On the Role of Context Granularity in LLM-Driven Program Repair

Tyler Holloway*
Department of Computer Science
Harvard University
Cambridge, MA
tylerholloway@g.harvard.edu

Ethan R. Elenberg
Permanence AI
New York, NY
ethan@permanence.ai

Abstract

Recent advances in Large Language Models (LLMs) have created new opportunities for scalable Automated Program Repair (APR). However, an underexplored aspect of APR is the impact of the surrounding code on patch correctness. We propose a novel context granularity based on backward static slicing, capturing lines on which the buggy line is data- and control-dependent. We then evaluate its performance against five commonly used APR context granularities as well as state-of-the-art APR systems. Using GPT-4, we assess all six context granularities on 109 single-line bugs from the Defects4J dataset. Our results show that the sliced context achieves the highest Correct/Plausible ratio (79%) in the dataset, suggesting that a more focused context improves the generation of semantically accurate patches per passed test case. In contrast, larger contexts, such as entire files, produce more correct patches overall but at a lower ratio. We propose that future work explore combining smaller, focused contexts like slicing with larger ones to enhance both semantic accuracy and the total number of correct patches, as well as investigating context granularity strategies tailored to specific bug types.

1 Introduction

Software systems are pervasive, underpinning critical infrastructure across various industries. However, these systems remain highly vulnerable to bugs, as demonstrated by software-related failures such as the Ariane 5 crash [2], Toyota's unintended acceleration incidents [10], and the Knight Capital Group trading glitch [21]. The economic impact of these vulnerabilities is also staggering. In 2020, software defects cost the U.S. economy $2.08 trillion, rising to $2.41 trillion by 2022, according to the Consortium for Information & Software Quality (CISQ) [4]. As systems become more complex, Automated Program Repair (APR) has emerged as a promising solution for automatically fixing bugs, reducing manual effort, and minimizing downtime.

Recent advancements in machine learning, including Large Language Models (LLMs) such as StarCoder [13] and AST-T5 [5], have introduced scalable approaches to APR by leveraging historical code changes and large-scale datasets [7, 24, 17]. However, a crucial yet underexplored aspect of both traditional and learning-based APR methods is the role of context—specifically, the code surrounding a buggy line—in the success of generated patches [22].

Studies in natural language processing (NLP) highlight the crucial role of context in the performance of LLMs across various tasks. Liu et al. [16] show that model performance may degrade when key information appears in the middle of lengthy contexts. Similarly, Ettinger et al. [19] find that irrelevant details can negatively impact predictions, as models often rely on semantic similarity and

*Work done while at Permanence AI.

Machine Learning for Systems Workshop at NeurIPS 2024.

word positioning. Despite these insights, how different context granularities impact LLM performance in APR tasks remains an open question. Here, we define *context granularity* as the amount of code surrounding the buggy line that is provided to the repair system.

In this paper, we propose a novel context granularity based on backward static slicing [23] and evaluate its performance in generating single-line patches using GPT-4 [18]. Our study compares this approach with five traditional context granularities, using the widely adopted Defects4J dataset [8], as follows:

- `Single-line`: the buggy line.
- `Fixed`: the five lines above and below the buggy line.
- `Method`: the enclosing method of the buggy line.
- `Class`: the enclosing class of the buggy line.
- `Sliced`: the lines that are data- and control-dependent on the buggy line, obtained through backward static slicing [23].
- `File`: the entire file that contains the buggy line.

Our findings reveal that while the `File` context generates the highest number of correct patches, this increase is minimal compared to the substantial increase in context size. More focused contexts, such as `Method` and `Sliced`, achieve higher semantic accuracy per passed test case, as indicated by the ratio of Correct patches to Plausible patches. Notably, the `Sliced` context achieves the highest Correct/Plausible ratio across all evaluated contexts, even surpassing state-of-the-art systems like AlphaRepair [24].

Contributions. This paper makes the following contributions:

1. We propose a novel context granularity based on backward static slicing, which retains only the lines on which the buggy line is data- and control-dependent.
2. We systematically investigate the impact of 6 different context granularities, including the proposed slicing granularity, on the performance of LLMs in APR tasks.
3. We compare the performance of our slicing granularity and the other five context settings when used with GPT-4 against state-of-the-art APR techniques.

2 Background

Automated Program Repair is a subfield of software engineering focused on fixing software bugs with minimal human intervention. Recent advancements have redefined program repair as a Neural Machine Translation task, in which frameworks such as CURE [7] and CoCoNut [17] "translate" buggy code into corrected code by learning bug-fixing patterns from large code corpora. Building on this, more recent approaches, including AlphaRepair [24], leverage large language models, such as GPT and CodeBERT [3], in a zero-shot learning setting. This setup enables patch generation without extensive fine-tuning.

In contrast, traditional APR methods often rely on rule-based or heuristic-driven strategies to generate patches. For instance, TBar [15] and AVATAR [14] use template-based repair, applying common bug-fixing patterns derived from previous patches. FixMiner [11] builds on this by analyzing large datasets of historical bug fixes to guide patch generation. GenProg [12] takes a different approach, using genetic programming to explore fixes through mutation and crossover operations. While these methods can produce valid patches, they often depend on templates, heuristics, or exhaustive searches, which can limit their flexibility in addressing new or complex bugs.

