An Empirical Study on the Effectiveness of Iterative LLM-Based Improvements for Static Analysis Issues

João Carlos Gonçalves
Universidade Federal de Uberlândia
Uberlândia - MG, Brasil
jcarlosptc@live.com

Marcelo de Almeida Maia
Universidade Federal de Uberlândia
Uberlândia - MG, Brasil
marcelo.maia@ufu.br

ABSTRACT

Maintaining and evolving software systems often demands more effort than their initial development. Improving source code quality through automated support can significantly reduce technical debt, increase maintainability, and enhance developer productivity. This paper presents an experimental approach that integrates static analysis with Large Language Models (LLMs) to automate source code improvement. The proposed pipeline iteratively processes Java classes by extracting issues detected by SonarQube and transforming them into prompts for LLMs, which generate improved code versions. Each version is reanalyzed, and the process repeats until convergence or a predefined iteration limit is reached.

The experimental setup includes multiple configurations combining two LLMs (GPT-4-mini and Gemini), variation in temperature, prompt style, and number of iterations. Evaluations were conducted using multiple Java datasets, with three repeated runs for the Commons Lang repository to identify behavioral patterns. The analysis focuses on the number of issues reduction, decrease in technical debt (measured in a SonarQube metric), and the evolution of issue severity. Functional correctness was assessed manually by inspecting and executing the improved code to ensure behavior preservation.

The results demonstrate that combining SonarQube with LLMs is effective in reducing code issues—achieving over 58% average reduction in key scenarios—while preserving functionality. The iterative process proved successful in guiding the models to incrementally improve code quality based on real static analysis feedback. This work contributes a reproducible and extensible pipeline, offering insights into the impact of LLM configurations and supporting further research in the integration of AI and software quality engineering.

KEYWORDS

Code Quality, Code Readability, Static Analysis, Software Engineering, LLM, ChatGPT, Gemini, SonarQube, Issues.

1 Introduction

Maintenance and evolution are core activities in the software development lifecycle, often consuming more resources than the initial system construction [5]. In this context, the continuous improvement of source code quality becomes essential to ensure readability, maintainability, and the reduction of technical debt. Among the strategies widely adopted to achieve these goals are static analysis tools, such as SonarQube [42], and, more recently, the use of Large Language Models (LLMs) as support tools for code improvement and correction tasks [4, 11, 28].

LLMs specifically trained on programming data, such as Codex [37], CodeGen [2], StarCoder [31], and Code LLaMA [35], have shown promising performance in source code generation, explanation, and correction [10, 14, 16, 38]. Recent studies have explored the use of these tools as assistants in software development, demonstrating advances in contextual understanding, defect identification, and the automated application of patches [20]. Despite their potential, the practical adoption of LLMs in real-world environments still faces challenges related to the reliability of their suggestions, the preservation of original functionality, and integration with traditional code quality analysis tools [24].

Moreover, SonarQube remains one of the most widely used tools for static analysis, extensively adopted in both public and private software projects [43]. Its rule-based approach enables the detection of security flaws, vulnerabilities, maintainability issues, and code smells. However, despite offering detailed and contextualized diagnostics, SonarQube does not perform automatic fixes, requiring developers to manually apply the recommended improvements.

In this context, given the relevance of SonarQube for software development, it is worthwhile to investigate approaches that combine the precision of static analysis with the generative capabilities of LLMs.

This work proposes and evaluates an experimental approach for automatic source code improvement through the integration of SonarQube and LLMs in an iterative, automated, and controlled process. To operationalize this proposal, a pipeline was developed to orchestrate static analysis, prompt generation for the models, code replacement, reanalysis, and result logging in successive cycles. SonarQube diagnostics serve as the underlying instructions for constructing prompts submitted to the LLMs, which return refined versions of the code while preserving its original functional behavior. Each generated version is reanalyzed, forming a continuous improvement cycle that proceeds until convergence or the maximum number of iterations is reached.

The experiments were conducted using different models (GPT-4-mini and Gemini) and configurations (temperature, prompt, number of iterations), applied to widely used Java repositories such as Commons Lang, Commons IO, and Google Guava. The analysis of the results considered the reduction in the total number of issues, the decrease in estimated technical debt, and the evolution of issue severity across iterations. Functional preservation was verified manually through inspection and execution of the improved methods.

The results demonstrate the potential of combining static analysis with LLMs to automate the code improvement process, reducing the manual effort associated with fixing the code and contributing to the enhancement of software quality. The proposed approach aims to address existing gaps in the literature, particularly regarding the iterative use of LLMs guided by static analysis tools, the evaluation of the impact of different configurations, and the systematic execution of experiments with functional assessment.

The remainder of this paper is organized as follows. Section 2 introduces the background concepts. Section 3 describes the methodology, including the proposed pipeline, the experimental setup, and the evaluation metrics. Section 4 presents and discusses the results. Section 5 reviews related work. Section 6 outlines the threats to validity and limitations. Finally, Section 7 presents the conclusions.

2 Background

This section presents fundamental concepts for understanding the proposal of this work.

2.1 Static Code Analysis

Static code analysis is a widely used technique for inspecting source code without executing it, aiming to identify defects, bad practices, and potential vulnerabilities [32]. This approach provides an efficient way to detect quality issues in the early stages of development, helping to reduce maintenance costs and increase software reliability [34].

