EAI Endorsed Transactions
on Internet of Things

Research Article EAI.EU

When Does Context Help? A Controlled Study of LLM-Based Bug Fixing

Dinesh Kumar Gummadavelli[1], Xianshan Qu[1*], Xiaopeng Li[1], Xin Zhang[2], Yuqi Song[2], Bokai Yang[3]

[1]University of Central Oklahoma, Edmond, OK, USA
[2]University of Southern Maine, Portland, ME, USA
[3]University of Wisconsin Eau Claire, Eau Claire, WI, USA

Abstract

Large language models (LLMs) have shown promise for automated program repair, but it remains unclear which debugging signals are most useful and when additional context becomes distracting, costly, or ineffective. We present a controlled empirical study of LLM-based bug fixing on FIXEVAL, comparing three model families–GPT, Claude, and DeepSeek–under five prompt settings: code-only repair, problem description, failed test cases, passed and failed test cases, and all available context. We further investigate whether structured execution feedback improves repair in a second attempt after an initial patch fails. Across 400 benchmark instances spanning Wrong Answer, Runtime Error, Time Limit Exceeded, and Memory Limit Exceeded bugs, we evaluate repair accuracy, response time, computation cost and second-attempt recovery. The results show that contextual information generally improves first-attempt accuracy, but its benefit depends strongly on bug type and model family. Full context is often most effective for Wrong Answer, Runtime Error, and Time Limit Exceeded bugs, whereas Memory Limit Exceeded bugs remain difficult across settings, suggesting that resource-limit failures often require deeper algorithmic redesign. Passed tests provide limited benefit unless paired with stronger semantic or failing-test signals. Structured failure feedback improves second-attempt repair most when it exposes concrete behavioral mismatches or runtime failures, but it is less effective for coarse resource-limit signals. These findings clarify when context helps LLM-based repair and provide practical guidance for designing cost-aware, feedback-driven repair pipelines.

Received on 09 September 2026; accepted on 10 September 2026; published on 22 September 2026

Keywords: Bug Fixing, Contextual Information, Large Language Models, Problem Descriptions, Test Cases, Feedback

doi:10.4108/eetiot.15050

1. Introduction

Automated program repair (APR) aims to generate patches that correct software defects with limited human intervention. Traditional APR techniques typically search over predefined edit spaces, mined repair templates, or constraint-based transformations, and their effectiveness is often bounded by the expressiveness of the underlying repair space. Large language models (LLMs) have changed this design space by generating candidate patches directly from code, natural-language requirements, and execution evidence. Early studies on Codex, ChatGPT, and other code LLMs showed that pretrained models can repair benchmark bugs without task-specific repair templates, but they also exposed persistent limitations in correctness, semantic reasoning, and reliability [1–3].

Recent work has therefore moved from single-turn patch generation toward execution-grounded and agentic repair. Benchmarks such as FIXEVAL [4] and SWE-bench [5] support more realistic execution-based evaluation by pairing buggy programs or repository issues with tests that validate generated fixes. Systems such as ChatRepair [3], SWE-agent [6], AutoCodeRover [7], ThinkRepair [8], and RepairAgent [9] further show that LLMs can benefit from tool use, repository navigation, repeated test execution, and feedback-driven refinement. These systems demonstrate the promise of

*Corresponding author. Email: xqu1@uco.edu

feedback-driven repair, but they also combine multiple mechanisms such as retrieval, planning, test execution, prompt engineering, and iterative search–making it difficult to isolate which debugging signals actually help the model produce a correct patch.

A key open question is therefore not only whether LLMs can repair bugs, but also when specific forms of context help. Human developers rarely debug from source code alone. They use problem statements, failing inputs, expected outputs, passing examples, stack traces, and feedback from previous attempts. LLMs may similarly benefit from these signals, yet additional context does not necessarily improve repair. More information can introduce irrelevant details, increase latency and cost, and distract the model from the most diagnostic evidence. Moreover, different bug types may require different signals: a Wrong Answer bug may be explained by a failing input-output pair, whereas a Memory Limit Exceeded bug may require algorithmic redesign that is not obvious from a small number of examples.

