ARE AGENTS LEAVING YOUR CODE STUPID? PREDICTIVE REFACTORING FOR POST-REPAIR HARDENING

Anonymous authors
Paper under double-blind review

ABSTRACT

Software engineering agents are evaluated mainly by issue-resolution accuracy, and their workflows are reactive: the agent fixes the reported bug and stops, even when repository history marks the surrounding code as defect-prone. In tangled, high-churn legacy code, a patch can then pass the benchmark while leaving brittle interfaces, recurring defect hotspots, and hidden temporal coupling in place. We propose **Memory-Augmented Predictive Refactoring** (MAPR), a two-phase workflow that treats a solved repair as the starting point for regression-aware hardening. MAPR builds *repository memory* from historical pull requests, summarizing bug density, author-linked defect patterns, temporal coupling, review coverage, and code churn. After Phase 1 resolves the issue, Phase 2 replays the Phase 1 patch in a fresh sandbox and hardens the repaired code under a test-driven RED–GREEN–VERIFY protocol that targets guard clauses, diagnostic error messages, and a controlled blast radius. We evaluate MAPR on Phase 1-solved SWE-bench Verified instances with four backbones, together with a 348-instance MiniMax-M2.5 run over the full benchmark. Full Phase 2 raises guard density by a factor of 4.5–7.0 on every backbone and keeps 96.4–100% of repairs on the three backbones with complete harness logs; on the 348-instance run, it raises guard clauses from 68 to 798 while keeping 96.5% of repairs. The RED–GREEN–VERIFY scaffold accounts for most of the gain in hardening quality: removing it lowers the share of error messages that report runtime state on every backbone, for example from 79.7% to 30.4% on MiniMax-M2.5. Separating the two stages also matters: a single-session agent with the same scaffold and step budget loses 6–14 repairs on these backbones, versus 0–2 for MAPR. Falling back to the Phase 1 patch whenever Phase 2 breaks a repair restores every repair and retains at least 97% of the added guards. The anonymized code, data, and analysis artifacts are available at `https://figshare.com/s/b1f96ed457df4694a5a4`.

1 INTRODUCTION

Industrial software evolves in aging repositories shaped by urgent fixes, ownership changes, and accumulated technical debt. A bug report often points into a maintenance hotspot: a high-churn file, a validation boundary with repeated failures, or a module that co-changes with other risky components (Fowler, 2018; Mens & Tourwé, 2004; Tornhill, 2015; 2018). Software engineering agents now resolve a substantial fraction of real GitHub issues, especially when equipped with richer interfaces, search, or stronger foundation models (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2024; Xia et al., 2025a; Antoniades et al., 2024; Xia et al., 2025b). Their objective, however, remains reactive: fix the reported bug and stop once the target tests pass. In maintenance practice, a fix is also an occasion to harden the surrounding code before the next failure (Fowler, 2018). We ask whether an agent that has already repaired an issue can use repository history to decide where behavior-preserving hardening is worthwhile, without losing the repair. The issue text alone is a weak guide for this decision: it rarely says which files repeatedly attract bug fixes, which modules co-change, or which local patterns have caused prior defects. We therefore expose repository history to the agent as *repository memory*, a compact file-level summary of bug density, temporal coupling, recurring defect patterns, review coverage, and churn.

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.

Running example. In an `astropy` issue, removing a required column from a `TimeSeries` object raised a self-contradictory exception stating that the object expected `time` but found `time`. A conventional repair agent fixes the message by reporting the full expected and actual column lists. The patch passes the issue's regression test, and a reactive workflow stops there. The repaired code, however, remains a fragile validation boundary. Repository memory shows that the timeseries module appears frequently in bug-fixing pull requests, co-changes with a companion sampled-series module, and has prior defects involving weak validation of column-list attributes. These signals do not indicate that the fix is wrong. They indicate that the repaired neighborhood has a history of related failures, which is the information an agent needs to decide whether a small hardening edit is worth attempting.

This paper introduces *Memory-Augmented Predictive Refactoring* (MAPR), a two-phase workflow for post-repair hardening. Phase 1 runs a standard issue-driven repair agent. Phase 2 starts in a fresh sandbox, reapplies the Phase 1 patch, conditions on the issue and the retrieved repository memory, and follows a RED–GREEN–VERIFY protocol: it writes temporary tests for missing defensive behavior, adds guard clauses and diagnostic errors until these tests pass, and verifies that the original repair still passes. In the running example, Phase 2 rejects a bare string where a list of required columns is expected, fails early on non-iterable values, and reports the offending element and its index when the list contains a non-string entry (Appendix A). Because any second stage can regress a solved instance, we also evaluate a fallback that returns the Phase 1 patch whenever Phase 2 breaks a passing repair. Figure 1 summarizes the workflow.

We evaluate MAPR on Phase 1-solved SWE-bench Verified instances with four backbones: MiniMax-M2.5 (n=64), Gemini-3.1-Flash-Lite (n=55), GPT-5-mini (n=54), and Qwen3.5-Plus (n=69), and we add a 348-instance MiniMax-M2.5 run over the full benchmark. Full Phase 2 raises guard density by a factor of 4.5–7.0 on every backbone and keeps 96.4–100% of repairs on the three backbones with complete harness logs. On MiniMax-M2.5, for example, it raises the guard clauses in the submitted patch from 7 to 137 without a regression, whereas the variant without the TDD scaffold adds only 39 guards and incurs two regressions. Our contributions are:

- We formulate post-repair predictive hardening as an agentic maintenance task with regression safety as an explicit constraint: after a bug is fixed, the agent uses repository history to target defensive edits while preserving the repair, and both outcomes are measured.
- We present an auditable two-stage protocol (Algorithm 1) in which a predictions file is the only interface between repair and hardening, together with a regression-safe output-selection rule. The protocol leaves the base agent loop and its tool interface unchanged.
- We report a controlled attribution study across four backbones. It shows that the RED–GREEN–VERIFY scaffold drives hardening quality, that repository memory has a small and inconsistent effect on safety, and that a single-session agent with the same scaffold and step budget regresses more repairs than the two-stage protocol on three of four backbones.

2 METHOD

2.1 TASK FORMULATION

MAPR targets the interval after a repair has been produced and before it is deployed. Given an issue x, a repository R, and a Phase 1 patch p_1 that passes the relevant benchmark tests, Phase 2 must output a merged patch p_2 that preserves the repair and adds behavior-preserving defensive edits. The merged patch is evaluated as a complete submission rather than as a delta from p_1, so the workflow remains compatible with standard SWE-bench evaluation.

The base agent loop is unchanged: it receives a rendered prompt, executes shell commands, edits files, and submits a patch. All new behavior enters through runner orchestration and prompt context. The experiments therefore measure whether repository memory and TDD-style verification improve post-repair behavior without introducing a new agent architecture.

Phase 2 is not a second attempt to solve the original issue. It receives the issue only as semantic context, and its editable target is the repaired region together with nearby historically risky code. We therefore restrict hardening to a bounded transformation family: fail-fast guards, safer access paths, boundary checks, and diagnostic messages that expose the violated condition and the runtime state. Broader rewrites, test edits, and public API changes are outside this family.

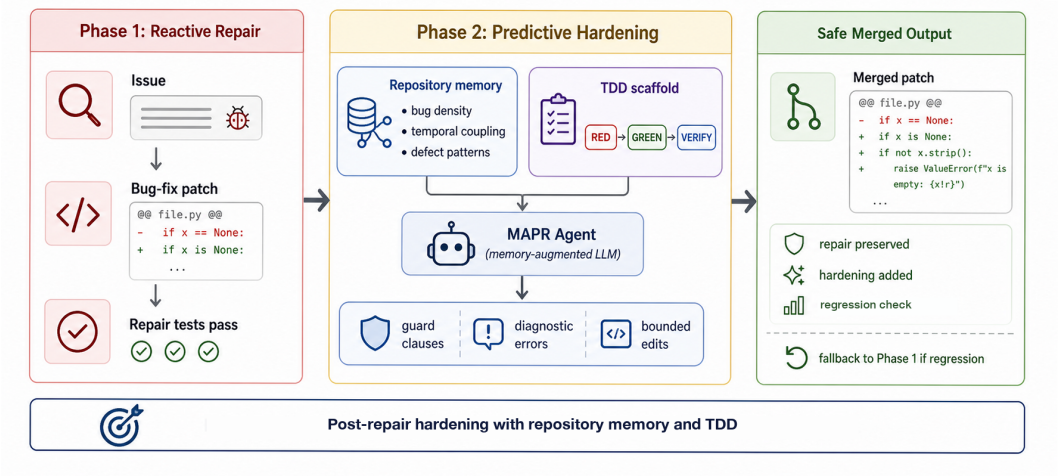


Figure 1: Overview of MAPR. Phase 1 repairs the issue and admits only solved instances to Phase 2, which replays the repair in a fresh sandbox, conditions the agent on repository memory and a RED–GREEN–VERIFY scaffold, and submits a merged patch.

