REPAIR BEFORE REGENERATING: SCOPE-CONTROLLED RECOVERY FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents increasingly solve programming tasks by writing, executing, and revising code. In these workflows, a failed intermediate program interrupts subsequent actions, making efficient recovery an essential part of task execution. Recovery begins with two useful resources: an existing candidate that encodes the current approach to the task, and execution feedback that reveals where its behavior departs from the specification. Existing pipelines often insert reflection or supervision before regenerating a complete module, adding model calls and repeatedly producing code beyond the correction itself. We introduce *RepairAgent*, an execution-grounded local-repair framework that turns concrete execution discrepancies into targeted changes to the current program. The framework separates two recovery decisions that existing pipelines typically couple: what evidence the model receives and how much of the artifact it regenerates. For supported public equality checks, RepairAgent reads the failed expression and its actual and expected values directly, then proposes exact-text replacements for selected spans. The interface validates replacement targets and the resulting module's syntax, preserving text outside the edited spans while concentrating generation on the correction. Public execution validates each candidate and supplies evidence for further repair, with complete-module generation available as a second-attempt fallback. On executable repair tasks from QuixBugs, MBPP, and HumanEval, RepairAgent reaches a 93.3% held-out pass rate at 14.14 seconds of dataset-macro latency, reducing complete-pipeline latency by 49.5–88.6% relative to adapted Supervisor-Agent, Reflexion, and Self-Refine baselines. Matched controls isolate the scope effect: local edits reduce latency by 15.6–18.7% under fixed feedback.

1 INTRODUCTION

LLM agents solve tasks by interleaving reasoning, tool execution, and interaction with their environment (Yao et al., 2023; Yang et al., 2024). When an intermediate artifact fails, the agent must recover before the surrounding workflow can continue. Execution feedback provides a basis for this recovery process, enabling agents to revise their behavior after observing unsuccessful attempts (Shinn et al., 2023). Effective recovery requires agents to identify the cause of failure and generate corrections from the available execution evidence.

Existing approaches improve agent correction by enriching the feedback available for subsequent attempts. They mainly follow two directions. *Language-based refinement* introduces an explicit feedback stage before revision: Self-Refine alternates self-feedback with revision, while Reflexion converts task outcomes into verbal reflections retained across attempts (Madaan et al., 2023; Shinn et al., 2023). *Runtime supervision* introduces a supervisory mechanism around the executing agent; SupervisorAgent guides actions, corrects observations, and requests verification during execution (Lin et al., 2026). Beyond these approaches, tool- and execution-grounded methods obtain corrective evidence directly from the environment, such as CRITIC, Self-Debugging, and LDB (Gou et al., 2024; Chen et al., 2024; Zhong et al., 2024). Together, these approaches explore different ways of constructing and using feedback to help agents recover from failed executions.

Feedback-and-regeneration pipelines couple two decisions that determine the recovery path. *First, they route execution evidence through another reasoning stage even when the check already exposes the discrepancy needed for the next action. Second, complete-module generation can turn a local*

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.

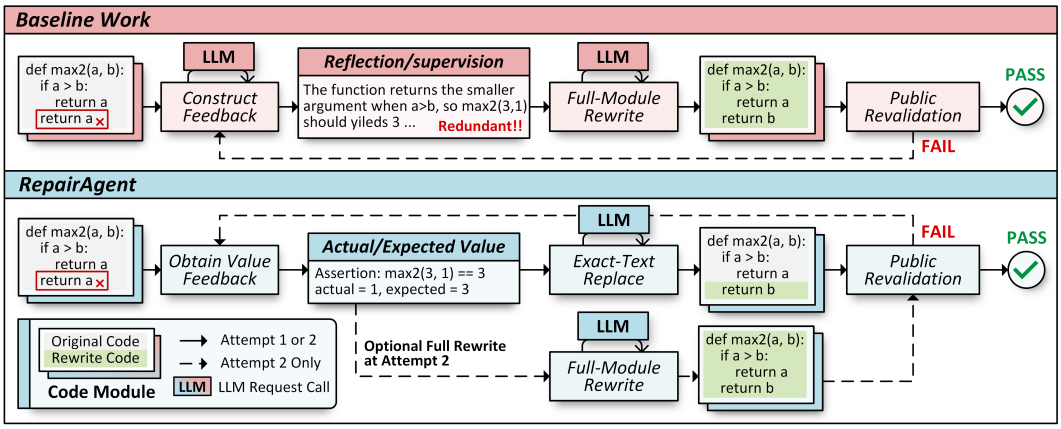


Figure 1: RepairAgent replaces an auxiliary feedback-and-regeneration loop with values exposed by the existing public check and validated exact-text edits; complete-module rewriting remains an optional fallback on the second attempt.

failure into regeneration of an otherwise useful artifact. Separating these decisions makes it possible to connect execution-time evidence to a targeted action while measuring how much of the current artifact the repair should change.