This paper presents a controlled empirical study of contextual signals and feedback refinement for LLM-based bug fixing. We evaluate three model families–GPT, Claude, and DeepSeek under a shared repair framework on FIXEVAL [4]. We compare five prompt settings: a basic code-only prompt, problem description, failed test cases, passed and failed test cases, and all contextual signals combined. We then examine a two-attempt setting in which an initially failed patch is revised using structured execution feedback containing the failure type, failing input, expected output, and observed output. This design allows us to separate the effect of initial context from the effect of feedback-based refinement.

Our study is guided by three research questions:

- RQ1. What is the value of different contextual signals for LLM-based bug fixing?
- RQ2. How does the usefulness of context vary across model families and bug types?
- RQ3. To what extent does structured failure feedback improve second-attempt repair accuracy?

The paper makes three contributions. First, it provides a controlled comparison of problem descriptions, failing tests, passing tests, and combined context for LLM-based bug fixing across multiple model families and bug categories. Second, it analyzes the trade-offs among repair accuracy, response time, and computation cost, showing when richer prompts justify their overhead. Third, it evaluates structured failure feedback in a fixed two-attempt repair setting, clarifying when execution feedback supports convergence and when it remains insufficient.

Prior work has examined contextual signals or feedback mechanisms individually. However, many studies focus on a single model family [10–13]. Others [3, 7, 9] combine feedback with additional mechanisms, such as planning and tool use, making it difficult to isolate the effect of feedback. This study is, to our knowledge, the first to jointly compare context type, model family, and bug category within a single controlled framework. This joint comparison allows us to observe interactions, such as which context types are most useful for which bug types, that single-model or single-context studies cannot reveal.

2. Background

LLM-based bug fixing differs from traditional automated program repair in that the repair process is largely prompt-driven. Instead of searching a constrained patch space, the model generates a candidate fix based on the information provided in the prompt. As a result, the quality and type of contextual information can strongly influence repair behavior.

2.1. Contextual Signals for Bug Fixing

Contextual signals describe information beyond the buggy code that may help the model infer the intended behavior and identify the cause of failure. In this study, we consider three common types of contextual signals: problem descriptions, failed test cases, and passed test cases.

A problem description explains the intended behavior, input-output format, and constraints. It can help the model infer the correct algorithm, especially when the bug is difficult to identify from code alone. Failed test cases provide negative behavioral evidence by showing inputs on which the program fails, along with expected and observed outputs. They can directly expose incorrect behavior, but may be less informative for resource-limit bugs when the feedback only indicates timeout or memory exhaustion. Passed test cases provide positive behavioral evidence by showing behavior that should be preserved. They may help avoid regressions, but they usually do not identify the fault directly.

These signals can be used separately or together. Richer prompts may provide more complete information, but they can also add noise, latency, and cost. Thus, full context is not necessarily better than selective context.

2.2. Iterative Repair and Failure Feedback

Bug fixing is often iterative. After generating a patch, developers run tests, inspect failures, and revise the code. LLM-based repair can follow the same process: if an initial patch fails validation, the model can use failure feedback to generate a revised patch.

Failure feedback may be provided as raw logs or as structured summaries. Raw logs can be lengthy and inconsistent, while structured feedback highlights the key information, such as failure type, failing input, expected output, and observed output.

In this study, we use a two-attempt design. The first attempt evaluates the effect of contextual signals. If the patch fails, the second attempt provides structured failure feedback together with the original context. This design allows us to examine both initial repair performance and feedback-based refinement under a fixed repair budget.

3. Experimental Setup

3.1. Dataset

We use the FIXEVAL[4] dataset, derived from CodeNet[14], which contains competitive programming submissions from AtCoder and Aizu Online Judge. The CodeNet dataset compiles programs in over 50 languages, and FIXEVAL augments these programs with open-source unit tests. FIXEVAL is well suited for our study because it supports controlled sampling at scale. It also includes more complex bugs than datasets such as QuixBugs[15], including multi-line logic errors.

We evaluate four bug categories: Wrong Answer, Memory Limit Exceeded, Time Limit Exceeded, and Runtime Error. For each category, we randomly sample 100 instances, resulting in a total of 400 evaluation instances. Each instance contains the buggy program, problem description, and test cases, which we use to construct different prompt contexts.