One of the most commonly used tools for static analysis is SonarQube, an open-source platform that automatically evaluates source code based on predefined rules, classifying issues into different severity levels (e.g., *bugs*, *code smells*, and *security vulnerabilities*) [8]. In addition, SonarQube provides quantitative metrics such as cyclomatic complexity, test coverage, code duplication, and technical debt [1, 42].

Despite its effectiveness in identifying problems, SonarQube does not automatically apply fixes. This leaves developers responsible for manually reviewing and resolving the reported issues, which can be time-consuming and error-prone — especially in large-scale projects [1]. This scenario has motivated research exploring the automation of the repair process based on reports generated by tools like SonarQube [11].

2.2 Large Language Models

Large Language Models (LLMs) have been gaining increasing attention in the field of Software Engineering. Trained on massive amounts of natural language and source code data, these models are capable of understanding and generating code with a high degree of fluency and coherence [13, 14]. Popular examples include Codex [37], used in GitHub Copilot [29], as well as CodeT5 [26] and CodeBERT [27].

LLMs have been applied to a variety of software development tasks, such as code generation, function summarization, auto-completion, code explanation, and even automated program repair [14, 19]. In the latter case, the model receives as input either a fault description or a defective code snippet and returns a corrected version. Recent studies show that LLMs can achieve promising success rates in code repair tasks, especially when guided by carefully designed prompts [6, 7, 33].

In the context where LLMs and static analysis intersect, opportunities emerge for hybrid solutions that use tools like SonarQube to detect issues and then trigger an LLM to propose fixes — an approach adopted in this work — considering that LLMs are not yet capable of fully replacing static analysis tools [3]. This orchestration model represents a step toward intelligent automation in software maintenance by combining the precise diagnostics of static analysis with the generative capabilities of language models [4, 9].

The application of LLMs in iterative code improvement cycles — progressively adapting to static analysis reports — has also been investigated as an alternative to traditional manual improvement approaches, offering a new paradigm for quality-driven development [17, 33].

These investigations reinforce the potential of integrating static analysis with generative models as a promising path for automating maintenance tasks and improving overall source code quality.

3 Methodology

This research adopted an experimental approach using both quantitative and qualitative data analysis to investigate the effectiveness of Large Language Models (LLMs) in the automated improvement of source code based on recommendations issued by a static analysis tool. The study was conducted in four main stages: (i) the development of an automated iterative improvement pipeline, (ii) the systematic execution of experiments on real-world repositories, (iii) the quantitative analysis of results based on metrics extracted from SonarQube evaluations applied to the improved versions, and (iv) the qualitative analysis of functionality preservation after the code improvements.

3.1 Automated Iterative Improvement Pipeline

The automated pipeline developed for this research was responsible for orchestrating the analysis, generation, and evaluation of improvements. Its goal was to ensure reproducible, iterative, and scalable executions over different combinations of experimental configurations. The pipeline architecture consisted of the following main modules:

3.1.1 *Input File Reading:*

This module initiates the iterative improvement cycle for each code class in the selected dataset. The pipeline operates over a set of files organized by repository and experiment.

During execution, each Java file is read as raw text using the `r+ead_file` function. Its content is copied into a predefined project structure, replacing the content of a fixed main file — typically `Main.java` — located at `code/src/main/java/project/`. This allows isolated static analysis using SonarQube without requiring multiple Java projects to be configured.

This modular design makes the pipeline reusable and independent of the original repository structure, enabling standardized iterative processing across thousands of classes. Additionally, using a single entry point per analysis simplifies version tracking, metric collection per class, and validation of improvements at each iteration.

3.1.2 *Static Analysis with SonarQube:*

Static code analysis is performed using SonarQube, executed through the *SonarScanner*. After each iteration and class update, the scanner is triggered to assess the improved code based on the project's preconfigured quality rules.

Once the scan is completed, the pipeline waits for 60 seconds (`sleep(60)`) to ensure that analysis data is indexed and made available by the SonarQube instance (running in a Docker container). This delay accounts for propagation time between analysis completion and API data availability.

The pipeline then queries the SonarQube API to retrieve the list of issues identified in the last scan. These results guide the filtering, prompt generation, and metric registration steps in the following iterations.

Examples of real SonarQube messages retrieved during the experiments include:

- `Remove this unused "STRING_ON_OFF" private field. (line 17)`
- `Replace the usage of the literal "true" by a named constant or an enum. (line 32)`
- `Add a default case to this switch. (line 41)`

3.1.3 *Custom Prompt Construction:*

After issue identification, the pipeline dynamically constructs a natural language prompt and sends it to the selected LLM to guide the code improvement. The prompt aims to clearly communicate the issues found in the code and specify the required improvements.

Two prompt templates were used across experiments:

Prompt 1 (zero-shot)

Apply the following improvements to the code and return only the Java code: {issues_message}.

Prompt 2 (role-based)

You are acting as a software development assistant and must apply improvements to the code using the following list of issues: {issues_message}. You should only modify the code to fix the reported issues, without removing any functionality unless explicitly stated. Only the Java code must be returned, with no additional explanations or text, and no loss of any previously implemented behavior.

The variable `{issues_message}` is dynamically filled with the list of issue messages extracted from SonarQube in the current iteration. This prompt variation was used as an experimental parameter to evaluate the impact of instruction formulation on the quality of the generated improvements.

3.1.4 *Interaction with the Configured LLM:.*

The constructed prompt is sent to the LLM (GPT-4-mini or Gemini) via API, along with defined parameters (temperature, token limit, etc.). The expected output is the improved code only, without any additional text.