Table 1: Repository-memory signals used to prioritize Phase 2 hardening. The prompt receives a compressed summary of these signals rather than the raw pull-request history.

Signal	Role in Phase 2
Bug density	Prioritizes files repeatedly involved in bug-fixing pull requests.
Temporal coupling	Highlights files that often change with the repaired file.
Error patterns	Summarizes recurring defect motifs near the patch neighborhood.
Review coverage	Downweights areas with stronger historical review evidence.
Churn	Flags volatile files where small hardening edits may be valuable.

2.2 Repository Memory

Repository memory is built from historical pull-request records. An advisor computes five file-level risk signals (Table 1), namely bug density, temporal coupling, author-linked error patterns, review coverage, and code churn, and compresses them into a bounded prompt summary of high-risk files, common bug patterns, and temporal-coupling alerts. The summary biases where Phase 2 looks without placing raw repository history in the prompt.

Retrieval is patch-scoped. Before Phase 2 starts, the runner extracts the files modified by the Phase 1 patch and queries memory for that neighborhood, which keeps the second stage focused on code connected to the repair. If memory is unavailable, the runner omits the memory block, and the prompt reduces to the no-memory variant. Retrieval is also time-scoped: the runner uses only pull requests merged before the target issue's pull request was opened. The runs in Section 4 predate this restriction; Appendix B.10 quantifies its effect on the memory summaries.

2.3 Two-Phase Execution

Phase 1 is a conventional issue-driven repair run: it receives the problem statement and produces a source patch. Only instances whose Phase 1 patch passes the benchmark evaluation are eligible for Phase 2, because post-repair hardening requires a repaired behavior to preserve.

Phase 2 always runs in a fresh sandbox. The runner loads the Phase 1 prediction for the instance, applies the patch to the clean repository, and injects into the agent template the original issue, the replayed Phase 1 patch, the benchmark's fail-to-pass and pass-to-pass test names, and, when enabled, the memory summary. Because the Phase 1 patch is applied without being committed, the final `git diff` contains both the repair and the hardening edits.

Algorithm 1 MAPR two-phase runner

Require: issue x, repository R, Phase 1 prediction file P_1
run Phase 1 repair agent on (x, R) to produce patch p_1
evaluate p_1 with the benchmark harness
if p_1 does not resolve x **then**
    skip Phase 2 for this instance
end if
create a fresh sandbox for R and apply p_1
extract modified files from p_1
retrieve patch-scoped repository memory m if enabled
inject $(x, p_1, m, \text{tests})$ into the Phase 2 template
run RED–GREEN–VERIFY hardening agent
return merged patch $p_2 \leftarrow$ `git diff`

In Algorithm 1, the predictions file is the only interface between the stages: it supplies the exact Phase 1 patch to replay. The workflow is auditable, and the Phase 1 gate can be computed outside the agent. Phase 2 never inherits a live container from Phase 1, so no hidden state is shared.

2.4 TDD-Driven Predictive Refactoring

The full Phase 2 prompt imposes a RED–GREEN–VERIFY scaffold. In RED, the agent writes temporary tests to a scratch directory that is excluded from the submitted patch. Guard-clause tests expose missing preconditions, diagnostic-error tests require messages that carry runtime context and state the expected condition, and a blast-radius report records how widely the patch spreads. The first two test families should fail before hardening. In GREEN, the agent adds narrowly scoped defensive edits, such as fail-fast checks, safer access paths, and context-rich exceptions, until these tests pass. In VERIFY, it reruns the temporary tests and the benchmark tests of the instance before submission. The blast-radius report keeps the edit scope visible without rejecting multi-file hardening.

The scaffold does not add human-curated tests to the benchmark. It turns the hardening goal into executable checks that the agent must satisfy, while the benchmark tests protect the repaired behavior. Because the temporary tests are excluded from the submission, the comparison remains source patch against source patch: any gain in guard counts or diagnostic ratios must appear in production code, not in auxiliary tests that downstream users never receive.

2.5 Regression-Safe Output Selection

For deployment, we also evaluate a conservative fallback: if raw Phase 2 regresses an instance that Phase 1 solved, the output for that instance is the original Phase 1 patch. This rule is a post-processing step over already generated artifacts and requires no additional model calls. We report both views: raw outputs for attribution and fallback outputs for deployment.

When to harden. Post-repair hardening is not always preferable to a minimal repair. Extra guards and error messages enlarge the diff that maintainers review, and they can change behavior on inputs that the tests do not cover. MAPR therefore treats Phase 2 as an optional stage bounded by three rules. First, Phase 2 starts only after the repair passes its tests, so the minimal Phase 1 patch is always available as the default output. Second, hardening targets patch neighborhoods that repository memory flags as historically risky. Our configuration flags a file when its bug density exceeds 0.4; when no touched file is flagged, the minimal patch can be shipped without running Phase 2. Third, the fallback returns the Phase 1 patch whenever the hardened patch fails a test that the repair passed. Settings that require minimal diffs, such as security backports or release branches, should skip Phase 2. In our experiments, Phase 2 runs on every repaired instance so that its effect can be measured.

3 EXPERIMENTAL DESIGN

3.1 EXPERIMENTAL SETUP

We run four backbones on the first 100 SWE-bench Verified instances; this restriction reflects our compute budget. Post-repair hardening is defined only for instances that Phase 1 resolves, so the Phase 1 gate is part of the task rather than a convenience filter. The gate contains 73 instances for MiniMax-M2.5, 55 for Gemini-3.1-Flash-Lite, 56 for GPT-5-mini, and 72 for Qwen3.5-Plus. To test whether the conclusions depend on the first-100 restriction, we also run MiniMax-M2.5 on all 348 instances that its Phase 1 run resolves across the full benchmark (Section 4.1).

All metrics of an instance are computed from the patch that the SWE-bench harness scored and from its per-instance report, so hardening and safety always refer to the same patch. An instance enters a backbone's *matched slice* if it is in the gate and every compared variant either was scored by the harness on its released patch or submitted an empty patch, which we count as a regression. All rows of a backbone, including Phase 1, use the same slice: n=64 for MiniMax-M2.5, 55 for Gemini-3.1-Flash-Lite, 54 for GPT-5-mini, and 69 for Qwen3.5-Plus. We ran Phase 2 on every instance with a Phase 1 patch and applied the gate at analysis time. Because instances are processed independently, this is equivalent to running Phase 2 on gated instances only.

Phase 1 runs for at most 250 steps and Phase 2 for at most 150; the single-stage variant receives the sum. The backbone, sandbox, and tool interface are the same for all variants.

3.2 COMPARISON VARIANTS

We compare Phase 1 against four Phase 2 variants and one single-stage baseline:

- **full**: repository memory plus TDD-driven refactoring;
- **wo_memory**: TDD-driven refactoring without repository memory;
- **wo_tdd**: repository memory without the RED–GREEN–VERIFY scaffold;
- **wo_memory+wo_tdd**: neither repository memory nor the TDD scaffold (MiniMax-M2.5 only);
- **single-stage**: one session that first repairs the issue and then follows the RED–GREEN–VERIFY scaffold without memory, with the combined step budget of both stages (400 steps). It has no predictions-file handoff, no sandbox reset, and no Phase 1 gate.

3.3 METRICS

We combine official SWE-bench resolution outcomes with patch-level hardening metrics. **Pass-to-pass** (P2P) is the fraction of Phase 1-solved instances that remain solved after Phase 2, and the **regression** count is the number of solved instances that Phase 2 breaks. P2P normalizes safety across slices, whereas the regression count gives the absolute deployment cost. **Guards** counts the guard clauses, that is, `if`-`raise` and early-`return` checks, in the added lines of the submitted patch. Because absolute counts grow with patch size, we also report **guard density**, the number of guards per added line. The **diagnostic-context ratio** is the fraction of added `raise` statements whose messages include runtime state such as values, types, or indices, and the **actionable-error ratio** is the fraction whose messages state the violated condition or the expected input. Together, these two ratios separate generic exceptions from errors that help a maintainer diagnose a failed contract. All patch metrics are computed from unified diffs rather than from a semantic analysis of the repository. This keeps the evaluator robust across heterogeneous projects, and we interpret the metrics as measures of explicit defensive structure, not of maintainability as a whole (Appendix B.7).