3.2. Models

We evaluate three representative LLM families using GPT-5, Claude Sonnet 4.5, and DeepSeek-Coder. For reproducibility, all model calls were made through the corresponding provider APIs using fixed model identifiers: GPT-5 via the OpenAI API, Claude Sonnet 4.5 via the Anthropic API, and DeepSeek-Coder via the DeepSeek API. All models are evaluated on the same benchmark instances, prompt settings, validation procedure, and repair budget. Model-specific changes are limited to API compatibility. Each model is allowed at most two repair attempts per instance. The first attempt evaluates the effect of different contextual signals. If the generated patch fails validation, the model receives one additional attempt with structured failure feedback. This second attempt is used to evaluate the effect of that feedback.

For all experiments, we configure the decoding parameters of each model to ensure deterministic and reproducible outputs. For Claude Sonnet 4.5 and DeepSeek Coder, temperature is set to 0 to suppress randomness and produce consistent repairs across runs. GPT-5 does not expose a temperature parameter in the same way and is used with its default decoding configuration. The maximum output token limit is set to 4,096 for all three models, which is sufficient to accommodate complete corrected code for the programming problems in our dataset. No explicit stop conditions or top-p sampling constraints are applied beyond the default API behavior. All models are prompted using the chat completion interface with a fixed system prompt and a single user message per repair attempt, with no multi-turn conversation history maintained between attempts.

3.3. Evaluation Metrics

We evaluate repair effectiveness using execution-based correctness: a patch is considered successful if the modified program passes all evaluation tests. In addition, we report response time per problem, computation cost per problem, and second-attempt recovery accuracy. For each problem, cost is calculated as (input tokens × input token rate) + (output tokens × output token rate), using each provider's published pricing for the corresponding model at the time of experimentation. Together, these metrics allow us to assess both repair effectiveness and practical efficiency.

3.4. Contextual Prompt Settings

We study five prompt settings that vary the amount and type of context provided to the model. All settings use the same system instruction as follows:

```
You help in correcting code. Always correct the
code and return it strictly in this JSON format:
{"code": "x", "explanation": "y"} in a single
line. Ensure valid JSON with properly escaped
quotes. Do not include anything outside of the
JSON object.
Keep the explanation concise. The code should be
a complete, corrected version of the input code.
```

Basic Prompt (BP). This setting assesses the repair ability using a straightforward prompt that contains only the buggy code snippet and a general repair instruction, without any additional contextual guidance. This serves as the baseline condition and allows us to measure how well the model can identify defects and infer the required repair from code alone.

Basic Prompt with Problem Description (PD). To examine whether semantic information improves debugging effectiveness, we augment the user prompt with a detailed problem description while keeping the system prompt unchanged. The user prompt is extended as follows:

```
Correct this code {code}.
The code is supposed to do this: {description}.
```

Here, {code} is the buggy code snippet and {description} is the problem statement retrieved from AtCoder, including constraints and input/output formats.

Basic Prompt with Failed Test Cases (FT). Because failing tests often reveal the exact behavior that is incorrect, we study how the model performs when supplied with failed test cases and their corresponding outputs. This setting includes the specific input, expected output, and actual output for each failing test case, along with the reason for failure such as wrong answer, runtime error, timeout, or memory limit exceeded. The user prompt is as follows:

```
Correct this code {code}.
Failing test cases (input/expected/actual):
Case 1 - reason: {reason}, Input: {input},
Expected: {expected}, Actual: {actual}.
```

Here, {reason} is the failure type, and {input}, {expected}, and {actual} are the test case details. To maintain consistency and avoid overwhelming the models, we include at most three failing test cases for each task. Including all available test cases was impractical because it would substantially increase prompt length, inference cost, and response time.

Basic Prompt with Passed and Failed Test Cases (PFT). In addition to failed tests, human developers often rely on passed tests to understand which program behaviors are already correct and should be preserved. To reflect this perspective, we evaluate the model under a setting where the prompt is augmented with both passing and failing test cases. The user prompt is as follows:

```
Correct this code {code}.
Passed test cases (examples of correct
behavior):
Passed Case 1 - Input: {input}, Expected:
[expected].
Failed test cases (input/expected/actual):
Failed Case 1 - reason: {reason}, Input:
{input}, Expected: {expected}, Actual: {actual}.
```

The passing cases serve as behavioral constraints that help the model preserve already-correct behavior while fixing the identified bugs. Because of token limitations, and to maintain consistency across tasks and models, we include up to two passing test cases and three failing test cases for each task.

