GiottoBugFixer: an effective and scalable easy-to-use framework for fixing software issues in a DevOps pipeline

Placido Pellegriti[1], Carmine Cisca[1] and Fabio Previtali[1,*]

[1]AlmavivA S.p.A., Via di Casal Boccone 188/190, Rome, 00137, Italy

Abstract

Developing software is one of the most important and crucial activity in the IT domain. It is an important, challenging and time consuming activity due to many factors that spaces from software complexity up to testing and deployment phases. In the past decades, a plethora of tools have been released for helping developers in coding faster, however they are now becoming ineffective and unable to keep up with the change affecting the IT development.

This paper investigates the potential of generative AI in the realm of software development, focusing on how these technologies can augment the coding process, from initial concept to final deployment. It begins by delineating the fundamental mechanisms through which generative AI models, such as code completions and automated code generation can enhance developer productivity, reduce error rates and streamline the software development lifecycle. We conducted an experimentation on several repositories obtaining around 25% of software issues automatically fixed with a 17x speed up.

Keywords

Platform Engineering, Software Automation, Generative AI

1. Introduction

In the rapidly evolving field of software engineering, understanding the intricacies of the software development process is crucial for delivering high-quality, efficient and reliable software solutions. This paper delves into the comprehensive study of the software development lifecycle, focusing on pivotal aspects such as code quality, implementation and testing. By dissecting these elements, we aim to offer insights into optimizing the development process, ensuring that software not only meets but exceeds the rigorous demands of applications to be realized.

At the heart of any software project lies the quality of its code, which serves as the cornerstone for functionality, maintainability, and scalability. We explore methodologies and practices such as code reviews, static code analysis, and adherence to coding standards that contribute to enhancing code quality. By integrating these practices, developers can reduce bugs, facilitate easier updates, and ensure a robust foundation for the software's architecture. The phases of implementation and testing are critical for transforming conceptual designs into functioning software. **Contributions.** This paper examines how generative AI models have been integrated in a DevOps pipeline for helping in improving the quality of the software released. We conducted an experimentation on several repositories in Java and C# and we demonstrated that our solution is able to fix around 25% of software issues 17x faster than a developer.

Ital-IA 2024: 4th National Conference on Artificial Intelligence, organized by CINI, May 29-30, 2024, Naples, Italy

*Corresponding author.

p.pellegriti@almaviva.it (P. Pellegriti); c.cisca@almaviva.it (C. Cisca); f.previtali@almaviva.it (F. Previtali)

2. Related Work

Developing an automatic code fixer is key for enhancing programming productivity [1] and is an active area of research [2, 3, 4].

This trend has gained increasing popularity in recent years. Examples include Google's Tricorder [5], Facebook's Getafix [6] and Zoncolan and Microsoft's Visual Studio IntelliCode. The techniques underlying these tools can be classified into broadly two categories: logical, rule-based techniques [5] and statistical, data-driven techniques [7, 6, 8]. The former uses manually written rules capturing undesirable code patterns and scans the entire codebase for these classes of bugs. The latter learns to detect abnormal code from a large code corpus using deep neural networks.

Despite great strides, however, both kinds of tools are limited in generality because they target error patterns in specific codebases or they target specific bug types. For instance, Zoncolan's rules are designed to be specifically applicable to Facebook's codebases, and deep learning models target specialized bugs in variable naming [7] or binary expressions [6]. Moreover, the patterns are relatively syntactic, allowing them to be specified by human experts using logical rulesor learnt from a corpus of programs.

In this paper, we propose an effective and scalable easy-to-use framework for fixing software issues in a DevOps pipeline by means of an LLM model (i.e., GPT3.5[1]).

