NOT EVERY FAILURE NEEDS A PATCH: PAUSE FOR SELECTIVE PROGRAM REPAIR UNDER UNCERTAIN EXECUTION FEEDBACK

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based automated program repair relies on execution feedback to guide patch generation. However, systems often treat each failure as evidence that the program should be modified. Because test and runtime signals may be incomplete, noisy, or environmentally induced, this assumption can cause unnecessary patches, ineffective retries, and unreliable termination. We present PAUSE, a selective program repair framework that determines whether modification is warranted, whether the candidate is acceptable, and whether continued repair remains worthwhile. PAUSE coordinates specialized agents to analyze repair context, reason over execution evidence, decide between PATCH and NOOP, and learn from repair trajectories. Its dual-condition governor evaluates candidate acceptability and continuation value. Trajectories are distilled into reusable decision constraints rather than patch templates. Experiments on three benchmarks spanning Python and Java and dedicated control cases show that PAUSE improves confirmed repair rate by 2.5–3.0 percentage points over the strongest controlled baseline, limits over-repair and unsupported acceptance to 1.7% and 2.0%, respectively, and reduces model calls by 44.6% relative to fixed-budget execution. These results demonstrate that selective action and trajectory-aware governance can improve repair reliability while reducing unnecessary computation. The code is available at `https://anonymous.4open.science/r/PAUSE`.

1 INTRODUCTION

Automated program repair (APR) has been formulated as a patch generation problem: given a failing program, the goal is to synthesize a modification that makes the program pass validation. However, with the emergence of LLM-based repair agents, this assumption is increasingly challenged by interactive environments where failures may provide incomplete or misleading evidence Jimenez et al. (2024); Ding et al. (2023); Liu et al. (2024); Wu et al. (2024a). Recent benchmarks and training environments further reveal substantial variation in issue descriptions, repository configurations, and executable validation conditions, making reliable software repair more than a code-generation problem Vergopoulos et al. (2025); Pan et al. (2025).

Recent LLM-based repair agents organize perception, reasoning, tool interaction, and memory into iterative workflows Hong et al. (2024); Qian et al. (2024); Tao et al. (2024); Zhang et al. (2024a); Islam et al. (2024). They inspect the program and its environment, propose a modification, execute the candidate, and use the resulting feedback to guide subsequent attempts. This workflow makes execution feedback a key link between successive repair decisions.

Agents use execution feedback to guide repair. Reflection, self-refinement, and tool-assisted critique help LLMs revise outputs based on environmental observations Shinn et al. (2023); Madaan et al. (2023); Gou et al. (2024). In programming tasks, runtime traces and intermediate states can offer more diagnostic evidence than final outputs Zhong et al. (2024). Yet repair agents that react to failures rarely ask whether a failure warrants a response. Empirical evaluations show that runtime information may be unhelpful in some debugging cases, while generated tests aid patch screening without providing a complete correctness specification Tian et al. (2024); Mündler et al. (2024). A

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.

failed execution may indicate a source-level defect, timeout, unstable infrastructure, missing dependencies, or insufficient specification evidence. Yet passing tests cannot prove correctness.

Under uncertain execution feedback, this organization exposes a missing decision layer: whether to modify the program, preserve the state, continue repair, or terminate safely. Once a failure is observed, the loop typically advances from execution feedback to code modification, even though the system must determine whether the evidence justifies a patch, whether the program can be safely returned, and whether another attempt remains worthwhile. Conflating these decisions can cause unnecessary changes to correct programs, acceptance based on incomplete evidence, or repeated attempts after the trajectory has ceased to improve. Reliable repair therefore requires explicit governance over the repair process rather than relying on stronger patch generation.

To address this problem, we propose PAUSE, a selective program repair framework under uncertain execution feedback. Architecturally, PAUSE coordinates five role-specialized agents with an experience-retrieval module and a repository of distilled decision constraints. Its planner then selects between PATCH and NOOP, allowing the system to abstain when modification is insufficiently supported. Execution outcomes are normalized into typed observations and interpreted against the accumulated repair trajectory. Finally, a dual-condition governor separately assesses the acceptability of the current program and the expected benefit of continued repair, producing ACCEPTPATCH, PRESERVE, CONTINUE, or REJECT. Informative trajectories are distilled into reusable decision constraints without treating previous patches as modification templates.

Our contributions are summarized as follows:

- We challenge the assumption that execution failures should automatically trigger code modifications and formulate LLM-based program repair under uncertain feedback as a selective decision process, where modification, preservation, and continuation are explicit choices.
- We propose PAUSE, a selective repair framework that introduces abstention, typed execution feedback interpretation, trajectory-level reasoning, dual-condition governance, and experience-based decision constraint distillation to enable reliable autonomous repair.
- We conduct an evaluation on 2,580 faults from three APR benchmarks and controlled uncertainty scenarios, demonstrating that selective improves confirmed effectiveness while reducing harmful modifications, unsupported acceptance, and wasted repair attempts.

2 RELATED WORK

Automated Program Repair APR has progressed from traditional search-based, template-based, and semantics-based approaches to neural and LLM-based repair paradigms Huang et al. (2024); Xia et al. (2023); Jiang et al. (2023). Representative traditional systems include SimFix, which reuses existing code ingredients, TBar, which applies fix patterns, and CURE, which incorporates code-aware representations into neural repair Jiang et al. (2018); Liu et al. (2019); Jiang et al. (2021). Despite their different patch-generation strategies, these methods commonly rely on executable tests for validation. A test-passing patch may therefore remain inconsistent with the intended behavior under incomplete specifications, leading to patch overfitting and motivating research on patch correctness assessment Ismayilzada et al. (2024); Petke et al. (2024); Fei et al. (2025). Existing studies mainly improve candidate generation or assess patches after they have been produced. In contrast, PAUSE focuses on whether uncertain execution evidence justifies modification in the first place, treating abstention as a valid repair decision.

LLM-Based and Agentic Repair LLMs substantially expand the repair search space through task-specific adaptation, contextual reasoning, and iterative feedback. RepairLLaMA improves repair through parameter-efficient fine-tuning, while ChatRepair, ThinkRepair, ITER, and ContrastRepair use conversational feedback, repair reasoning, partial-patch refinement, or contrastive test evidence to guide subsequent candidates Silva et al. (2025); Xia & Zhang (2024); Yin et al. (2024); Ye & Monperrus (2024); Kong et al. (2025). Recent approaches further exploit static–dynamic context and runtime traces to construct repair hypotheses and refine patches from execution behavior Cheng et al. (2026); Wu et al. (2026).

Agentic systems extend these pipelines by allowing LLMs to navigate repositories, invoke analysis and execution tools, and adapt actions according to intermediate outcomes Yang et al. (2024); Zhang

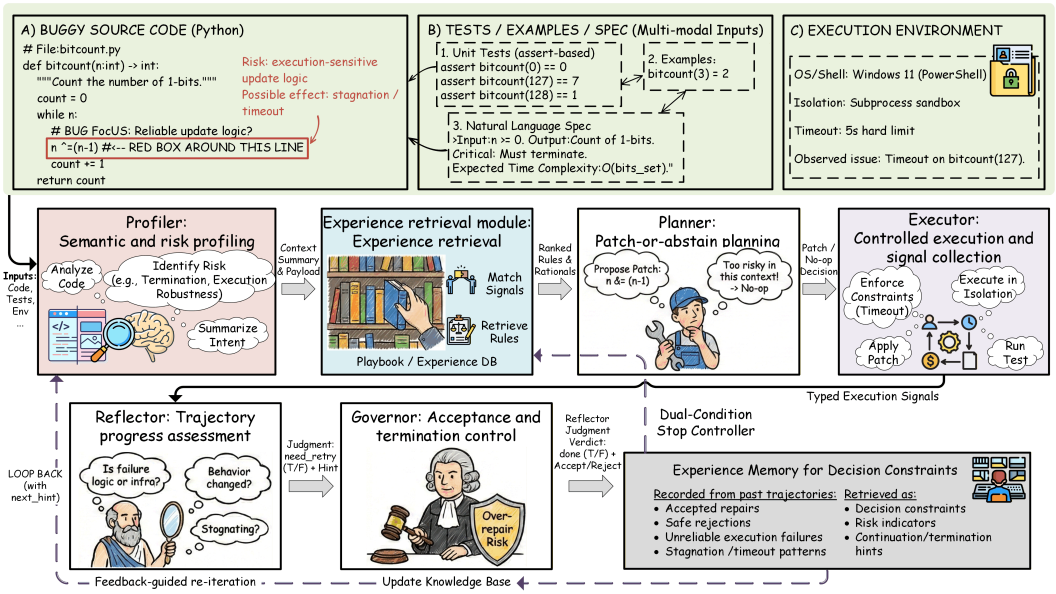

Figure 1: Overview of PAUSE, comprising five role-specialized agents, an experience-retrieval module, and a repository of distilled decision constraints. The workflow supports selective planning, typed execution interpretation, trajectory reflection, and dual-condition governance.

et al. (2024b); Bouzenia et al. (2025); Xia et al. (2024). Recent multi-agent research also studies process-level failure localization and responsibility attribution in collaborative code generation Geng et al. (2026). Across these directions, execution evidence is primarily used for diagnosis, candidate generation, or patch validation. The decision of whether a failure should trigger a patch, another attempt, or termination remains underexplored. To address this gap, PAUSE interprets execution outcomes as typed and potentially unreliable observations, explicitly deciding whether to patch or abstain and whether continued repair remains worthwhile.

3 METHOD

3.1 PROBLEM FORMULATION AND OVERVIEW

We study selective program repair under uncertain execution feedback. Given an initial program P_0, specification signals $\mathcal{S}$, and an execution environment $\mathcal{E}$, the goal is to determine both how to repair the program and whether the available evidence justifies modification. The specification signals may include tests, examples, and natural-language requirements, while $\mathcal{E}$ defines the execution conditions and resource constraints used for validation.

Conventional repair pipelines typically treat a failure as a direct trigger for patch generation and continue until the tests pass or the repair budget is exhausted. This strategy becomes unreliable when failures arise from incomplete specifications, non-termination, infrastructure instability, or restrictive execution conditions. PAUSE instead separates code modification from repair governance.

As illustrated in Figure 1, PAUSE consists of five role-specialized agents—Profiler, Planner, Executor, Reflector, and Governor—together with an experience-retrieval module and an experience repository. Profiler constructs risk-aware context; Planner selects between PATCH and NOOP; Executor returns typed execution evidence; Reflector evaluates trajectory progress; and Governor produces ACCEPTPATCH, PRESERVE, CONTINUE, or REJECT. The retrieval module supplies relevant experience-derived decision constraints without generating patches.

At iteration t, the repair state is represented as

$$s_t = \big(P_t, \mathcal{S}, \mathcal{E}, \mathcal{H}_t, \mathcal{M}_t\big), \tag{1}$$

where P_t is the current program, $\mathcal{H}_t$ records the repair trajectory, and $\mathcal{M}_t$ is the experience repository. These components form a closed loop in which the Planner determines whether modification is warranted, while the Reflector and Governor determine whether the current state is acceptable and whether another attempt remains worthwhile.

The bitcount example illustrates this distinction. A timeout indicates abnormal execution, but does not reveal whether its cause is defective source code, an incomplete specification, or a restrictive environment. Rather than patching automatically, PAUSE treats the timeout as uncertain evidence and uses repair context and trajectory to decide whether to modify the program or terminate.

3.2 CONTEXT PROFILING AND SELECTIVE PLANNING

Each repair iteration begins with risk-aware context profiling. Profiler analyzes the current program, specification signals, execution environment, and repair history:

$$c_t = \mathrm{Profile}\left(P_t, \mathcal{S}, \mathcal{E}, \mathcal{H}_t\right). \tag{2}$$

The resulting profile c_t summarizes the inferred program intent, relevant specification evidence, and execution-sensitive regions, such as loop updates, termination conditions, boundary checks, state mutations, and exception paths. Rather than generating a patch, Profiler characterizes how the current observation affects modification risk and identifies the evidence required for subsequent planning. For example, the same suspicious statement may require different treatment depending on whether the observed outcome is an assertion failure, compilation error, timeout, or infrastructure failure. This structured profile separates contextual understanding from code modification.

The experience-retrieval module complements the profile by querying the repository for relevant decision constraints:

$$\mathcal{R}_t = \mathrm{Recall}\left(c_t, \mathcal{H}_t, \mathcal{M}_t\right). \tag{3}$$

The retrieved entries may describe accepted repairs, safe rejections, repeated timeouts, execution instability, ineffective modifications, or contexts in which further repair was unlikely to provide benefit. These entries support repair control rather than serving as patch templates. This separation prevents task-specific historical modifications from directly biasing the current patch.

Using the context profile, retrieved constraints, and repair trajectory, the Planner selects an action:

$$\mathcal{A} = \{\textsc{Patch}, \textsc{NoOp}\}, \qquad a_t \in \mathcal{A}. \tag{4}$$

When $a_t = \textsc{Patch}$, the Planner produces a candidate modification Δ_t. When $a_t = \textsc{NoOp}$, the current program is preserved. The resulting candidate program is

$$\widehat{P}_t = \begin{cases} \mathrm{Apply}(P_t, \Delta_t), & a_t = \textsc{Patch}, \\ P_t, & a_t = \textsc{NoOp}. \end{cases} \tag{5}$$

NOOP is a valid abstention action rather than a failed patch attempt. The Planner selects it when the failure cannot be reliably attributed to the source code, the specification is insufficient for safe modification, or patching poses greater risk than benefit. The unchanged program then proceeds to execution and governance, which may preserve it, continue repair, or reject the task. Thus, NOOP is an iteration-level action, whereas PRESERVE is a terminal decision.

3.3 TYPED EXECUTION FEEDBACK AND TRAJECTORY REFLECTION

Executor evaluates $\widehat{P}_t$ against the specification signals $\mathcal{S}$ in execution environment $\mathcal{E}$:

$$o_t = \mathrm{Execute}\left(\widehat{P}_t, \mathcal{S}, \mathcal{E}\right) = (\tau_t, \eta_t), \tag{6}$$

where τ_t denotes the observation type and η_t contains the associated diagnostics. We distinguish seven observation types: $\mathrm{TestPass}$, $\mathrm{TestFailure}$, $\mathrm{BuildFailure}$, $\mathrm{RuntimeFailure}$, $\mathrm{Timeout}$, $\mathrm{InfrastructureFailure}$, and $\mathrm{SpecificationIssue}$.

TestPass and TestFailure describe outcomes under the available executable checks. BuildFailure and RuntimeFailure capture errors before or during execution, while Timeout represents non-termination or resource-limit violations. InfrastructureFailure and SpecificationIssue distinguish failures caused

by the execution environment or insufficient validation evidence from source-level defects. The diagnostic component η_t may include failed tests, compiler messages, stack traces, execution time, outputs, and infrastructure logs.

These observations are treated as uncertain evidence rather than definitive correctness judgments. For example, a timeout may indicate faulty loop logic or an overly restrictive execution limit. Likewise, passing the available tests supports acceptance under the observed specification but does not establish complete semantic correctness.