Basic Prompt with Problem Description and Test Cases (ALL). In our final contextual setting, we combine all sources of information used in the study: the problem description, failing test cases, and passing test cases. The purpose of this setting is to determine whether a richer, more comprehensive context yields a synergistic effect that improves bug-fixing performance, or whether too much information introduces noise and reduces the model's ability to focus on the most relevant debugging cues. The user prompt is extended as follows:

```
Correct this code {code}.
The code is supposed to do this: {description}
Passed test cases (examples of correct
behavior):
Passed Case 1 - Input: {input}, Expected:
[expected].
Failed test cases (input/expected/actual):
Failed Case 1 - reason: {reason}, Input:
{input}, Expected: {expected}, Actual: {actual}.
```

Here, the test case fields follow the same format as the FT and PFT settings. Up to two passing test cases and up to three failing test cases are included per task alongside the full problem description. This setting represents the maximum context condition and serves as the upper bound for contextual augmentation in our evaluation.

3.5. Failure Feedback Design

To evaluate whether execution feedback helps models recover from an unsuccessful repair, we use a fixed two-attempt protocol. In the first attempt, the model receives the Problem Description (PD) prompt and generates a complete corrected program. The generated program is then compiled and executed against the validation tests. If it passes all tests, the instance is counted as repaired and no second attempt is made. If the first attempt fails, the model is given one additional opportunity to revise its patch using failure feedback derived from the test execution results of the failed attempt. The Round 1 corrected code is used as input for Round 2, along with the original problem description and the collected failure information. This second attempt is designed to measure whether the model can use feedback to improve an unsuccessful repair.

We select Problem Description (PD) instead of Basic Prompt (BP) as the feedback-stage baseline for two reasons. First, BP provides little information about the program's intended behavior, making it hard to tell whether improvements come from failure feedback or from the model learning the task in the first attempt. Using PD avoids this problem because the model already has access to the problem description in both attempts, so any improvement in the second attempt can be more directly attributed to the failure feedback itself. Second, although the Basic Prompt with Problem Description and Test Cases (ALL) setting sometimes achieves higher accuracy than PD, as shown in Table 1, it does so at a noticeably higher cost (Table 2). In several cases, PD and ALL produce similar accuracy,

such as GPT-5 on Wrong Answer (83% vs. 85%) and DeepSeek on Memory Limit Exceeded (11% vs. 11%), while in other cases ALL performs meaningfully better, such as DeepSeek on Time Limit Exceeded and Claude on Wrong Answer. We therefore use PD as a practical baseline that balances cost and accuracy, and we acknowledge this as a deliberate trade-off in our experimental design rather than claiming PD is optimal in every case.

The failure feedback is structured into a compact format containing the failure type, the specific test case input, the expected output, and the actual output produced by the failed patch. In this stage, we include at most three representative failing test cases per task because of token limitations. An example of the structured failure feedback is shown below:

```
Status: Failed
Failure Type: Wrong answer
Affected Test Case: These are test cases where
    the program still performs incorrectly.
Expected Output: [correct output]
Observed Output: [incorrect output]
Note: Ensure previously correct behaviors remain
    unchanged
```

4. Results and Analysis

This section presents the empirical findings of the study in two stages. First, we study how contextual signals affect first-attempt repair accuracy, response time, and cost. Second, we evaluate whether structured failure feedback improves repair accuracy on a second attempt.

4.1. Contextual Signals: Accuracy, Response Time, and Cost

We first evaluate how the five contextual prompt settings affect first-attempt bug-fixing performance. For each bug type, we report the repair success rate (accuracy) for the three model families considered in this study: GPT, Claude, and DeepSeek. Table 1 summarizes these first-attempt accuracy results across models, bug types, and contextual settings. Overall, contextual information improves repair accuracy compared with code-only prompting, but the benefit depends strongly on the type of context, the model family, and the bug category.

Across all models and bug types, the all-context setting achieves the highest average first-attempt accuracy. Averaged over the four bug categories, GPT-5 improves from 42.75% under the basic prompt to 62.00% under the all-context setting, Claude Sonnet 4.5 improves from 19.25% to 50.25%, and DeepSeek-Coder improves from 24.75% to 46.00%. These results suggest that combining semantic information from the problem description with behavioral evidence from test cases provides complementary repair signals. The improvement is especially visible for Wrong Answer, Time Limit Exceeded, and Runtime Error bugs, where the all-context setting usually provides the best or near-best performance.