3.1.5 *Code Replacement and Reevaluation:*

The returned code replaces the original version, and a new SonarQube analysis is triggered to assess the impact of the improvement in the current iteration.

3.1.6 *Metric Logging and Execution Control:*

Iteration data (e.g., number of issues, severity levels, technical debt, execution time) is stored in `.csv` files per experiment and execution. All improved versions are also saved as `.txt` files for further validation and historical tracking.

3.1.7 *Convergence or Iteration Limit:*

The iterative process terminates when either (i) no new issues are reported, or (ii) the maximum number of iterations is reached (2 or 5, depending on the experimental configuration).

3.2 Research Questions

To investigate the effectiveness of using Large Language Models (LLMs) in the automated improvement of source code based on static analysis feedback, this study was guided by the following research questions:

- **RQ1: What is the percentage reduction of issues?** This question evaluates the ability of LLMs to reduce the number of defects identified by static analysis tools after applying automated improvements.
- **RQ2: Does the reduction in issues proportionally impact the reduction of technical debt?** This question examines whether the observed reduction in defects directly translates into a corresponding decrease in the estimated technical debt of the project.
- **RQ3: How does the experimental configuration impact code improvement?** This question analyzes the effect of varying experimental parameters (language model, temperature, prompt formulation, and number of iterations) on the quality of the generated improvements.
- **RQ4: Is there any functional breakage throughout the iterations of the improvement process?** This question investigates whether the iterative improvement process compromises the original functionality of the code, introducing regressions or semantic faults.

These research questions were formulated to provide a comprehensive analysis of the impacts, limitations, and potential of LLMs as agents for continuous software quality improvement.

3.3 Experimental Configurations and Design Justification

The experiments were systematically designed to evaluate the effectiveness of LLM-assisted automatic improvement under different parameter combinations, as well as to understand how the variation of datasets influences the results. Each experimental setup involved four main parameters: the language model used (GPT-4-mini or Gemini), the generation temperature (0.1 for more deterministic behavior or 0.3 for more exploratory responses), the prompt formulation (direct vs. contextualized), and the maximum number of iterations per class (2 or 5). These combinations defined eight unique experimental scenarios, numbered 1 through 8.

The selection of these scenarios followed a structured experimental design methodology based on *Fractional Factorial Design* (FFD) [36]. FFD was adopted to enable a systematic yet cost-effective exploration of the experimental space, allowing the estimation of the **main effects** and selected **interaction effects** of the studied factors.

By evaluating a representative subset of all possible combinations, FFD made it feasible to assess the influence of multiple parameters — such as model, temperature, prompt style, and number of iterations — on the performance of the refactoring process, while reducing the total number of required executions and computational effort.

This approach was crucial for:

- Isolating the individual impact of each factor (e.g., model, prompt type, iteration count) on the reduction of issues and technical debt;
- Identifying significant interactions between factors, such as whether the effect of a higher number of iterations depends on the model or prompt used;
- Maximizing experimental robustness with a feasible number of executions, given resource and time constraints.

Each experiment was executed over a set of Java classes. For the `Commons Lang` dataset, all eight configurations were executed three times to ensure statistical robustness, enabling the analysis of means, standard deviations, and behavioral patterns. For the exploratory phase using `Commons IO` and `Guava`, each configuration was executed once to provide a complementary perspective on how code diversity and project style influence improvement effectiveness.

The entire experimental process was fully automated through the developed pipeline, which handled input reading, static analysis via SonarQube, prompt generation, model interaction, code replacement, and metric logging — including the number of issues, severity levels, estimated effort (technical debt), and iteration count. The output data were stored in structured `.csv` files, and each code version generated throughout the iterations was saved in `.txt` format to enable inspection, validation, and historical traceability.

3.4 Repositories and Data Sources

The experiments used real Java repositories extracted from popular open-source projects with active maintenance histories. The main repositories selected were Apache Commons Lang, Apache Commons IO, and Google Guava.

Apache Commons Lang extends core Java functionalities with utilities for string, number, and object manipulation. Apache Commons IO offers I/O utilities for file operations. Google Guava is a comprehensive core library set from Google, featuring collections, immutables, graph support, and concurrency, hashing, and string utilities.

These repositories were selected due to their popularity, diversity of code constructs, active maintenance history, and compatibility with SonarQube. Additionally, the presence of automated test suites (although not used in this study) and documented code quality issues provided a rich environment for evaluating LLM-based improvements. These factors contribute to both the relevance and the potential generalizability of the findings within the Java ecosystem.

3.5 Evaluation Metrics

To assess improvement effectiveness, the following metrics were extracted per iteration:

- Total number of issues (SonarQube)
- Percentage reduction in issues
- Reduction by severity (BLOCKER, CRITICAL, MAJOR)
- Reduction in technical debt (estimated effort in minutes)

Code functionality preservation was evaluated manually through the execution of improved methods and visual inspection. Although no functional failures were observed in the evaluated samples, a systematic semantic validation was not performed, which constitutes a methodological limitation.

4 Results and Discussion

This section presents the results obtained from the application of Large Language Models (LLMs) in the task of automated source code improvement based on issues reported by SonarQube.

4.1 RQ1 — What is the percentage reduction of issues?

To answer RQ1, we evaluated the percentage of issues identified by SonarQube that were successfully corrected by Large Language Models (LLMs) through iterative cycles of improvements. The analysis considered the percentage difference between the number of issues in the initial iteration (0) and the last recorded iteration in each execution. Due to financial and computational resource limitations, most experiments were executed only once; thus, the analysis prioritizes the variation across different datasets and experimental configurations, providing a comparative view of the models' effectiveness.