Reflector interprets the current observation with respect to the accumulated repair trajectory:

$$(r_t, \chi_t) = \text{Reflect}\left(c_t, a_t, \widehat{P}_t, o_t, \mathcal{H}_t\right), \tag{7}$$

where r_t is the reflective assessment and $\chi_t \in \{0, 1\}$ is the continuation judgment. A value of $\chi_t = 1$ indicates that another repair attempt may still provide meaningful benefit, whereas $\chi_t = 0$ indicates that continued repair is not sufficiently justified.

Reflection is not another patch-generation step. It compares the current outcome with previous iterations to determine whether the candidate changes relevant behavior, resolves failures, introduces regressions, or repeats an ineffective attempt. It also identifies trajectory patterns such as recurring timeouts, equivalent patches, stagnation, and diminishing returns. The resulting assessment r_t and continuation judgment χ_t are passed to Governor for the final acceptability and governance decision.

3.4 Dual-Condition Governance and Experience Distillation

Governor combines candidate acceptability with the continuation judgment produced by Reflector. It first assesses whether the current candidate is acceptable under the available specification, execution evidence, and repair context:

$$\alpha_t = \text{Assess}\left(\widehat{P}_t, a_t, o_t, r_t, \mathcal{H}_t\right). \tag{8}$$

Here, $\alpha_t \in \{0, 1\}$, where $\alpha_t = 1$ indicates that the available evidence is sufficient to accept the current candidate. This judgment supports a reliable terminal decision without claiming complete semantic correctness.

Governor combines α_t with the continuation judgment χ_t and the selected action a_t:

$$d_t = \begin{cases} \text{AcceptPatch}, & \alpha_t = 1 \wedge a_t = \text{Patch}, \\ \text{Preserve}, & \alpha_t = 1 \wedge a_t = \text{NoOp}, \\ \text{Continue}, & \alpha_t = 0 \wedge \chi_t = 1, \\ \text{Reject}, & \alpha_t = 0 \wedge \chi_t = 0. \end{cases} \tag{9}$$

AcceptPatch returns an accepted patched program, whereas Preserve returns the unchanged program when modification is unnecessary or insufficiently justified. Continue initiates another repair iteration, while Reject terminates without claiming that a reliable repair has been found. NoOp and Preserve belong to different decision levels: NoOp is an iteration-level action selected by Planner, whereas Preserve is a terminal governance decision. A NoOp action may therefore lead to Preserve, Continue, or Reject.

The two conditions serve complementary purposes. Acceptability determines whether the current candidate can be returned, while continuation estimates whether another attempt remains worthwhile. Their separation prevents premature termination when the trajectory is still improving and avoids ineffective retries after repair has stagnated. A maximum iteration and execution budget is retained only as a safety bound. Once the budget is exhausted, continuation is disabled and an unacceptable candidate is rejected.

After governance, the repair history is updated as

$$\mathcal{H}_{t+1} = \mathcal{H}_t \cup \left\{(\widehat{P}_t, a_t, o_t, r_t, \alpha_t, \chi_t, d_t)\right\}. \tag{10}$$

This record preserves the attempted action, observed outcome, reflective assessment, and governance rationale required for subsequent trajectory reasoning.

Dataset	Lang.	Published	Raw	Eligible
QuixBugs	Python	40	40	40
ConDefects	Python	1,625	2,864	1,885
Defects4J v2.0	Java	835	835	655

Table 1: Statistics of the evaluated APR benchmarks.

For an informative intermediate or terminal outcome, PAUSE distills a reusable experience entry:

$$e_t = \mathrm{Distill}\left(c_t, a_t, o_t, r_t, \alpha_t, \chi_t, d_t\right). \tag{11}$$

The entry summarizes the context, execution pattern, repair action, and resulting decision. It may describe an accepted patch, a preserved program, a safe rejection, unreliable execution, or a stagnating trajectory. The repository is then updated by

$$\mathcal{M}_{t+1} = \mathrm{Update}\left(\mathcal{M}_t, e_t\right). \tag{12}$$

To avoid redundant accumulation, the repository retains entries that contribute new evidence. Semantically similar entries are consolidated, while both supporting and conflicting observations are preserved to calibrate the reliability of each decision constraint. The resulting experience informs subsequent action selection without prescribing the patch itself, thereby keeping patch generation separate from repair governance.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Datasets. We evaluate PAUSE on QuixBugs Lin et al. (2017), ConDefects Wu et al. (2024b), and Defects4J v2.0 Just et al. (2014), covering Python and Java programs at different scales (Table 1). We use all 40 Python QuixBugs programs; after reproducibility screening, 1,885 of 2,864 ConDefects pairs and 655 of 835 Defects4J faults remain eligible. Screening criteria and exclusions are reported in the Appendix, and all repair rates use the eligible populations as denominators.

Implementation Details. PAUSE is implemented in Python 3.10 using GPT-5.5 as the default backbone. Five role-specialized agents use dedicated prompts and share a structured repair trajectory, while a separate non-agent retrieval module queries distilled decision constraints. Each instance is limited to five iterations with temperature 0.2 and top-p 0.95. Python candidates run in isolated subprocesses and Java candidates in a Defects4J-compatible environment, both with fixed timeouts. Additional implementation and hardware details are provided in the Appendix.

Metrics. We report plausible repair rate ($\mathrm{PRR} = \mathrm{PR}/N_{\mathrm{eligible}}$), confirmed repair rate ($\mathrm{CRR} = \mathrm{CR}/N_{\mathrm{eligible}}$), confirmation precision ($\mathrm{CP} = \mathrm{CR}/\mathrm{PR}$), and first-attempt confirmed repair rate (C@1). Efficiency is measured by calls, tokens, time, and cost, while selective behavior is assessed using NOOP accuracy, over-repair rate, safe-rejection precision, unsupported-acceptance rate, and premature-stop rate. For paired repair outcomes, we apply McNemar's test with Holm correction and report 95% bootstrap confidence intervals for repair-rate differences and matched-pairs odds ratios. For continuous metrics, we use the Wilcoxon signed-rank test with matched-pairs rank-biserial correlation.

4.2 OVERALL REPAIR EFFECTIVENESS AND EFFICIENCY

We compare PAUSE with Direct Repair, ChatRepair Xia & Zhang (2024), and RepairAgent Bouzenia et al. (2025) under the same GPT-5.5 backbone, externally observable task evidence, decoding settings, five-attempt budget, and validation protocol. RepairAgent is adapted only to the Python execution interfaces of QuixBugs and ConDefects while retaining its original repair workflow. On Defects4J, we additionally evaluate RepairLLaMA Silva et al. (2025), TBar Liu et al. (2019), SimFix Jiang et al. (2018), and CURE Jiang et al. (2021) using their released implementations. These

Table 2: Repair effectiveness (%). PRR: plausible repair rate; CRR: confirmed repair rate; CP: confirmation precision. Bold marks the best applicable result; underline marks the second-best.

Method	**QuixBugs** ($N = 40$)			**ConDefects** ($N = 1{,}885$)			**Defects4J** ($N = 655$)		
	PRR	**CRR**	**CP**	**PRR**	**CRR**	**CP**	**PRR**	**CRR**	**CP**
TBar		N/A			N/A		17.7	10.7	60.3
SimFix		N/A			N/A		15.9	9.3	58.7
CURE		N/A			N/A		29.9	22.4	75.0
RepairLLaMA		N/A			N/A		35.0	27.6	79.0
Direct Repair	85.0	67.5	79.4	42.0	31.0***	73.7	38.2	28.9***	75.6
ChatRepair	90.0	72.5	80.6	46.4	36.0***	77.6	42.9	33.9***	79.0
RepairAgent†	<u>92.5</u>	<u>75.0</u>	<u>81.1</u>	<u>48.5</u>	<u>38.2</u>***	<u>78.8</u>	<u>45.2</u>	<u>35.9</u>*	<u>79.4</u>
PAUSE	**95.0**	**77.5**	**81.6**	**51.2**	**41.2**	**80.4**	**48.4**	**38.9**	**80.4**

N/A denotes an unsupported Python benchmark configuration. †RepairAgent uses a Python-compatible execution-interface adaptation. Stars beside controlled-baseline CRR values denote Holm-adjusted McNemar comparisons with PAUSE: $^{*}p_{\mathrm{adj}} < 0.05$, $^{**}p_{\mathrm{adj}} < 0.01$, and $^{***}p_{\mathrm{adj}} < 0.001$. Full confidence intervals and effect sizes are reported in the Appendix.

Table 3: Inference efficiency over 2,580 eligible faults. Calls, tokens, and costs are means per fault; time is the median. Lower is better.

Method	**Calls**	**Tok. (k)**	**Time (s)**	**$/Bug**	**$/CR**
Direct Repair	**2.8**	**19.6**	**39.6**	**0.29**	**0.94**
ChatRepair	<u>3.9</u>	<u>27.8</u>	<u>58.2</u>	<u>0.45</u>	1.25
RepairAgent	8.7	67.5	126.4	1.12	2.93
PAUSE	4.1	28.9	63.7	0.49	<u>1.19</u>

Java-oriented implementations are not evaluated on QuixBugs-Python or ConDefects-Python because they do not support the corresponding Python benchmark configurations without substantive changes to their original repair pipelines. Such entries are marked as N/A rather than unsuccessful repairs. Original tests determine plausibility, while confirmation requires the frozen validation protocol. Control instances are analyzed separately for selective repair and governance.

As shown in Table 2, PAUSE achieves the highest PRR, CRR, and CP on all three benchmarks. Relative to RepairAgent, the strongest controlled baseline, it improves CRR by 2.5 percentage points on QuixBugs and 3.0 points on both ConDefects and Defects4J. Its gains over Direct Repair remain approximately 10 points across datasets, indicating benefits beyond the shared LLM backbone; on Defects4J, it also exceeds RepairLLaMA, the strongest applicable released neural baseline, by 11.3 points. CP reaches 81.6% on QuixBugs and 80.4% on both larger benchmarks, indicating that more plausible patches survive independent confirmation. Moreover, PAUSE achieves a pooled C@1 of 34.2%, compared with 24.6% for Direct Repair, 28.8% for ChatRepair, and 30.8% for RepairAgent, showing that its advantage is present from the first attempt. The trend across Python and Java suggests that the gains are not confined to one language or execution environment, although the QuixBugs result warrants caution because it contains only 40 programs.

The efficiency results in Table 3 reveal a different trade-off across baselines. Direct Repair has the lowest inference cost but a 10.2-point lower pooled CRR than PAUSE. Compared with ChatRepair, PAUSE incurs slightly more calls, tokens, time, and cost per fault, while reducing cost per confirmed repair from $1.25 to $1.19 through its higher confirmed repair rate. Relative to RepairAgent, PAUSE reduces model calls by 52.9%, tokens per fault by 57.2%, median wall-clock time by 49.6%, and cost per confirmed repair by 59.4%. These results suggest that selective action and governed termination avoid unproductive attempts, giving PAUSE the strongest effectiveness–efficiency balance among the controlled methods.

4.3 Ablation Study

We ablate the Profiler, Retrieval, typed observation normalization, and Reflector, and replace the specialized reasoning roles with a *Unified Reasoner*. All variants retain NoOp, Executor, and Gov-

Table 4: Pooled ablation results over 2,580 eligible faults. CRR and C@1 are percentages; Δ denotes the CRR drop from PAUSE. Tok./CR is in thousands.

Variant	CRR	Δ	C@1	Calls	Tok./CR
w/o Profiler	37.7	3.5	29.8	3.9	72.2
w/o Retrieval	39.0	2.2	31.8	4.0	72.0
w/o Typed Obs.	34.6	6.6	26.9	4.9	103.4
w/o Reflector	36.1	5.1	**34.2**	4.5	90.2
Unified	35.5	5.7	28.7	**3.8**	97.7
PAUSE	**41.2**	–	**34.2**	4.1	**70.2**

Table 5: Generalization across LLM backbones. Each entry reports Direct Repair/PAUSE. Dataset columns and pooled CRR report confirmed repair rates; CP and C@1 are also percentages. Calls reports the mean number of model calls per fault.

Backbone	CRR			Pooled	CP	C@1	Calls
	Quix.	ConD.	D4J	CRR			
GPT-5.5	67.5/**77.5**	31.0/**41.2**	28.9/**38.9**	31.0/**41.2**	74.3/**80.5**	24.6/**34.2**	2.8/4.1
DeepSeek-V4-Pro	65.0/**75.0**	29.2/**39.0**	27.0/**36.9**	29.2/**39.0**	73.5/**80.2**	22.8/**32.1**	2.9/4.0
Qwen3.7-Plus	62.5/**72.5**	27.7/**37.4**	26.0/**35.4**	27.8/**37.4**	72.7/**79.8**	21.2/**30.6**	2.7/4.2

ernor, and use the same GPT-5.5 backbone, inputs, five-attempt budget, and validation protocol. Table 4 reports pooled results; dataset-level results are provided in the Appendix.

The ablation results indicate that every ablation lowers CRR. Removing typed observations causes the largest drop (-6.6 points) and increases Tok./CR from 70.2k to 103.4k, indicating that structured execution feedback helps avoid ineffective retries. The Unified Reasoner makes fewer calls but reduces CRR by 5.7 points and consumes 97.7k tokens per confirmed repair, suggesting that role specialization improves evidence use. This confirms the value of role-specialized reasoning.

Removing the Reflector leaves C@1 unchanged but lowers final CRR by 5.1 points, showing that trajectory reflection mainly benefits later attempts. Profiler removal primarily harms first-attempt performance, while Retrieval provides a smaller but consistent 2.2-point CRR gain through experience-derived constraints. Together, these components achieve the highest CRR and lowest Tok./CR.

4.4 GENERALIZATION ACROSS LLM BACKBONES

We instantiate PAUSE and Direct Repair with GPT-5.5, DeepSeek-V4-Pro, and Qwen3.7-Plus to examine backbone dependence. Within each backbone, both methods share the task evidence, decoding settings, five-attempt budget, and validation protocol; PAUSE uses the same prompts, observation schema, and governance rules across backbones. Metrics are pooled over all 2,580 eligible faults. We omit cross-provider token and cost comparisons because tokenization and pricing differ.

Table 5 shows that PAUSE improves pooled CRR over same-backbone Direct Repair by 10.2 points with GPT-5.5, 9.8 points with DeepSeek-V4-Pro, and 9.6 points with Qwen3.7-Plus; dataset-level gains range from 9.4 to 10.2 points. Pooled CP remains between 79.8% and 80.5%, versus 72.7%–74.3% for Direct Repair, indicating that more plausible patches pass independent confirmation. C@1 improves by 9.3–9.6 points, showing that the advantage emerges in the first repair iteration. PAUSE averages 4.0–4.2 calls per fault, only 1.1–1.5 more than Direct Repair, so substantially longer trajectories are unlikely to explain its CRR gain. Absolute performance still varies by backbone: pooled CRR falls from 41.2% with GPT-5.5 to 37.4% with Qwen3.7-Plus. Thus, PAUSE consistently improves repair across the evaluated backbones, although absolute repair rates vary.