Table 1. First-attempt repair accuracy (%) across models, bug types, and context settings

Model	Bug	BP	PD	FT	PFT	ALL
GPT-5	Wrong Answer	70	83	82	80	**85**
	Memory Limit Exceeded	8	**19**	10	10	13
	Time Limit Exceeded	43	56	54	55	**64**
	Runtime Error	50	**76**	67	64	86
Claude-sonnet-4-5	Wrong Answer	28	62	57	55	**81**
	Memory Limit Exceeded	5	**15**	14	13	12
	Time Limit Exceeded	10	37	22	22	**45**
	Runtime Error	34	60	36	33	**63**
Deepseek-coder	Wrong Answer	42	67	64	62	**77**
	Memory Limit Exceeded	9	11	**16**	15	11
	Time Limit Exceeded	26	32	36	39	**56**
	Runtime Error	22	35	28	32	**40**

However, the results also show that more context is not universally better. Memory Limit Exceeded is the clearest exception. For this category, accuracy remains low across all models and prompt settings. GPT-5 performs best with the problem description rather than all context, achieving 19% under the problem-description setting compared with 13% under all context. Claude Sonnet 4.5 also performs best with the problem description, reaching 15% compared with 12% under all context. DeepSeek-Coder performs best with failed test cases, reaching 16%, while all context drops to 11%. These results suggest that memory-limit bugs are less amenable to example-guided repair because they often require algorithmic redesign, data-structure replacement, or complexity reduction rather than localized correction of output behavior.

The usefulness of each context type also depends on the model. For GPT, the problem description alone is already strong for Wrong Answer and Runtime Error bugs, with ALL providing smaller additional gains. Claude benefits substantially from richer context, especially for Wrong Answer bugs. DeepSeek also improves under ALL for most categories, but for Memory Limit Exceeded, failed test cases alone perform better than the full context. These results show that context effectiveness is both model-specific and bug-type-specific.

Passing test cases provide limited additional benefit when used without the problem description. In several cases, PFT performs similarly to or worse than FT, suggesting that passing examples may introduce extra information without clearly identifying the repair target. Passing tests appear most useful when combined with the problem description and failing cases in the ALL setting.

Table 2. Average response time and computation cost per problem across context settings

(a) Average response time (s)

Model	BP	PD	FT	PFT	ALL
GPT-5	17.85	**16.42**	16.87	21.47	24.34
Claude-sonnet-4-5	**14.22**	15.05	16.55	15.74	15.25
Deepseek-coder	**15.12**	16.56	15.99	16.62	17.09

(b) Average computation cost (¢)

Model	BP	PD	FT	PFT	ALL
GPT-5	**1.98**	2.50	2.15	2.37	2.63
Claude-sonnet-4-5	**0.65**	2.87	1.24	1.44	3.49
Deepseek-coder	**0.05**	0.23	0.08	0.09	0.27

Table 3. First- and second-attempt repair accuracy (%) under the Problem Description (**PD**) setting

Model	Bug	1st Attempt	2nd Attempt
GPT-5	Wrong Answer	83	**85**
	Memory Limit Exceeded	19	19
	Time Limit Exceeded	56	**64**
	Runtime Error	76	**82**
Claude-sonnet-4-5	Wrong Answer	62	**81**
	Memory Limit Exceeded	15	**19**
	Time Limit Exceeded	37	**44**
	Runtime Error	60	**65**
Deepseek-coder	Wrong Answer	67	**71**
	Memory Limit Exceeded	11	11
	Time Limit Exceeded	32	**41**
	Runtime Error	35	**44**

Table 2a reports response time, and Table 2b reports computation cost. The per-problem cost is averaged across all evaluated instances within each context setting to produce the values reported in Table 2b.

GPT shows the clearest latency increase under the ALL setting, while Claude and DeepSeek remain relatively stable across context settings. Cost varies by model and prompt setting, but it is not strictly monotonic with context size. Overall, the efficiency results suggest that richer context can improve accuracy, but the gain should be weighed against latency and cost, especially for large-scale automated repair pipelines.

4.2. Failure Feedback and convergence

We next evaluate the effect of failure feedback on the second repair attempt. Table 3 reports first- and second-attempt accuracy after adding structured failure feedback under the Problem Description (PD) setting. PD setting is selected as described in Section 3.5.

Failure feedback improves repair accuracy in most model-category pairs, but the gains are uneven. The largest improvement occurs for Claude on Wrong Answer bugs, increasing from 62% to 81%. Time Limit Exceeded and Runtime Error also improve consistently across all three models, suggesting that execution feedback is especially useful when the failure exposes a concrete runtime behavior or performance issue.

