom.

Observation

A failing generated test does not necessarily reproduce the reported issue. A valid reproduction requires the failure to be caused by the reported bug, rather than by an unrelated assertion.

G.2 WHOLE-TEST VALIDITY: SCIKIT-LEARN-14496

> **Issue report (excerpt):** *[BUG] Optics float `min_samples` NN instantiation*
>
> When passing `min_samples` as a float to optics l439 & 440 execute to bring it into integer ranges, but don't convert to int:
>
> ```
> if min_samples <= 1:
>     min_samples = max(2, min_samples * n_samples)
> ```
>
> When instantiating the `NearestNeighbours` class with a float it raises due to the float (l448).
>
> ...

Reported behavior. `OPTICS` accepts a fractional `min_samples` value as a proportion of the dataset size. For 100 samples, `min_samples=0.1` should therefore correspond to 10 neighbors. However, the buggy implementation produces the floating-point value `10.0` and passes it to `NearestNeighbors`, which requires an integer and raises a `TypeError`.

Trajectory comparison. Both models encounter the reported `TypeError` during generation. However, the base student also includes an invalid input, `2.5`, as a later case in the same test. The first exception stops execution before this input is reached. In contrast, LOEC generates a different test that avoids this invalid input.

Base student: overlooks an invalid input

Action 7. Construct the test.
The test loops over `[0.1, 0.5, 1.0, 2.5]` and checks the size of `labels_` after each call to `fit`.

Action 9. Run the test.
The first call, using `0.1`, raises the reported `TypeError`. Later inputs are not reached.

Action 10. Accept the observed failure.
"The test fails as expected with the current code. Now let me verify the test file is properly formatted and then submit the changes."

Action 12. Submit.
The student submits the test with all four inputs unchanged.

LOEC: uses valid test inputs

Action 18. Construct the final test.
The test uses fractional inputs `0.1`, `0.5`, and `0.05`, plus the integer input `1`.

Action 20. Run the test.
The first call, using `0.1`, also raises the reported `TypeError` on the buggy repository.

Action 21. Probe integer conversion.
The student computes an integer neighbor count and passes it to `OPTICS`. This probe does not run the complete test on the reference-fixed repository.

Action 25. Submit.
The student inspects the test diff and submits the test.

Figure 5: Selected trajectory excerpts for `scikit-learn-14496`. Both models observe the target exception during generation.

Generated tests. Both models modify `sklearn/cluster/tests/test_optics.py`.

Base student: includes the invalid input `2.5`

```
def test_float_min_samples():
    rng = np.random.RandomState(0)
    n_samples = 100
    X = rng.randn(n_samples, 2)
    for min_samples in [0.1, 0.5, 1.0, 2.5]:
        clust = OPTICS(metric='minkowski', n_jobs=-1, min_samples=
    min_samples)
        clust.fit(X)
        assert clust.labels_.size == n_samples
```

LOEC: uses valid test inputs

```python
def test_float_min_samples():
    from sklearn.datasets import make_blobs
    X, y = make_blobs(n_samples=100, centers=3, random_state=42)
    clust = OPTICS(metric='minkowski', min_samples=0.1)
    clust.fit(X)
    assert clust.labels_ is not None
    assert len(clust.labels_) == len(X)
    clust2 = OPTICS(metric='minkowski', min_samples=0.5)
    clust2.fit(X)
    assert clust2.labels_ is not None
    assert len(clust2.labels_) == len(X)
    clust3 = OPTICS(metric='minkowski', min_samples=0.05)
    clust3.fit(X)
    assert clust3.labels_ is not None
    assert len(clust3.labels_) == len(X)
    clust4 = OPTICS(metric='minkowski', min_samples=1)
    clust4.fit(X)
    assert clust4.labels_ is not None
    assert len(clust4.labels_) == len(X)
```

Figure 6: Generated tests for `scikit-learn-14496`.

Why the base-student test still fails after the fix. On the buggy repository, the first input, `0.1`, raises the reported `TypeError`. Execution stops before the first `labels_` assertion or any later input is reached. After the reference fix, the iterations for `0.1`, `0.5`, and `1.0` complete. However, the final input, `2.5`, raises an input-validation `ValueError`. The test therefore still fails, but for a different reason from feedback not available to the student during generation.

Base student test execution results

```
Buggy:
TypeError: n_neighbors does not take <class 'float'> value, enter integer value

Reference-fixed:
ValueError: min_samples must be a positive integer or a float between 0 and 1.
    Got 2.5
```

How LOEC avoids this test error. The LOEC test uses `0.1`, `0.5`, and `0.05`, plus the integer `1`, without including the invalid input `2.5`. At Action 21, the student also computes `int(round(max(2, min_samples * n_samples)))` and passes the resulting integer.

Generation-time observation: LOEC, Action 21

```
Fixed min_samples: 10 (type: int)
```

Model	Outcome	Result
Base student	FAIL → FAIL	Not reproduced
LOEC	FAIL → PASS	Reproduced

What this case shows. Both tests initially trigger the correct exception, but their outcomes differ after the fix. The difference lies in later test inputs, which are not reached when the first call fails.