3.4 RESEARCH QUESTIONS

The evaluation addresses three questions. **RQ1** asks whether full Phase 2 improves defensive hardening while preserving Phase 1 repairs. **RQ2** asks how the TDD scaffold and repository memory contribute to the observed gains and whether separating repair from hardening into two stages is necessary. **RQ3** asks how much hardening the regression-safe fallback preserves.

Table 2: Main results on the matched slice of each backbone: Phase 1-resolved instances among the first 100 SWE-bench Verified instances for which every listed variant was scored by the harness or submitted an empty patch (n). All metrics are computed from the scored patches. **Guards**: guard clauses in the added lines of the submitted patch; **Dens.**: guards per added line; **Ctx./Act.**: diagnostic-context and actionable-error ratios over added `raise` statements; **P2P**: fraction of the n instances still resolved; **Reg.**: instances no longer resolved, with empty submissions counted as regressions; **Safe**: guards retained under the regression-safe fallback, whose P2P is 100% by construction. Ctx./Act. are omitted for Phase 1, whose patches add at most two `raise` statements per backbone.

Variant	Guards	Dens.	Ctx.	Act.	P2P	Reg.	Safe
MiniMax-M2.5 (n=64)							
Phase 1	7	0.020	–	–	–	–	–
full	137	0.132	79.7%	48.0%	100.0%	0	137
wo_memory	148	0.119	87.1%	28.0%	98.4%	1	148
wo_tdd	39	0.040	30.4%	8.7%	96.9%	2	39
wo_memory+wo_tdd	42	0.044	24.1%	10.3%	98.4%	1	41
single-stage	77	0.089	80.3%	42.6%	90.6%	6	74
Gemini-3.1-Flash-Lite (n=55)							
Phase 1	5	0.030	–	–	–	–	–
full	77	0.210	75.7%	81.4%	96.4%	2	75
wo_memory	79	0.237	84.6%	87.2%	85.5%	8	73
wo_tdd	39	0.100	8.7%	78.3%	90.9%	5	33
single-stage	48	0.150	68.3%	90.2%	74.6%	14	38
Qwen3.5-Plus (n=69)							
Phase 1	10	0.021	–	–	–	–	–
full	258	0.141	96.8%	6.8%	97.1%	2	257
wo_memory	255	0.144	97.1%	1.2%	97.1%	2	252
wo_tdd	236	0.114	16.2%	45.9%	97.1%	2	214
single-stage	137	0.117	90.4%	4.8%	82.6%	12	131
GPT-5-mini (n=54)							
Phase 1	12	0.016	–	–	–	–	–
full[†]	93	0.072	86.3%	41.1%	$\leq$68.5%	$\geq$17	–
wo_memory	118	0.087	88.5%	48.4%	59.3%	22	103
wo_tdd	46	0.022	36.5%	30.8%	85.2%	8	48
single-stage	101	0.088	86.0%	42.1%	61.1%	21	74

[†]17 of the 54 submissions are empty patches, and the other 37 were not scored by the harness. We therefore report only bounds and exclude this row from the safety comparisons.

4 Experimental Results

This section answers the three research questions of Section 3.4. We first test whether full Phase 2 adds hardening while preserving solved repairs, then isolate the contributions of memory, TDD, and the two-stage split, and finally evaluate the regression-safe fallback and the cost of Phase 2.

4.1 RQ1: Effectiveness

Analysis procedure. We compare the Phase 1 patch with the full Phase 2 patch on each matched slice. The hardening outcomes are guard clauses, guard density, and the diagnostic-context and actionable-error ratios of the submitted patch; the safety outcomes are P2P and the regression count.

Results. Table 2 shows that full Phase 2 adds defensive structure on every backbone. Guard clauses in the submitted patch rise from 7 to 137 on MiniMax-M2.5, from 5 to 77 on Gemini-3.1-Flash-Lite, from 10 to 258 on Qwen3.5-Plus, and from 12 to 93 on GPT-5-mini, and 75.7–96.8% of the added `raise` statements report runtime state. Guard density rises by a factor of 4.5–7.0, so the increase is not only a consequence of larger patches. On the three backbones with complete harness logs, full Phase 2 keeps 64/64, 53/55, and 67/69 repairs: MiniMax-M2.5 incurs no regression, whereas Gemini-3.1-Flash-Lite and Qwen3.5-Plus incur two each. GPT-5-mini is the weakest Phase 2 backbone: 17

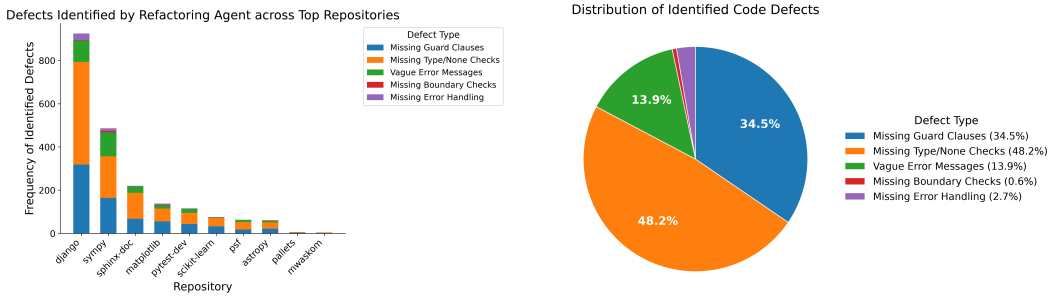


Figure 2: Hardening edits added by full Phase 2 on MiniMax-M2.5 (346 Phase 1-resolved SWE-bench Verified instances with non-empty patches). Left: edits per repository, grouped by the defect class each edit addresses. Right: distribution of the edits over defect classes.

of its 54 full submissions are empty patches, which bounds its P2P at 68.5%. Actionable-error ratios vary more across backbones (6.8% on Qwen3.5-Plus versus 81.4% on Gemini-3.1-Flash-Lite), which indicates that models differ in whether they state the violated expectation even when they report runtime state. On the 348-instance run described below, the hardening edits concentrate on missing guards, missing type or `None` checks, and low-context exceptions rather than on broad rewrites (Figure 2), which matches the intended edit family.

Scale check. To test whether these observations depend on the first-100 slice, we run the full variant and the joint ablation on all 348 instances that MiniMax-M2.5 resolves in Phase 1 across SWE-bench Verified. Full Phase 2 raises guard clauses from 68 to 798 (density 0.026 to 0.122) with a diagnostic-context ratio of 85.5%, versus 250 guards and 15.9% for `wo_memory+wo_tdd`. Safety does not favor the full variant: it keeps 336/348 repairs (96.5% P2P), versus 340/348 (97.7%) for the joint ablation. The hardening gain therefore holds at benchmark scale, whereas the zero-regression result of the first-100 MiniMax-M2.5 slice does not.

Hardening patterns. Figure 2 counts those edits by defect class on the 346 non-empty patches of the same MiniMax-M2.5 run. The edits concentrate in Django and SymPy, the two repositories that contribute the most Phase 1 repairs, and they address missing type or `None` checks and missing guard clauses far more often than missing boundary checks. The released patches realize this distribution through three recurring motifs and one recurring failure mode.

Fail-fast validation adds precondition checks that reject `None`, malformed collections, or invalid scalar ranges before the original computation proceeds. These edits raise the guard count and usually keep the blast radius small because they can be inserted locally. Diagnostic enrichment turns a generic exception into a specific one that reports expected and observed types, the invalid value, or the violated contract, which raises the diagnostic ratios. Safer access paths replace brittle lookups and indexing with guarded or defaulted accessors. The recurring failure mode is the opposite of these motifs: the agent broadens the semantic footprint of the repair, rewriting behavior around the repaired region or generalizing the logic beyond what the tests constrain. This failure mode is why the method pairs the TDD scaffold with the regression-safe fallback.

