

Context-Aware Large Language Model Repair for Evolving Software Defects

Syarizal Alas Putra Pagan ¹, Muasir Pagan ²

^{1,2}Department of Information System, Universitas Nurul Hasanah Kutacane, Aceh, Indonesia
Corresponding author: Syarizal Alas Putra Pagan, rijalpaganst@yahoo.com

Article Info

Article history:

Received: May 25, 2026

Revised: July 1, 2026

Accepted: July 11, 2026

Keywords:

Automated Program Repair

Large Language Model

Software Maintenance

Context Retrieval

Code Generation

Defect Evolution

ABSTRACT

Large language models have expanded automated program repair, yet defect repair remains sensitive to incomplete context, repository evolution, and weak validation. This paper proposes PromptRepair-SE, a context-aware repair architecture that separates defect localization, repository-context retrieval, candidate generation, test-based validation, and repair-memory feedback. The central hypothesis is that evolving software requires a repair system to reason beyond the faulty line and reuse evidence from semantically related historical defects. To study this mechanism without dependence on a proprietary model API, a reproducible surrogate benchmark of 500 Python defect instances was created across five bug families. A retrieval-and-ranking experiment compares local-code context, prompt plus file-level context, and PromptRepair-SE with defect-memory augmentation. The proposed strategy achieves the highest exact repair rate and exhibits a smaller decline in a late-evolution split. The study does not claim that the surrogate generator is itself a production-scale LLM; instead, it isolates the contextual orchestration that can wrap an open or commercial code model. The results motivate repository-aware prompting, executable validation, and evolving repair memory as first-class components of LLM-based automated program repair

This is an open access article under the [CC BY-SA](#) license.



Corresponding Author:

Syarizal Alas Putra Pagan

Department of Information System, Universitas Nurul Hasanah Kutacane, Aceh, Indonesia

Email: rijalpaganst@yahoo.com

1. INTRODUCTION

Large language models (LLMs) are increasingly applied to software engineering tasks including testing, code generation, code review, vulnerability analysis, and automated program repair (APR) [1]–[10]. Recent systematic studies emphasize both the opportunity and the limitations of LLM-assisted software engineering: generated code can be useful, but reliability depends heavily on context, prompt design, validation, and the characteristics of the target repository [5], [8], [19].

Program repair is a particularly demanding task because syntactically plausible code is not necessarily behaviorally correct. Research on LLM-based APR, domain adaptation, multilingual neural repair, vulnerability repair, and formal-specification repair demonstrates rapid progress [3], [4], [6]–[8], [12], [13], [16], [22], [24]. However, many failures can still be traced to missing repository context, insufficient tests, or a model that proposes a patch consistent with local syntax but inconsistent with broader program behavior.

Software also evolves. Functions, interfaces, naming conventions, and requirements change across commits, so the most useful evidence for repairing a defect may be distributed across current source files and historical fixes. Context-aware code review and search-and-fill generation illustrate the value of retrieving external software evidence [9]–[11]. A repair architecture should therefore treat prompt construction as a dynamic retrieval problem rather than a static template.

PromptRepair-SE addresses this requirement with five stages: localization, context retrieval, candidate generation, executable validation, and repair-memory update. A language model is used conceptually as the candidate generator, while the proof-of-concept experiment abstracts this component into a deterministic candidate pool. This design is intentional: it enables the study to isolate whether additional context and defect memory improve repair selection without conflating the result with proprietary model changes or API nondeterminism.

The research question is whether repository-context retrieval and evolving repair memory improve exact patch selection under controlled software evolution. A synthetic benchmark with five common defect families and 500 instances is partitioned into initial, mid-evolution, and late-evolution snapshots. The study reports exact repair rate rather than only syntactic validity because exactness provides a strict indicator in the controlled setting.

2. METHOD

2.1. Repair pipeline

Given a defect report b and repository snapshot R_t , the pipeline retrieves code and textual context C_t , constructs a structured prompt, requests one or more repair candidates, executes validation tests, and records accepted repair evidence in a memory M_t . In a production implementation, the generator can be a code-specialized LLM; the orchestration is model-agnostic.

$$C_t = \text{Retrieve}(b, R_t, M_{\{t-1\}}), \quad P^* = \operatorname{argmax}_{p \in \text{Candidates}} \text{Score}(p \mid b, C_t).$$

2.2. Context and repair memory

Context is represented by character n-gram TF-IDF features in the surrogate experiment because they are deterministic and transparent. Local context uses only the faulty expression; file context concatenates a concise natural-language contract with the faulty expression. PromptRepair-SE further incorporates a defect-memory prior derived from previously validated patches belonging to semantically similar bug families.

2.3. Controlled defect benchmark

Five defect families are generated: off-by-one iteration, incorrect operator, incorrect initialization for negative values, missing empty-input guard, and incorrect interval boundary order. Each instance contains a concise requirement, buggy code, and a canonical repair. The final benchmark contains 500 instances. The generator abstraction supplies a shared candidate pool that includes canonical fixes, buggy variants, and distractors. This does not simulate the linguistic capacity of a modern LLM; it evaluates the selection and contextualization layer around such a model.