The works most closely related to ours are [22] and [20]. Katana [22] applies control and data flow analysis to extract context through dual slicing, converts these slices into AST-based graphs, and employs a Graph Neural Network (GNN) to train an APR model. However, it does not systematically explore how different context granularities impact LLM performance—a gap our study addresses. The experiments in [20] examine the role of local context in repair success, investigating factors such as context length and its placement around the bug by training and evaluating various Transformer-based models. In contrast, our study systematically examines a broader range of context granularities, from

single-line to full-file. Additionally, we analyze the impact of context on repair success specifically within the zero-shot setting of APR using LLMs.

2.1 Backward Static Slicing

Backward static slicing is a program analysis technique that identifies all statements in a program that could influence the value of a specific variable at a given point by tracing control and data dependencies backward through the program's control flow graph (CFG). Our implementation follows the slicing algorithm introduced by Weiser [23], who likened program slicing to the mental process programmers use when debugging code. This process begins at the buggy line and traverses backward through the control flow graph to locate statements that directly or indirectly affect relevant variables or execution paths. By isolating these critical dependencies, backward static slicing can greatly simplify the program. Additional slicing techniques that may be used to create new context granularities are detailed in Appendix B.

3 Methodology

In this study, we systematically evaluate the impact of different context granularities on the success of single-line patches generated by an LLM for Automated Program Repair. Our approach formulates the problem as follows: given a buggy line with a specified context—a token sequence consisting of the buggy line and surrounding lines—generate an output token sequence that aligns with a ground truth patch, thereby converting previously failing tests to passing.

We define context granularity as the amount of code surrounding a buggy line that is provided to the repair system. As previously described, we evaluate six context granularities: `Single-line`, `Fixed`, `Method`, `Class`, `Sliced`, and `File`. AST parsing using Tree-sitter is employed for context extraction in all granularities except for `File`, which includes all lines in the file.[2]

We evaluate each context granularity using four key metrics: **# Compilable**, which counts how many generated patches adhere to programming syntax; **# Plausible**, which counts patches that convert previously failing test cases to passing ones; and **# Correct**, which counts patches that are semantically equivalent to the ground truth fix provided by a human developer. Additionally, we use the **Correct/Plausible** ratio to assess the semantic accuracy of the patches. A high Correct/Plausible ratio suggests that the model generates patches that are not only syntactically valid and functionally correct but also semantically aligned with the intended fix. Conversely, a low ratio may indicate overfitting, where the model produces patches that pass tests but diverge from the intended logic. This ratio may be especially useful in applications with limited human intervention, where plausible but incorrect patches—while they may pass tests—are insufficient.

4 Evaluation

In this section, we evaluate the performance of GPT-4 across different context granularities to answer the following research questions:

- **RQ1**. Does increasing context length improve precision of LLM-based APR, as measured by the Correct/Plausible ratio?
- **RQ2**. Is there a correlation between token count and the number of compilable patches?
- **RQ3**. What factors may contribute to the `Sliced` context achieving the best tradeoff between the Correct/Plausible ratio and token count?
- **RQ4**. How does the `Sliced` context compare to state-of-the-art APR systems in terms of the number of correct patches and Correct/Plausible ratio?

(RQ1) The `File` context, which has the largest token count (331,383 tokens), generates the highest number of plausible patches (47) and correct patches (28), as shown in Table 1. However, its Correct/Plausible ratio is modest at 60%, suggesting that although the large context provides more information, it may also introduce noise. A similar trend is observed for the `Class` context, which

[2] `https://tree-sitter.github.io/tree-sitter/`

Context	# Compilable	# Plausible	# Correct	Token Count	Correct/Plausible
Single-line	61	23	11	**2,070**	48%
Fixed	67	34	23	6,554	68%
Method	**83**	45	25	16,077	56%
Class	78	41	24	281,320	59%
File	71	47	28	331,383	60%
AlphaRepair [24]	–	**50**	**36**	–	72%
Recoder [25]	–	23	11	–	48%
TBar [15]	–	25	8	–	32%
Sliced (Ours)	82	28	22	47,766	**79%**

Table 1: `Sliced` (Ours) achieves the highest Correct/Plausible ratio (79%), outperforming both smaller contexts (`Single-line`, `Method`) and larger contexts (`Class`, `File`). While AlphaRepair generates more correct patches overall, `Sliced` produces a higher proportion of correct patches relative to plausible patches, indicating a higher likelihood of generating plausible patches that are semantically equivalent to the human developer's fix. **Bold** values indicate the best method, and underlined values indicate the second-best method.

produces 41 plausible patches and 24 correct patches, yielding a Correct/Plausible ratio of 59%. This aligns with findings from Liu et al. [16] and Ettinger et al. [19], which suggest that irrelevant context can degrade model performance.