Answer to RQ1: *Full Phase 2 raises guard clauses and guard density on every backbone and keeps 96.4–100% of repairs on the three backbones with complete harness logs. The hardening gain persists on the 348-instance run, which keeps 96.5% of repairs, and the added edits follow the intended fail-fast and diagnostic family rather than broad rewrites.*

4.2 RQ2: Attribution

Analysis procedure. The `full` versus `wo_tdd` gap isolates the TDD scaffold with memory held fixed, and the `full` versus `wo_memory` gap isolates memory with the scaffold held fixed. The joint ablation, available for MiniMax-M2.5, removes both. The single-stage variant tests whether the two-stage split is necessary. Because it uses the scaffold without memory, its two-stage counterpart is `wo_memory`, and it receives the combined step budget of both stages (400 steps).

TDD drives hardening quality. Removing TDD lowers the diagnostic-context ratio on every backbone, by 49.3 points on MiniMax-M2.5, 67.0 on Gemini-3.1-Flash-Lite, 80.6 on Qwen3.5-Plus, and 49.8 on GPT-5-mini. It also lowers guard density on every backbone, for example from 0.132 to 0.040 on MiniMax-M2.5. It increases regressions on MiniMax-M2.5 (2 vs. 0) and Gemini-3.1-Flash-Lite (5 vs. 2) and leaves them unchanged on Qwen3.5-Plus (2 vs. 2). The context-ratio drop appears even when guard counts stay high: without TDD, Qwen3.5-Plus still adds 236 guards, but only 16.2% of its new errors report runtime state. The joint ablation on MiniMax-M2.5 (42 guards, 24.1% context) is close to `wo_tdd`, consistent with TDD being the main source of hardening quality.

Repository memory. Memory has no consistent effect on hardening volume. Without memory, guard counts stay within 8% of full on three backbones (148 vs. 137, 79 vs. 77, and 255 vs. 258) and rise by 27% on GPT-5-mini, and guard density differs from full by at most 0.027. The difference lies in safety: removing memory increases regressions on MiniMax-M2.5 (1 vs. 0) and Gemini-3.1-Flash-Lite (8 vs. 2) and leaves them unchanged on Qwen3.5-Plus (2 vs. 2). These differences involve few instances, and on the 348-instance run the joint ablation regresses fewer instances than full (8 vs. 12). The safety effect of memory is therefore suggestive rather than established.

Why two stages. With the same scaffold, no memory, and a matched step budget, the single-stage agent regresses more repairs than two-stage `wo_memory` on three of four backbones: 6 vs. 1 on MiniMax-M2.5, 14 vs. 8 on Gemini-3.1-Flash-Lite, and 12 vs. 2 on Qwen3.5-Plus. The two are comparable on GPT-5-mini (21 vs. 22). Here a regression is an instance that the same backbone solves with the Phase 1 prompt alone but that the single session fails. The single session also adds fewer guards on every backbone (77 vs. 148, 48 vs. 79, 137 vs. 255, and 101 vs. 118). Compared with full MAPR, which loses 0, 2, and 2 repairs on the first three backbones, the single session loses 6, 14, and 12. One explanation, which we do not test directly, is that hardening edits interfere with the repair when both share one trajectory, whereas the handoff lets Phase 2 start from a verified repair.

Answer to RQ2*: The RED–GREEN–VERIFY scaffold is the main driver of hardening quality on every backbone. Repository memory does not change hardening volume, and its association with fewer regressions is small and does not hold on the 348-instance run. With the same scaffold and step budget, separating repair from hardening yields more guards than a single session on every backbone and fewer regressions on three of four backbones.*

4.3 RQ3: Deployment Safety

Analysis procedure. We apply the regression-safe policy of Section 2.5, which reverts each regressed instance to its Phase 1 patch, and measure how much hardening it preserves.

Results. The fallback reaches 100% P2P by construction, so the question is how much hardening survives. Full Phase 2 regresses 0, 2, and 2 instances on the fully evaluated backbones, so the fallback replaces at most two patches per backbone and retains 137/137, 75/77, and 257/258 guard clauses (column Safe of Table 2). On the 348-instance run, it replaces 12 patches and retains 776 of 798 guards (97.2%). Ablations can lose more hardening: `wo_tdd` retains 214 of 236 guards on Qwen3.5-Plus, and `wo_memory` retains 103 of 118 on GPT-5-mini. The rule detects regressions with the benchmark tests; a deployment would use the repository's own tests. The safe view measures how much hardening such a test gate preserves, not correctness on unseen tests.

Answer to RQ3*: The regression-safe fallback restores all repairs and retains at least 97% of the guard clauses added by full Phase 2 on every fully evaluated slice, including the 348-instance run.*

Cost. On the matched slices, full Phase 2 uses 66.7, 35.0, 68.6, and 19.5 model calls per instance on MiniMax-M2.5, Gemini-3.1-Flash-Lite, Qwen3.5-Plus, and GPT-5-mini. This is 0.9–1.5 times the calls of Phase 1, with median wall-clock times of 2.5–9.1 minutes (Appendix Table 3). Because Phase 2 runs only on repaired instances, its total cost scales with the number of repairs, not with benchmark size. Appendix B.8 reports trajectory statistics and a failure-mode analysis.

5 Related Work

Software engineering agents. Recent software engineering agents combine repository retrieval, structured search, shell interaction, and validation to resolve real issues. RepoCoder, AutoCodeRover, and RepoGraph use repository context or graph structure during repair (Zhang et al., 2023; 2024; Ouyang et al., 2024); Agentless, OpenHands, SWE-Search, Mini-SWE-agent, and Live-SWE-agent show that diverse localize–edit–validate loops can reach strong SWE-bench performance (Xia et al., 2025a; Wang et al., 2024; Antoniades et al., 2024; SWE-agent Team, 2025; Xia et al., 2025b). Multi-agent and planning-oriented systems further decompose the task into role assignment, localization, editing, testing, or adaptive test-time compute (Hong et al., 2024; Li et al., 2025; 2023; Tao et al., 2024; Gao et al., 2025), and stronger regression tests can change benchmark conclusions (Yu et al., 2025). This literature mostly treats the issue report as the objective: localize the relevant code, produce a patch, and stop when the target tests pass. Repository context serves to localize or explain the current bug, not to decide what should happen after the bug is fixed. General-purpose coding agents such as Codex and Claude Code let users combine project memory files, test-first instructions, and review passes in one session, but the effect of such combinations on hardening has not been measured separately from their effect on regressions. Our single-stage baseline (Section 4.2) is a controlled instance of this pattern. MAPR instead separates the two steps: Phase 1 remains a conventional repair stage, and Phase 2 re-enters a fresh sandbox after the repair has passed, replays the patch, and makes small hardening edits around historically risky code.

Repository history and refactoring. Mining software repositories has long shown that project history predicts future defects and maintenance hotspots. Prior work links fixes to bug-inducing commits (Sliwerski et al., 2005) and finds that churn, ownership, and diff complexity are informative defect predictors (Moser et al., 2008; Hassan, 2009; Kamei et al., 2013; Herzig et al., 2016). Hotspot analysis brings these ideas into maintenance practice by helping developers prioritize areas where refactoring effort is likely to pay off (Tornhill, 2015; Borg et al., 2024). MAPR uses the same family of signals in a narrower, online setting: the agent starts from a passing Phase 1 patch, and history determines where to look rather than licensing a broad rewrite.

Automated software refactoring. Automated refactoring has long studied code smells, rule-based transformations, search-based opportunity detection, and selective developer practice (Fowler, 2018; Mens & Tourwé, 2004; Agnihotri & Chug, 2020; O'Keeffe & Ó Cinnéide, 2008; Mohan & Greer, 2018; Murphy-Hill et al., 2012; Golubev et al., 2021). Recent LLM work shows that models can recommend or perform behavior-preserving refactorings, but also exposes risks of behavior-altering edits and weak validation (Midolo et al., 2026; Liu et al., 2024; Reich & Maalej, 2023). Multi-agent systems such as RefAgent, MUARF, and RefactorGPT coordinate smell detection, planning, editing, and validation (Oueslati et al., 2025; Xu, 2025; Karabiyik, 2025), while benchmarks show that stateful multi-file refactoring remains difficult (Pucho et al., 2025; Gautam et al., 2025; Xu et al., 2026). MAPR is deliberately narrower. It does not target design improvement, API migration, or code-smell removal; its edits are limited to defensive guards and diagnostic errors within a bounded blast radius, and its RED–GREEN–VERIFY scaffold tests these edits before submission. The task is post-repair hardening, which is useful only if the original repair remains solved.