[1]https://openai.com

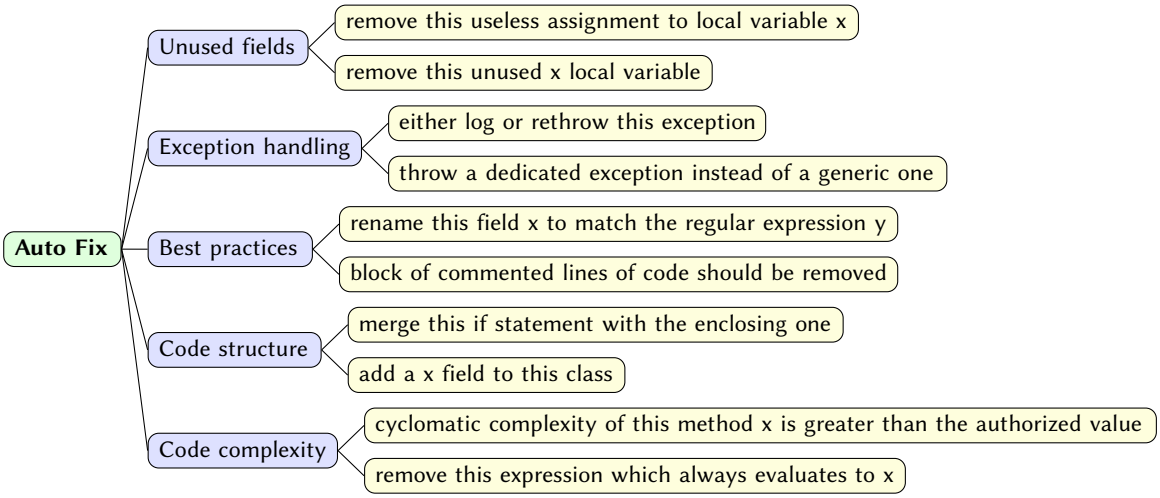

Figure 1: Issue distribution being fixed by the proposed approach among five classes that we defined.

3. Modelling Approach

In this section, we describe the LLM models that have been used, how the prompt has been engineered so that it effectively performs for our task as well as the classification of the issues based on a taxonomy that we defined.

3.1. Model Selection

We evaluated the following models:

1. (OpenAI) `gpt-3.5-turbo-0613`
2. (OpenAI) `gpt-3.5-turbo-1106`
3. (OpenAI) `gpt-4-0613`
4. (MetaAI) `llama-2-7b-hf`

We used OpenAI models via API on Cloud while we fine-tuned the Llama 2 model. Fine-tuning has been carried out by giving examples of snippets pairs `incorrect code/correct code` extracted from our internal repositories.

3.2. Framework

We conducted an analysis about the distribution of issues being fixed by the proposed approach among five classes that we defined (see Figure 1). On `unused fields`, `best practices` and `code structure` classes the proposed solution is able to **correct around 50% of the issues** whilst on the remaining two classes the fixing rate is around 30%.

3.3. Prompt

Engineered Prompt	
System	You are ChatGPT, a code snippet fixer. Your task is to generate a fix for the provided code snippet based on the given error message. Do not alter the code snippet other than fixing the error. Incomplete code should remain incomplete. Submit your response in JSON format with the keys: `corrected_code`, `correction_flag`, `explanation`, `renamed_variables`. `corrected_code` should be contained in double quotes, and all double quotes in the code snippet should be escaped with a backslash. `correction_flag` should be 1 if you have corrected the code snippet, 0 otherwise. The explanation field should contain a brief explanation of the correction. `renamed_variables` should be a Python dictionary containing the names of custom user defined functions or variables that you have renamed as keys, and their new names as values. Do not add any builtin functions you might have changed to `renamed_variables`.
User	I have encountered an error. Error message: `"System.Exception" should not be thrown by user` Code snippet: `if (archiveResult.Result <= 0) { await sess.AbortTrans(); throw new Exception("Fail"); }` Please fix the error in the code snippet without completing it. The code must remain incomplete and indented as in the original snippet. Please provide a JSON response.

3.4. Post-Processing

Following an analysis of common issues observed in code returned by generative models, a series of post-processing functions have been implemented to enhance the quality of the response both in terms of writing style and integration with actual code. This manipulation occurs before the code is inserted into files, prior to undergoing quality checks and software compilation.