Figure 1 presents the average percentage reduction of issues per experiment for the Apache Commons Lang dataset. It is observed that seven out of eight experiments achieved reductions above 50%, with one experiment exceeding 80%. Figure 2 illustrates the results

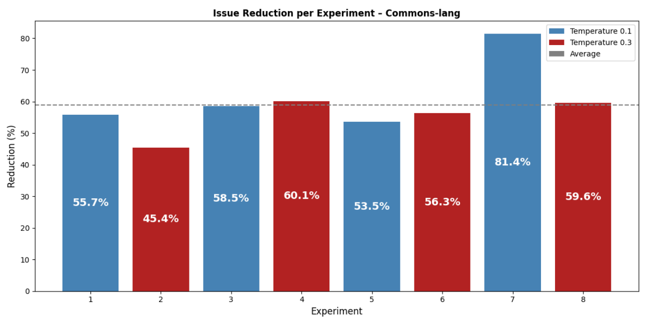

Figure 1: Average issue reduction observed in the Apache Commons Lang dataset.

obtained from experiments using the Apache Commons IO dataset. In this case, an average reduction greater than 70% was achieved, with variations between 55% and 83% depending on the configuration. Figure 3 depicts the results obtained from experiments using the Google Guava dataset, where an average reduction of 46% was observed, with variations between 41% and 51% depending on the configuration adopted. The differences in average reduction percentages across datasets can be explained by the number and severity of issues present in each case. Although such differences exist, we observed that the configuration directly impacts the results. For example, the parameters (temperature, prompt type, and

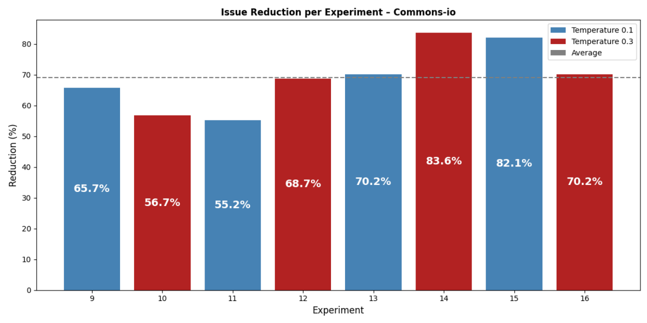

Figure 2: Average issue reduction observed in the Apache Commons IO dataset.

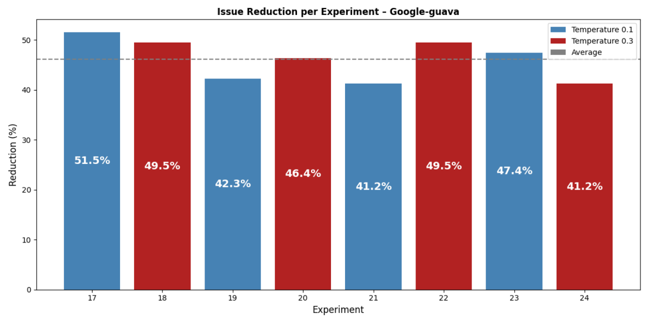

Figure 3: Average issue reduction observed in the Google Guava dataset.

number of iterations) used in Experiments 1 and 2 for the Commons Lang dataset are the same as those used in Experiments 9, 10, 17, and 19, resulting in similar trends across executions.

Another important aspect in the analysis of issue reduction is the severity of the issues, as some types of problems are inherently more complex to correct. Cyclomatic complexity, for instance, requires splitting a function into several others while preserving the original behavior. Table 1 presents the average reduction by severity level, considering all experiments. These results indicate that the models are more effective at correcting issues of intermediate and high severity (CRITICAL and MAJOR), but were not able to efficiently handle BLOCKER-type issues, which even showed a slight increase in one of the experiments. This suggests that deeper structural improvements still present a significant challenge for LLMs.

Thus, although LLMs proved effective in addressing a substantial portion of problems detected by static analysis — often surpassing 50% issue reduction — their effectiveness varies according to the severity of the issue and the experimental configuration, highlighting the need for carefully tuned parameters to achieve optimal results.

4.2 RQ2 — Does the Reduction of Issues Proportionally Impact the Reduction of Technical Debt?

To assess the proportionality between issue reduction and technical debt reduction, the percentage reduction of each metric was calculated for the experiments conducted on three distinct datasets: Commons Lang, Commons IO, and Google Guava. The analysis considered the comparison between the values from the first and last iteration of each experimental execution, using a single run per configuration.

Table 2 summarizes the average values of issue reduction, technical debt reduction, the absolute difference between these metrics, and the relative proportion of technical debt reduction in relation to issue reduction. The data indicate that although issue reduction is strongly correlated with technical debt reduction, this relationship is not proportional in all cases. The Commons IO dataset showed the highest relative efficiency, with an average reduction of 69.03% in issues and 63.33% in technical debt, resulting in an absolute difference of only 5.70 percentage points and a proportion of 0.91, indicating that nearly every corrected issue contributed to the reduction of technical debt.

In contrast, the Commons Lang dataset presented a more significant difference: despite achieving a 58.81% reduction in issues, the reduction in technical debt was only 42.08%, resulting in a difference of 16.73 percentage points and a proportion of 0.71. This suggests that, in this context, language models tend to prioritize fixing issues with lower technical impact, leaving problems with higher estimated effort (in minutes) partially or fully unaddressed.