6 Conclusion

We presented MAPR, a two-phase workflow that hardens code after a repair has passed its tests. Phase 2 replays the Phase 1 patch in a fresh sandbox, conditions on patch-scoped repository memory, and follows a RED–GREEN–VERIFY scaffold. It raises guard density by a factor of 4.5–7.0 while keeping 96.4–100% of repairs on the backbones with complete harness logs, and a fallback to the Phase 1 patch restores the remaining repairs while retaining at least 97% of the added guards. The TDD scaffold drives hardening quality, and separating repair from hardening regresses fewer repairs than a single session with the same budget. Appendix B.10 discusses threats to validity.

AI use statement

In this work, we used generative AI tools only to polish the writing of the manuscript, that is, to improve the grammar, wording, and readability of text drafted by the authors. We did not use

generative AI tools for any other purpose in preparing this work. The language models evaluated in Section 4 are the backbones of the studied agents and are the object of study rather than writing aids. We have reviewed all AI-polished text. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

Reproducibility statement

The anonymized code, data, and analysis artifacts are available as a review artifact (Anonymous Authors, 2026). The artifact includes, for every backbone, the Phase 1 gate, the instance IDs of each matched slice, the prediction files and SWE-bench harness logs of every variant, the benchmark configurations, and the script that recomputes Table 2 from these logs. Appendix B lists the artifacts required to reproduce each comparison slice, Appendix B.5 describes the runner and sandbox configuration, Appendices B.6 and B.7 define the evaluation pipeline and metrics, and Appendix B.4 summarizes the prompt configuration of every evaluated variant.

References

Mansi Agnihotri and Anuradha Chug. A systematic literature survey of software metrics, code smells and refactoring techniques. *Journal of Information Processing Systems*, 16(4), 2020.

Anonymous Authors. Replication package for predictive refactoring study, 2026. URL https://figshare.com/s/b1f96ed457df4694a5a4.

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

Markus Borg, Marwa Ezzouhri, and Adam Tornhill. Ghost echoes revealed: Benchmarking maintainability metrics and machine learning predictions against human assessments. In *IEEE International Conference on Software Maintenance and Evolution (ICSME), Industry Track*, 2024. URL https://ieeexplore.ieee.org/abstract/document/10795019. Preprint available at https://arxiv.org/abs/2408.10754.

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2nd edition, 2018. ISBN 978-0-13-468599-1.

Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. Trae Agent: An LLM-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*, 2025. URL https://arxiv.org/abs/2507.23370.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. RefactorBench: Evaluating stateful reasoning in language agents through code. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2503.07832.

Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: A large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1303–1313, 2021.

Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pp. 78–88, 2009. doi: 10.1109/ICSE.2009.5070510.

Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21(2):303–336, 2016.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2308.00352.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi: 10.1109/TSE.2012.70.

Muhammed Abdulhamid Karabiyik. RefactorGPT: A ChatGPT-based multi-agent framework for automated code refactoring. *PeerJ Computer Science*, 11:e3257, 2025.

Chenliang Li, He Chen, Ming Yan, Weizhou Shen, Haiyang Xu, Zhikai Wu, Zhicheng Zhang, Wenmeng Zhou, Yingda Chen, Chen Cheng, et al. Modelscope-agent: Building your customizable agent system with open-source large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 566–578, 2023.

Zongwei Li, Zhonghang Li, Zirui Guo, Xubin Ren, and Chao Huang. DeepCode: Open agentic coding. *arXiv preprint arXiv:2512.07921*, 2025. URL https://arxiv.org/abs/2512.07921.

Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. An empirical study on the potential of LLMs in automated software refactoring. *arXiv preprint arXiv:2411.04444*, 2024. URL https://arxiv.org/abs/2411.04444.

Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. doi: 10.1109/TSE.2004.1265817.

Alessandro Midolo, Emiliano Tramontana, and Massimiliano Di Penta. From human to machine refactoring: Assessing GPT-4's impact on Python class quality and readability. *arXiv preprint arXiv:2601.13139*, 2026. URL https://arxiv.org/abs/2601.13139.

Michael Mohan and Des Greer. A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, 6(1):3, 2018.

Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pp. 181–190, 2008. doi: 10.1145/1368088.1368114.

Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012. doi: 10.1109/TSE.2011.41.

Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008. doi: 10.1016/j.jss.2007.06.003.

Khouloud Oueslati, Maxime Lamothe, and Foutse Khomh. RefAgent: A multi-agent LLM-based framework for automatic software refactoring. *arXiv preprint arXiv:2511.03153*, 2025. URL https://arxiv.org/abs/2511.03153.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. *arXiv preprint arXiv:2410.14684*, 2024. URL https://arxiv.org/abs/2410.14684.

Alexander Puma Pucho, Alexandre Mello Ferreira, Elder José Reioli Cirilo, and Bruno B. P. Cafeo. Refactoring Python code with LLM-based multi-agent systems: An empirical study in ML software projects. In *Simpósio Brasileiro de Engenharia de Software (SBES)*, pp. 678–684. SBC, 2025.

Pavel Reich and Walid Maalej. Testability refactoring in pull requests: Patterns and trends. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023. doi: 10.1109/ICSE48619.2023.00098. URL https://arxiv.org/abs/2303.14253.

Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005. doi: 10.1145/1082983.1083147. URL https://doi.org/10.1145/1082983.1083147.

SWE-agent Team. Mini-SWE-agent: A minimal agent for software engineering tasks. https://github.com/SWE-agent/mini-swe-agent, 2025.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: LLM-based multi-agent framework for GitHub issue resolution. *Advances in Neural Information Processing Systems*, 37:51963–51993, 2024.

Adam Tornhill. *Your Code as a Crime Scene*. Pragmatic Bookshelf, 2015.

Adam Tornhill. Assessing technical debt in automated tests with codescene. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 122–125. IEEE, 2018.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, et al. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024. URL https://arxiv.org/abs/2407.16741.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025a.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly?, 2025b. URL https://arxiv.org/abs/2511.13646.

Yisen Xu. MUARF: Leveraging multi-agent workflows for automated code refactoring. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 226–227. IEEE, 2025.

Yisen Xu, Jinqiu Yang, et al. SWE-Refactor: A repository-level benchmark for real-world LLM-based code refactoring. *arXiv preprint arXiv:2602.03712*, 2026. URL https://arxiv.org/abs/2602.03712.

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 (NeurIPS)*, 2024.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-Bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. URL https://aclanthology.org/2025.acl-long.189.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2471–2484. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

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 (ISSTA)*, pp. 1592–1604. ACM, 2024. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

A Motivation

A.1 A Motivating Example

We illustrate post-repair hardening with issue #13033 of the public `astropy` repository. Phase 1 resolves the reported symptom, but repository history indicates that the repaired code remains fragile, and Phase 2 hardens it before the next related defect is reported.

The original bug. The issue reports that removing a required column from a `TimeSeries` object whose required columns are `time` and `flux` produces a misleading exception (Listing 1):

Listing 1: Trigger code and the misleading error before Phase 1.

```
ts._required_columns = ["time", "flux"]
ts.remove_column("flux")
# ValueError: TimeSeries object is invalid -
# expected 'time' as the first columns but found 'time'
```

The message claims to have found `'time'` where `'time'` was expected. This self-contradictory report hides the actual cause, namely that the `flux` column is missing. The fault lies in the core timeseries module: the check compares the full list of leading columns, but the message reports only the first expected column and the first actual column, which here are both `time`.

Phase 1: reactive bug fix. A standard repair agent rewrites the message to report the complete expected and actual column lists and the first mismatching index (Listing 2). The patch resolves the reported symptom and passes the column-validation regression test of the instance, at which point a standard agent workflow terminates, leaving the validation logic unchanged.

Listing 2: Phase 1 patch: corrected error-message formatting.

```
# Before (buggy):
raise ValueError(
    "{} object is invalid - expected '{}' as the first"
    " column{} but found '{}'"
    .format(cls, required_columns[0], plural, colnames[0])
)

# After Phase 1 (fixed):
raise ValueError(
    f"{cls} object is invalid - expected leading columns:"
    f" {expected_leading!r}; found: {actual_leading!r};"
    f" first mismatch at index {first_diff_index}:"
    f" expected {expected_leading[first_diff_index]!r},"
    f" got {actual_leading[first_diff_index]!r}"
)
```

What repository memory reveals. For the core timeseries module, repository memory returns the following signals derived from historical pull requests:

- **High bug density** (0.61): 61% of the pull requests that modify the file are bug fixes.
- **Temporal coupling**: the module co-changes with a companion sampled-series module in 73% of nearby pull requests, so a defensive change in one module may need to be mirrored in the other.
- **Recurring pattern**: memory flags missing input-type validation on column-list attributes, a pattern that appears in three prior bug-fix pull requests for this file.