4.5 SELECTIVE REPAIR AND GOVERNANCE

We assess whether PAUSE avoids unsupported modifications and terminates unproductive repair trajectories. The control set contains 300 verified-correct programs and 300 faulty programs exposed to

Table 6: Selective repair and governance results. Rates are percentages, and arrows indicate the preferred direction.

(a) Safety on control instances

Variant	NoOp Acc. ↑	ORR ↓	SRP ↑	UAR ↓
w/o NoOp	0.0	11.7	92.0	3.3
w/o Safe Rej.	90.7	2.3	–	9.7
Fixed Budget	91.0	3.0	93.8	2.3
First Plausible	88.3	6.7	81.5	12.7
PAUSE	**91.3**	**1.7**	**94.1**	**2.0**

(b) Stopping and repair effectiveness

Variant	Prem. ↓	CRR ↑	Calls ↓
w/o NoOp	2.9	40.7	4.7
w/o Safe Rej.	2.5	40.9	4.6
Fixed Budget	0.0*	**41.6**	7.4
First Plausible	8.9	37.0	**3.2**
PAUSE	3.1	41.2	4.1

ORR: over-repair rate; SRP: safe-rejection precision; UAR: unsupported-acceptance rate; Prem.: premature-stop rate. CRR and Calls are pooled over 2,580 eligible faults. *Zero by construction.

unreliable feedback, with 75 instances each of missing, truncated, conflicting, and non-deterministic observations. Complete evidence is withheld during repair and used only for offline decision labeling. These controls are excluded from PRR and CRR, so the governance analysis does not alter the benchmark repair rates and instead isolates selective behavior under uncertainty.

We compare four variants under the same backbone, inputs, decoding settings, attempt budget, and confirmation protocol. *w/o NoOp* forces modification whenever repair continues; *w/o Safe Rejection* removes the evidence-based REJECT branch; *Fixed Budget* runs every unresolved trajectory to five attempts; and *First Plausible* accepts the first patch passing the original tests. Together, these variants probe abstention, safe rejection, continuation, and test-based acceptance. The frozen confirmation protocol is used only to score final outputs.

NoOp Acc. and ORR measure initial abstention and terminal over-repair on correct programs, while SRP and UAR measure rejection precision and unsupported terminal acceptance under unreliable feedback. Table 6(a) shows that PAUSE selects NOOP for 91.3% of correct programs and limits accepted over-repair to 1.7%; forcing modification increases ORR to 11.7%. Under unreliable feedback, PAUSE achieves 94.1% rejection precision and a 2.0% unsupported-acceptance rate. Removing safe rejection raises UAR to 9.7%, while accepting the first plausible patch raises it to 12.7%. These results distinguish the roles of NOOP, which avoids an unsupported modification during planning, and REJECT, which prevents an unsupported terminal acceptance during governance.

Premature Stop counts unconfirmed terminal decisions whose paired continuation from the same state reaches a confirmed repair within the remaining budget. This paired continuation tests whether stopping forfeits a repair that was still attainable, rather than counting every early termination as premature. Table 6(b) shows that running every unresolved case to the full budget improves CRR by only 0.4 points but increases calls from 4.1 to 7.4. Governed termination therefore reduces calls by 44.6% while retaining about 99% of the confirmed repairs obtained by Fixed Budget. First Plausible uses fewer calls, but lowers CRR to 37.0% and produces the highest UAR. The slightly lower premature-stop rates of the two permissive ablations are likewise accompanied by higher over-repair or unsupported acceptance. Together, planning-level abstention and governance-level rejection address complementary risks while avoiding most calls spent on unproductive continuation.

5 CONCLUSION

We presented PAUSE, a selective program repair framework that combines first-class abstention with typed execution feedback and trajectory-aware governance. PAUSE treats PATCH and NOOP as explicit actions, interprets heterogeneous execution outcomes as uncertain observations, and separately assesses candidate acceptability and the value of continued repair. Experiments on QuixBugs, ConDefects, and Defects4J demonstrate improved confirmed repair effectiveness, while controlled evaluations on correct programs and unreliable feedback further validate the safety and efficiency of the proposed design under uncertain execution conditions.

Limitations and Future Work. PAUSE is evaluated on Python and Java benchmarks in controlled environments, while its confirmation protocol cannot ensure semantic correctness. Future work will extend to repository-level and multilingual repair, richer specification signals, and learned governance with broader experience transfer across diverse repair tasks and execution environments.

AI USE STATEMENT

Generative AI tools were used in this work in two distinct roles. First, large language models are integral to the proposed PAUSE framework. They support role-specialized agents in analyzing repair context, selecting between PATCH and NOOP, interpreting execution feedback, assessing repair trajectories, and governing repair decisions. The models, experimental settings, and agent procedures are described in the main text and appendix. Second, generative AI tools assisted with language polishing, wording refinement, and improving the readability of the manuscript.

All AI-assisted text was reviewed and revised by the authors. The authors verified the reported methods, experimental results, figures, tables, and cited references, and take full responsibility for the final paper. Generative AI tools were not listed as authors and did not replace the independent confirmation protocol used to evaluate repair outcomes.

REFERENCES

Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. RepairAgent: An autonomous, LLM-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2188–2200, 2025. doi: 10.1109/ICSE55347.2025.00157.

Yu Cheng, Zhongxin Liu, Zhenchang Xing, Chao Ni, Qing Huang, and Xiaoxue Ren. PracRepair: LLM-empowered automated program repair inspired by human-like debugging practices, 2026. URL https://arxiv.org/abs/2606.17612.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46701–46723, 2023.

Zhiwei Fei, Jidong Ge, Chuanyi Li, Tianqi Wang, Yuning Li, Haodong Zhang, LiGuo Huang, and Bin Luo. Patch correctness assessment: A survey. *ACM Transactions on Software Engineering and Methodology*, 34(2):55:1–55:50, 2025. doi: 10.1145/3702972.

Mingyang Geng, Shanzhi Gu, Zhipeng Liu, Chuanfu Xu, Zhaoyang Qu, and Haotian Wang. Failure localization in multi-agent code generation via knowledge-guided and transferable reasoning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(1):318–326, 2026. doi: 10.1609/aaai.v40i1.36993.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Sx038qxjek.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. Evolving paradigms in automated program repair: Taxonomy, challenges, and opportunities. *ACM Computing Surveys*, 57(2), 2024. doi: 10.1145/3696450.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4912–4944. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.269. URL https://aclanthology.org/2024.acl-long.269/.

Elkhan Ismayilzada, Md Mazba Ur Rahman, Dongsun Kim, and Jooyong Yi. Poracle: Testing patches under preservation conditions to combat the overfitting problem of program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):44:1–44:39, 2024. doi: 10.1145/3625293.

Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2018. doi: 10.1145/3213846.3213871.

Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1161–1173. IEEE, 2021. doi: 10.1109/ICSE43902.2021.00107.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, pp. 1430–1442, 2023. doi: 10.1109/ICSE48619.2023.00125.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014. doi: 10.1145/2610384.2628055.

Jiaolong Kong, Xiaofei Xie, Mingfei Cheng, Shangqing Liu, Xiaoning Du, and Qi Guo. ContrastRepair: Enhancing conversation-based automated program repair via contrastive test case pairs. *ACM Transactions on Software Engineering and Methodology*, 34(8):216:1–216:31, 2025. doi: 10.1145/3719345.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Companion Proceedings of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pp. 55–56, 2017. doi: 10.1145/3135932.3135941.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 31–42. Association for Computing Machinery, 2019. doi: 10.1145/3293882.3330577.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=pPjZIOuQuF.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, et al. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-bench: Testing and validating real-world bug-fixes with code agents. In *Advances in Neural Information Processing Systems*, volume 37, pp. 81857–81887, 2024. doi: 10.52202/079017-2601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/94f093b41fc2666376fb1f667fe282f3-Abstract-Conference.html.

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. PMLR, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Justyna Petke, Matias Martinez, Maria Kechagia, Aldeida Aleti, and Federica Sarro. The patch overfitting problem in automated program repair: Practical magnitude and a baseline for realistic benchmarking. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 452–456. Association for Computing Machinery, 2024. doi: 10.1145/3663529.3663776.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, et al. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

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*, volume 36, pp. 8634–8652, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

André Silva, Sen Fang, and Martin Monperrus. RepairLLaMA: Efficient representations and fine-tuned adapters for program repair. *IEEE Transactions on Software Engineering*, 51(8):2366–2380, 2025. doi: 10.1109/TSE.2025.3581062.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: LLM-based multi-agent framework for GitHub issue resolution. In *Advances in Neural Information Processing Systems*, volume 37, pp. 51963–51993, 2024. doi: 10.52202/079017-1647. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d1f02132ef51602adf07000ca5b6138-Abstract-Conference.html.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, et al. DebugBench: Evaluating debugging capability of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 4173–4198, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.247. URL https://aclanthology.org/2024.findings-acl.247/.

Konstantinos Vergopoulos, Mark Niklas Müller, and Martin Vechev. Automated benchmark generation for repository-level coding tasks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 61240–61265. PMLR, 2025. URL https://proceedings.mlr.press/v267/vergopoulos25a.html.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 53270–53290. PMLR, 2024a. URL https://proceedings.mlr.press/v235/wu24a.html.

Jiaqing Wu, Tong Wu, Manqing Zhang, Yunwei Dong, and Bo Shen. Runtime execution traces guided automated program repair with multi-agent debate, 2026.

Yonghao Wu, Zheng Li, Jie M. Zhang, and Yong Liu. ConDefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 642–646, 2024b. doi: 10.1145/3663529.3663815.

Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 819–831, 2024. doi: 10.1145/3650212.3680323.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, pp. 1482–1494, 2023. doi: 10.1109/ICSE48619.2023.00129.

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

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601.

He Ye and Martin Monperrus. ITER: Iterative neural repair for multi-location patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 10:1–10:13, 2024. doi: 10.1145/3597503.3623337.

Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. ThinkRepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1274–1286, 2024. doi: 10.1145/3650212.3680359.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13643–13658. Association for Computational Linguistics, 2024a. doi: 10.18653/v1/2024.acl-long.737. URL https://aclanthology.org/2024.acl-long.737/.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024b. doi: 10.1145/3650212.3680384.

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*, pp. 851–870, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.49. URL https://aclanthology.org/2024.findings-acl.49/.

A Method Specification and Pseudocode

A.1 Repair State and Decision Space

Given an initial program P_0, available specification evidence $\mathcal{S}$, and initial execution evidence $\mathcal{E}_0$, PAUSE formulates repair as an iterative decision process. At iteration t, the repair state is

$$s_t = (P_t, \mathcal{S}, \mathcal{E}_t, \mathcal{H}_t, \mathcal{M}_t)\,, \tag{13}$$

where P_t is the current program, $\mathcal{S}$ contains the available specification evidence, $\mathcal{E}_t$ contains the latest execution evidence, $\mathcal{H}_t$ is the accumulated repair trajectory, and $\mathcal{M}_t$ is the experience repository available at the current iteration.

The specification evidence may include a task description, function signature, visible tests, input constraints, expected outputs, and dataset-specific metadata. The execution evidence is obtained by compiling, executing, or testing the current candidate. Because this evidence may be incomplete or unreliable, PAUSE treats execution as decision evidence rather than as a semantic correctness oracle.

The trajectory is represented as

$$\mathcal{H}_t = [h_0, h_1, \ldots, h_{t-1}]\,, \tag{14}$$

where an iteration record is

$$h_i = \left(P_i, c_i, R_i, a_i, \Delta_i, \widehat{P}_i, o_i, r_i, g_i\right). \tag{15}$$

Here, c_i is the context profile, R_i is the retrieved experience, a_i is the planning action, Δ_i is an optional source modification, $\widehat{P}_i$ is the candidate program, o_i is the typed execution observation, r_i is the trajectory assessment, and g_i is the governance decision.

The Planner chooses between two actions:

$$a_t \in \{\mathrm{Patch}, \mathrm{NoOp}\}\,. \tag{16}$$

For a Patch action, the candidate is constructed as:

$$\widehat{P}_t = \mathrm{Apply}\,(P_t, \Delta_t)\,. \tag{17}$$

Table 7: Interfaces of the six functional roles in PAUSE.

Role	Input	Output	Function
Profiler	P_t, $\mathcal{S}$, $\mathcal{E}_t$, and $\mathcal{H}_t$	Context profile c_t	Identifies program intent, suspicious behavior, relevant code regions, and risk-sensitive structures that should be considered before modification.
Recaller	c_t, $\mathcal{H}_t$, and $\mathcal{M}_t$	Decision constraints R_t	Retrieves experience relevant to the current context and trajectory. The retrieved experience constrains decisions rather than supplying a patch template.
Planner	c_t, $\mathcal{E}_t$, $\mathcal{H}_t$, and R_t	Action a_t and optional edit Δ_t	Chooses between Patch and NoOp. When a patch is justified, it proposes a localized source modification.
Executor	$\widehat{P}_t$, $\mathcal{S}$, and the execution configuration	Typed observation o_t	Compiles, executes, or tests the candidate and normalizes the raw output into a structured execution observation.
Reflector	P_t, $\widehat{P}_t$, o_t, and $\mathcal{H}_t$	Trajectory assessment r_t	Compares the current outcome with previous iterations and assesses progress, regression, stagnation, evidence reliability, and the value of another attempt.
Governor	$\widehat{P}_t$, o_t, r_t, and the remaining budget	Decision g_t	Combines current-state acceptability with continuation value to produce $\mathrm{AcceptPatch}$, $\mathrm{Preserve}$, $\mathrm{Continue}$, or Reject.

For a NoOp action, the source program remains unchanged:

$$\widehat{P}_t = P_t, \qquad \Delta_t = \varnothing. \tag{18}$$

A NoOp action means that the current evidence does not justify an immediate source-level modification. It is not itself a terminal decision. The current program remains subject to execution, reflection, and governance.

The Governor produces one of four decisions:

$$g_t \in \{\mathrm{AcceptPatch}, \mathrm{Preserve}, \mathrm{Continue}, \mathrm{Reject}\}\,. \tag{19}$$

Their meanings are as follows:

- $\mathrm{AcceptPatch}$ terminates the trajectory and accepts a candidate that differs from the initial program;
- $\mathrm{Preserve}$ terminates without modifying the initial program because the available evidence supports retaining it;
- $\mathrm{Continue}$ starts another iteration because the current state is not yet acceptable but further repair remains justified;
- Reject terminates without accepting a generated patch because the current state is unsupported and continuation is no longer justified.

These are repair-time decisions based only on the evidence available to PAUSE. Confirmed correctness is assessed separately by the evaluation protocol.

A.2 ROLE INTERFACES

PAUSE contains six functional roles: Profiler, Recaller, Planner, Executor, Reflector, and Governor. Table 7 summarizes their interfaces.