Observation

An early failure can hide errors later in the test. Triggering the reported exception does not establish that the complete test will pass after the bug is fixed.

G.3 SUCCESSFUL SUBMISSION: matplotlib-25287

> **Issue report:** *[Bug]: `offsetText` is colored based on `tick.color` instead of `tick.labelcolor`*
> In version 3.6.3, when setting `ytick.labelcolor` / `xtick.labelcolor` in styles / `rcParams`, it does not change the color of the exponent label as well. It will be colored based on `xtick.color`.
> ...

Reported behavior. Axis offset text should use the tick-label color set by `xtick.labelcolor` or `ytick.labelcolor`. Instead, the buggy version uses the tick color. When the label color is `inherit`, using the tick color is expected.

Trajectory comparison. Both models generate tests for the offset-text color. The base student repeatedly runs its unchanged tests and ends without submitting them. In contrast, LOEC recognizes two target failures and a passing control as expected on the buggy repository, then submits its tests.

Base student: repeats tests without submitting

Action 18, 21. Create and place the tests. The student writes three tests and places them in the repository's test directory. Their content remains unchanged thereafter.

Action 22. Run the test file. **Observed:** three failures: offset text expected to be red or blue is black or green instead.

Action 26. Check test discovery. **Observed:** `collected 3 items`.

Action 30-later. Repeat the same command. The student runs the identical pytest command ten consecutive times without changing the tests.

LOEC: interprets the results and submits

Action 33, 34. Run and integrate the tests. A temporary test run gives two failures and one pass. The student then adds the tests to `test_axes.py`.

Action 37. Interpret the passing control. "*All three tests are working as expected - two fail (reproducing the bug) and one passes (the inherit case works correctly).*"

Action 40, 41. Inspect and rerun. The student checks the final test diff and reruns the tests. **Observed:** `2 failed, 1 passed`.

Action 42. Submit.

Figure 7: Trajectories for `matplotlib-25287`. The base student does not submit its tests.

Generated tests. The base-student listing below is the not submitted test left in its final workspace. The LOEC listings show its submitted y-axis test and `inherit` control.

Base student: y-axis test (not submitted)

```
def test_offset_labelcolor_rcparams_yaxis():
    original_ytick_labelcolor = plt.rcParams['ytick.labelcolor']
    try:
        plt.rcParams['ytick.labelcolor'] = 'red'
        fig, ax = plt.subplots()
        ax.plot([1.01e9, 1.02e9, 1.03e9])
        offset_color = ax.yaxis.get_offset_text().get_color()
        assert offset_color == 'red', f"Expected offset color to be 'red',
 got '{offset_color}'"
    finally:
        plt.rcParams['ytick.labelcolor'] = original_ytick_labelcolor
```

LOEC: submitted y-axis test

```
def test_ytick_offset_text_color_uses_labelcolor():
    plt.rcParams['ytick.color'] = 'yellow'
    plt.rcParams['ytick.labelcolor'] = 'blue'
    fig, ax = plt.subplots()
    ax.plot([1.01e9, 1.02e9, 1.03e9])
    fig.canvas.draw()
    assert ax.yaxis.offsetText.get_color() == 'blue', \
        f"Expected offset text color to be 'blue' (labelcolor), got '{ax.
yaxis.offsetText.get_color()}'"
```

LOEC: submitted inherit control

```python
def test_offset_text_color_uses_inherit_when_labelcolor_is_inherit():
    plt.rcParams['xtick.color'] = 'yellow'
    plt.rcParams['xtick.labelcolor'] = 'inherit'
    fig, ax = plt.subplots()
    ax.plot([1.01e9, 1.02e9, 1.03e9])
    fig.canvas.draw()
    assert ax.xaxis.offsetText.get_color() == 'yellow', \
        f"Expected offset text color to be 'yellow' (color), got '{ax.
    xaxis.offsetText.get_color()}'"
```

Figure 8: Generated tests for `matplotlib-25287`. The base-student test is not a formal submission.

Why the base student does not complete the task. By Action 21, the tests are already in place and remain unchanged. Nevertheless, the student repeats the same pytest command ten times at the remain steps. The command and final observations are shown below.

Base student, Actions 30 and later: repeated command

```
cd /testbed && python -m pytest lib/matplotlib/tests/test_offset_rcparams.py --
    tb=no -q
```

Command Execution Results

```
3 failed in 0.30s
```

How LOEC completes the task. The LOEC test run produces two failures for the label-color cases and one pass for the `inherit` control. The control checks behavior that already works on the buggy repository. The student recognizes this combination as expected, checks the final diff, and submits the tests at Action 42.

Model	Submission	Evaluation
Base student	Not submitted	Recovered patch: reproduced
LOEC	Submitted	Submitted patch: reproduced

What this case shows. The base student's failure is a failure to submit, not an inability to construct a valid reproduction test. The LOEC run completes submission. This comparison concerns task completion; it does not show that LOEC uses fewer actions overall.

Observation

A valid test patch still needs to be submitted. A test patch can satisfy the reproduction criterion yet remain in the workspace without becoming the agent's submitted result.