The Google Guava dataset, in turn, exhibited the lowest absolute reduction rates (46.13% for issues and 42.14% for technical debt), but maintained a high relative proportion (0.92), demonstrating a more balanced behavior between the two metrics, albeit with a lower overall impact.

These results highlight that while issue reduction can serve as a good indicator of code improvement, it does not, by itself, guarantee a proportional reduction of technical debt. The observed discrepancy between the metrics underscores the importance of considering both aspects in automated software quality assessments assisted by LLMs.

4.3 RQ3 — How Does the Experimental Configuration Impact Code Improvement?

This research question investigates the extent to which experimental configurations — composed of the combination of the LLM model, generation temperature, prompt type, and number of iterations — influence the effectiveness of automated code improvement. The analysis was conducted based on three executions per experiment to obtain more stable estimates of the average behavior for each configuration.

Table 3 presents the average percentage reduction of issues for each experiment, along with the corresponding standard deviations. The results demonstrate that experimental configurations substantially influence the effectiveness of automated code improvement. Among all the evaluated combinations, Experiment 7 — which used the Gemini model with a temperature of 0.1, prompt type 1, and five iterations — achieved the best average performance, reaching an 81.29% reduction in issues. This outcome suggests that combining a robust model, a more conservative temperature, greater iterative depth, and a well-formulated prompt provides a favorable environment for effective improvement.

Table 1: Analysis of Issue Reduction by Severity

Dataset	Blocker Reduction (%)	Critical Reduction (%)	Major Reduction (%)
Commons Lang	-1.25	67.45	60.00
Commons IO		50.66	76.30
Google Guava		42.61	47.17

Table 2: Comparison of Issue Reduction and Technical Debt Reduction by Dataset

Dataset	Reduction of Issues (%)	Reduction of Technical Debt (%)	Absolute Difference (%)	Relative Proportion (x)
Commons IO	69.03	63.33	5.70	0.91
Commons Lang	58.81	42.08	16.73	0.71
Google Guava	46.13	42.14	3.99	0.92

Table 3: Average Results of Percentage Reduction of Issues per Experiment

Experiment	Model	Temperature	Prompt	Iterations	Average Reduction (%)	Standard Deviation (%)
1	GPT-4-mini	0.1	1	2	57.65	4.54
2	GPT-4-mini	0.3	2	2	49.49	5.52
3	GPT-4-mini	0.1	2	5	62.58	1.94
4	GPT-4-mini	0.3	1	5	70.41	7.52
5	Gemini	0.1	2	2	53.40	6.89
6	Gemini	0.3	1	2	61.90	5.14
7	Gemini	0.1	1	5	81.29	7.94
8	Gemini	0.3	2	5	70.26	5.91

At the opposite extreme, Experiment 2, configured with GPT-4-mini, a temperature of 0.3, prompt type 2, and only two iterations, resulted in the lowest average reduction (49.49%). This configuration highlights the risks associated with suboptimal parameter combinations, where a higher temperature — while potentially beneficial for response diversity — when combined with a less targeted prompt and fewer cycles, undermines the model's ability to converge toward qualified solutions.

Moreover, the results reveal internal consistency among configurations with multiple iterations, particularly in Experiments 3, 4, 7, and 8, reinforcing the notion that controlled repetition of the analysis and improvement cycle tends to improve outcomes. However, the observed variability between models and the interaction between parameters demonstrate that performance does not depend on any single factor in isolation but rather on the synergy among all experimental elements.

4.3.1 *Model Impact.*

The comparison between the models — GPT-4-mini and Gemini — reveals relevant differences in both average performance and stability across executions. The Gemini model, particularly in configurations with five iterations (Experiments 7 and 8), achieved the best overall results, with Experiment 7 standing out by obtaining the highest average issue reduction (81.29%) among all tested configurations. Furthermore, even in less optimized setups (such as Experiment 6, with only two iterations), Gemini maintained competitive performance (61.90%), indicating greater robustness and resilience to variations in experimental parameters.

On the other hand, GPT-4-mini demonstrated greater sensitivity to temperature, prompt type, and the number of iterations. Its average performance varied significantly across experiments, ranging from 49.49% (Experiment 2) to 70.41% (Experiment 4). This amplitude reveals that the model can either perform excellently or yield limited results depending on the calibration of the experimental parameters. In particular, the combination of a higher temperature and a less effective prompt (as in Experiment 2) negatively impacted performance, even compared to simpler configurations of the Gemini model.

These findings indicate that, while both models are capable of code improvement with some level of success, the Gemini model tends to be more consistent and reliable, especially in scenarios with greater iterative depth. Conversely, GPT-4-mini exhibits higher potential variability, requiring greater attention in experimental design to achieve satisfactory results.

4.3.2 *Temperature and Prompt: A Determinant Interaction.*

The analysis of experiments shows that generation temperature and prompt type should not be evaluated independently but as interdependent components that, when well combined, can maximize — or compromise — the quality of automatic improvement.

Generally, higher temperatures, such as 0.3, increase the diversity and fluency of model responses, which can be beneficial for tasks requiring creativity, such as code rewriting. However, this additional freedom demands clearer, more objective, and better-structured prompts capable of guiding the model despite the broader generation space. This is evidenced when comparing Experiments 2 and 4, both using GPT-4-mini with a temperature of 0.3 but different prompts: Experiment 4 (Prompt 1) achieved a 70.41% reduction in issues, while Experiment 2 (Prompt 2) achieved only 49.49%. The difference of over 20 percentage points, despite using the same model and temperature, highlights that the prompt was decisive in guiding the improvement process.