The roles are deliberately separated by responsibility. In particular, the Planner proposes whether and how to modify the program, whereas the Governor independently determines whether the resulting state should be accepted, preserved, continued, or rejected.

Table 8: Typed execution outcomes used by PAUSE.

Type	Observed condition	Repair implication
Pass	The candidate completes all currently available compilation, execution, and test checks.	Provides positive repair-time evidence, but does not by itself establish correctness beyond the available checks.
AssertionFail	Execution completes, but an observed output violates an expected value or test assertion.	Provides behavioral evidence that can support a localized repair.
CompileError	The candidate cannot be parsed, compiled, linked, or loaded.	Indicates syntactic, typing, dependency-resolution, or build-level invalidity.
RuntimeError	Execution terminates with an exception, crash, or abnormal exit.	Indicates an invalid execution path or state reached at runtime.
Timeout	Execution exceeds the configured time or resource limit without a conclusive outcome.	Provides incomplete evidence that may result from either the program or the execution environment.
NonTermination	Repeated or instrumented execution indicates persistent looping, recursion, or lack of execution progress.	Provides stronger evidence of non-terminating program behavior than an isolated timeout.
Instability	Repeated executions under the same configuration produce materially different outcomes.	Indicates non-determinism, flaky behavior, or unstable execution evidence.

A.3 TYPED EXECUTION FEEDBACK

The Executor converts heterogeneous execution output into a typed observation

$$o_t = (y_t, q_t, d_t)\,, \tag{20}$$

where y_t is the execution-outcome type, q_t describes feedback quality, and d_t stores the available diagnostic payload. When a task-level outcome is observable, the implementation assigns one of seven types:

$$\begin{aligned} y_t \in \big\{&\mathrm{Pass}, \mathrm{AssertionFail}, \mathrm{CompileError}, \\ &\mathrm{RuntimeError}, \mathrm{Timeout}, \mathrm{NonTermination}, \mathrm{Instability}\big\}. \end{aligned} \tag{21}$$

Table 8 summarizes their operational interpretations.

Feedback quality is represented separately from the task-level outcome:

$$q_t = \left(q_t^{\mathrm{available}}, q_t^{\mathrm{complete}}, q_t^{\mathrm{consistent}}, q_t^{\mathrm{stable}}\right). \tag{22}$$

The four attributes indicate whether the feedback is available, complete, mutually consistent, and stable across repeated observations. They allow PAUSE to represent four unreliable-feedback conditions:

- *missing*, when no usable outcome or diagnostic is returned;
- *truncated*, when only part of the diagnostic or test output is available;
- *conflicting*, when different observations provide incompatible evidence;
- *non-deterministic*, when repeated executions produce materially different outcomes.

The outcome type and feedback quality are kept separate because the same task-level outcome can have different levels of reliability. For example, an assertion failure may be complete and stable or may be reported through truncated and conflicting feedback.

The payload d_t contains the available outcome-specific information, such as a failing test identifier, expected and actual values, compiler diagnostic, exception message, stack location, elapsed time, or repeated execution outcomes. Missing fields remain explicitly unavailable and are not inferred by the Executor.

Algorithm 1 PAUSE Selective Program Repair

Require: Initial program P_0, specification evidence $\mathcal{S}$, initial execution evidence $\mathcal{E}_0$, experience repository $\mathcal{M}_0$, iteration budget T
Ensure: Terminal program $P^\star$, decision g, trajectory $\mathcal{H}$

$\mathcal{H}_0 \leftarrow [\,]$
for $t = 0, \ldots, T-1$ **do**
  $c_t \leftarrow \textsc{Profiler}(P_t, \mathcal{S}, \mathcal{E}_t, \mathcal{H}_t)$
  $R_t \leftarrow \textsc{Recaller}(c_t, \mathcal{H}_t, \mathcal{M}_t)$
  $(a_t, \Delta_t) \leftarrow \textsc{Planner}(c_t, \mathcal{E}_t, \mathcal{H}_t, R_t)$
  if $a_t = \mathrm{Patch}$ **then**
    $\widehat{P}_t \leftarrow \textsc{Apply}(P_t, \Delta_t)$
  else
    $\widehat{P}_t \leftarrow P_t$
    $\Delta_t \leftarrow \varnothing$
  end if
  $o_t \leftarrow \textsc{Executor}(\widehat{P}_t, \mathcal{S})$
  $r_t \leftarrow \textsc{Reflector}(P_t, \widehat{P}_t, a_t, o_t, \mathcal{H}_t)$
  $b_t \leftarrow T - t - 1$
  $g_t \leftarrow \textsc{Governor}(\widehat{P}_t, P_0, o_t, r_t, b_t)$
  $h_t \leftarrow (P_t, c_t, R_t, a_t, \Delta_t, \widehat{P}_t, o_t, r_t, g_t)$
  $\mathcal{H}_{t+1} \leftarrow \mathcal{H}_t \,\|\, [h_t]$
  if $g_t = \mathrm{AcceptPatch}$ **then**
    return $(\widehat{P}_t, \mathrm{AcceptPatch}, \mathcal{H}_{t+1})$
  else if $g_t = \mathrm{Preserve}$ **then**
    return $(P_0, \mathrm{Preserve}, \mathcal{H}_{t+1})$
  else if $g_t = \mathrm{Reject}$ **then**
    return $(P_0, \mathrm{Reject}, \mathcal{H}_{t+1})$
  else
    $P_{t+1} \leftarrow \widehat{P}_t$
    $\mathcal{E}_{t+1} \leftarrow o_t$
    $\mathcal{M}_{t+1} \leftarrow \mathcal{M}_t$
  end if
end for
return $(P_0, \mathrm{Reject}, \mathcal{H}_T)$

A.4 Selective Repair Procedure

Algorithm 1 presents the end-to-end procedure. The iteration budget T is an upper bound: PAUSE may terminate before consuming all available iterations.

A.5 Selective Repair Procedure

Algorithm 1 summarizes the selective repair procedure. The iteration budget T is an upper bound: PAUSE may terminate before using all available iterations.

The symbol $\perp$ indicates that no generated patch is accepted. In this case, the original artifact is left unchanged and the instance is reported as rejected rather than repaired.

When the Governor selects $\mathrm{Continue}$, the candidate and its execution observation become the state of the next iteration. The full history remains available, allowing later roles to compare the new state with earlier candidates and outcomes.

A.6 Trajectory Reflection and Dual-Condition Governance

The Reflector evaluates the latest outcome in the context of the entire trajectory. Its output is represented as

Table 9: Dual-condition governance in PAUSE.

α_t	ν_t	Budget	Decision
1	0 or 1	Any	Accept or preserve
0	1	$b_t > 0$	Continue
0	0	Any	Reject
0	1	$b_t = 0$	Reject

$$r_t = (p_t, \sigma_t, \nu_t, z_t), \tag{23}$$

where p_t describes trajectory progress, σ_t represents stagnation or regression signals, ν_t is the continuation judgment, and z_t contains the supporting rationale.

The progress assessment is selected from

$$p_t \in \{\text{Improved}, \text{Unchanged}, \text{Regressed}, \text{Uncertain}\}. \tag{24}$$

Progress may include eliminating a compilation error, reducing observed test failures, replacing an uninformative timeout with a localized failure, or obtaining more complete and stable evidence. Repeatedly producing equivalent edits, reproducing the same failure without new evidence, oscillating between previous states, or introducing new failures contributes to σ_t.

The continuation judgment is

$$\nu_t \in \{0, 1\}, \tag{25}$$

where $\nu_t = 1$ means that another attempt remains justified by actionable evidence, recent progress, or an unexplored repair direction. The condition $\nu_t = 0$ indicates that the trajectory has stagnated or that the available evidence does not support another modification.

The Governor independently evaluates whether the current candidate is sufficiently supported. Let

$$\alpha_t = \text{Acceptable}\left(\widehat{P}_t, o_t, r_t\right), \qquad \alpha_t \in \{0, 1\}. \tag{26}$$

The judgment $\alpha_t = 1$ requires that the current outcome and its feedback quality provide sufficient repair-time support for terminal acceptance or preservation. It is not a formal proof of program correctness.

Let b_t denote the number of remaining iterations. The governance rule is

$$g_t = \begin{cases} \text{AcceptPatch}, & \alpha_t = 1 \wedge \widehat{P}_t \neq P_0, \\ \text{Preserve}, & \alpha_t = 1 \wedge \widehat{P}_t = P_0, \\ \text{Continue}, & \alpha_t = 0 \wedge \nu_t = 1 \wedge b_t > 0, \\ \text{Reject}, & \text{otherwise}. \end{cases} \tag{27}$$

This rule separates current-state acceptability from the value of future repair. A supported state terminates immediately, even when unused iterations remain. An unsupported state continues only when another attempt remains justified and the iteration budget has not been exhausted. Otherwise, the trajectory is rejected.

The final distinction between AcceptPatch and Preserve depends on whether the accepted state differs from the initial program. Consequently, a NoOp decision at a later iteration does not automatically imply preservation if an earlier accepted modification remains in the current candidate.

A.7 EXPERIENCE RETRIEVAL AND DISTILLATION

The experience repository stores reusable decision constraints rather than source-level patch templates. At iteration t, the Recaller retrieves relevant experience as

$$R_t = \text{Retrieve}\left(c_t, \mathcal{H}_t, \mathcal{M}_t\right). \tag{28}$$

An experience entry is represented as

$$m_j = (\kappa_j, \phi_j, \eta_j)\,, \tag{29}$$

where κ_j is a context signature, ϕ_j is an execution or trajectory pattern, and η_j is a reusable decision constraint.

For example, an experience constraint may state that incomplete or conflicting evidence should not independently trigger a patch, that repeated equivalent failures indicate stagnation, or that a stable and localized failure can support a targeted modification. Such constraints guide the Planner, Reflector, and Governor, but do not provide code to be copied into the current candidate.

After an eligible trajectory terminates, its decision-relevant information may be distilled as

$$\mathcal{M}' = \mathrm{Distill}\left(\mathcal{M}, \mathcal{H}_{t+1}, g_t\right). \tag{30}$$

The distilled experience may originate from an accepted repair, a justified preservation decision, a safe rejection, an unreliable feedback pattern, or a stagnating trajectory. Instance-specific source code and generated patch text are excluded from the reusable constraint.

The definitions and rules in this appendix are operational specifications of PAUSE, not theoretical guarantees. Explanations of why selective planning, typed feedback, or dual-condition governance may reduce unsupported modification are design motivations whose effects are evaluated empirically in the main experiments and ablation studies.

B Background and Operational Terminology

B.1 Automated Program Repair and Patch Correctness

APR aims to construct a program modification that resolves an observed defect under an available behavioral specification. In test-suite-based APR, this specification is typically represented by tests that expose at least one failure in the original program. A conventional repair workflow includes identifying relevant program context, generating candidate modifications, and validating the resulting candidates against the available testsHuang et al. (2024).

A candidate that passes the tests used during repair is commonly described as a *plausible patch*. Plausibility is defined relative to the available test suite and does not establish correctness for inputs or behaviors not covered by that suite. A plausible patch may therefore satisfy the observed tests while retaining or introducing incorrect behavior. This limitation is commonly referred to as *patch overfitting* Fei et al. (2025).

In this paper, a *plausible repair* is a generated patch that passes the original repair-time tests. A *confirmed repair* is a plausible patch that additionally satisfies the frozen confirmation protocol used in our evaluation. The confirmation protocol is applied only after repair and is not visible to the roles in PAUSE.

B.2 Iterative LLM-Based Repair

LLM-based program repair commonly provides a model with a faulty program, available specification information, and observed failure evidence. A single-turn method generates a patch from the initial context, whereas an iterative method incorporates the outcomes of earlier candidates into subsequent attempts.

Figure 2 illustrates a generalized agent-driven repair loop, in which program context and execution feedback inform successive repair actions.

Conversation-based repair, for example, uses compiler or test feedback to guide repeated interactions with a language model Xia & Zhang (2024). Other approaches incorporate explicit reasoning or accumulated repair experience into iterative generation Yin et al. (2024). More generally, reflection-based language agents use previous actions and outcomes to inform later decisions within the same trajectoryShinn et al. (2023).

In many iterative repair workflows, execution feedback is primarily used to improve the next candidate patch. In PAUSE, the same feedback also supports repair-control decisions. The method

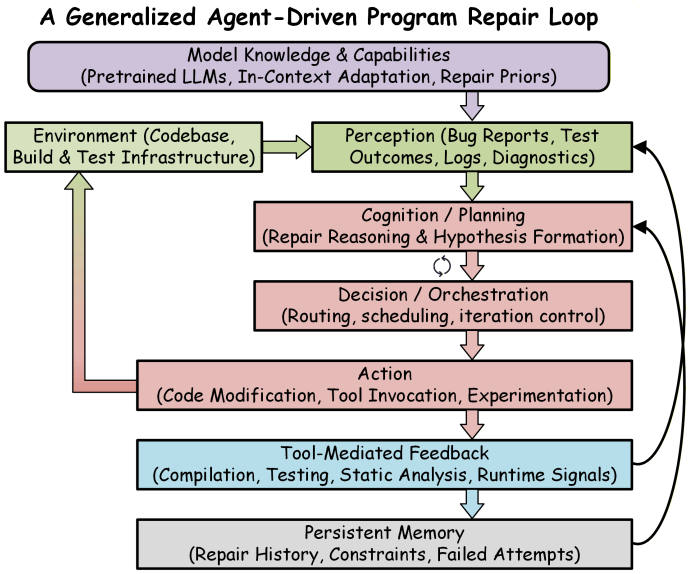

Figure 2: A generalized agent-driven program repair loop connecting program context, repair actions, execution feedback, and memory.

considers not only how a candidate might be modified, but also whether modification is currently justified, whether another attempt remains useful, and whether the trajectory should terminate without accepting a new patch.

B.3 Agentic Repair and Functional Decomposition

A software engineering agent combines language-model reasoning with actions performed in an external environment, such as inspecting code, editing source files, compiling programs, and running tests. Agentic repair systems organize these interactions into structured workflows that repeatedly observe the environment and act on the resulting evidence Bouzenia et al. (2025).

The six components in PAUSE are best understood as functional roles within one shared repair process rather than as six independent repair systems. The Profiler extracts repair-relevant context, the Recaller retrieves decision constraints, the Planner selects Patch or NoOp, the Executor produces typed execution observations, the Reflector evaluates the repair trajectory, and the Governor determines whether to accept, preserve, continue, or reject.

This decomposition separates patch generation from repair governance. In particular, the role that proposes a modification does not independently decide whether that modification should be accepted. The exact interfaces of the six roles are specified in Appendix A.2.

B.4 Execution Evidence under Uncertainty

Execution feedback provides concrete information about candidate behavior, but different outcomes have different implications for repair. An assertion failure may expose an expected–actual mismatch, a compilation error may identify syntactic or typing invalidity, and a runtime exception may reveal an invalid execution path. A timeout or an unstable result may provide less localized evidence and may require additional interpretation.