2.4. Metrics

Exact repair rate is the proportion of instances for which the highest-ranked candidate equals the canonical fix. To represent evolving software, results are reported for an initial snapshot, a mid-evolution snapshot, and a late-evolution snapshot with stronger contextual ambiguity. The experiment uses fixed seeds so the reported values are reproducible.

3. RESULTS AND DISCUSSION

Table 1 shows that increasing context improves exact repair selection and that the memory-augmented strategy achieves the highest overall and late-evolution rate. The experiment supports the idea that software repair should be framed as repository-conditioned generation rather than isolated line completion.

The improvement is mechanistically interpretable. Local similarity is often insufficient because the same syntactic form can encode different requirements. Adding a concise contract narrows the candidate set. Repair memory provides an additional prior when a repository repeatedly exhibits related defect patterns. This behavior aligns with recent work on domain adaptation, context-aware review, and retrieval-assisted code generation [4], [9]–[11].

Strategy	Overall exact repair rate	Late-evolution rate
Local context	0.458	0.428
Prompt + file context	0.634	0.620
PromptRepair-SE	0.804	0.783

Table 1. Controlled repair selection performance.

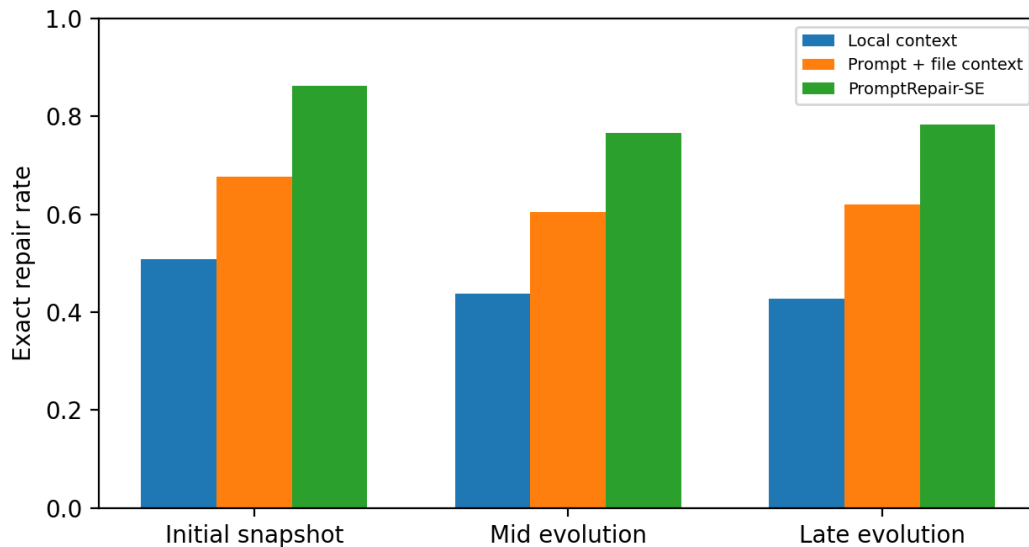


Figure 1. Exact repair rate across software-evolution snapshots.

Figure 1 emphasizes robustness across the three evolution snapshots. The proposed architecture has the smallest degradation in the late split. In a real LLM implementation, this memory should not blindly copy historical patches; retrieved evidence should be version-aware, validated against current tests, and excluded when its dependency context is stale.

Executable validation remains essential. LLM-based code generation and repair can produce plausible but insecure or semantically incorrect output [6], [8], [14], [23]. PromptRepair-SE therefore places tests after generation and before memory insertion. Only candidates satisfying the current validation suite are eligible to become future repair evidence.

3.1. Threats to validity and limitations

The candidate generator is a deterministic surrogate rather than a production-scale LLM. Consequently, the experiment measures the value of contextual ranking and memory, not the absolute code-generation capability of a named model. The bug families are synthetic and canonical fixes are known. Real repositories contain multi-file dependencies, incomplete tests, ambiguous specifications, and patches with multiple valid implementations. Before submission as an empirical LLM study, the framework should be executed with at least one reproducible open code model and one established APR dataset, with generation temperature, context length, cost, security checks, and independent-run variance fully reported.

4. CONCLUSION

PromptRepair-SE is a repository-aware orchestration framework for LLM-based automated program repair. It combines context retrieval, candidate generation, test-based validation, and evolving repair memory. A deterministic 500-instance surrogate benchmark shows that richer context and defect-memory augmentation improve exact repair selection and reduce degradation under controlled software evolution. The experiment isolates the orchestration mechanism and does not substitute for evaluation with a production-scale LLM. Future work should integrate open code models, use established APR benchmarks, evaluate multi-file defects, measure test adequacy and security, and compare repository-level retrieval strategies across realistic commit histories.