These signals indicate that the code region is fragile beyond the specific formatting bug.

Phase 2: TDD-driven predictive refactoring. Guided by these signals, MAPR Phase 2 adds guard clauses at the entry point of the validation logic, so that invalid values of the required-columns attribute are rejected before the column check runs (Listing 3):

Listing 3: Phase 2 patch: guard clauses added by MAPR.

```
# Added by MAPR Phase 2:
if isinstance(self._required_columns, (str, bytes)):
    raise TypeError(
        f"_required_columns must be a list/tuple of str,"
        f" got {type(self._required_columns).__name__}:"
        f" {self._required_columns!r}"
    )
try:
    iter(self._required_columns)
except TypeError:
    raise TypeError(
        f"_required_columns must be iterable, got"
        f" {type(self._required_columns).__name__}"
```

```
    )
for idx, col in enumerate(self._required_columns):
    if not isinstance(col, str):
        raise TypeError(
            f"elements of _required_columns must be str,"
            f" got {type(col).__name__} at index {idx}: {col!r}"
        )
```

The guards reject three kinds of misuse, namely a bare string, a non-iterable value, and a list with non-string elements, each of which would otherwise surface as a cryptic downstream failure. The TDD scaffold first writes a failing test for each guard (RED), then adds the guards until the tests pass (GREEN), and finally verifies that the benchmark tests of the instance still pass (VERIFY).

Observation. A standard agent workflow stops after Phase 1, once the immediate symptom is resolved, although repository history shows that the repaired region remains fragile. MAPR uses historical pull-request signals, namely bug density, temporal coupling, and recurring defect patterns, to *predict* where defensive hardening is most valuable, and it uses TDD to realize this hardening as concrete, verifiable code changes rather than broad rewrites.

B Additional Implementation and Evaluation Details

B.1 Terminology and Artifact Mapping

The implementation stores repository history as pull-request graph indices and coupling tables, from which a scorer ranks files and patterns and an advisor renders a bounded prompt summary. We call this pipeline *repository memory* because the method conditions post-repair behavior on persistent signals from repository history; it does not learn graph representations.

Artifact names predate the terminology of the paper. Result directories and configuration files use `graph` for memory and `test` or `tdd` for the TDD scaffold: `wo_graph` denotes `wo_memory`; `wo_test` denotes `wo_tdd`; `wo_graph_wo_tdd` and `wo_test_wo_graph` denote `wo_memory+wo_tdd`; and `phase0` denotes the single-stage baseline. We verified each mapping against the saved prompts: `wo_memory` trajectories contain the RED–GREEN–VERIFY scaffold but no memory block, and `wo_tdd` trajectories contain the memory block but no scaffold.

B.2 Reproducibility and Artifact Inventory

Core artifacts. Reproducing the main analyses requires four classes of artifacts:

- a Phase 1 predictions file with its saved trajectories;
- one or more Phase 2 predictions files with their trajectories;
- the passed-instance gate derived from the Phase 1 evaluation logs;
- the SWE-bench evaluation directories with per-instance harness reports for every variant.

The Phase 1 predictions file is the only interface between the stages: its patch field is the object replayed in the Phase 2 sandbox. Phase 2 trajectories additionally record the temporary tests, the hardening commands, and the final verification trace, which support qualitative inspection; the quantitative evaluation consumes only the final patches and reuses the test outcomes written by the SWE-bench harness instead of rerunning tests (Appendix B.6).

Evidence boundary. The quantitative claims rest on five slices: the four first-100 matched slices of Table 2 (MiniMax-M2.5, n=64; Gemini-3.1-Flash-Lite, n=55; Qwen3.5-Plus, n=69; GPT-5-mini, n=54) and the 348-instance MiniMax-M2.5 run over the full benchmark. The artifact lists the instance IDs of every slice, and a single script recomputes every table cell from the harness logs and records each regressed instance. A pairing audit compares each predictions file with the `patch.diff` files of all evaluation-log directories of the same model. When a released predictions file covers an instance, the instance counts only if the scored patch is identical to the released one. Two slices therefore rely on the patches stored with the harness logs: the MiniMax-M2.5 single-stage row, whose released predictions come from a later re-run of the same configuration, and the part of the

348-instance run outside the first 100 instances. The audit also identifies an artifact that no reported number uses: the Gemini-3.1-Flash-Lite `wo_memory+wo_tdd` directory, which duplicates `wo_memory`.

B.3 Repository Memory Construction

Storage and loading. Repository memory is implemented by the graph-advisor modules of the agent code. The advisor reads either a single JSON graph or a per-repository JSONL index. In index mode, repository histories are loaded on demand and held in a small least-recently-used cache, so a benchmark run over many repositories does not keep every history in memory.

Signals and prompt summary. The scorer combines five signals into a composite file-level risk score with fixed weights: bug density (0.35), author-linked error patterns (0.25), temporal coupling (0.20), review coverage (0.15), and code churn (0.05). Bug density is the fraction of the pull requests modifying a file that are bug fixes, and a file counts as high-risk when its bug density exceeds 0.4. The prompt receives a bounded summary with three parts: up to ten high-risk files ranked by bug density, up to five common bug patterns extracted from issue and pull-request labels, and up to five temporal-coupling alerts for files that often change together.

Patch-scoped retrieval and degradation. The runner passes the files touched by the Phase 1 patch to the advisor, which restricts high-risk ranking and temporal coupling to this neighborhood. If the repository has no history, the advisor cannot be initialized, or retrieval fails for an instance, the runner omits the memory block, and the template does not render the memory section. An instance without usable history thus runs with the no-memory prompt instead of failing.

B.4 Benchmark Prompts and Variants

The artifact contains six redacted benchmark configurations in the `prompt_snapshots/` directory. This section summarizes their contracts; the YAML files contain the complete prompt text.

Phase 1 repair. The Phase 1 configuration (`swebench.redacted.yaml`) defines a conventional shell-based repair agent: analyze the codebase, reproduce the issue, edit non-test source files, rerun the reproduction, and submit a patch over the edited source files. Restricting edits to non-test sources prevents the agent from improving benchmark outcomes by rewriting tests or packaging files. The prompt contains neither repository memory nor hardening tests.

Phase 2 hardening. The full Phase 2 configuration (`swe_refactor.redacted.yaml`) assumes that the Phase 1 patch is already applied and frames the task as behavior-preserving defensive hardening. Before the first turn, it provides the original issue, the replayed Phase 1 patch, the fail-to-pass and pass-to-pass test names, and, when available, the memory summary. It prioritizes the files modified by the Phase 1 patch over memory-retrieved files, forbids changes to external behavior and function signatures, and requires the RED–GREEN–VERIFY workflow of Section 2.4 with the temporary tests kept outside the submission. The requested edit family is narrow: guard clauses, safer access paths, context-rich exceptions, boundary checks, and lightweight resource-safety improvements. The agent submits a merged patch that contains the repair and the accepted hardening edits.

Single-stage variant. The combined configuration (`swe_combined.redacted.yaml`) runs repair and hardening in one session. The agent is first instructed to fix the bug without submitting and then follows most of the Phase 2 scaffold in the same trajectory. The configuration injects no repository memory, has no predictions-file handoff or Phase 1 gate, and has a step budget of 400, the sum of the Phase 1 (250) and Phase 2 (150) budgets. Its comparison with `wo_memory` thus isolates the stage boundary while holding the prompt ingredients and the total budget fixed.

Ablations. Each ablation removes one or both ingredients of the full Phase 2 prompt and keeps the rest of the runner unchanged. The file names use `wo_test` for `wo_tdd`:

- `wo_memory` removes memory injection and retains the TDD scaffold and the merged-patch contract (`swe_refactor_wo_memory.redacted.yaml`);

- `wo_tdd` removes the RED–GREEN–VERIFY scaffold and retains memory (`swe_refactor_wo_test.redacted.yaml`);
- `wo_memory+wo_tdd` removes both ingredients (`swe_refactor_wo_test_wo_memory.redacted.yaml`).