This behavior is also observed with the Gemini model, although with less variability. Comparing Experiments 7 and 8, both with five iterations but different prompts, shows that Prompt 1, when combined with a temperature of 0.1 (Experiment 7), led to the best overall performance (81.29%), while Prompt 2 (Experiment 8) resulted in 70.26%. Although both presented high reduction rates, the difference further reinforces the moderating role of the prompt over the effects of temperature.

Thus, empirical evidence demonstrates that temperature enhances generation flexibility but only produces real gains when associated with prompts that adequately contextualize and constrain the task scope. Models exposed to vague or generic prompts, even with greater creative freedom, tend to produce inconsistent results. Therefore, the temperature's effect is not linear, and its effectiveness is intrinsically tied to the quality of the prompt engineering employed.

4.3.3 *Number of Iterations: Incremental and Conditioned Impact*.

The data analysis reveals that the number of iterations exerts a positive incremental effect on the quality of improvement, but this impact is conditioned on the presence of other good experimental decisions. Configurations with five iterations, such as Experiments 3 and 4, achieved higher average reductions (62.58% and 70.41%, respectively) compared to configurations with only two iterations, such as Experiments 1, 2, and 5.

However, increasing the number of iterations alone is not sufficient to guarantee better results. For instance, Experiment 2, with only two iterations, showed the worst performance (49.49% average reduction) — a result attributed to the combination of low iterativity with an ineffective prompt. In contrast, Experiment 4, combining more iterations with a clearer prompt and a temperature that favored diversity, achieved one of the best results.

These findings suggest that additional iterations enable a progressive refinement process, where the model can address residual problems left by earlier cycles. Nevertheless, when the experimental foundation is poorly configured — for example, with vague instructions or inadequate generation parameters — the repetition may simply propagate flaws or yield only superficial adjustments.

Thus, the number of iterations should be interpreted as a quality amplification factor, whose impact is most expressive when embedded within a favorable experimental context. It is therefore a strategic variable that can be exploited to intensify the iterative improvement process, provided that the other elements of the experiment are properly calibrated.

Overall, the analysis shows that the experimental configuration directly and substantially impacts the effectiveness of LLM-driven automated improvement. Model performance cannot be dissociated from the accompanying parameters. The Gemini model proved to be more robust and consistent, whereas GPT-4-mini showed greater variability, being more sensitive to prompt quality, temperature, and the number of iterations.

Moreover, the interaction between temperature and prompt was shown to be determinant, with clear evidence that well-structured prompts are essential to extract the maximum potential of models operating with greater creative freedom. Meanwhile, the number of iterations acts as an amplifier, contributing to the maturation of improvements over multiple cycles.

These findings reinforce that the effectiveness of LLM-based improvement depends on the orchestration of multiple factors, requiring systematic planning and experimental design to select the ideal configuration according to the project's objectives and context.

It is important to note that these findings are based on a limited set of configurations and datasets. Further experimentation is needed to confirm whether similar patterns hold in larger or more diverse codebases.

4.4 RQ4 — Is There Any Functional Breakage Throughout the Iterations of the Improvement Process?

This research question investigated whether the iterative improvement process assisted by LLMs compromises the original functionality of the source code at any point. Since the goal of the study is to apply improvements based on SonarQube recommendations, it is crucial to ensure that such modifications do not introduce regressions or remove expected behaviors.

To address this question, a manual and exploratory evaluation was conducted on the code versions generated throughout the iterations. The analysis consisted of visually inspecting the changes made by the models and executing the improved methods whenever possible. The primary objective was to identify indications of functional failures, such as improper removal of code fragments, unjustified semantic changes, or unexpected modifications in control logic.

The results indicate that, in the evaluated samples, no explicit functional breakages were observed. Overall, the LLMs maintained a conservative behavior, performing punctual improvements such as code reorganization, renamings, and extraction of auxiliary methods, without impacting the functional logic of the classes.

A representative example can be observed in Figures 4 and 5. Figure 4 presents the original version of the `lastIndexOf(...)` function, while Figure 5 shows the improved version, in which an auxiliary method named `handleStringLastIndexOf(...)` was extracted. This modification fully preserves the original functional behavior while improving readability, promoting modularization, and facilitating future code maintenance.

Although the results are positive, it is important to highlight that the functional evaluation conducted has its limitations. The absence of a comprehensive suite of automated tests or formal semantic validation prevents an exhaustive verification of behavioral preservation at scale. Although the selected projects include automated test suites, these were not integrated into the pipeline due to technical constraints in test setup isolation per file and the heterogeneous structure of the repositories. Future work will address this by incorporating mutation testing or oracle-based validation strategies to improve functional assessment at scale.

```java
static int lastIndexOf(final CharSequence cs, final
     CharSequence searchChar, int start) {
    if (searchChar == null || cs == null) {
        return NOT_FOUND;
    }
    if (searchChar instanceof String) {
        if (cs instanceof String) {
            return ((String) cs).lastIndexOf((String)
                searchChar, start);
        }
        if (cs instanceof StringBuilder) {
            return ((StringBuilder) cs).lastIndexOf((
                String) searchChar, start);
        }
        if (cs instanceof StringBuffer) {
            return ((StringBuffer) cs).lastIndexOf((
                String) searchChar, start);
        }
    }
    ...
}
```