We make this separation explicit through two design principles. *First, actionable execution evidence can guide correction directly:* concrete execution values can expose the residual discrepancy needed for correction. For example, in a return-order error, the mismatch between actual and expected tuples identifies the remaining mistake that a generic assertion failure cannot convey. *Second, repair scope is a controllable part of recovery:* a repair process can preserve the information encoded in the current artifact by changing selected spans. With identical feedback, broader regeneration can overwrite correct behaviors, whereas local edits preserve text outside the replacement spans.

We therefore present *RepairAgent*, an execution-grounded local repair framework that couples the evidence path and repair scope in a single execution-to-action interface (Figure 1). For supported equality assertions, the existing public check yields the failed expression together with its actual and expected values, so the repair model receives a concrete discrepancy without an auxiliary feedback-generation stage. Given that evidence, the model emits one to three exact-text replacements; the system accepts them only when each target is unique and the resulting module parses, while a complete-module action remains available as a controlled second-attempt fallback. This interface makes the repair action explicit: execution supplies the discrepancy, and the policy selects and validates the spans that change in the current artifact.

The interface is also the unit of analysis: varying evidence and repair scope independently makes the cost of additional reasoning separable from the cost of rewriting the artifact.

Our contributions are threefold:

1. **An execution-to-action interface.** We make the repair boundary explicit by separating the evidence exposed by execution from the scope of the artifact action, turning both into measurable recovery controls.
2. **A scope-controlled repair interface.** We instantiate this formulation with concrete value feedback and validated exact-text edits, preserving text outside the selected repair spans while retaining complete-module regeneration as a controlled second-attempt fallback.
3. **A matched evaluation and mechanism analysis.** RepairAgent achieves a 93.3% held-out pass rate at 14.14 seconds of dataset-macro latency, reducing complete-pipeline latency by 49.5–88.6% relative to adapted feedback-driven baselines. Matched controls isolate a 15.6–18.7% latency reduction from local scope, while concrete values expose a residual discrepancy left unresolved by raw execution feedback.

2 Related Work

Agentic reasoning and software engineering. ReAct couples reasoning with tool use and observations (Yao et al., 2023), while SWE-agent demonstrates the importance of agent–computer interfaces in software engineering (Yang et al., 2024). The repository-level RepairAgent of Bouzenia et al. (2025) plans tool use to inspect code, construct patches, and validate fixes. Our work studies the recovery step within these workflows, with execution evidence guiding local changes to an existing failing artifact.

Reflection and runtime supervision. Reflection and supervision methods construct additional guidance for subsequent attempts. Reflexion converts task feedback into verbal reflections (Shinn et al., 2023), Self-Refine alternates self-feedback with revision (Madaan et al., 2023), and SupervisorAgent introduces runtime supervision around execution (Lin et al., 2026). We adapt these approaches as baselines under matched recovery budgets to compare the cost of additional guidance with direct execution-grounded repair.

Tool- and execution-grounded correction. Tool- and execution-grounded methods extract corrective evidence from the environment. CRITIC validates and revises outputs through external tools (Gou et al., 2024), Self-Debugging combines execution results with code explanations (Chen et al., 2024), and LDB inspects runtime values for step-by-step verification (Zhong et al., 2024). RepairAgent pairs concrete actual–expected discrepancies with validated exact-text replacements and execution-based verification. Its execution-to-action interface makes the evidence path and repair scope independently measurable within the same repair loop, allowing the cost and preserved context of local correction to be evaluated directly.

3 Motivation

3.1 Preliminaries

An episode consists of a task specification x, a failing Python module c_0, and public checks V_{pub}. Given these inputs, a repair policy produces a candidate artifact $\hat{c}$ within at most two repair attempts. The returned artifact is evaluated using separate held-out checks V_{hid}, which are never exposed to the policy. A public pass terminates the repair process but does not guarantee correctness on held-out checks. The policy receives neither the reference implementation nor the mutation location.

We measure recovery efficiency using end-to-end task latency, including all costs incurred before returning the final artifact:

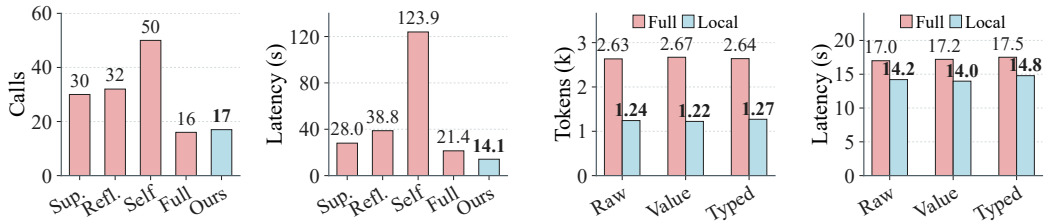
$$T_{\text{task}} = T_{\text{setup}} + \sum_{r=1}^{R} T_{\text{repair}}^{(r)}, \tag{1}$$

where $R \leq 2$ is the number of repair attempts. T_{setup} covers sandbox initialization and the initial public check. $T_{\text{repair}}^{(r)}$ includes feedback processing, correction generation, edit application, and validation in attempt r. Held-out grading and model cold loading are excluded.