B.5 Runner and Sandbox Configuration

Fresh sandbox and patch replay. Phase 2 always runs in a fresh environment. The runner locates the Phase 1 patch of the instance, creates the benchmark environment, and replays the patch before the agent acts. The patch is base64-encoded, decoded inside the sandbox, and piped to `git apply`, which avoids shell-escaping errors and hangs on large multiline patches. The runner records missing Phase 1 predictions, empty Phase 1 patches, and failed patch applications as distinct early-exit states, so instances that never enter the Phase 2 loop are not counted as model failures.

Template variables. After setup, the runner injects the replayed patch, the fail-to-pass and pass-to-pass test lists parsed from the dataset, and the memory summary into the template variables of the agent; the agent core is not modified. Templates are rendered in strict mode, in which an undefined variable raises an error, so every optional section, such as the memory block, is guarded by a template condition. A missing variable therefore cannot silently corrupt a prompt.

Isolation. The runner deep-copies the configuration before constructing each instance's environment, so settings attached during environment creation do not leak between parallel workers. All reported runs use Docker sandboxes with a clean working tree per instance, so Phase 2 sees only the replayed repair and the memory summary, not residual state from earlier runs.

B.6 Evaluation Pipeline and Comparison Slices

For each slice, the evaluator takes a Phase 1 predictions file, one or more Phase 2 predictions files with their SWE-bench evaluation logs, and the Phase 1 gate. It processes all compared variants under one accounting scheme and writes per-variant JSON summaries, comparison reports, and figures.

Diff-based analysis. The evaluator analyzes unified diffs directly instead of applying patches to checked-out repositories. Local repository heads may differ from the base commits used by the harness, which makes patch application unreliable at scale; diff analysis yields reproducible structural measurements without reconstructing repository state.

Gate and matched slice. Only instances that the Phase 1 harness logs mark as resolved are eligible. An instance must also be evaluable for every compared variant: either the harness scored it, or the variant submitted an empty patch, which counts as a regression. Per-instance metrics use the `patch.diff` stored with each harness report, so a cell never pairs the patch of one run with the test outcome of another. For MiniMax-M2.5, the gate contains 73 instances and the matched slice 64; each of the nine excluded instances lacks a scored submission for at least one variant.

Regression-safe outputs. A post-processing step writes regression-safe prediction files that replace each regressed Phase 2 output with its Phase 1 patch. They are kept apart from the raw Phase 2 outputs, so each reported number traces to either the generator or the selection rule.

B.7 Metric Definitions

All patch metrics are computed from the added lines of unified diffs, not from a syntax tree of the repository, and thus capture explicit defensive structure better than semantic reorganization.

Resolution and pass-to-pass. Resolution is taken from the SWE-bench evaluation logs. Pass-to-pass is the fraction of Phase 1-solved instances that remain solved after Phase 2, and the regression count is the number of these instances that fail after Phase 2. Neither quantity applies to a Phase 1 row, which is the reference for both, so Table 2 marks those cells with a dash.

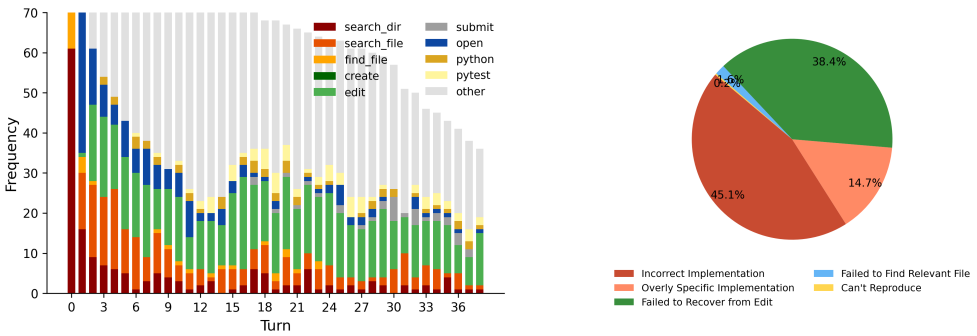


Figure 3: Process-level trajectory diagnostics. Left: action frequency at each turn on solved MiniMax-M2.5 trajectories, grouped into broad functional buckets. Right: failure modes for unresolved instances, labeled with a lightweight classifier and rule-based fallbacks.

Guard clauses. The evaluator counts `if ...: raise` and `if ...: return` patterns in single-line and multiline form. All reported numbers are unweighted, although the implementation also weights early returns by 1.5. Guard density divides the guards of a slice by its added lines.

Diagnostic error quality. Both ratios are computed over added `raise` statements. A message carries runtime context if it interpolates runtime values, for example through an f-string or string formatting, and it is actionable if it states the expected condition, for example in the form "expected X, got Y". The implementation also flags exception types that do not match the described violation, such as a value-range constraint raised as a `TypeError`.

Blast radius. Blast radius counts the touched files and hunk headers of a patch. It is descriptive rather than normative: a broad patch is not necessarily worse when the hardening spans coupled files, but a large blast radius helps identify diffuse editing.

Correspondence with the Phase 2 prompt. The metrics correspond to the edit family requested in Phase 2: guard clauses to fail-fast validation, the diagnostic ratios to informative exceptions, and blast radius to targeted edits. A high score thus indicates that a patch realizes the requested hardening style, not that this hardening is the best one possible.

B.8 Agent Trajectory and Failure Analysis

Section 4.1 discusses the defect-class distribution of the 348-instance MiniMax-M2.5 run (Figure 2). Following prior SWE-agent analyses (Yang et al., 2024), we report two process-level statistics (Figure 3): the frequency of action types at each turn on solved instances, and the distribution of failure modes over unresolved instances. The action frequencies are obtained by extracting the commands from saved trajectories, grouping them into broad functional categories, and aggregating counts by turn index; commands outside these categories fall into an `other` bucket. The failure modes are obtained by an LM-assisted labeling step over unresolved trajectories, with a rule-based fallback, using the SWE-agent taxonomy (e.g., Incorrect Implementation, Failed to Recover from Edit). Both procedures are heuristic, so we use them for qualitative diagnosis only.

B.9 Regression-Safe Deployment

The fallback leaves generation unchanged and only selects which patch is shipped, so the raw Phase 2 outputs remain available for analyzing the trade-off between hardening and regressions. For the full variant, the fallback is triggered by no instance on the first-100 MiniMax-M2.5 slice, by two Django instances on Gemini-3.1-Flash-Lite, and by one Astropy and one Django instance on Qwen3.5-Plus. On the 348-instance MiniMax-M2.5 run, it is triggered by 12 instances: four from Django, four from SymPy, three from scikit-learn, and one from Matplotlib. The fallback protects the repair but does not make the generator itself more reliable, so a deployment should retain both outputs.

B.10 Threats to Validity

Internal validity. Within each backbone, all variants share the runner, sandbox, step budgets, evaluation scripts, and matched slice. Regression counts are small (at most 22 per slice), so we do not claim statistical significance for differences of a few instances. The TDD effects on the diagnostic-context ratio and guard density, in contrast, are large and hold on all four backbones.

Construct validity. The patch metrics are syntactic proxies computed from diffs. They may miss semantic changes that a reviewer would notice, and the regression signal is only as strong as the tests of each instance. The RED-phase tests are generated by the agent rather than curated by humans; we treat them as process evidence, not as an independent certification of robustness.

Repository-memory temporal validity. Because MAPR uses history predictively, memory must predate the issue. The runs in Section 4 built memory from the full pull-request corpus. The released runner restricts memory per instance to pull requests merged before the target pull request was opened, excluding the target pull request and every later-numbered one.

To quantify the effect of this restriction, we re-rendered the memory summary of every matched instance with and without it, using the patch-scoped query of the runner. The unrestricted re-renderings reproduce the high-risk files of the saved prompts for 504 of 507 checked instances. The restriction changes the summaries substantially: the set of high-risk files is unchanged for only 18–35% of instances (mean Jaccard similarity 0.69–0.76), the bug-pattern ranking is unchanged for 48–58%, and the coupling alerts are almost always unchanged. The unrestricted summaries cite the target pull request as an example in 5–12% of instances and later pull requests in 62–72%.

The memory-conditioned rows (`full` and `wo_tdd`), and hence the `full` versus `wo_memory` comparison, are therefore exposed to temporal leakage. Two properties bound this risk. First, memory never enters grading: Phase 2 starts from an already solved issue and is scored only by the benchmark tests. Second, the conclusions that do not involve memory are unaffected. The single-stage comparison uses no memory, and the TDD effect also holds among memory-free variants: on MiniMax-M2.5, `wo_memory` reaches 148 guards and an 87.1% context ratio, versus 42 and 24.1% for `wo_memory+wo_tdd`.