Memory Limit Exceeded remains the least responsive category. GPT and DeepSeek show no improvement, while Claude improves modestly from 15% to 19%. This supports the first-attempt finding that memory-limit bugs are difficult to repair using limited contextual or execution feedback, likely because they often require deeper algorithmic changes.

The results also show that second-attempt gains depend on first-attempt accuracy. GPT starts with the strongest first-attempt performance and therefore has fewer remaining recoverable cases, leading to smaller absolute gains. Claude begins lower but benefits more from feedback, particularly on Wrong Answer bugs. DeepSeek shows meaningful gains on Time Limit Exceeded and Runtime Error bugs, but smaller gains on Wrong Answer bugs and none on Memory Limit Exceeded bugs.

Overall, structured failure feedback helps models refine unsuccessful patches, especially when the feedback identifies a concrete mismatch between expected and observed behavior. However, feedback is less effective when the failure signal is too coarse, as in Memory Limit Exceeded, where knowing that memory was exceeded does not indicate which data structure or algorithmic choice should be changed.

5. Related Work

Prior work on LLM-based software engineering has explored code review, code refinement, and automated program repair. However, existing studies have mostly evaluated whether LLMs can repair bugs under a single prompt setting or benchmark, while less attention has been paid to how different contextual signals and feedback mechanisms affect repair performance across models and bug types.

5.1. LLMs for Code Review and Refinement

Several studies have examined the ability of LLMs to review, refine, and improve code [7, 10–12, 16–18]. Fu et al.[10] evaluated ChatGPT on software vulnerability detection, classification, and repair, finding that the model struggled to reliably identify vulnerabilities and generate correct patches. Ribeiro et al. [11] investigated the use of GPT-3 for debugging OCaml programs, while Guo et al. [12] studied ChatGPT for code refinement tasks and showed that prompt design can substantially affect performance. Other work has explored GPT-based support for programming education. For example, Crandall et al.[17] applied GPT

to automated code review for student programming assignments, and Jalil et al. [18] examined ChatGPT's usefulness in software testing education.

These studies show that LLMs can assist with code understanding, review, and refinement, but they also highlight limitations in reliability and correctness. In contrast, our work focuses specifically on executable bug fixing and studies how different forms of repair context affect model behavior.

5.2. LLMs for Automated Program Repair

LLMs have also been increasingly studied for automated program repair. Early work evaluated ChatGPT and related models on QuixBugs [15], a small benchmark of 40 buggy programs. Wuisang et al. [19] and Sobania et al. [2] found that LLM-based repair can outperform traditional Automated Program Repair (APR) baselines [20] on this benchmark. Additional studies evaluated ChatGPT and other LLMs for bug fixing, localization, and patch generation [1, 13, 21].

More recent work has moved beyond single-turn repair toward more autonomous repair workflows. Zhang et al. [7] proposed AutoCodeRover for autonomous program improvement. Chandramohan et al. [22] explored the use of multiple tools in LLM-based repair. Bouzenia et al. [9] introduced RepairAgent, an autonomous LLM-based repair agent, while Xia et al. [3] studied conversational program repair. Other work has investigated self-directed automated repair [8], agent-based repair on SWE-bench [23], cost-efficient repair with LLMs [24], and the role of semantic reasoning in program repair [25].

These studies demonstrate the promise of LLMs for program repair, especially when models can interact with external tools, tests, or feedback. However, many evaluations either focus on a single model family, a single prompt design, or a limited benchmark setting.

5.3. Context and Feedback in LLM-Based Repair

Context and execution feedback are central to LLM-based repair. Prior studies show that prompt design can substantially affect code refinement and debugging performance, and that the same model may behave differently depending on whether it receives only code, natural-language intent, test cases, or execution errors [12, 13]. However, many studies evaluate either one specific type of error or a full repair system, which makes it difficult to identify the marginal contribution of individual context types for different bugs.

Feedback-driven repair has been explored in conversational and agentic systems. ChatRepair uses a dialogue style loop in which test failures and previous repair attempts guide subsequent patch generation [3]. SWE-agent emphasizes the role of agent-computer interfaces that allow models to inspect files, edit code, and execute commands [6]. AutoCodeRover combines LLMs with code search to localize and modify relevant program locations [7], while RepairAgent studies autonomous LLM-based repair over established APR benchmarks [9]. Recent refinement work also shows that deciding which candidate patch to refine involves an exploration-exploitation trade-off when test feedback is available [26].