The reliability of feedback is distinct from its outcome type. For example, an assertion failure may be complete and reproducible, or it may be truncated and inconsistent across repeated executions. Similarly, an apparent passing outcome may be insufficiently reliable when only part of the intended execution is observed.

The experimental setting considers four forms of unreliable feedback:

Table 10: Operational terminology used in PAUSE.

Term	Category	Operational meaning
Patch	Planner action	Generate and apply a source-level modification supported by the current context and execution evidence. The resulting candidate must still be executed, reflected upon, and governed.
NoOp	Planner action	Introduce no source-level modification in the current iteration because the available evidence does not justify an immediate patch. It does not by itself terminate the trajectory.
AcceptPatch	Governor decision	Terminate the trajectory and accept a sufficiently supported candidate that differs from the initial program.
Preserve	Governor decision	Terminate while retaining the initial program without accepting a new source-level modification.
Continue	Governor decision	Begin another repair iteration because the current state is not yet acceptable but further progress remains justified.
Reject	Governor decision	Terminate without accepting a generated patch because the current state is unsupported and further repair is not justified or no budget remains.
Plausible repair	Evaluation label	A generated patch that passes the original repair-time tests.
Confirmed repair	Evaluation label	A plausible patch that additionally satisfies the frozen offline confirmation protocol.
Stagnation	Trajectory property	A pattern in which repeated iterations fail to produce meaningful progress, new actionable evidence, or a distinct repair direction.

- *missing feedback*, where no usable execution outcome or diagnostic is returned;
- *truncated feedback*, where only part of the diagnostic or test output is available;
- *conflicting feedback*, where different observations support incompatible interpretations;
- *non-deterministic feedback*, where repeated executions under the same configuration produce materially different outcomes.

These conditions characterize the evidence available to the repair method rather than the underlying defect itself. A faulty program may be accompanied by reliable or unreliable feedback, and an otherwise correct program may be exposed to an unreliable failure signal.

Accordingly, PAUSE represents an execution observation using both a task-level outcome and a feedback-quality record. The complete schema is defined in Appendix A.3.

B.5 OPERATIONAL TERMINOLOGY

Table 10 summarizes the central terms used throughout the paper. The main distinction is between *iteration-level actions*, which are selected by the Planner, and *trajectory-level decisions*, which are produced by the Governor.

A NoOp action and a Preserve decision occur at different stages of the method. The Planner may select NoOp, after which the unchanged program is still executed and evaluated. The Governor selects Preserve only when the available evidence supports terminal retention of the initial program.

Likewise, Reject does not indicate that the initial program has been verified as correct. It indicates only that PAUSE does not accept a generated patch under the current evidence and repair budget.

B.6 BACKGROUND REFERENCES FOR REPLICATION

Readers seeking a broader introduction to APR may consult the taxonomy and survey by Huang et al. Huang et al. (2024). The survey of patch correctness assessment by Fei et al. provides additional background on plausibility, correctness assessment, and patch overfitting Fei et al. (2025). Conversation-based repair and ThinkRepair provide representative examples of iterative LLM-based

Table 11: Benchmark populations used in the experiments.

Dataset	Language	Instance unit	Raw	Eligible	Excluded	Primary evaluation role
QuixBugs	Python	Algorithmic program	40	40	0	Controlled evaluation on compact programs with executable reference implementations.
ConDefects	Python	Faulty–repaired pair	2,864	1,885	979	Larger-scale Python evaluation with naturally collected faulty–repaired pairs.
Defects4J	Java	Project fault	835	655	180	Cross-language evaluation on real-world projects and build environments.

repair Xia & Zhang (2024); Yin et al. (2024), while Reflexion provides background on trajectory-level reflection in language agents Shinn et al. (2023). RepairAgent provides a representative reference for agentic program repairBouzenia et al. (2025).

These references provide conceptual background only. The operational definitions, prompts, configurations, dataset versions, and evaluation procedures required to reproduce PAUSE are specified in the corresponding appendices of this paper.

Scope of statements. The definitions in this appendix describe the terminology and experimental interpretation used in the paper. Statements concerning why a particular design may improve repair control are design motivations rather than theoretical guarantees. The effectiveness of selective planning, typed execution feedback, trajectory reflection, and dual-condition governance is evaluated empirically in the main experiments and ablation studies.

C DATASETS, BASELINES, AND EVALUATION CONTROLS

C.1 BENCHMARK SELECTION

We evaluate PAUSE on QuixBugs, ConDefects, and Defects4J.

QuixBugs. QuixBugs contains compact algorithmic programs with faulty and corrected implementations in multiple programming languages Lin et al. (2017). We use all 40 programs from its official Python version. The compact programs and executable reference implementations make QuixBugs suitable for examining repair behavior under controlled algorithmic tasks and for constructing supplementary differential tests.

ConDefects. ConDefects provides faulty–repaired program pairs designed as a complementary benchmark for LLM-based fault localization and program repair Wu et al. (2024b). We use its Python subset to evaluate whether the repair procedure generalizes beyond the small programs in QuixBugs. The repository snapshot contains 2,864 candidate pairs, of which 1,885 remain eligible after reproducibility screening.

Defects4J. Defects4J contains reproducible faults from real-world Java projects together with project-specific build and test infrastructure Just et al. (2014). We use Defects4J v2.0 to evaluate repair under larger codebases, project dependencies, and more complex test environments. Of the 835 faults in the frozen release, 655 satisfy our build and execution requirements.

Table 11 summarizes the benchmark populations. The raw count denotes the population available in the frozen repository snapshot, whereas the eligible count denotes the instances included in the reported repair-effectiveness denominators.

C.2 Eligibility Screening and Preprocessing

Eligibility screening is performed before repair and is independent of the output of evaluated method. An instance is retained only when the benchmark artifacts required for execution and evaluation can be reproduced in the frozen environment.

The common screening procedure checks whether:

1. the buggy and reference artifacts are available and can be parsed, loaded, or checked out;
2. the required dependencies and test harness can be initialized without an unrecoverable infrastructure failure;
3. at least one supplied test exposes the faulty behavior of the buggy version;
4. the corresponding repaired or fixed version satisfies the required benchmark checks; and
5. the instance can be executed within the predefined resource limits without relying on an unavailable external service.

For QuixBugs, all 40 official Python programs satisfy these conditions. The preprocessing procedure normalizes program and test locations but does not alter source semantics or add benchmark-specific repair hints.

For ConDefects, each faulty–repaired pair is executed independently. Pairs are excluded when the files cannot be parsed, the supplied tests cannot be executed, the buggy behavior cannot be reproduced, the paired repair does not satisfy the required checks, or an unavailable dependency prevents controlled execution. This screening reduces the raw population from 2,864 to 1,885 eligible pairs.

For Defects4J, each fault is checked out through the benchmark interface and evaluated in its project-specific environment. A fault is excluded when checkout fails, the project cannot be compiled, the triggering failure cannot be reproduced, the relevant tests cannot complete under the resource limits, or the fixed revision cannot be validated in the frozen environment. This procedure retains 655 of the 835 faults in Defects4J.

C.3 Plausibility and Frozen Confirmation

We distinguish repair-time plausibility from offline confirmation. A generated patch is labeled *plausible* when it passes the original tests available during repair. A plausible patch is labeled *confirmed* only when it additionally passes the frozen dataset-specific confirmation protocol.

For QuixBugs, confirmation combines the official tests with differential checking against the reference implementation on a fixed collection of random and boundary inputs. The supplementary inputs and random seeds are generated before the evaluated runs and remain identical across methods.

For ConDefects, confirmation uses the paired repaired program as the reference for fixed supplementary inputs in addition to the original test evidence. Inputs for which the reference program does not produce a valid outcome are excluded before evaluation rather than filtered according to a candidate patch.

For Defects4J, plausibility is determined by the original project tests used during repair. Confirmation additionally requires the candidate to pass the frozen project-level regression and supplementary validation checks prepared before the repair runs.

The same confirmation artifacts are applied to every method. They are not included in prompts, execution observations, retrieved experience, or Governor decisions.

C.4 External Baselines

The baseline set covers direct LLM repair, conversational repair, agentic repair, repair-specific language models, and established non-LLM APR paradigms. Table 12 summarizes their benchmark applicability.

Table 12: Applicability of PAUSE and external baselines. N/A indicates that the released method does not support the benchmark language or execution setting; it does not denote a repair failure.

Method	Category	QuixBugs	ConDefects	Defects4J	Implementation note
Direct Repair	Direct LLM	Yes	Yes	Yes	Uses the same backbone and repair-time evidence without the role-specialized control workflow.
ChatRepair	Conversational LLM	Yes	Yes	Yes	Uses iterative execution feedback under the common attempt budget.
RepairAgent	Agentic repair	Yes	Yes	Yes	Only the execution interface is adapted for the Python benchmarks.
RepairLLaMA	Fine-tuned repair model	N/A	N/A	Yes	Evaluated under its supported Java repair setting.
TBar	Template-based APR	N/A	N/A	Yes	Uses its Java-specific repair templates and localization workflow.
SimFix	Similarity-based APR	N/A	N/A	Yes	Uses its supported Java project and test infrastructure.
CURE	Neural APR	N/A	N/A	Yes	Evaluated through its Java-oriented repair pipeline.
PAUSE	Selective LLM repair	Yes	Yes	Yes	Uses the same selective planning and governance procedure across languages, with dataset-specific execution adapters.

Direct Repair. Direct Repair prompts the same backbone model with the program, specification evidence, and currently available failure information, but does not use role decomposition, experience retrieval, typed trajectory reflection, or explicit governance. It provides a controlled comparison for measuring the contribution of the complete PAUSE workflow.

ChatRepair. ChatRepair represents conversational LLM-based repair, in which test or compiler feedback from a candidate is supplied in a subsequent repair interaction Xia & Zhang (2024). It supports iterative refinement but does not include the explicit PATCH/NOOP planning decision or the dual-condition Governor used by PAUSE.

RepairAgent. RepairAgent is an autonomous LLM-based program repair agent that interacts with source code and execution toolsBouzenia et al. (2025). For QuixBugs and ConDefects, we adapt only its execution interface to support the Python test environments. Its high-level repair workflow and decision logic are otherwise retained. The Java-compatible configuration is used for Defects4J.

RepairLLaMA. RepairLLaMA uses repair-oriented representations and fine-tuned adapters for patch generationSilva et al. (2025). It is evaluated on Defects4J under its supported Java setting. It is not applied to the Python benchmarks because doing so would require a new cross-language training or adaptation procedure.

TBar, SimFix, and CURE. TBar is a template-based repair methodLiu et al. (2019), SimFix constructs patches using existing patches and similar codeJiang et al. (2018), and CURE uses code-aware neural machine translation for repair Jiang et al. (2021). These systems are evaluated on Defects4J, for which their Java-oriented implementations and repair assumptions are applicable. They are not reimplemented for the Python benchmarks.

Table 13: Controlled variants used in the selective-repair and governance analysis.

Variant	Changed component	Operational behavior
w/o NoOp	Removes the Planner's abstention action.	Whenever repair continues, the Planner must propose a source-level modification, even when the available evidence does not clearly justify one.
w/o Safe Rejection	Removes the evidence-based REJECT branch.	An unresolved trajectory cannot terminate through safe rejection while repair budget remains.
Fixed Budget	Removes adaptive early termination for unresolved trajectories.	Every unresolved instance is processed for all five permitted attempts, regardless of detected stagnation.
First Plausible	Replaces governed acceptance with test-suite plausibility.	The first generated patch that passes the original repair-time tests is accepted without applying the complete governance decision.

C.5 BASELINE ALIGNMENT

For LLM-based comparisons, Direct Repair, ChatRepair, RepairAgent, and PAUSE use the same backbone model wherever the corresponding method permits backbone substitution. They receive the same program, specification evidence, and initial execution feedback. They also use the same decoding configuration, maximum generation length, model-call accounting, execution limits, and maximum repair-attempt budget.

The common attempt budget is an upper bound of five repair iterations per instance. A method may terminate earlier according to its own stopping rule. The baselines are not given access to the frozen confirmation evidence or offline correctness labels.

Method-specific mechanisms are retained rather than forcibly rewritten into the PAUSE architecture. In particular, ChatRepair preserves its conversational feedback loop, RepairAgent preserves its agentic workflow, and the Java APR baselines preserve their original patch-generation paradigms. Dataset adapters only normalize source paths, test invocation, execution status, and result logging.

For methods that do not support a benchmark language or execution environment, the corresponding entry is reported as N/A. Such entries are excluded from cross-method statistical comparisons for that benchmark.

C.6 GOVERNANCE CONTROLS

In addition to external baselines, we evaluate four controlled variants of PAUSE to isolate the effect of selective planning and governance. All variants use the same backbone, inputs, decoding configuration, execution interface, attempt budget, and confirmation protocol as the complete method.

These variants are not treated as independent methods with separately tuned configurations. They are controlled interventions on the complete PAUSE workflow and are used only to identify the contribution of the removed decision mechanism.

D PROMPT TEMPLATES AND IMPLEMENTATION DETAILS

D.1 PROMPT ORGANIZATION

Each agent prompt contains four parts: a fixed role instruction, instance-specific inputs, role-specific decision rules, and a structured output schema. The role instruction and output schema remain fixed across datasets, while program text, execution commands, test information, trajectory history, and retrieved experience are inserted at run time.

The prompts request concise, evidence-linked rationales rather than unrestricted reasoning traces. Each agent is instructed to use only the information included in its input and to avoid inferring unavailable execution results or offline correctness labels.

Table 14 summarizes the principal variables inserted into the prompts.

Table 14: Run-time variables used to assemble the agent prompts.

Variable	Source	Content
PROGRAM	Current repair state	The source program supplied to the current agent. For the Executor and later agents, this may be the candidate produced by the Planner.
SPECIFICATION	Dataset adapter	The available task description, function signature, visible tests, input constraints, and expected behavior permitted during repair.
EXECUTION_EVIDENCE	Executor or initial benchmark run	The typed execution outcome, feedback-quality attributes, and available diagnostic payload.
TRAJECTORY_HISTORY	Shared trajectory store	Structured records of previous actions, candidates, observations, reflections, and governance decisions.
CONTEXT_PROFILE	Profiler	The decision-focused summary of program intent, constraints, suspicious regions, risk-sensitive structures, and evidence gaps.
EXPERIENCE_CONSTRAINTS	Experience Repository	Retrieved control constraints relevant to the current context and trajectory. Source-level patch text is not included.
EXECUTION_COMMANDS	Dataset adapter	The fixed compilation, execution, and test commands for the current benchmark instance.
EXECUTION_LIMITS	Experiment configuration	The compilation, execution, test, and resource limits applied by the Executor.
REMAINING_BUDGET	Repair controller	The number of repair iterations remaining after the current iteration.