(RQ2) In contrast, smaller context granularities like `Method` and `Sliced` outperform larger contexts in compilability and Correct/Plausible ratios. The `Method` context (16,077 tokens) generates the most compilable patches (83), surpassing the `File` context. The `Sliced` context, with 47,766 tokens, produces 82 compilable patches and achieves the highest Correct/Plausible ratio (79%), the highest among all evaluated contexts. Interestingly, the `Fixed` context achieves the second-highest Correct/Plausible ratio (68%).

(RQ3) We conjecture that the high Correct/Plausible ratio of the `Sliced` context is due to two factors. First, focusing the model's attention on key dependencies reduces the risk of overfitting on irrelevant patterns, enabling it to generate more semantically accurate patches. Second, the buggy line is always positioned at the end of the `Sliced` context, which may help prevent the model from losing focus on critical information. Furthermore, the progression in Correct/Plausible ratios—from `File` to `Sliced`—may suggest that retaining code in close proximity to the buggy line, even if not all of it is directly relevant to control structures or data flows, enhances the model's ability to produce patches that are not only compilable but also more aligned with the developer's intended fix.

(RQ4) When compared to the state-of-the-art system AlphaRepair, which achieves a Correct/Plausible ratio of 72%, the `Sliced` context slightly surpasses this benchmark with a ratio of 79%. This suggests that, despite generating fewer correct patches overall (22), the `Sliced` context, which provides smaller, more targeted input, likely reduces noise and ambiguity, increasing the likelihood that plausible patches are also semantically equivalent to the developer's intended fix.

5 Future Work

Future work could explore hybrid strategies that combine the specificity of smaller contexts (e.g., `Sliced` and `Method`) with the broader scope of larger contexts (e.g., `File`) to improve both the Correct/Plausible ratio and the total number of correct patches. It may also be useful to tailor context strategies to specific bug types, with simpler bugs potentially benefiting from focused contexts and more complex bugs requiring broader scopes. Additionally, the impact of context granularities based on forward or dynamic slicing could be investigated.

Furthermore, we plan to test the generality of our technique across diverse benchmarks, encompassing various bug types, programming languages, and paradigms such as object-oriented and functional programming. Additionally, a thorough evaluation of latency and resource consumption will help determine whether the complex control flow graph computations required for backward slicing are feasible for real-time applications.

Acknowledgments and Disclosure of Funding

The authors would like to thank Jonathan Weese for helpful suggestions and feedback.

References

[1] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[2] M. Dowson. The Ariane 5 software failure. *ACM SIGSOFT Softw. Eng. Notes*, 22:84, 1997.

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[4] Consortium for Information and Software Quality (CISQ). The cost of poor software quality in the us: A 2022 report. `https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf`, 2022.

[5] Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. Ast-t5: Structure-aware pretraining for code generation and understanding. *arXiv preprint arXiv:2401.03003*, 2024.

[6] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[7] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[8] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[9] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[10] Phil Koopman. A case study of toyota unintended acceleration and software safety. *Presentation. Sept*, page 17, 2014.

[11] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25:1980–2024, 2020.

[12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE transactions on software engineering*, 38(1):54–72, 2011.

[13] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[14] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE, 2019.

[15] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 31–42, 2019.

[16] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[17] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[18] OpenAI. Gpt-4 technical report, 2023. Accessed: 2023-09-16.

[19] Lalchand Pandia and Allyson Ettinger. Sorting through the noise: Testing robustness of information processing in pre-trained language models. *arXiv preprint arXiv:2109.12393*, 2021.

[20] Julian Aron Prenner and Romain Robbes. Out of context: How important is local context in neural program repair? In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–13. IEEE, 2024.

[21] U.S. Securities and Exchange Commission. SEC Charges Knight Capital With Violations of Market Access Rule. `https://www.sec.gov/news/press-release/2013-222`, Oct 2013. Accessed: [Your access date].

[22] Mifta Sintaha, Noor Nashid, and Ali Mesbah. Katana: Dual slicing based context for learning bug fixes. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–27, 2023.

[23] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[24] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022.

[25] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 341–353, 2021.

A Implementation Details

The implementation was developed in Python, using the Tree-sitter library to parse Java code and extract its AST. This approach can be adapted to other programming languages by updating the AST generator. We constructed the CFG from scratch, with the AST serving as the basis for identifying nodes and edges that represent control flow between different statements and blocks.

Once the context is extracted, the model prompt is generated in the following format:

```
 Buggy line: [insert buggy line] Context: [insert extracted context]

Please generate the fixed buggy line only. Do not add any additional comments.
```

B Additional Slicing Techniques

In this paper, we evaluate a novel context granularity based on backward static slicing. Other slicing techniques offer different insights into program behavior. *Forward slicing* identifies all program statements that may be affected by changes to a specific variable or statement at a given point in the code [6]. *Dynamic slicing*, in comparison to static slicing, focuses on a specific execution trace [1]. It identifies statements influencing a variable's value for a particular execution run, offering greater precision but reduced generalizability due to its reliance on test inputs for trace generation. *Symbolic execution*, another analysis technique, explores program paths by treating variables as symbolic values, enabling reasoning about the conditions under which specific paths are taken [9].