Autocompletion errors prevention: Generative models often tend to complete the input code, which frequently consist of incomplete fragments, such as if or for statements without subsequent blocks, or portions that lack logical coherence when considered out of context. To address this issue, lines generated as completions of these snippets can be removed, considering the error occurs midway through the original snippet. Using Greedy String Tiling, a metric employed in literature for comparing code strings, the last lines of the generated code are compared with those from the input. If a match with the original code's final line is identified, only the preceding part up to that line is retained for insertion into the file.

Indentation correction: The generated code often loses the information regarding indentation levels, resulting in snippets where the indentation style and depth may differ from the original code. This discrepancy can include variations in both indentation style (such as tabs versus spaces) and indentation depth within the snippet.

Despite the flexible rules regarding indentation in currently supported languages, a method has been implemented to address this issue. This approach, again based on Greedy String Tiling, compares lines between input and output code to identify and apply a base indentation level that aligns with the indentation found in the received snippet. This ensures improved readability and quality of the generated code snippet, which is guaranteed to have consistent indentation with the surrounding code.

4. Experimental Evaluation

In this section, we report a study on how issues are distributed and results on two languages that are the most widely used by developers.

4.1. Issue Distribution

We conducted an analysis about the distribution of issues being fixed by the proposed approach among five classes that we defined (see Figure 2).
Looking at the plot, on three classes the proposed solution is able to correct around 50% of the issues whilst on the remaining two classes the fixing rate is around 30%.

Repo	Issue Fixed	Tec. Debit Red.	Speed Up
# 1	**100.0 %**	**63.0 %**	10.3x
# 2	41.0 %	13.1 %	2.4x
# 3	36.6 %	10.3 %	2.2x
# 4	32.5 %	20.0 %	2.0x
# 5	46.5 %	26.6 %	2.9x
# 6	58.3 %	46.4 %	**17.0x**
# 7	47.3 %	26.7 %	2.0x
Avg	**51.7 %**	**29.4 %**	**5.5x**

Table 1
Results on Java repositories.

Unused variables/fields: in this class there are five SonarQube rules. In order to better understand what kind of issues belong to this class, here two examples: 1) *remove this useless assignment to local variable x* and 2) *remove this unused x local variable.*

Exception handling: in this class there are six SonarQube rules. The type of issues belonging to this class are for example: 1) *either log or rethrow this exception* and 2) *throw a dedicated exception instead of a generic one.*

Best practices/conventions: in this class there are twenty-seven SonarQube rules. The type of issues belonging to this class are for example: 1) *rename this field x to match the regular expression y* and 2) *block of commented lines of code should be removed.*

Code structure/elements: in this class there are thirty-five SonarQube rules. The type of issues belonging to this class are for example: 1) *merge this if statement with the enclosing one* and 2) *add a x field to this class.*

Code complexity: in this class there are ten SonarQube rules. The type of issues belonging to this class are for example: 1) *the cyclomatic complexity of this method x is greater than the authorized value* and 2) *remove this expression which always evaluates to x.*

4.2. Performance Results

We report the quantitative evaluation of the proposed solution on the two languages of the experimentation. In Table 1, we summarize the results on Java language on which an average debit reduction of 29,4% has been reached, with a peak of 63.0%. The pipeline executes on average 5.5 times faster than developers with a peak of 17 times. In Table 2, we summarize the results on C# language on which an average debit reduction of 25,9% has been obtained, with a peak of 42.9%. The pipeline executes on average 2.4 times faster than developers with a peak of 4.8 times. Results on C# are slightly worst because code is more complex and for building and analyzing the code more time is required with respect to Java.