ROLE
You are the Profiler agent in PAUSE, a selective program repair framework. Your responsibility is to construct a decision-focused profile of the current program and its execution context. You must not generate or recommend a patch.
INPUTS
1. Current program: <PROGRAM>
2. Available specification evidence: <SPECIFICATION>
3. Latest typed execution observation: <EXECUTION_EVIDENCE>
4. Previous repair trajectory: <TRAJECTORY_HISTORY>
OBJECTIVE
Identify the program intent, constraints, suspicious behavior, and risk-sensitive code structures that are relevant to the current repair decision.
INSTRUCTIONS
1. Summarize the intended behavior using only the supplied specification and observable program context.
2. Identify constraints that a valid program should preserve.
3. Identify code regions related to the observed failure.
4. Highlight risk-sensitive structures, including boundary conditions, loop updates, termination conditions, state mutations, exception paths, and resource-management operations when applicable.
5. Distinguish observed evidence from uncertainty or missing information.
6. Do not propose source-code edits.
7. Do not infer information that is absent from the inputs.
8. Return only the required structured output.

Figure 3: Prompt template used by the Profiler agent.

D.2 Agent Prompt Templates

Profiler. The Profiler converts the current program, specification, execution evidence, and trajectory history into a decision-focused context profile. Its prompt explicitly prohibits patch generation and asks the agent to distinguish observed evidence from unresolved uncertainty. The complete template is shown in Figure 3.

Planner. The Planner receives the current program, context profile, latest execution evidence, trajectory history, and retrieved experience constraints. It must select exactly one of Patch and NoOp. A patch is requested only when the evidence supports a concrete source-level modification; otherwise, the Planner records a no-op decision. The Planner is not permitted to accept or reject its own candidate. Its complete prompt is shown in Figure 4.

Executor. The Executor interacts with the dataset-specific compilation, execution, and test interfaces. Its prompt specifies the permitted commands, resource limits, repetition policy, outcome vocabulary, and feedback-quality schema. The Executor reports what was observed but does not

ROLE
You are the Planner agent in PAUSE. Your responsibility is to decide whether the available evidence justifies a source-level modification. You must choose exactly one action: Patch or NoOp.
INPUTS
1. Current program: <PROGRAM>
2. Context profile: <CONTEXT_PROFILE>
3. Latest typed execution observation: <EXECUTION_EVIDENCE>
4. Previous repair trajectory: <TRAJECTORY_HISTORY>
5. Retrieved experience constraints: <EXPERIENCE_CONSTRAINTS>
6. Required patch representation: <PATCH_FORMAT>
OBJECTIVE
Select the action best supported by the current evidence. If a patch is justified, propose a localized modification. Otherwise, select NoOp.
INSTRUCTIONS
1. Choose Patch only when the available evidence supports a concrete source-level modification.
2. Choose NoOp when the evidence is missing, conflicting, insufficient, or does not establish that source modification is currently warranted.
3. Treat retrieved experience as decision constraints, not as patch templates.
4. For Patch, make the smallest modification that addresses the identified behavior while preserving unrelated functionality.
5. Do not modify tests, specifications, expected outputs, or execution infrastructure.
6. Do not decide whether the candidate should be accepted, preserved, continued, or rejected. That decision belongs to the Governor.
7. Do not claim semantic correctness.
8. Return only the required structured output.

Figure 4: Prompt template used by the Planner agent.

ROLE
You are the Executor agent in PAUSE. Your responsibility is to execute The current candidate under the supplied configuration and convert the raw result into a typed observation. You must not edit the program or make a terminal repair decision.
INPUTS
1. Candidate program: <CANDIDATE_PROGRAM>
2. Available specification and tests: <SPECIFICATION>
3. Build and execution commands: <EXECUTION_COMMANDS>
4. Resource and timeout limits: <EXECUTION_LIMITS>
5. Repetition policy: <REPETITION_POLICY>
OBJECTIVE
Obtain reproducible execution evidence and normalize it into the typed observation schema used by PAUSE.
INSTRUCTIONS
1. Use only the supplied build, execution, and test commands.
2. Preserve the raw compiler, runtime, and test outputs.
3. Classify the observable task-level outcome as exactly one of: Pass, AssertionFail, CompileError, RuntimeError, Timeout, NonTermination, or Instability.
4. Use NonTermination only when repeated or instrumented evidence supports persistent lack of execution progress. An isolated time-limit violation should be classified as Timeout.
5. Assess whether the returned feedback is available, complete, consistent, and stable.
6. Do not reconstruct missing diagnostics.
7. Do not interpret unavailable output as either success or failure.
8. Do not propose a patch.
9. Do not decide whether the candidate should be accepted.
10. Return only the required structured output.

Figure 5: Prompt template used by the Executor agent.

propose a new patch or issue a terminal governance decision. Figure 5 presents the complete template.

Reflector. The Reflector compares the latest candidate and observation with the complete trajectory. It identifies improvement, regression, repetition, stagnation, and changes in feedback quality. Its output includes a continuation judgment, but not the final AcceptPatch, Preserve, Continue, or Reject decision. The complete Reflector prompt is shown in Figure 6.

Governor. The Governor receives the current candidate, typed execution observation, trajectory reflection, action history, and remaining budget. It first assesses whether the current state is sufficiently supported and then combines this assessment with the Reflector's continuation judgment. The prompt encodes the four permitted decisions and their consistency constraints. It also prohibits the use of hidden tests, reference repairs, and offline confirmation labels. Figure 7 shows the complete template.

D.3 Structured Agent Outputs

All agents return structured records. Table 15 lists the required top-level fields. The complete field definitions and enumerated values appear in the prompt figures.

The structured schemas reduce ambiguity between agent responsibilities. For example, the Planner may return only Patch or NoOp, whereas only the Governor may return a terminal decision. Likewise, the Reflector reports whether continuation appears worthwhile but does not independently terminate the trajectory.

ROLE
You are the Reflector agent in PAUSE. Your responsibility is to assess the latest candidate and execution observation in the context of the complete repair trajectory. You evaluate progress and continuation value, but you do not make the final governance decision.
INPUTS
1. Previous program: <PREVIOUS_PROGRAM>
2. Current candidate: <CANDIDATE_PROGRAM>
3. Latest Planner action: <PLANNER_ACTION>
4. Latest typed execution observation: <EXECUTION_EVIDENCE>
5. Previous repair trajectory: <TRAJECTORY_HISTORY>
6. Retrieved experience constraints: <EXPERIENCE_CONSTRAINTS>
OBJECTIVE
Determine whether the current iteration shows progress, regression, stagnation, or unresolved uncertainty, and whether another repair attempt remains worthwhile.
INSTRUCTIONS
1. Compare the latest outcome with earlier candidates and observations.
2. Treat removal of a previous failure, reduction of failing tests, localization of an earlier uninformative failure, or improvement in feedback quality as possible progress.
3. Treat repeated equivalent edits, repeated failures without new evidence, oscillation between previous outcomes, or newly introduced failures as possible stagnation or regression.
4. Assess the reliability of the evidence used in the comparison.
5. Determine whether another attempt has a distinct and evidence-based direction.
6. Do not generate a patch.
7. Do not issue AcceptPatch, Preserve, Continue, or Reject. The final decision belongs to the Governor.
8. Return only the required structured output.

Figure 6: Prompt template used by the Reflector agent.

ROLE
You are the Governor agent in PAUSE. Your responsibility is to make the final repair-control decision by separately considering current-state acceptability and the value of another repair attempt.
INPUTS
1. Initial program: <INITIAL_PROGRAM>
2. Current candidate: <CANDIDATE_PROGRAM>
3. Latest Planner action: <PLANNER_ACTION>
4. Latest typed execution observation: <EXECUTION_EVIDENCE>
5. Trajectory-level reflection: <TRAJECTORY_REFLECTION>
6. Previous repair trajectory: <TRAJECTORY_HISTORY>
7. Remaining iteration budget: <REMAINING_BUDGET>
OBJECTIVE
Return exactly one decision: AcceptPatch, Preserve, Continue, or Reject.
INSTRUCTIONS
1. First determine whether the current candidate is sufficiently supported by the available repair-time evidence.
2. Do not treat a passing observation as sufficient when the feedback is missing, incomplete, conflicting, or unstable.
3. Do not use any hidden test, reference repair, or offline confirmation result.
4. Select AcceptPatch only when the current candidate is sufficiently supported and differs from the initial program.
5. Select Preserve only when the current state is sufficiently supported and the returned program is the initial program.
6. Select Continue only when the current state is not sufficiently supported, the Reflector reports that another attempt remains worthwhile, and the remaining budget is greater than zero.
7. Select Reject when the current state is unsupported and continuation is not justified or no iteration budget remains.
8. Do not generate or revise a patch.
9. Return only the required structured output.

Figure 7: Prompt template used by the Governor agent.

D.4 Experience Repository Interface

The Experience Repository provides two operations: Retrieve and Distill. Its interface is shown in Figure 8.

For retrieval, the repository constructs a query signature from the Profiler output, latest execution evidence, and trajectory pattern. It returns a bounded set of context-compatible decision constraints, together with their provenance and confidence. Retrieved entries do not contain source-level patch text.

For distillation, an eligible completed trajectory is reduced to its context pattern, execution pattern, trajectory pattern, and decision-relevant consequence. Instance-specific source code, generated patches, and benchmark identifiers are removed before storage. Under the frozen evaluation protocol, experience derived from one evaluated instance is not made available to subsequent evaluated instances.

D.5 Module Interfaces and Implementation Mapping

Table 16 maps the conceptual components in the paper to their implementation responsibilities. Dataset-specific logic is isolated in the execution adapter; the five agent interfaces and the Experience Repository remain unchanged across datasets.

Table 15: Required top-level fields in the five agent outputs.

Agent	Required fields	Downstream use
Profiler	intent_summary, relevant_constraints, failure_summary, suspicious_regions, risk_sensitive_structures, evidence_gaps	Provides decision-relevant context to the Planner and Experience Repository.
Planner	action, evidence_used, target_location, patch_description, patch, expected_effect, rationale	Determines whether a candidate modification is generated and supplies the candidate to the Executor.
Executor	outcome_type, feedback_quality, tests, diagnostics, execution, observation_summary	Provides normalized execution evidence to the Reflector and Governor.
Reflector	progress, progress_evidence, regression_signals, stagnation_signals, feedback_assessment, continuation_judgment, next_iteration_focus	Provides the trajectory-level assessment and continuation judgment to the Governor and, when applicable, the next iteration.
Governor	current_state_acceptable, acceptability_evidence, continuation_supported, decision, selected_artifact, rationale	Determines terminal acceptance, preservation, continuation, or rejection.

ROLE
The Experience Repository is a structured non-agent component. It stores and retrieves reusable decision constraints. It does not generate source-code patches and does not make terminal decisions.
RETRIEVAL INPUTS
1. Context profile: <CONTEXT_PROFILE>
2. Current typed execution observation: <EXECUTION_EVIDENCE>
3. Current trajectory signature: <TRAJECTORY_SIGNATURE>
RETRIEVAL PROCEDURE
1. Construct a query from the task context, failure type, feedback quality, trajectory progress, and stagnation signals.
2. Retrieve the top-k most relevant experience entries.
3. Exclude entries whose context or execution conditions are incompatible with the current instance.
4. Return decision constraints rather than source-code fragments.
5. Preserve the provenance and confidence of each retrieved entry.

Figure 8: Retrieval and distillation interfaces of the Experience Repository. The repository is a structured component rather than a sixth agent.

D.6 Prompt Assembly and Output Validation

At each invocation, the implementation combines the fixed role instruction with only the run-time variables required by that agent. For example, the Planner receives the context profile and retrieved experience constraints, whereas the Governor receives the typed observation, trajectory reflection, and remaining budget. Offline confirmation evidence is not inserted into any agent prompt.

Agent responses are parsed against the corresponding structured schema. Required fields, enumerated values, and cross-field consistency are checked before the response enters the shared trajectory. Examples of cross-field checks include the following:

- a Planner output with action NoOp must not contain a source modification;
- an Executor observation must contain exactly one task-level outcome type;
- a Reflector continuation judgment must use one of the two permitted values;
- a Governor decision of AcceptPatch must select the candidate artifact;
- a Governor decision of Preserve must select the initial program;
- Continue is invalid when no iteration budget remains.

A response that cannot be parsed or fails schema validation is recorded as an invalid agent output. It is not interpreted as positive evidence and cannot independently support terminal acceptance. The raw response, parsed record, prompt version, model identifier, and iteration identifier are retained in the experiment log.

Table 16: Mapping from the conceptual architecture to the implementation modules.

Component	Implementation responsibility	Primary input and output	Paper correspondence
Profiler module	Assembles the Profiler prompt, validates the profile schema, and stores the context profile.	Program, specification, execution evidence, and trajectory → context profile.	Context profiling in Appendix A.2 and Figure 3.
Planner module	Assembles the Planner prompt, validates the selected action, and applies a generated edit when the action is Patch.	Profile, evidence, trajectory, and experience constraints → action and optional patch.	Selective planning in Appendix A.5 and Figure 4.
Execution adapter and Executor module	Invokes dataset-specific build and test commands, captures raw output, and validates the typed observation.	Candidate and execution configuration → typed execution observation.	Typed feedback in Appendix A.3 and Figure 5.
Reflector module	Assembles the trajectory context and validates the progress and continuation fields.	Candidate, observation, and trajectory → trajectory assessment.	Trajectory reflection in Appendix A.6 and Figure 6.
Governor module	Applies the governance prompt, validates decision consistency, and updates the terminal or continuing repair state.	Candidate, observation, reflection, and remaining budget → governance decision.	Dual-condition governance in Appendix A.6 and Figure 7.
Experience Repository	Indexes distilled control experience, retrieves compatible constraints, and records provenance.	Context and trajectory signature ↔ decision constraints.	Experience retrieval and distillation in Appendix A.7 and Figure 8.
Trajectory store	Maintains structured records across iterations and supplies the relevant history to each agent.	Agent outputs and candidate states → ordered repair trajectory.	State definition in Appendix A.1.

D.7 Prompt Versioning and Reproducibility

The five agent templates and the Experience Repository interface are versioned with the experiment configuration. All methods using the same PAUSE configuration receive identical prompt templates; only instance-specific program, specification, execution, and trajectory content changes across runs.

The released artifact includes the machine-readable prompt templates, structured schemas, dataset adapters, and prompt-to-module mapping. Prompt versions are recorded with each run so that results can be traced to the exact role instruction and output format used during the experiment.

E Experimental Protocol and Hyperparameters

E.1 Experimental Protocol

For PAUSE, a repair trajectory contains at most $T = 5$ iterations. Each iteration may invoke the Profiler, Planner, Executor, Reflector, and Governor agents. The Experience Repository is queried before planning but does not constitute an additional model call. A trajectory may terminate before reaching the maximum budget through $\mathrm{AcceptPatch}$, $\mathrm{Preserve}$, or Reject.

The maximum iteration count is an upper bound rather than a fixed number of mandatory attempts. When the Governor selects $\mathrm{Continue}$, the resulting candidate and typed execution observation are passed to the next iteration. When no budget remains, an unsupported state is rejected rather than accepted solely because the trajectory has reached its limit.