3.2 Observations

We organize recovery around two design variables: the execution evidence that guides the next action, and the artifact scope that the action is allowed to change.

Observation 1: Actionable execution evidence reduces the need for additional feedback reasoning. For supported equality checks, the failed expression and its actual and expected values expose a concrete discrepancy at the repair boundary. Figures 2a and 2b compare the resulting recovery pipelines: feedback-driven adaptations use 30–50 model calls and incur 28.0–123.9 seconds of macro latency, whereas RepairAgent uses 17 calls and 14.1 seconds under the same mainsuite contract. These pipeline comparisons include both feedback and output-scope differences; the matched contrasts below isolate repair scope. Concrete execution values also remain useful when additional correction is required: in the independent local-edit comparison, value feedback resolves



(a) Main-suite calls **(b)** Main-suite latency **(c)** Matched output tokens **(d)** Matched latency

Figure 2: Recovery efficiency. (a,b) Main-suite calls and macro latency. (c,d) Independent-suite output tokens and macro latency under matched raw, value, and typed feedback. Blue denotes RepairAgent or local edits; pink denotes controls or full-module repair.

a residual return-order discrepancy that raw-error feedback leaves unresolved. These observations motivate a repair policy that consumes execution evidence directly and spends model calls on the correction itself.

Observation 2: Local edits preserve useful artifact context and improve recovery efficiency. Figures 2c and 2d show that, under matched feedback conditions, local edits reduce suite-total output from roughly 2.6–2.7k tokens to 1.2–1.3k tokens across the independent matched suite. They also reduce macro latency by 16.4%, 18.7%, and 15.6% for raw errors, value feedback, and typed diagnosis, respectively. These savings accompany a smaller generated artifact: replacement spans rather than a complete module. Across controlled comparisons, local edits preserve the surrounding artifact text while allocating generation to the correction, making repair scope a measurable component of recovery rather than an incidental property of the model output.

4 REPAIRAGENT FRAMEWORK

Figure 3 illustrates RepairAgent's workflow through a two-attempt repair of `sum_to`: an exception message first guides a local initialization fix, after which an actual–expected mismatch guides a local correction to the accumulation loop. Both attempts preserve the surrounding module, and the illustrated repair succeeds without complete-module rewriting.

4.1 EXECUTION-TO-ACTION FORMULATION

RepairAgent treats recovery as a structured transition from an execution observation to a repair action. Let c_r denote the current candidate at attempt r, and let $e_r = \mathcal{E}(c_r, V_{\text{pub}})$ denote the evidence returned by the existing public execution. A repair policy is factorized as

$$\pi_{f,s} : (x, c_r, e_r) \longmapsto a_r, \qquad f \in \{\text{RAW}, \text{VALUE}, \text{TYPED}\}, \quad s \in \{\text{FULL}, \text{EDIT}\}, \tag{2}$$

where f specifies how the execution evidence is exposed and s specifies the scope of the generated artifact. RepairAgent is the concrete policy $\pi_{\text{VALUE,EDIT}}$: it consumes values already produced by the public check and returns validated local edits. This factorization makes the recovery path explicit and turns evidence representation and repair scope into two independently testable design variables.

For the proposed policy, the available action space depends on the attempt:

$$\mathcal{Z}_r = \begin{cases} \mathcal{A}_{\text{edit}}(c_r), & r = 1, \\ \mathcal{A}_{\text{edit}}(c_r) \cup \mathcal{A}_{\text{full}}(x, c_r, e_r), & r = 2. \end{cases} \tag{3}$$

The first attempt therefore uses the local action interface by default. The second attempt keeps the same local action available and adds complete-module regeneration as a recovery path when the first correction is insufficient. Repair stops as soon as the public checks pass; held-out checks are used only for independent evaluation.

The factorization also makes the efficiency target explicit. We decompose the per-attempt critical path as

$$T^{(r)}_{\text{repair}} = T^{(r)}_{\text{feedback}}(f) + T^{(r)}_{\text{action}}(s) + T^{(r)}_{\text{validate}}, \tag{4}$$