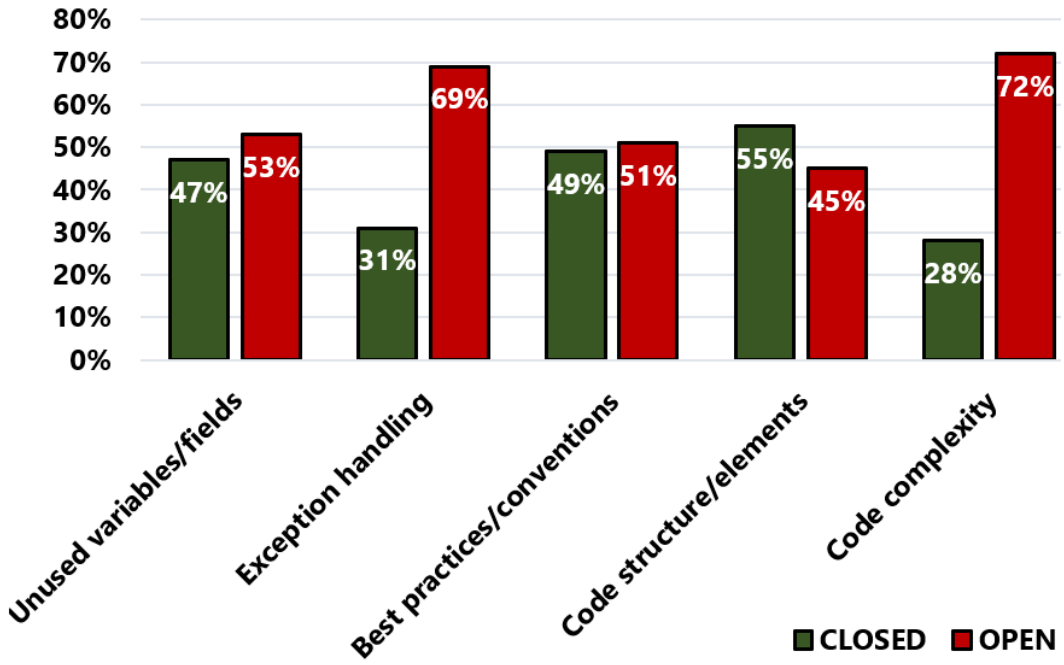

Figure 2: Distribution of the fixed issues on five classes.

Repo	Issue Fixed	Tec. Debit Red.	Speed Up
# 1	46.7 %	36.4 %	2.5x
# 2	30.6 %	12.8 %	1.6x
# 3	39.6 %	18.5 %	1.3x
# 4	32.4 %	14.3 %	0.7x
# 5	37.1 %	34.5 %	2.7x
# 6	38.2 %	21.8 %	3.0x
# 7	**61.2 %**	**42.9 %**	**4.8x**
Avg	**40.8 %**	**25.9 %**	**2.4x**

Table 2
Results on C# repositories.

5. Conclusions

In conclusion, our comprehensive study elucidates the multifaceted nature of the software development process, offering insights into optimizing development practices to meet and exceed the demanding requirements of today's applications. The integration of generative AI models into the software development lifecycle marks a significant advancement, showcasing the potential to revolutionize how software is developed, tested, and maintained. This paper contributes to the body of knowledge by demonstrating the effectiveness of these models in improving software quality and development efficiency, setting a precedent for future research and application in the field of software engineering.

References

[1] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, R. Bowdidge, Programmers' build errors: a case study (at google), in: 36th International Conference on Software Engineering, 2014, pp. 724–734.

[2] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, K. Wang, Hoppity: learning graph transformations to detect and fix bugs in programs, in: International Conference on Learning Representations (ICLR), 2020.

[3] Y. Ding, B. Ray, P. Devanbu, V. J. Hellendoorn, Patching as translation: the data and the metaphor, in: 35th ACM International Conference on Automated Software Engineering, 2020, pp. 275–286.

[4] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, E. Aftandilian, Deepdelta: learning to repair compilation errors, in: 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 925–936.

[5] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, C. Winter, Tricorder: building a program analysis ecosystem, in: 37th International Conference on Software Engineering, volume 1, 2015, pp. 598–608.

[6] J. Bader, A. Scott, M. Pradel, S. Chandra, Getafix: learning to fix bugs automatically, ACM on Programming Languages 3 (2019) 1–27.

[7] M. Allamanis, M. Brockschmidt, M. Khademi, Learning to represent programs with graphs, arXiv preprint arXiv:1711.00740 (2017).

[8] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, R. Singh, Neural program repair by jointly learning to localize and repair, arXiv preprint arXiv:1904.01720 (2019).