Figure 4: Code before improvement.

```java
static int lastIndexOf(final CharSequence cs, final
     CharSequence searchChar, int start) {
    if (searchChar == null || cs == null) {
        return NOT_FOUND;
    }
    if (searchChar instanceof String) {
        return handleStringLastIndexOf(cs, (String)
            searchChar, start);
    }
    ...
}

private static int handleStringLastIndexOf(final
     CharSequence cs, final String searchChar, int start)
      {
    if (cs instanceof String) {
        return ((String) cs).lastIndexOf(searchChar,
            start);
    }
    if (cs instanceof StringBuilder) {
        return ((StringBuilder) cs).lastIndexOf(
            searchChar, start);
    }
    if (cs instanceof StringBuffer) {
        return ((StringBuffer) cs).lastIndexOf(searchChar
            , start);
    }
    return NOT_FOUND;
}
```

Figure 5: Code after improvement through method extraction.

Despite the positive results, it is important to emphasize that this analysis was conducted without a formal suite of automated tests, which limits the ability to generalize the findings. Functional verification was based on direct execution and visual analysis of the improved versions, which does not guarantee the detection of all types of regressions, particularly those involving subtle behaviors or external context dependencies.

Therefore, it can be concluded that, in the evaluated samples, no perceptible functional breakages were observed, and the models proved capable of performing behaviorally safe improvements. Nevertheless, the absence of automated tests or formal semantic validation represents a relevant limitation of the study. Future work may incorporate mechanisms such as semantic equivalence verification, specification-based testing, or mutation testing to more rigorously validate functional preservation throughout the iterative improvement process..

5 Related Work

The use of *Large Language Models* (LLMs) — such as GPT-3 [39], GPT-4 [40], CodeT5 [26], and Codex [37] — has shown great promise in tasks such as code improvement, automated repair, and code generation [18, 23]. Zhang *et al.* [18] conducted a systematic review on the use of LLMs in *Automated Program Repair* (APR), organizing dozens of recent studies, proposing a taxonomy of existing approaches, and exploring scenarios such as semantic bug fixing and security vulnerability mitigation.

Complementarily, Hou *et al.* [23] discussed the use of LLMs with a special focus on vulnerability repair, emphasizing the importance of pre-training models on code-specific datasets. Models like CodeBERT [27] and CodeT5 [26] have been applied to supervised repair tasks, trained with explicit defect-solution patterns, highlighting the need for representative datasets to maximize model effectiveness.

Seeking to reduce dependence on supervised training, several studies [22, 33, 41] demonstrated that LLMs can autonomously repair code defects through inference, without requiring manual labels. However, although these models can reflect on code behavior and propose corrections, there are no formal guarantees of correctness. As highlighted by Cai *et al.* [24], this limitation has motivated the development of complementary mechanisms to guide and validate LLM-generated corrections, mitigating the risk of functional degradation or the introduction of new defects [12].

Addressing the contextual limitations often faced by LLMs — especially when code depends on information dispersed across multiple files or external libraries — recent works have explored mechanisms based on static analysis and advanced prompt engineering [15, 22].

Among these, the works of Agrawal *et al.* [12], Ahmed *et al.* [21], and Hao *et al.* [25] stand out. Agrawal *et al.* proposed the *Monitor-Guided Decoding* (MGD) technique, where a monitor acts as a stateful interface between the LLM and static analysis tools such as linters and LSP servers. During code generation, the monitor intercepts code snippets at predefined trigger points, queries these tools, and transforms their feedback into constraints applied to subsequent decoding steps.

Ahmed *et al.* [21] leveraged outputs from tools like SonarQube — including violated rules, severity, and defect locations — to select relevant historical examples and compose contextually rich few-shot prompts, improving generalization capabilities and repair accuracy on unseen projects.

In a distinct line of research, Hao *et al.* [25] explored the potential of guiding LLMs through the symbolic execution of pseudocode embedded directly into prompts. Their approach enables the model to reason about variables, control flow, and method calls, simulating aspects of static analysis internally without external instrumentation.

Beyond pointwise analysis and repair strategies, recent studies have explored *iterative improvement cycles*, where suggestions generated by LLMs are applied, evaluated, and refined over multiple rounds. Barke *et al.* [19] investigated this interactive approach using LLMs integrated into GitHub Copilot [29], while other proposals designed fully automated pipelines combining static analysis, suggestion generation, and continuous reassessment [33, 41].

These approaches collectively highlight the growing maturity of LLM usage in software engineering, reinforcing the importance of static analysis integration, contextual prompt curation, and continuous iteration to overcome inherent limitations and enhance model robustness in real-world development scenarios.

6 Threats to Validity

Although the results obtained in this research demonstrate the potential of Large Language Models (LLMs) for automated source code improvement, some limitations and threats to validity must be acknowledged.

6.1 External Validity

This study was conducted using a specific set of open-source Java repositories (Apache Commons Lang, Apache Commons IO, and Google Guava), which may limit the generalizability of the findings to other application domains, programming languages, or coding styles. In addition, only two LLMs (GPT-4-mini and Gemini) were evaluated under a restricted set of configurations. Although multiple executions were carried out to reduce variability, the explored parameter space remains limited compared to the full range of LLM behaviors and prompt engineering possibilities.

6.2 Internal Validity

One of the main threats concerns the functional evaluation of the improved code. Although the iterative process significantly reduced issues, functionality preservation was verified manually. The absence of automated testing limits the generalization of the results. Although no failures were observed in manual execution, we recognize that this form of validation does not guarantee the absence of semantic regressions. This limitation will be addressed in future work by integrating mutation testing and differential validation.