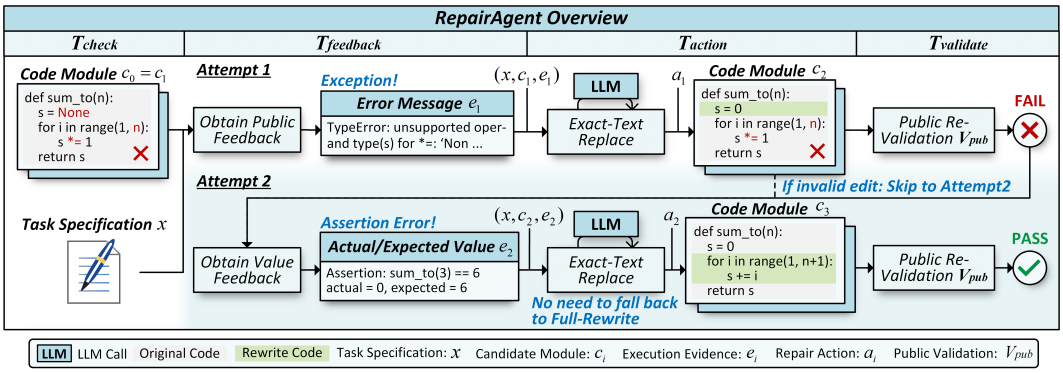


Figure 3: RepairAgent overview. Each attempt reads the latest public-check result and applies validated local edits. The example first fixes an initialization error, then uses the resulting actual–expected mismatch to correct the accumulation loop. Complete-module rewriting is available on the second attempt but is not used here. Public validation controls repair termination; held-out checks remain outside the repair loop.

where the terms cover auxiliary feedback construction, action generation and application, and public validation of the resulting candidate. The initial public check is performed once before the first attempt; subsequent attempts reuse the preceding validation result. For value feedback, there is no additional feedback-model stage: supported equality checks emit actual and expected values during public execution. Local edits scope the generated artifact to replacement spans. All realized costs, including initialization, the initial check, unsuccessful attempts, and retries, remain in the end-to-end measurement of Equation 1.

4.2 Execution-Grounded Evidence and Scope-Controlled Action

For a supported equality assertion, the public validator returns the failed expression together with its actual and expected values. We represent this evidence as

$$e_r = (g_r, y_r, \bar{y}_r), \qquad y_r = g_r(c_r), \tag{5}$$

where g_r is the checked expression, y_r is the observed value, and $\bar{y}_r$ is the expected value. The tuple exposes the residual discrepancy $y_r \neq \bar{y}_r$ in the same execution that establishes public failure. The repair model can therefore act on a concrete mismatch rather than waiting for a separate feedback model to restate it. The public check thus serves as the evidence interface: the policy still receives only x, the current candidate, and public execution evidence, while the discrepancy is represented in a form that directly supports the next action.

When the check does not expose a supported equality, e_r contains the ordinary error message or the bounded diagnostic evidence used by the corresponding control. Raw feedback retains the error as emitted by the runtime. Typed diagnosis selects a fixed error type before collecting values or a bounded trace. The policy never synthesizes a new test and never receives V_{hid}, the reference implementation, or the mutation location, so the value path changes the representation of observable evidence without changing the task's information boundary.

Given (x, c_r, e_r), the local action emits a JSON object containing one to three pairs of `old` and `new` strings. Let

$$\Delta_r = \big((o_j, n_j)\big)_{j=1}^{m}, \qquad 1 \le m \le 3, \tag{6}$$

be the ordered edit sequence. Replacements are applied sequentially to a temporary candidate:

$$c_r^{(0)} = c_r, \qquad c_r^{(j)} = \mathrm{replace}\big(c_r^{(j-1)}, o_j, n_j\big), \tag{7}$$

provided that o_j occurs exactly once in $c_r^{(j-1)}$. The feasible local-action set is therefore

$$\mathcal{A}_{\text{edit}}(c_r) = \Big\{\Delta_r : \ \mathrm{count}_{c_r^{(j-1)}}(o_j) = 1 \ \forall j, \ \mathrm{parse}(c_r^{(m)}) = 1\Big\}. \tag{8}$$

If a target is missing, duplicated, or produces a syntax-invalid candidate, the edit is rejected and the current candidate remains unchanged for that attempt. A syntactically valid edit must still pass public execution for early termination. The model chooses the correction span, while the interface enforces unique targets and a syntactically valid module. Text outside the selected replacement spans is preserved by construction. Complete-module generation is represented by $\mathcal{A}_{\text{full}}$ and is available through the second-attempt branch in Equation 3 for RepairAgent.

4.3 Two-Attempt Execution and Mechanism Isolation

Starting from $c_1 = c_0$, RepairAgent executes the initial public check once, then follows the loop in Figure 3:

1. Read e_r from the latest public execution, or the application error if the preceding action was rejected.
2. Select $a_r \in \mathcal{Z}_r$ using Equation 2. For a local action, apply the sequential validation in Equations 7–8; for a full action, parse the generated module directly.
3. Execute an accepted candidate on V_{pub}. If it passes, return it; otherwise retain the candidate and execution evidence for the next attempt. A rejected action leaves the candidate unchanged and supplies its application error. Increment r and continue while $r \leq 2$.