These systems demonstrate that feedback and interaction can improve repair, but they typically combine multiple mechanisms, including retrieval, localization, planning, tool use, and repeated validation. In contrast, our study isolates a smaller set of debugging signals under a fixed repair budget. This controlled design allows us to compare problem descriptions, failing tests, passing tests, full context, and structured failure feedback across model families and bug types.

6. Limitations

This study has several limitations. First, our evaluation is based on FIXEVAL [4], which contains competitive programming tasks. Although this benchmark provides executable tests and diverse bug categories, its programs are generally smaller and more self-contained than real-world software systems. As a result, the findings may not fully generalize to large codebases that involve multiple files, dependencies, build systems, or complex runtime environments. However, the qualitative patterns we observe, such as the limited benefit of additional context for resource-limited bugs and the varying effectiveness of structured feedback on signal specificity, are properties of how language models use contextual information rather than properties specific to competitive programming. We therefore expect these patterns to extend to other software repair settings, although this remains to be validated on real-world codebases.

Second, we evaluate a fixed set of contextual signals: problem descriptions, failed test cases, and passed test cases. Other forms of debugging information, such as stack traces, compiler diagnostics, code comments, commit history, issue reports, or repository-level context, may affect repair performance differently. Our results therefore characterize the studied context types rather than all possible repair signals.

Third, the study uses a fixed prompt structure and a two-attempt repair budget. Different prompt templates, feedback formats, sampling parameters, or additional repair attempts could lead to different results. In particular, more interactive or agent-based repair systems may benefit from repeated execution, tool use, or search strategies that are outside the scope of this controlled setup.

Fourth, using only a subset of failing and passing test cases in the prompts may increase the risk of

patch overfitting, where a model fixes the observed failures but fails to generalize to unobserved cases. This risk is mitigated by our evaluation protocol, which requires each generated patch to pass all available tests; overfitted patches are therefore counted as unsuccessful. At the same time, providing only a subset of test cases may limit the amount of behavioral evidence available to the model, potentially underestimating the effectiveness of the corresponding context settings. Nevertheless, because the same test-selection strategy and evaluation protocol are applied consistently across all models and prompt settings, the comparative trends reported in this study remain valid.

7. Conclusion

In this study, we present a controlled empirical study of contextual signals and feedback refinement for LLM-based bug fixing. We compared three model families—GPT, Claude, and DeepSeek—on FIXEVAL [4] using five prompt settings: basic code-only prompting, problem description, failed test cases, passed and failed test cases, and all context combined. We further evaluated whether structured failure feedback improves repair accuracy in a second attempt.

Our results show that contextual information generally improves first-attempt repair accuracy, but its effectiveness depends on both the model family and bug type. Full context often performs well, especially for Wrong Answer, Time Limit Exceeded, and Runtime Error bugs, but it is not universally optimal. Memory Limit Exceeded remains challenging across models, suggesting that resource-limit bugs often require deeper algorithmic changes that are not easily inferred from problem descriptions or test examples alone.

The results also show that structured failure feedback can improve second-attempt repair accuracy, particularly when the feedback exposes concrete behavioral mismatches or runtime failures. However, the gains are uneven, and feedback is less effective when the failure signal is too coarse, as in Memory Limit Exceeded cases.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 2451420.

References

[1] Prenner JA, Babii H, Robbes R. Can OpenAI's Codex fix bugs? an evaluation on QuixBugs. In: IEEE/ACM International Workshop on Automated Program Repair (APR); 2022. p. 69-75.

[2] Sobania D, Briesch M, Hanna C, Petke J. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In: IEEE/ACM International Workshop on Automated Program Repair (APR); 2023. p. 23-30.

[3] Xia CS, Zhang L. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis; 2024. p. 819-31.

[4] Haque MMA, Ahmad WU, Lourentzou I, Brown C. FixEval: Execution-based Evaluation of Program Fixes for Programming Problems. In: IEEE/ACM International Workshop on Automated Program Repair (APR); 2023. p. 11-8.

[5] Jimenez CE, Yang J, Wettig A, Yao S, Pei K, Press O, et al. Swe-bench: Can language models resolve real-world github issues? arXiv preprint arXiv:231006770. 2023.