External validity. All runs use the Python repositories of SWE-bench Verified with memory derived from their pull-request histories, and four backbones. Generalization to other languages, private monorepos, and other model families remains to be tested. Memory is informative only when a repository's history yields stable signals about defect concentration or file coupling; for repositories with sparse history, the runner falls back to the no-memory prompt.

B.11 Compute and Per-Instance Cost

All experiments use Dockerized SWE-bench environments and the model-specific run configurations included in the artifact. Table 3 reports model calls and wall-clock time per instance on the matched slices of Table 2, computed from the saved trajectories. Wall-clock time spans the first to the last message of a trajectory and includes sandbox command execution. We do not report dollar cost because cost tracking was unavailable for some model endpoints. The saved MiniMax-M2.5 single-stage trajectories come from a re-run of the evaluated configuration, so we omit them.

Table 3: Mean model calls and median wall-clock minutes per instance on the matched slices.

	Model calls			Wall-clock (min)		
Backbone	Phase 1	Phase 2	Single	Phase 1	Phase 2	Single
MiniMax-M2.5	44.4	66.7	–	3.1	9.1	–
Gemini-3.1-Flash-Lite	37.8	35.0	57.3	1.2	2.7	3.5
Qwen3.5-Plus	62.0	68.6	82.8	4.2	6.3	8.2
GPT-5-mini	17.6	19.5	34.6	1.9	2.5	6.1

B.12 Artifact Redaction

The released prompt configurations (Appendix B.4) and pipeline files keep role definitions, task decomposition, behavioral constraints, verification procedures, and submission contracts intact. Redaction removes or normalizes the following information:

- service endpoints, hostnames, IP addresses, and internal API bases;
- API credentials, tokens, and authentication fields;
- absolute filesystem paths, volume names, and run identifiers, which are replaced by neutral placeholders such as `workspace root` and `evaluation logs`;
- author and reviewer identities in the pull-request history.

B.13 Broader Impact

Post-repair hardening can help developers make historically fragile code fail earlier and with more informative errors immediately after a repair. A system that proposes follow-up changes at scale can also be used to push broad modifications without adequate human review. MAPR anchors the second stage to a verified repair, bounds it with the benchmark tests and its own temporary tests, and reports raw and regression-safe outputs separately. We recommend that deployments keep both outputs visible so that users can weigh hardening against regression risk.

C Qualitative Analysis

We analyze four trajectories from the Qwen3.5-Plus full Phase 2 run: two localized hardening runs, one over-broad refactoring, and one run that exhausts its budget before submission. The admin-extension and float-formatting patches pass the harness. The enum-serialization run submits an empty patch and is one of the two full-variant regressions of Qwen3.5-Plus in Table 2. The FITS-card instance lies outside the gate because the harness marks its Phase 1 patch as unresolved; we include it because its trajectory illustrates the intended Phase 2 behavior.

Astropy FITS-card instance

Issue summary. The issue concerns inconsistent handling of doubled single quotes in the Astropy FITS `Card` type. Phase 1 changes the escaping logic in the card-parsing module but fails the harness tests. The Phase 2 task is not to change the escaping again, but to add defensive checks along the same parsing path so that malformed card images fail earlier and with clearer messages.

Interaction summary.

1. **Start from the patched file**: The agent records the baseline blast radius and focuses on the card-parsing module, the only file touched by the replayed Phase 1 patch.
2. **Attempt verification first**: It tries to run the relevant FITS header test, finds that the test runner is not installed in the sandbox, installs it, and spends several turns on test bootstrapping before changing any code.
3. **Add localized defensive checks**: It then inserts guard clauses at the main card-parsing entry points that reject `None`, wrong input types, empty card images, and too-short card bodies with explicit `TypeError`, `ValueError`, or `VerifyError` messages.
4. **Prepare a merged patch**: The agent creates a patch that contains both the Phase 1 escaping fix and the new checks, confirms that the scratch test directory is excluded, and submits.

Evaluation. This trajectory fits the intended Phase 2 behavior. The agent stays on the Phase 1 file, adds fail-fast validation at the relevant parsing entry points, and improves diagnostics without reopening the repaired escaping logic. Its main weakness is procedural: the run spends noticeable effort on test setup, which weakens the evidence of end-to-end verification.

DJANGO ADMIN-EXTENSION INSTANCE

Issue summary. The issue requests a new hook, `ModelAdmin.get_inlines()`, so that Django admin inlines can depend on the request or the edited object. After Phase 1 adds the hook, Phase 2 should harden it against invalid return values and degenerate configurations.

Interaction summary.

1. **Inspect the hook boundary**: The agent records the baseline diff, reads the Django admin options module, and identifies the new `get_inlines()` hook as the place for defensive validation.
2. **Write temporary tests around misuse**: It creates scratch tests that exercise invalid inline configurations, including non-sequence return values and missing inline declarations.
3. **Harden the hook contract**: The agent updates `get_inline_instances()` to validate the hook output before iterating over it, and updates `get_inlines()` so that `None` degrades to an empty list while values that are neither lists nor tuples raise a context-rich `TypeError`.
4. **Submit a narrow merged patch**: The final patch keeps the hook design of Phase 1 intact, excludes the temporary tests, and leaves the rest of the admin control flow unchanged.

Evaluation. Among the submitted runs, this is the clearest example of successful post-repair hardening. The agent identifies a newly added API boundary, strengthens it with type checks and a safe default for `None`, and keeps the change local to the extension point introduced by Phase 1. Despite some environment friction, the merged patch matches the Phase 2 objective.

ASTROPY FLOAT-FORMATTING INSTANCE

Issue summary. The Astropy FITS `Card` type may format floating-point values with unnecessarily long strings, which truncates comments. Phase 1 changes the core float-formatting routine to try the string representation of the value before falling back to explicit formatting. Phase 2 should therefore add only defensive checks around this path without widening the behavior of the routine itself.

Interaction summary.

1. **Follow the float-formatting trail**: The agent inspects the float-related FITS header tests and reads the card-module region that Phase 1 modified.
2. **Bootstrap tests, then widen scope**: After again spending turns on getting the test runner to work, it decides to harden both the float-formatting routine and a lower-level padding helper.
3. **Blend hardening with semantic edits**: The submitted patch adds some defensive validation, but it also expands docstrings and carries a large block of behavior-changing float-formatting logic in the same delta. The new type check in the padding helper also affects a widely reused function outside the local bug context.
4. **Submit after patch inspection**: The agent verifies that the patch excludes the temporary tests and submits the merged output.

Evaluation. This case illustrates a common Phase 2 failure mode among runs that reach submission: the model conflates a semantics-changing bug fix with behavior-preserving hardening. The artifact is a valid merged patch, but it is not a clean example of defensive refactoring, because its dominant delta remains float-formatting behavior rather than auxiliary guards. The run completes, yet the hardening signal is diluted by an unnecessarily broad semantic footprint.

DJANGO ENUM-SERIALIZATION INSTANCE

Issue summary. Django migrations serialize enum defaults by value rather than by name, which is brittle under translation. Phase 1 makes the serializer emit name-based enum literals. Phase 2 should harden the same serialization path while preserving the repaired format.

Interaction summary.

1. **Begin with verification**: The agent records the baseline diff and runs the migration-writer tests associated with the repaired serializer.
2. **Encounter a tool and expectation mismatch**: After finding that the `pytest` entry point is missing, it switches to Django's bundled test runner. These runs reveal that the replayed Phase 1 patch changes the serializer output, whereas the checked-in tests still expect the previous representation.
3. **Loop on the contradiction**: Instead of turning to safe hardening opportunities, the agent repeatedly re-reads the prompt, debates whether it may modify existing tests, and reruns the same failing checks.
4. **Exhaust the budget**: The trajectory never leaves this loop and ends when the step or cost limit is reached, without a submitted patch.

Evaluation. This is the clearest unsuccessful Phase 2 run in the slice. The model correctly detects that the replayed source patch and the checked-in test expectations disagree, but it cannot turn this diagnosis into a productive next step. The failure concerns coordination rather than search: once the verification boundary itself is inconsistent, the TDD scaffold becomes a trap, and the agent spends its entire budget without proposing a concrete refactoring.