On the second attempt, a complete-module action is available as $\mathcal{A}_{\text{full}}(x, c_2, e_2)$ when a local correction is not sufficient. The episode terminates after the second failed attempt, and the final candidate is graded independently with V_{hid}. The procedure changes only the inference-time action interface and requires no additional model training.

The evaluation mirrors this factorization with matched policy pairs. Holding the evidence path fixed, the scope effect is measured as

$$\Delta_{\text{scope}}(f) = \frac{\overline{T}_{\pi_{f,\text{FULL}}} - \overline{T}_{\pi_{f,\text{EDIT}}}}{\overline{T}_{\pi_{f,\text{FULL}}}}, \tag{9}$$

while holding the action scope fixed, the evidence effect is measured by comparing $\pi_{\text{RAW},s}$ and $\pi_{\text{VALUE},s}$. The raw, value, and typed policies therefore share the same task contract and two-attempt budget; only the evidence representation or output scope changes. Uniform tracing and matched re-execution provide additional controls for separating the information content of a diagnostic from the cost of obtaining it. This matched construction directly identifies the source of efficiency: a stable scope gap under fixed evidence attributes savings to local action, while any value gain is tied to discrepancies that execution exposes concretely.

5 Evaluation

5.1 Experimental Setup

Datasets and Models. We evaluate RepairAgent on native defects from QuixBugs (Lin et al., 2017) and controlled single-site mutations of reference programs from MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021). The main suite combines QuixBugs defects with MBPP and HumanEval mutations; an independent ablation suite uses disjoint defects and source programs, stratified by benchmark and initial failure type. All policies use Qwen3.5-27B at temperature zero. Each episode allows at most two repair attempts and nine model calls, with a limit of 2,048 generated tokens per repair or auxiliary call.

Baselines. We compare RepairAgent with Reflexion (Shinn et al., 2023), Self-Refine (Madaan et al., 2023), and SupervisorAgent (Lin et al., 2026), adapted to a common repair environment. Reflexion retains execution-triggered reflection and task-local memory; Self-Refine retains self-feedback, revision, and `NO_CHANGES` stopping; SupervisorAgent retains error-triggered supervision, official prompt construction, guidance, observation correction, and sandbox verification. Across the two suites, matched controls pair raw errors, value feedback, or typed diagnosis with full-module generation or exact-text editing. Uniform tracing tests diagnostic selection, while matched re-execution without tracing isolates the contribution of trace information. All policies share the task inputs, model, budgets, and hardware allocation.

Table 1: Main-suite recovery cost for all registered policies: macro latency including failures, total calls and output tokens, and latency overhead relative to RepairAgent.

Policy	Latency (s) ↓	Calls	Output tokens	Overhead (%)
RepairAgent (ours)	**14.14**	**17**	**1,484**	–
Raw / full	21.39	16	3,924	+51.2
Raw / local	16.37	18	1,584	+15.7
Typed / full	24.02	16	4,436	+69.8
Typed / local	14.45	17	1,471	+2.2
Uniform / local	15.27	17	1,563	+8.0
SupervisorAgent	28.02	30	3,722	+98.1
Reflexion	38.76	32	7,152	+174.0
Self-Refine	123.94	50	21,191	+776.3

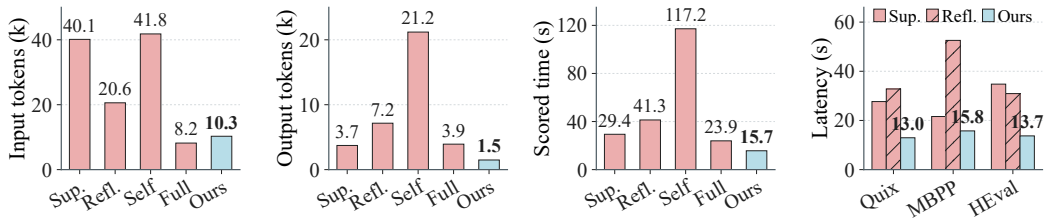
Evaluation Metrics. Held-out pass rate is the percentage of tasks whose final candidate passes all held-out checks, using all tasks in the corresponding suite as the denominator. We measure end-to-end repair latency from sandbox creation through candidate generation and public validation. Latency includes initial checks, diagnosis, all repair and auxiliary calls, edit application, and retries, but excludes model cold loading and held-out grading. Aggregate latency equally weights per-dataset task means and includes failed episodes. We quantify task-level uncertainty with 10,000 dataset-stratified paired bootstrap samples, with source-program clustering in the ablation suite. Additional measurements include grading-inclusive latency, input and output tokens, model calls, parsing failures, and edit size.

Environment. Experiments run on NVIDIA A800-SXM4 GPUs with 80 GB memory, with one resident model per worker. The main suite distributes disjoint tasks across two GPUs, while the ablation suite uses one. Policies for each task execute serially on the same GPU in a fixed randomized order. Program execution and held-out grading use separate CPU-only, network-isolated containers. Model loading and warm-up are recorded separately from task timing. Dataset construction, public/held-out partitions, and repair stopping rules are specified in Appendix B.1.