Table 17: Final configuration of PAUSE used in the main experiments.

Setting	Final value	Application
Backbone	GPT-5.5	Shared by the Profiler, Planner, Executor, Reflector, and Governor.
Maximum repair iterations	$T = 5$	Upper bound on the number of feedback-guided repair iterations per instance.
Temperature	0.2	Applied to all generative agent calls.
Top-p	0.95	Applied to all generative agent calls.
Profiler output limit	1,024 tokens	Context profile and evidence-gap summary.
Planner output limit	2,048 tokens	Action decision, rationale, and optional source modification.
Executor output limit	1,024 tokens	Typed execution observation and normalized diagnostic payload.
Reflector output limit	1,024 tokens	Trajectory progress, stagnation signals, and continuation judgment.
Governor output limit	512 tokens	Acceptability assessment and terminal or continuation decision.
Retrieved experience entries	At most $k = 3$	Decision constraints returned by the Experience Repository for one iteration.
Repeated executions	Up to 3	Used when the initial observation requires a stability check.
Invalid structured output retry	1 retry	A single format-correction request is issued when a response cannot be parsed against the required schema.

All methods are evaluated on the same frozen eligible populations and use the same dataset-specific execution and confirmation artifacts. For controlled LLM-based comparisons, the backbone, repair-time inputs, decoding configuration, and attempt budget are held constant whenever the baseline architecture permits these settings to be aligned. Method-specific repair logic is otherwise preserved.

E.2 FINAL MODEL CONFIGURATION

The default language-model backbone is GPT-5.5. The exact API model identifier and prompt version are recorded in the run manifest. All five agents use the same backbone but receive different role-specific prompts and structured output schemas. Table 17 reports the final configuration used in the main experiments.

Frequency and presence penalties are set to zero. Agent outputs are requested in the structured formats defined in Appendix D.3. The implementation does not use sampling-based candidate voting or generate multiple patches from the same Planner invocation. Each iteration therefore produces at most one candidate modification.

The Experience Repository is non-generative. Its retrieval operation uses the structured context and trajectory signature to return at most three decision constraints. Retrieved entries are inserted into the Planner input in ranked order and are subject to the same context budget across all evaluated instances.

E.3 HYPERPARAMETER DEVELOPMENT

Hyperparameters are selected on a fixed development pool that is disjoint from the instances used for the reported benchmark results. The development pool includes faulty programs with reproducible execution evidence and control instances representing correct programs or unreliable feedback. No reported test instance or frozen confirmation outcome is used to select the final configuration.

We vary only parameters that directly affect repair search, generation stability, experience retrieval, or evidence checking. Prompt wording, role definitions, typed observation categories, and Governor decision labels are treated as method specifications rather than tuned hyperparameters. Table 18 lists the values evaluated during development.

Table 18: Hyperparameter values evaluated during development.

Hyperparameter	Values evaluated	Selected	Selection consideration
Maximum iterations T	$\{1, 3, 5, 7\}$	5	Confirmed repair effectiveness and additional model calls after trajectory progress had saturated.
Temperature	$\{0.0, 0.2, 0.4, 0.6\}$	0.2	Balance between stable structured outputs and sufficient variation in candidate generation.
Top-p	$\{0.80, 0.90, 0.95, 1.00\}$	0.95	Repair effectiveness and structured-output stability under the selected temperature.
Experience retrieval k	$\{1, 3, 5\}$	3	Usefulness of retrieved constraints without introducing excessive or weakly related context.
Stability repetitions	$\{1, 3, 5\}$	3	Ability to identify unstable observations relative to additional execution cost.

The final setting is selected using a lexicographic criterion. We first prefer configurations with higher confirmed repair rate on the faulty development instances. Among configurations with comparable confirmed repair performance, we prefer the configuration with lower over-repair and unsupported-acceptance rates on the control instances. Remaining ties are resolved using the lower average number of model calls.

This selection rule reflects the objective of selective repair: a configuration is not preferred solely because it generates more plausible patches if the same setting also increases unsupported modification or unnecessary iterations.

No benchmark-specific prompt or decoding parameter is tuned after the global configuration is selected. Dataset-specific differences are limited to source-language handling, test invocation, build commands, and execution limits.

E.4 Randomness and Run Accounting

A master seed of 42 is used for deterministic preprocessing, development-set construction, control-instance generation, supplementary input generation, and bootstrap resampling. Component-specific seeds are derived from the master seed and the dataset, instance identifier, and experimental configuration.

The implementation initializes the random-number generators used by Python, NumPy, and any locally executed model component. When the model API supports a request-level seed, the derived instance seed is included in the request. When exact deterministic replay is not supported by the provider, the implementation records the complete prompt, model identifier, decoding parameters, request identifier, raw response, and parsed output.

Each main-table entry is computed from one complete run of the corresponding method and configuration over every eligible instance. Thus, the experimental unit for repair effectiveness is the benchmark instance rather than the mean of several partial benchmark runs. Governance variants are likewise evaluated once over the complete frozen control population. The number of runs used for each experiment is summarized in Table 19.

Variation and uncertainty in the reported results are estimated across paired benchmark instances rather than by treating repeated generations of the same instance as independent samples. Confidence intervals and statistical tests are specified in the metric and statistical-analysis appendix.

E.5 Execution Environments

Python and Java candidates are executed in isolated working directories. Each candidate receives a fresh process environment, fixed input files, and the dataset-specific test command. Source files, test outputs, process exit status, elapsed time, and timeout events are retained in the run log.

Table 19: Run accounting for the reported experiments.

Experiment	Runs per setting	Evaluation unit
Main benchmark	1	Eligible instance
External baseline	1	Eligible instance
Governance control	1	Control instance
Ablation	1	Eligible or control instance
Hyperparameter development	1	Development instance

Python execution. QuixBugs and ConDefects are executed using Python 3.10 in isolated subprocesses. Each candidate is parsed before test execution. A single test invocation is limited to 10 seconds, and the complete instance-level test procedure is limited to 60 seconds. The process memory limit is 4 GB. Standard output and standard error are captured separately and normalized by the Executor.

Java execution. Defects4J candidates are evaluated in a Defects4J v2.0-compatible environment. The project-specific Java version is selected according to the benchmark metadata, using OpenJDK 8 or OpenJDK 11 as required. Compilation is limited to 600 seconds, triggering-test execution to 300 seconds, and the complete regression-test procedure to 1,800 seconds. The process memory limit is 8 GB.

A timeout is reported when an execution exceeds the configured wall-clock limit without sufficient evidence of the cause. $\mathrm{NonTermination}$ is used only when repeated execution or available runtime evidence indicates persistent lack of progress. Environment or harness failures are retained as unreliable evidence and are not silently converted into program-level failures.

The same execution limits are used across methods on a given dataset. A method cannot modify the supplied tests, confirmation artifacts, resource limits, or execution commands.

F Evaluation Metrics and Statistical Analysis

F.1 Evaluation Population and Outcome Assignment

Let $\mathcal{I}_d$ denote the eligible instance set of dataset d, and let

$$N_d = |\mathcal{I}_d| \tag{31}$$

be its size. In our experiments, the eligible populations contain 40 QuixBugs programs, 1,885 ConDefects pairs, and 655 Defects4J faults, giving a pooled population of

$$N_{\text{all}} = 40 + 1{,}885 + 655 = 2{,}580. \tag{32}$$

Every eligible instance contributes exactly once to the denominator of the corresponding repair-effectiveness metric. A model error, invalid structured response, compilation failure, execution timeout, exhausted repair budget, or terminal rejection does not remove an eligible instance from evaluation.

For PAUSE, only a terminal $\mathrm{AcceptPatch}$ decision returns a program as a repaired output. A $\mathrm{Preserve}$ decision returns the initial program, while Reject returns no accepted repair. For each baseline, the final artifact produced by its own stopping rule is evaluated in the same manner.

Let $P_i^\star$ denote the final returned artifact for instance i. We define the plausible-repair indicator as

$$p_i = \mathbb{I}\left[P_i^\star \text{ is a modified program that passes the original tests}\right], \tag{33}$$

and the confirmed-repair indicator as

$$c_i = \mathbb{I}\left[p_i = 1 \text{ and } P_i^\star \text{ passes the frozen confirmation protocol}\right]. \tag{34}$$

Here, $\mathbb{I}[\cdot]$ is the indicator function. By construction,

$$c_i \le p_i. \tag{35}$$

A candidate generated during an intermediate iteration is not counted as a repair unless it becomes the method's final accepted output. This rule prevents methods that generate many discarded candidates from receiving credit for patches that they do not ultimately return.

F.2 Repair-Effectiveness Metrics

For dataset d, the numbers of plausible and confirmed repairs are

$$\mathrm{PR}_d = \sum_{i \in \mathcal{I}_d} p_i, \qquad \mathrm{CR}_d = \sum_{i \in \mathcal{I}_d} c_i. \tag{36}$$

The plausible repair rate is

$$\mathrm{PRR}_d = \frac{\mathrm{PR}_d}{N_d}, \tag{37}$$

and the confirmed repair rate is

$$\mathrm{CRR}_d = \frac{\mathrm{CR}_d}{N_d}. \tag{38}$$

PRR measures how often a method returns a modified program that satisfies the original repair-time tests. CRR applies the stronger frozen confirmation protocol and is the primary repair-effectiveness metric. Because both metrics use the complete eligible population as their denominator, failures to produce an accepted patch are retained as unsuccessful outcomes.

Confirmation precision is defined as

$$\mathrm{CP}_d = \frac{\mathrm{CR}_d}{\mathrm{PR}_d}, \tag{39}$$

when $\mathrm{PR}_d > 0$. If a method produces no plausible repair, CP is reported as N/A rather than zero. CP describes the proportion of returned plausible repairs that also satisfy the confirmation protocol. It is therefore interpreted together with PRR and CRR rather than as a standalone measure of repair coverage.

To measure early repair effectiveness, let $c_i^{(1)} \in \{0, 1\}$ indicate whether instance i obtains a confirmed repair in the first repair iteration. We report

$$\mathrm{C@1}_d = \frac{1}{N_d} \sum_{i \in \mathcal{I}_d} c_i^{(1)}. \tag{40}$$

The first repair iteration includes all role invocations and execution steps required to produce and govern the first candidate. Thus, C@1 refers to the first repair iteration rather than the first individual model call.

For pooled reporting, counts are first summed across datasets:

$$\mathrm{PRR}_{\mathrm{all}} = \frac{\sum_d \mathrm{PR}_d}{\sum_d N_d}, \qquad \mathrm{CRR}_{\mathrm{all}} = \frac{\sum_d \mathrm{CR}_d}{\sum_d N_d}. \tag{41}$$

The pooled values are thus micro-averages over all 2,580 eligible instances. They are not arithmetic means of the three dataset-level percentages.

F.3 Selective-Behavior Metrics

Selective behavior is evaluated separately from repair effectiveness. Let $\mathcal{C}$ denote the set of 300 verified-correct control programs and let $\mathcal{U}$ denote the set of 300 faulty programs exposed to unreliable feedback.

Initial NoOp accuracy. For a correct control instance $i \in \mathcal{C}$, let $n_i = 1$ when the Planner selects NoOp in the initial iteration and $n_i = 0$ otherwise. Initial NoOp accuracy is

$$\mathrm{NoOpAcc} = \frac{1}{|\mathcal{C}|} \sum_{i \in \mathcal{C}} n_i. \tag{42}$$

This metric measures whether the method initially abstains from modification when the program is already correct. It does not require the trajectory to terminate immediately.

Over-repair rate. Let $m_i = 1$ when a method terminates by accepting a modified version of a verified-correct program, and $m_i = 0$ otherwise. The over-repair rate is

$$\mathrm{ORR} = \frac{1}{|\mathcal{C}|} \sum_{i \in \mathcal{C}} m_i. \tag{43}$$

ORR is a terminal metric. A temporary modification that is subsequently discarded is not counted as over-repair, whereas an accepted source modification to an initially correct program is counted regardless of whether it passes the visible tests.

Safe-rejection precision. For an unreliable-feedback instance $i \in \mathcal{U}$, let $r_i = 1$ when the method terminates with Reject. Let $s_i^{\mathrm{rej}} = 1$ when that rejection is labeled appropriate by the frozen complete evidence, and $s_i^{\mathrm{rej}} = 0$ otherwise. Safe-rejection precision is

$$\mathrm{SRP} = \frac{\sum_{i \in \mathcal{U}} r_i s_i^{\mathrm{rej}}}{\sum_{i \in \mathcal{U}} r_i}. \tag{44}$$

SRP is conditional on the method issuing a rejection. If a method never selects Reject, SRP is reported as N/A. A high SRP indicates that issued rejections are generally supported by the complete offline evidence; it does not by itself indicate how frequently the method rejects.

Unsupported-acceptance rate. Let $a_i = 1$ when a method terminates with an acceptance-type decision, namely $\mathrm{AcceptPatch}$ or $\mathrm{Preserve}$. Let $u_i^{\mathrm{acc}} = 1$ when the artifact selected by that decision is not supported by the frozen complete evidence. The unsupported-acceptance rate is

$$\mathrm{UAR} = \frac{\sum_{i \in \mathcal{U}} a_i u_i^{\mathrm{acc}}}{|\mathcal{U}|}. \tag{45}$$

For $\mathrm{AcceptPatch}$, the selected artifact is the generated candidate. For $\mathrm{Preserve}$, it is the initial program. UAR therefore captures both acceptance of an unsupported patch and unsupported preservation of a faulty input under unreliable feedback.

Premature stopping. A terminal decision may be safe with respect to the current candidate but still end a trajectory for which further repair would have been productive. To identify such cases, we replay a paired continuation from the exact terminal state using the remaining iteration budget and the frozen continuation protocol.

Let $e_i = 1$ when an instance terminates before exhausting its budget without a confirmed output, and let $f_i = 1$ when the paired continuation reaches a confirmed repair within the unused budget. Premature Stop is reported as

$$\mathrm{PrematureStop} = \frac{\sum_{i \in \mathcal{U}} e_i f_i}{|\mathcal{U}|}. \tag{46}$$

This metric distinguishes evidence-based early termination from termination that forgoes a repair obtainable from the same state and remaining budget. The paired continuation is used only for offline analysis and is not available to the evaluated method.

NoOp Acc. and ORR are computed only on verified-correct controls. SRP, UAR, and Premature Stop are computed only on unreliable-feedback controls. The control instances are not included in PRR, CRR, CP, or C@1.

F.4 Efficiency Metrics

Efficiency is measured over the same eligible instances used for repair effectiveness. For instance i, let L_i denote the number of language-model calls made before terminal decision. Calls from the Profiler, Planner, Executor, Reflector, and Governor are included, as are format-correction retries. Experience retrieval and local program execution are not counted as model calls.