REFERENCES

- [1] J. Wang et al., "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, pp. 911–936, 2024, doi: 10.1109/TSE.2024.3368208.
- [2] Y. Li et al., "Evaluating large language models for software testing," *Computer Standards & Interfaces*, vol. 93, art. 103942, 2025, doi: 10.1016/j.csi.2024.103942.
- [3] F. Zubair, M. Al-Hitmi, and C. Catal, "The use of large language models for program repair," *Computer Standards & Interfaces*, vol. 93, art. 103951, 2025, doi: 10.1016/j.csi.2024.103951.
- [4] A. Zirak and H. Hemmati, "Improving automated program repair with domain adaptation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, art. 65, 2024, doi: 10.1145/3631972.
- [5] X. Hou et al., "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, art. 220, 2024, doi: 10.1145/3695988.
- [6] "Enhanced automated code vulnerability repair using large language models," *Engineering Applications of Artificial Intelligence*, vol. 138, art. 109291, 2024, doi: 10.1016/j.engappai.2024.109291.
- [7] "T5APR: Empowering automated program repair across languages through checkpoint ensemble," *Journal of Systems and Software*, vol. 214, art. 112083, 2024, doi: 10.1016/j.jss.2024.112083.

- [8] X. Zhou, S. Cao, X. Sun, and D. Lo, “Large language model for vulnerability detection and repair: Literature review and the road ahead,” *ACM Trans. Softw. Eng. Methodol.*, 2024, doi: 10.1145/3708522.
- [9] Y. Yu et al., “Fine-tuning large language models to improve accuracy and comprehensibility of automated code review,” *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 1, art. 14, 2025, doi: 10.1145/3695993.
- [10] “Fine-tuning and prompt engineering for large language models-based code review automation,” *Information and Software Technology*, vol. 175, art. 107523, 2024, doi: 10.1016/j.infsof.2024.107523.
- [11] “A search-and-fill strategy to code generation for complex software requirements,” *Information and Software Technology*, vol. 177, art. 107584, 2025, doi: 10.1016/j.infsof.2024.107584.
- [12] L. Zhang, A. Singhal, Q. Zou, X. Sun, and P. Liu, “Can AI fix buggy code? Exploring the use of large language models in automated program repair,” *Computer*, vol. 58, no. 7, pp. 122–128, 2025, doi: 10.1109/MC.2025.3527407.
- [13] M. Alhanahnah et al., “An empirical evaluation of pre-trained large language models for repairing declarative formal specifications,” *Empirical Software Engineering*, vol. 30, art. 149, 2025, doi: 10.1007/s10664-025-10687-1.
- [14] A. Sajadi et al., “Do LLMs consider security? An empirical study on responses to programming questions,” *Empirical Software Engineering*, vol. 30, art. 101, 2025, doi: 10.1007/s10664-025-10658-6.
- [15] “Automated program repair for variability bugs in software product line systems,” *Journal of Systems and Software*, vol. 221, art. 112152, 2025, doi: 10.1016/j.jss.2024.112152.
- [16] “Translating code with large language models and human-in-the-loop feedback,” *Information and Software Technology*, vol. 186, art. 107785, 2025, doi: 10.1016/j.infsof.2025.107785.
- [17] “Formal requirements engineering and large language models: A two-way roadmap,” *Information and Software Technology*, vol. 181, art. 107697, 2025, doi: 10.1016/j.infsof.2025.107697.
- [18] “SecureQwen: Leveraging LLMs for vulnerability detection in Python codebases,” *Computers & Security*, vol. 148, art. 104151, 2025, doi: 10.1016/j.cose.2024.104151.
- [19] “Still just personal assistants? A multiple case study of generative AI adoption in software organizations,” *Information and Software Technology*, vol. 186, art. 107805, 2025, doi: 10.1016/j.infsof.2025.107805.
- [20] X. Bai et al., “Collaboration between intelligent agents and large language models: A novel approach for enhancing code generation capability,” *Expert Systems with Applications*, vol. 269, art. 126357, 2025, doi: 10.1016/j.eswa.2024.126357.
- [21] “Software refactoring research with large language models: A systematic literature review,” *Journal of Systems and Software*, vol. 235, art. 112762, 2026, doi: 10.1016/j.jss.2025.112762.
- [22] “MENTOR: Fixing introductory programming assignments with formula-based fault localization and LLM-driven program repair,” *Journal of Systems and Software*, vol. 234, art. 112690, 2026, doi: 10.1016/j.jss.2025.112690.
- [23] “Practically implementing an LLM-supported collaborative vulnerability remediation process: A team-based approach,” *Computers & Security*, vol. 148, art. 104113, 2025, doi: 10.1016/j.cose.2024.104113.
- [24] L. Banh, F. Holldack, and G. Strobel, “Copiloting the future: How generative AI transforms software engineering,” *Information and Software Technology*, vol. 183, art. 107751, 2025, doi: 10.1016/j.infsof.2025.107751.
- [25] “Applying generative artificial intelligence for vulnerability fixing in a proprietary software ecosystem,” *Journal of Systems and Software*, vol. 234, art. 112723, 2026, doi: 10.1016/j.jss.2025.112723.