5.2 Overall Recovery Efficiency

Separating evidence from action scope. The main-suite comparison in Table 1 places RepairAgent at 14.14 seconds of dataset-macro latency, reducing complete-pipeline recovery time by 49.5%, 63.5%, and 88.6% relative to SupervisorAgent, Reflexion, and Self-Refine, respectively. It uses 17 model calls versus 30, 32, and 50 for these baselines, with all auxiliary calls, public validation, retries, and unsuccessful episodes included. On the independent suite, matched controls isolate the scope effect: replacing full-module generation with local edits reduces latency by 15.6–18.7% under fixed feedback, alongside the corresponding output-token reductions. The baseline comparisons measure the complete recovery pipeline, while the matched controls hold the evidence path fixed and vary the artifact scope. Together they show that recovery cost is governed by both how a discrepancy is exposed and how much of the current artifact is regenerated in response.

Preserved task correctness. The held-out evaluation shows that RepairAgent matches the three feedback-driven baselines at a 93.3% pass rate. These checks are unavailable during repair, so they assess behavior beyond the public feedback used to construct a candidate. Typed and uniform tracing with local edits reach the same rate, while raw and typed full-module repair each attain 86.7%. Raw feedback with local edits also reaches 86.7%, indicating that the observed correctness depends on the evidence available to guide the edit as well as its scope. The typed-feedback pair also associates the narrower output with a higher held-out pass rate. The edit interface targets the faulty computation while retaining surrounding text, allowing local correction to preserve implementation choices that remain useful beyond the public check. The overall comparison therefore pairs lower recovery cost with the same observed held-out correctness as the adapted baselines.



(a) Input tokens **(b)** Output tokens **(c)** Including grading **(d)** Across datasets

Figure 4: Main-suite resource use and latency; all episodes included. (a,b) Total input and output tokens. (c) Episode-weighted task time including held-out grading. (d) Mean task latency on QuixBugs (Quix), MBPP, and HumanEval (HEval). Blue denotes RepairAgent; pink denotes controls.

5.3 Resource Use and Cross-Dataset Consistency

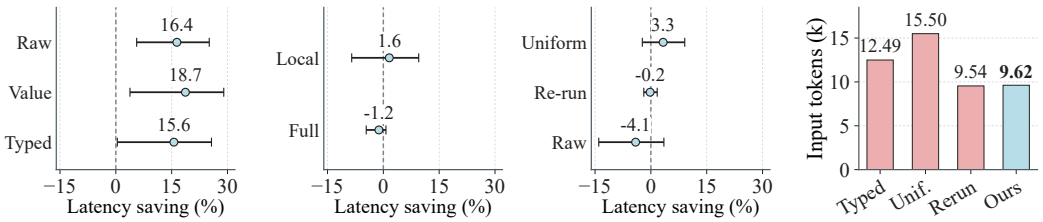
Less model traffic. The first two panels of Figure 4 separate the prompt traffic consumed during recovery from the tokens generated in response. RepairAgent consumes 10,280 input tokens and generates 1,484 output tokens, reducing the two totals by 50.1–75.4% and 60.1–93.0%, respectively, relative to the three feedback-driven baselines. Its output is also smaller than the 3,924 tokens generated by raw full-module repair. The two reductions reflect complementary aspects of the recovery path: direct execution feedback avoids auxiliary feedback exchanges, while local edits concentrate generated code on the correction. This distinction matters because the current program remains available as input even when the output contains only replacement spans. The model can use the surrounding implementation to formulate a repair without reproducing that implementation in its response.

Consistent gains across accounting choices and datasets. The final two panels of Figure 4 examine the gain under broader timing boundaries and at the dataset level. Including held-out grading, RepairAgent takes 15.69 seconds per episode, compared with 29.40, 41.32, and 117.17 seconds for SupervisorAgent, Reflexion, and Self-Refine, yielding reductions of 46.6–86.6%. This episode-weighted measure includes the final evaluator's work as well as the recovery process. The dataset breakdown also favors RepairAgent: mean task latency falls by 53.2% on QuixBugs, 27.1% on MBPP, and 55.6% on HumanEval against the fastest feedback-driven baseline for each dataset. That comparator is SupervisorAgent on QuixBugs and MBPP and Reflexion on HumanEval. The larger reductions on QuixBugs and HumanEval show that the magnitude varies across workloads, while the direction is consistent. Including unsuccessful episodes, the advantage persists both when grading is charged and when each benchmark is examined separately.