Let K_i^{in} and K_i^{out} denote the total input and output tokens across all model calls. Total token consumption is

$$K_i = K_i^{\mathrm{in}} + K_i^{\mathrm{out}}. \tag{47}$$

Calls and tokens are reported as means per eligible instance:

$$\overline{L} = \frac{1}{N_d} \sum_{i \in \mathcal{I}_d} L_i, \qquad \overline{K} = \frac{1}{N_d} \sum_{i \in \mathcal{I}_d} K_i. \tag{48}$$

End-to-end repair time T_i^{wall} is measured from the first repair invocation to the terminal decision. It includes model latency, local compilation, test execution, output parsing, and repair orchestration, but excludes offline confirmation. Because wall-clock time is typically skewed by API and execution latency, it is summarized using the median and interquartile range rather than only the mean.

The monetary cost of instance i is computed from the recorded input and output tokens and the frozen model pricing:

$$C_i = K_i^{\text{in}} \lambda_{\text{in}} + K_i^{\text{out}} \lambda_{\text{out}}, \tag{49}$$

where λ_{in} and λ_{out} are the per-token input and output prices applicable to the reported experiment.

Average cost per eligible bug is

$$\$/\text{Bug} = \frac{\sum_{i \in \mathcal{I}_d} C_i}{N_d}, \tag{50}$$

and cost per confirmed repair is

$$\$/\text{CR} = \frac{\sum_{i \in \mathcal{I}_d} C_i}{\text{CR}_d}, \tag{51}$$

when $\text{CR}_d > 0$. Cost per confirmed repair is reported as N/A when no confirmed repair is obtained.

All attempted instances contribute to calls, tokens, time, and cost, including instances ending in model failure, rejection, or budget exhaustion. This avoids reporting efficiency only on successful repairs.

F.5 Confidence Intervals and Distributional Reporting

For a single binomial rate, including PRR, CRR, C@1, NoOp Acc., ORR, and UAR, we report a two-sided 95% Wilson confidence interval. These intervals describe uncertainty across the evaluated instances under the frozen experimental configuration.

For the paired difference between two methods, we use an instance-level paired bootstrap with 10,000 resamples. Each resampled instance retains the outcomes of both methods, preserving the paired experimental design. Dataset-level intervals resample within one benchmark. Pooled intervals use stratified resampling within QuixBugs, ConDefects, and Defects4J so that the benchmark composition remains fixed.

For example, the paired CRR difference between methods A and B is

$$\Delta_{\text{CRR}} = \text{CRR}_A - \text{CRR}_B. \tag{52}$$

Its bootstrap interval is reported in percentage points. The same procedure is used for differences in PRR, C@1, ORR, UAR, and other instance-level rates.

CP and SRP are conditional ratios whose denominators may differ across methods. They are therefore accompanied by their numerator and denominator counts and by bootstrap confidence intervals, rather than being treated as ordinary paired binary outcomes.

For calls, tokens, and cost, we report the mean together with a bootstrap 95% confidence interval. For wall-clock time, we report the median and interquartile range. Additional per-instance distributions are used in the supplementary result analysis where relevant.

Because each configuration is executed once over the complete frozen population, these intervals quantify variation across benchmark instances. They do not estimate variability across repeated stochastic runs of the same instance.

F.6 Statistical Tests and Effect Sizes

All hypothesis tests are two-sided with a family-wise significance level of $\alpha = 0.05$. Statistical comparisons are paired because the same eligible instances are evaluated by the compared methods.

Table 20: Repair outcomes of PAUSE across the eligible benchmark populations. PR and CR denote the numbers of plausible and confirmed repairs, respectively.

Dataset	Eligible	PR	CR	PRR (%)	CRR (%)	CP (%)	C@1 (%)	Calls
QuixBugs	40	35	31	87.5	77.5	88.6	70.0	6.70
ConDefects	1,885	943	788	50.0	41.8	83.6	34.9	7.90
Defects4J	655	286	244	43.7	37.3	85.3	31.1	9.20
Pooled	2,580	1,264	1,063	49.0	41.2	84.1	34.5	8.21

Binary repair outcomes. PRR, CRR, and C@1 are compared using McNemar's test. For two methods A and B, let n_{10} denote the number of instances on which A succeeds and B fails, and let n_{01} denote the reverse. McNemar's test evaluates whether the two discordant counts differ. When the number of discordant pairs is small, we use the exact two-sided form of the test. In addition to the adjusted p-value, we report the paired rate difference and matched-pairs odds ratio

$$\mathrm{OR}_{\mathrm{MP}} = \frac{n_{10}}{n_{01}}. \tag{53}$$

When either discordant cell is zero, a correction of 0.5 is added to both discordant counts for the reported odds ratio. The uncorrected counts remain visible.

The same paired binary procedure is used for control metrics defined on a common instance-level denominator, such as NoOp Acc., ORR, UAR, and Premature Stop. Conditional metrics such as CP and SRP are compared using paired bootstrap differences because their effective denominators depend on the method's outputs.

Continuous efficiency outcomes. Calls, tokens, wall-clock time, and monetary cost are compared using the Wilcoxon signed-rank test over paired per-instance values. Pairs with a zero difference do not contribute to the signed-rank statistic.

For each comparison, we additionally report the median paired difference and the matched-pairs rank-biserial correlation:

$$r_{\mathrm{rb}} = \frac{W^+ - W^-}{W^+ + W^-}, \tag{54}$$

where W^+ and W^- are the sums of positive and negative signed ranks. The sign is oriented so that a positive value favors PAUSE for repair-effectiveness metrics and lower resource use favors PAUSE for efficiency metrics.

Multiple comparisons. Holm correction is applied to the p-values within each research question and metric family. For example, comparisons of PAUSE against multiple baselines for CRR within the same benchmark form one family, while the corresponding efficiency comparisons form a separate family. This avoids combining unrelated hypotheses into a single correction family while still controlling multiplicity among directly comparable claims.

The primary significance tests are conducted separately for each benchmark. Pooled results over the 2,580 eligible instances are reported as aggregate summaries with stratified confidence intervals; they do not replace the dataset-level paired tests.

G Additional Experimental Results and Analysis

G.1 Benchmark-Level Repair Outcomes

Table 20 decomposes the repair results of PAUSE across the three benchmark populations. The pooled row is computed by aggregating instance-level counts rather than averaging the three dataset-level percentages.

Across the pooled eligible population, PAUSE returns 1,264 plausible repairs, of which 1,063 satisfy the frozen confirmation protocol. The resulting gap of 201 instances illustrates why repair-time

Table 21: Selective-governance results on the control populations. NoOp Acc. and ORR use the verified-correct controls; SRP, UAR, and Premature Stop use the unreliable-feedback controls.

Method	NoOp Acc. (%)	ORR (%)	SRP (%)	UAR (%)	Premature Stop (%)	Avg. attempts
PAUSE	96.3	1.7	95.0	2.0	1.3	1.91
w/o NoOp	N/A	18.7	88.4	7.7	2.7	2.21
w/o Safe Rejection	96.3	3.7	N/A	13.0	0.7	3.84
Fixed Budget	96.3	2.3	93.5	3.0	0.0	5.00
First Plausible	96.3	8.3	84.0	11.0	0.7	1.31

plausibility and offline confirmation are reported separately. A candidate passing the original tests is not automatically counted as confirmed.

Under the aligned comparison used in the main experiment, the strongest baseline obtains a pooled CRR of 38.5%, compared with 41.2% for PAUSE. The corresponding difference is 2.7 percentage points. The comparison is based on final returned artifacts rather than all intermediate candidates generated during a trajectory.

The pooled C@1 of 34.5% indicates that a substantial proportion of confirmed repairs are completed in the first iteration. The remaining confirmed repairs rely on additional execution evidence and subsequent repair decisions. C@1 is therefore interpreted together with final CRR: requiring more than one iteration is not necessarily undesirable when the initial candidate or its supporting evidence remains uncertain.

The confirmation precision of 84.1% indicates that most plausible outputs also satisfy the frozen confirmation protocol, although the remaining plausible-but-unconfirmed cases show that governed acceptance does not eliminate patch overfitting. The method is therefore evaluated as an empirical repair procedure rather than as a correctness-guaranteed system.

G.2 SELECTIVE GOVERNANCE RESULTS

Table 21 reports the complete selective-governance results for PAUSE and its four controlled variants. NoOp accuracy and over-repair rate are computed on the 300 verified-correct programs. Safe-rejection precision, unsupported acceptance, and premature stopping are computed on the 300 unreliable-feedback controls.

Effect of NoOp. The full method initially selects NoOp for 289 of the 300 verified-correct controls, giving a NoOp accuracy of 96.3%. Five correct programs ultimately receive an accepted modification, producing an ORR of 1.7%.

Removing NoOp forces a modification whenever repair continues. Under this intervention, ORR rises to 18.7%. This result does not imply that every forced modification is accepted, but it shows that removing iteration-level abstention substantially increases the number of correct programs that terminate with a modified artifact.

Effect of safe rejection. The full method correctly supports most of the rejections it issues, with an SRP of 95.0%. When the explicit Reject branch is removed, unresolved trajectories remain active until they reach another terminal outcome or consume the available budget. This variant has the largest average attempt count apart from Fixed Budget and produces an unsupported-acceptance rate of 13.0%.

SRP is reported as N/A for this variant because it cannot issue the decision on which the conditional metric is defined.

Effect of adaptive termination. Fixed Budget runs all unresolved trajectories for five attempts. It eliminates premature stopping by construction, but requires substantially more attempts than the full method. The additional attempts produce only a modest change in unsupported acceptance, suggesting that consuming the complete budget does not automatically resolve evidence uncertainty.

Table 22: Terminal decisions by repair iteration for 2,580 eligible instances. Iteration-5 rejections include unresolved trajectories that exhaust the repair budget.

Terminal decision	Repair iteration					Total
	1	2	3	4	5	
AcceptPatch	1,000	180	60	18	6	1,264
Preserve	45	6	2	1	0	54
Reject	500	400	220	100	42	1,262
All trajectories	1,545	586	282	119	48	2,580

Table 23: Final governance outcomes under unreliable execution feedback. UAR denotes unsupported-acceptance rate and SRP denotes safe-rejection precision.

Feedback condition	AcceptPatch	Preserve	Reject	Avg. attempts	UAR (%)	SRP (%)
Missing	4	1	70	1.42	1.3	97.1
Truncated	18	0	57	1.81	1.3	96.5
Conflicting	9	1	65	2.07	2.7	93.8
Non-deterministic	7	0	68	2.34	2.7	92.6
Overall	38	2	260	1.91	2.0	95.0

This comparison should not be interpreted as showing that early termination is always preferable. The relevant trade-off is whether fewer attempts preserve confirmed repair effectiveness while avoiding unsupported acceptance and missed repair opportunities.

Effect of governed acceptance. First Plausible has the lowest average attempt count because it accepts the first candidate passing the original tests. However, its ORR and UAR increase to 8.3% and 11.0%, respectively. The result reflects the difference between satisfying the repair-time tests and having sufficient support for terminal acceptance.

Overall, the complete method balances the five reported selective metrics rather than optimizing any one of them independently. In particular, a method can reduce premature stopping by attempting more repairs, or reduce attempts by accepting earlier, while worsening other forms of decision error.

G.3 Termination Distribution and Model Calls

Table 22 reports the iteration at which each main-benchmark trajectory terminates. An $\mathrm{AcceptPatch}$ decision corresponds to a plausible returned repair. A $\mathrm{Preserve}$ decision retains the original program, while Reject returns no accepted repair.

A total of 1,545 trajectories, or 59.9% of the eligible population, terminate in the first iteration. Only 48 trajectories reach the fifth iteration. The resulting average is approximately 1.66 completed repair iterations per instance.

The 1,264 $\mathrm{AcceptPatch}$ outcomes correspond to the plausible repairs reported in Table 20. Among them, 1,063 subsequently pass the frozen confirmation protocol. The 54 $\mathrm{Preserve}$ outcomes occur on benchmark inputs known offline to be faulty and are therefore unsuccessful repair outcomes, even though they are distinct from accepting a generated patch.

The early-termination distribution leads to an average of 8.21 model calls per eligible instance. The strongest aligned baseline uses 14.82 calls per instance under the same maximum attempt budget, corresponding to a reduction of approximately 44.6%.

This reduction is not attributed solely to first-iteration acceptance. Preservation and rejection also avoid additional calls when the current state is sufficiently supported or the trajectory lacks a distinct continuation direction. Because these decisions can themselves be incorrect, the call reduction is interpreted together with ORR, UAR, and Premature Stop rather than as an isolated measure of quality.

G.4 Results by Feedback Condition

Table 23 decomposes the final decisions on the unreliable-feedback controls. Each condition contains 75 faulty programs, so the counts can be compared directly.

The method accepts a generated patch in 38 of the 300 unreliable-feedback trajectories, preserves the initial program in two, and rejects the remaining 260. Six acceptance-type outcomes are labeled unsupported by the complete offline evidence, giving the overall UAR of 2.0%.

Missing feedback leads to the fewest accepted patches because it provides little direct support for a source modification. Truncated feedback retains partial diagnostic information and consequently supports more accepted repairs. These patterns are descriptive of the reported control construction and should not be generalized to every form of missing or truncated execution output.

Conflicting and non-deterministic observations require more attempts on average than missing or truncated feedback. In these cases, repeated execution can provide additional information about consistency and stability, but it may also leave the trajectory unresolved.

Of the 260 issued rejections, 247 are supported by the complete offline evidence, producing an overall SRP of 95.0%. The raw number of rejections is not interpreted as a success measure by itself. The remaining rejected cases include trajectories for which the offline continuation analysis identifies a productive direction or for which the current evidence was insufficient to justify terminal rejection.

G.5 Observed Failure Patterns

Inspection of unsuccessful trajectories reveals three recurring patterns that are directly relevant to the design of PAUSE.

Insufficient or unreliable evidence. Some failures are exposed by tests that do not sufficiently constrain the intended behavior, while others are accompanied by missing, truncated, conflicting, or unstable observations. In these cases, the available evidence may not support either a well-localized modification or a reliable terminal decision.

Ineffective or stagnating repair. Some trajectories repeatedly modify the same region, reproduce an equivalent failure, or alternate between previously observed outcomes. Other candidates remove one visible symptom while introducing a compilation, runtime, or behavioral regression. These cases may exhaust the iteration budget without reaching a supported candidate.

Plausible but unconfirmed acceptance. A candidate may pass the original repair-time tests and appear sufficiently supported under the available evidence, but fail the frozen confirmation protocol. Such cases contribute to PR but not CR and account for part of the observed difference between PRR and CRR. They also illustrate that typed execution feedback and governed acceptance reduce, but do not eliminate, the risk of test-suite overfitting.

These patterns are descriptive and may overlap within the same trajectory. They are not treated as mutually exclusive failure classes or as inputs to the five agents.

Overall, the supplementary results indicate that confirmed repair outcomes, selective-behavior controls, and computational efficiency should be interpreted jointly. The findings are limited to the evaluated benchmarks, control construction, model configuration, iteration budget, and frozen confirmation protocol, and do not constitute a formal correctness or safety guarantee.