[6] Yang J, Jimenez CE, Wettig A, Lieret K, Yao S, Narasimhan K, et al. Swe-agent: Agent-computer interfaces enable automated software engineering. Advances in Neural Information Processing Systems. 2024;37:50528-652.

[7] Zhang Y, Ruan H, Fan Z, Roychoudhury A. Autocoderover: Autonomous program improvement. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis; 2024. p. 1592-604.

[8] Yin X, Ni C, Wang S, Li Z, Zeng L, Yang X. Thinkrepair: Self-directed automated program repair. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis; 2024. p. 1274-86.

[9] Bouzenia I, Devanbu P, Pradel M. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In: Proceedings of the IEEE/ACM 47th International Conference on Software Engineering. ICSE '25. IEEE Press; 2025. p. 2188–2200. Available from: https://doi.org/10.1109/ICSE55347.2025.00157.

[10] Fu M, Tantithamthavorn C, Nguyen V, Le T. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? In: Asia-Pacific Software Engineering Conference (APSEC); 2023. .

[11] Ribeiro F, de Macedo JNC, Tsushima K, Abreu R, Saraiva J. GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair. In: Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering; 2023. p. 111-24.

[12] Guo Q, Cao J, Xie X, Liu S, Li X, Chen B, et al. Exploring the potential of ChatGPT in automated code refinement: An empirical study. In: 46th IEEE/ACM International Conference on Software Engineering; 2024. p. 1-13.

[13] Qu X, Zuo F, Li X, Rhee J. Context matters: Investigating its impact on ChatGPT's bug fixing performance. In: 2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA). IEEE; 2024. p. 17-23.

[14] Puri R, Kung DS, Janssen G, Zhang W, Domeniconi G, Zolotov V, et al.. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks; 2021.

[15] Lin D, Koppel J, Chen A, Solar-Lezama A. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In: Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications:

software for humanity; 2017. p. 55-6.
[16] Zuo F, Qian G, Qu X, Rhee J, Fu J. Revisiting the Capability of GPT in Solving Coding Problems: A Lesson from Programming with Recursion. In: Proceedings of the 26th ACM Annual Conference on Cybersecurity & Information Technology Education; 2025. p. 134-40.
[17] Crandall AS, Sprint G, Fischer B. Generative Pre-Trained Transformer (GPT) Models as a Code Review Feedback Tool in Computer Science Programs. Journal of Computing Sciences in Colleges. 2023;39(1):38-47.
[18] Jalil S, Rafi S, LaToza TD, Moran K, Lam W. ChatGPT and software testing education: Promises & perils. In: IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW); 2023. p. 4130-7.
[19] Wuisang MC, Kurniawan M, Santosa KAW, Gunawan AAS, Saputra KE. An Evaluation of the Effectiveness of OpenAI's ChatGPT for Automated Python Program Bug Fixing using QuixBugs. In: 2023 International Seminar on Application for Technology of Information and Communication (iSemantic); 2023. p. 295-300.
[20] Ye H, Martinez M, Durieux T, Monperrus M. A comprehensive study of automatic program repair on the QuixBugs benchmark. Journal of Systems and Software. 2021;171:110825.
[21] Do Viet T, Markov K. Using Large Language Models for Bug Localization and Fixing. In: 12th International Conference on Awareness Science and Technology (iCAST); 2023. p. 192-7.
[22] Chandramohan M, Jancic J, Zhang Y, Krishnan P. From Benchmark Data To Applicable Program Repair: An Experience Report. arXiv preprint arXiv:250816071. 2025.
[23] Rondon P, Wei R, Cambronero J, Cito J, Sun A, Sanyam S, et al. Evaluating agent-based program repair at google. In: 2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE; 2025. p. 365-76.
[24] Hidvégi D, Etemadi K, Bobadilla S, Monperrus M. Cigar: Cost-efficient program repair with llms. arXiv preprint arXiv:240206598. 2024.
[25] Le-Cong T, Le B, Murray T. Can LLMs reason about program semantics? a comprehensive evaluation of LLMs on formal specification inference. In: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers); 2025. p. 21991-2014.
[26] Tang H, Hu K, Zhou JP, Zhong S, Zheng WL, Si X, et al. Code repair with llms gives an exploration-exploitation tradeoff. Advances in Neural Information Processing Systems. 2024;37:117954-96.