5.4 Mechanism and Component Analysis

Repair scope drives the repeated latency reduction. The controlled contrasts in panel a of Figure 5 hold feedback fixed while varying the generated artifact in the independent suite. Local edits reduce macro latency by 16.4% with raw errors, 18.7% with value feedback, and 15.6% with typed diagnosis. Their exploratory 95% source-clustered intervals are [5.6, 25.1], [3.8, 29.0], and [0.4, 25.7]%, respectively, each with a positive lower bound. Each pair gives the model the same type of evidence, so its saving comes from changing the repair interface with the diagnostic format held fixed. The repeated effect across all three conditions makes repair scope a common source of efficiency across feedback designs. It also explains why local output benefits both the proposed value-based path and controls that retain raw errors or typed tracing.

Fewer generated tokens at the same call count. Table 2 separates model invocations from the amount of code generated. RepairAgent achieves the lowest measured macro latency and output-token total among the eight independent-suite policies: 13.98 seconds and 1,223 tokens. The two value-feedback policies both use 16 model calls, but local edits reduce output from 2,670 to 1,223 tokens and latency from 17.20 to 13.98 seconds, while input rises slightly from 9,271 to 9,619 tokens. Thus, the 18.7% latency saving accompanies 54.2% less generation at the same call count, showing that call counts alone obscure work per invocation. Typed tracing, uniform tracing, and



(a) Local vs. full **(b)** Value vs. raw **(c)** Typed vs. controls **(d)** Diagnostic input

Figure 5: Independent-suite mechanism comparisons. (a) Macro-latency savings of local edits over full-module output for raw errors, value feedback, and typed diagnosis. (b) Macro-latency savings of value feedback over raw errors for local and full-module output. (c) Macro-latency savings of typed diagnosis over uniform tracing, matched re-execution, and raw feedback, all with local edits. Positive values favor the first policy in each comparison. Markers show point estimates; error bars show exploratory 95% paired bootstrap intervals, clustered by source within dataset, without multiple-comparison correction. (d) Total input tokens for local-edit policies; Ours uses value feedback.

Table 2: Independent-suite recovery cost: macro latency including failures; calls and tokens are totals. Policy names denote evidence/output scope. Uniform uses uniform tracing; Re-run uses matched re-execution. RepairAgent uses value feedback with local edits.

Policy	Latency (s) ↓	Calls	Input tokens	Output tokens
RepairAgent (ours)	**13.98**	**16**	**9,619**	**1,223**
Raw / full	17.00	16	8,127	2,633
Raw / local	14.21	16	8,494	1,242
Value / full	17.20	16	9,271	2,670
Typed / full	17.51	16	12,143	2,639
Typed / local	14.79	16	12,493	1,271
Uniform / local	15.29	16	15,499	1,311
Re-run / local	14.75	17	9,541	1,301

matched re-execution with local edits take 14.79, 15.29, and 14.75 seconds, respectively, leaving the value-based policy at the fastest measured point in this suite.

Concrete evidence keeps correction focused. Panels b–d of Figure 5 separate diagnostic input volume from latency effects. Value feedback with local edits uses 9.62k input tokens, 23.0% less than typed tracing and 37.9% less than uniform tracing. Under fixed scope, its latency savings are 1.6% for local edits and -1.2% for full-module output; both exploratory intervals, as well as the three tracing contrasts, span zero. Its main role is therefore to describe the discrepancy compactly and guide the next edit. In Appendix A, the return-order case first fixes the counting logic but returns $(0, 6)$ instead of $(6, 0)$, directing the next attempt to swap the values while preserving the repaired code. In the trailing-space case, the local edit preserves treatment of the last space-delimited field, which full-module regeneration loses. Across the controlled comparisons, local scope supplies the repeated latency reduction, while concrete evidence focuses the remaining correction.

6 Conclusion

RepairAgent makes execution-to-action the organizing principle for recovery: public execution exposes a concrete discrepancy, and a scope-controlled interface turns that discrepancy into a validated change to the current artifact. On the main suite, it achieves a 93.3% held-out pass rate at 14.14 seconds of dataset-macro latency, reducing complete-pipeline latency by 49.5–88.6% relative to the adapted feedback-driven baselines. The independent suite separates the two design choices: local edits reduce latency by 15.6–18.7% under matched feedback, while concrete values expose a residual return-order discrepancy that raw errors leave unresolved. These results show how an execution-grounded repair policy can spend model computation on the correction itself while preserving useful context outside the selected repair spans.

AI USE STATEMENT

During manuscript preparation, the authors used generative artificial intelligence (AI) tools for language editing, manuscript organization, figure and table refinement, interpretation of experimental results, reference checking, and LaTeX debugging. AI-assisted suggestions for presentation and analysis were checked against the source data and implementation by the authors. The authors take responsibility for the final text, figures, claims, and experimental results.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024.

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. In *International Conference on Learning Representations*, 2024.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings of the SPLASH Companion*, pp. 55–56, 2017. doi: 10.1145/3135932.3135941.

Fulin Lin, Shaowen Chen, Ruishan Fang, Hongwei Wang, and Tao Lin. Stop wasting your tokens: Towards efficient runtime multi-agent systems. In *International Conference on Learning Representations*, 2026.

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. In *Advances in Neural Information Processing Systems*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

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. In *Advances in Neural Information Processing Systems*, 2024.

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

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, 2024.

A ADDITIONAL EVALUATION VIEWS

The dataset-level view shows selected policies and includes unsuccessful episodes; blue bars identify RepairAgent.

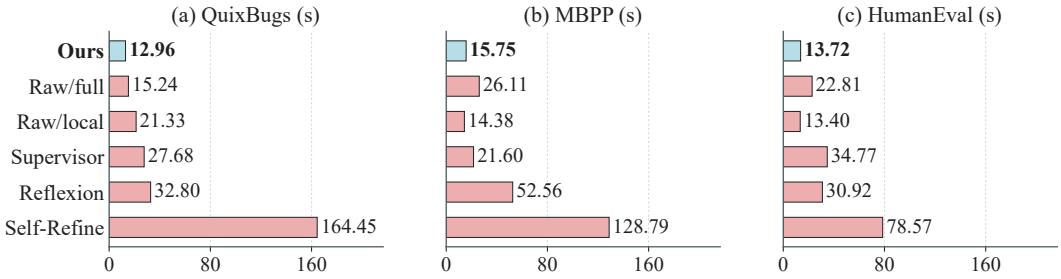


Figure 6: Main-suite mean task latency by dataset, including failures. The three panels share one latency scale. MBPP and HumanEval use controlled defects.

Repair trajectories. In the recorded `mbpp-343` episode, the first local edit fixes the counters but swaps the return order. The public check exposes `actual=(0,6)` and `expected=(6,0)`, guiding the replacement `return (d,l)` → `return (l,d)`; public and held-out checks then pass. On HumanEval/134, full-module policies replace the original last-field behavior with a search for the last nonempty word. This passes public checks but fails a held-out trailing-space case. Local edits repair the exception while preserving the original split-and-select logic and pass both partitions.

A.1 EXPLORATORY CROSS-TASK SCREEN

An additional eight-task screen (four MBPP and four HumanEval development tasks) used the same Qwen3.5-27B service and fixed repair environment. Direct feedback, Reflexion, and Self-Refine each solved 7/8 tasks; their mean delivery times were 13.87, 16.14, and 32.71 seconds per task, respectively. Direct feedback therefore reduced delivery latency by 14.1% relative to Reflexion and 57.6% relative to Self-Refine on these additional tasks.

B REPRODUCIBILITY DETAILS

B.1 EVALUATION PROTOCOL

Task selection. Mutation candidates follow a fixed abstract-syntax-tree traversal, considering at most 16 operator or small-constant changes per source program. The main suite selects the first publicly detected mutation; the ablation suite selects the first for each registered failure type under fixed dataset quotas. Selection is independent of model repair outcomes.

Checks and information boundaries. Public checks comprise the first MBPP assertion, the first two top-level assertions of eligible HumanEval programs, or the first two QuixBugs cases; the remaining checks are held out. Preflight retains instances whose defects are publicly detectable and whose reference programs pass both partitions within the execution budget. Deliberately incorrect programs provide additional scoring checks. Policies receive neither reference implementations, mutation locations, nor held-out results.

Repair execution. The local-edit interface requests one to three exact-text replacements through JSON-constrained generation. Edits require unique targets and a syntactically valid result; invalid edits leave the candidate unchanged and consume an attempt. The second attempt allows another patch or a full-module fallback. Ordinary repair stops on a public pass, while Self-Refine retains its feedback-based stopping rule.

B.2 Evaluation manifests

Versioned manifests record source identity, benchmark, mutation rule, and initial failure type. Defects from the same source form one cluster for ablation resampling. Public and held-out checks are disjoint within each instance; this separation does not control pretraining exposure.

B.3 Implementation and Controls

Baseline implementations. The repair-task adapters and common harness define the measured implementations. SupervisorAgent uses its official prompt construction and compression, a short JSON action interface, and the same 27B model for supervision; Reflexion retains reflections within a task.

Diagnostics and controls. Typed diagnosis maps `AssertionError` to wrong output, other error or exception strings to exceptions, and remaining nonzero termination to timeout/exit. Traces retain the latest six events, bounded variable summaries, and frequent execution locations; classification and diagnostic re-execution count toward task time. The no-trace control preserves this execution pathway to isolate trace information.

Independent-suite checks. The request audit found no held-out assertions in prompts; reference and deliberately incorrect controls passed scoring checks. Initial prompts and outputs agree across the paired diagnostic controls on their shared failure types, although later repairs can change the failure type.

C Discussion and Limitations

Scope of the evidence. The evaluation fixes the repair-time contract: one Qwen3.5-27B model, shared hardware allocation, a two-attempt budget, and disjoint benchmark suites. Under this contract, matched contrasts isolate how concrete execution evidence and bounded artifact scope allocate computation while retaining useful context. Broader model and repository-scale studies are extensions of this interface.