6.3 Construct Validity

Another threat relates to the reliance on SonarQube as the sole evaluation tool for measuring improvement. While SonarQube provides detailed static analysis results, it may not capture all relevant aspects of software quality, such as architectural design issues, code readability improvements not mapped to rules, or maintainability aspects beyond detected issues.

6.4 Conclusion Validity

Finally, the evaluated LLMs operate as black-box systems, lacking explicit reasoning regarding the software's broader architecture and context. As a result, the generated improvements are primarily focused on localized changes, potentially ignoring dependencies and interactions among components. This may affect the long-term maintainability and integration quality, even if short-term static analysis metrics are improved.

7 Conclusion

This work investigated the use of Large Language Models (LLMs) for automated source code improvement through an iterative process guided by SonarQube recommendations. The core proposal was operationalized through the development of an automated experimental pipeline that integrates static analysis, prompt generation, interaction with LLMs, and continuous quality reassessment after each iteration.

The developed pipeline played a fundamental role in conducting the research, enabling reproducible, scalable, and comparable execution of multiple experiments. By automating tasks from input file reading to post-improvement result collection, the pipeline made it possible to consistently apply the iterative process across different repositories, models, and configurations, while facilitating quantitative analysis of the results.

The collected data demonstrated that the models were capable of promoting significant reductions in the identified *issues*, with average reductions exceeding 58% across the experiments. The analysis segmented by severity revealed that the models tend to be more effective in resolving *MAJOR* and *CRITICAL* issues, which achieved the highest correction rates. This finding indicates that LLMs are capable of contributing to more technically relevant improvements, countering the assumption that they would be effective only in simple corrections.

The reduction of technical debt, measured in estimated effort minutes, partially followed the reduction of *issues*, although with variations among different configurations. Stronger correlations were observed when corrections were more concentrated on high-severity *issues*. This observation reinforces the importance of considering multiple quality metrics — both quantitative and qualitative — when evaluating automated improvement processes.

The verification of the preservation of functionality in the improved code was performed manually through the inspection and direct execution of the altered methods. Although no execution failures were identified in the evaluated samples, this strategy does not systematically guarantee semantic equivalence with the original behavior, representing an important limitation to be addressed by more robust validation approaches in future work.

Regarding experimental configurations, the results highlight the importance of synergy between parameters. The Gemini model, combined with five iterations, a well-structured prompt, and a conservative temperature setting (experiment 7), exhibited the best overall performance. GPT-4-mini, in turn, showed greater sensitivity to configuration, with results more dependent on fine-tuning temperature, prompt elaboration, and the number of iterations.

In summary, this study demonstrated that LLMs can be successfully used in automated pipelines for source code improvement, achieving tangible gains in the reduction of problems identified by static analysis tools. The developed pipeline proved essential for systematically structuring the process and enabling iterative experimentation across multiple scenarios. Despite the limitations related to functional validation, the obtained results reinforce the potential of LLM-based solutions as support tools for software development and maintenance, provided they are accompanied by sound engineering practices and appropriate validation mechanisms.

The proposed solution is not only experimentally effective but also practical: it can be integrated into real-world development pipelines to support automated refactoring based on static analysis feedback, reducing developer workload and increasing consistency in code quality improvements.

For future work, the following directions are proposed:

- Incorporating more rigorous functional and semantic validation metrics, such as specification-based tests or enhanced coverage analyses, to ensure behavior preservation;
- Expanding the experiments to different software contexts, including other programming languages, complex frameworks, and codebases with insufficient testing;
- Applying adaptive prompt engineering techniques, allowing the iterative evolution of the instructions provided to the model throughout the improvement process;
- Integrating the approach with version control systems to assess the impact of improvements within real development cycles (e.g., pull requests, merges, human reviews);
- Finally, investigating hybrid strategies that combine static heuristics, human suggestions, and LLM outputs to enhance confidence and explainability in assisted improvement processes.

These future directions aim not only to increase the reliability of LLMs in code improvement tasks but also to broaden their practical applicability in continuous development and real-world system maintenance scenarios.

ARTIFACT AVAILABILITY

All artifacts are available on Zenodo [30].

REFERENCES

[1] G. Ann Campbell and P. Patroklos Papapetrou. 2013. *SonarQube in Action*. Manning Publications.

[2] CodeGen. 2024. CodeGen AI Platform. https://www.codegen.com/. Accessed: 2024-04-18.

[3] Han Cui. 2024. Can large language model replace static analysis tools. In *International Conference on Computer Network Security and Software Engineering (CNSSE 2024)*, Vol. 13175. SPIE, 320–325.

[4] Igor Regis da Silva Simões and Elaine Venson. 2024. Evaluating Source Code Quality with Large Language Models: a comparative study. In *Proceedings of the XXIII Brazilian Symposium on Software Quality*. 103–113.

[5] Sayed Mehdi Hejazi Dehaghani and Nafiseh Hajrahimi. 2013. Which factors affect software projects maintenance cost more? *Acta Informatica Medica* 21, 1 (2013), 63.

[6] Aidan ZH Yang et al. 2024. Revisiting unnaturalness for automated program repair in the era of large language models. *arXiv preprint arXiv:2404.15236* (2024).

[7] Boshi Wang et al. 2023. Towards Understanding Chain-of-Thought Prompting: An Empirical Study of What Matters. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Toronto, Canada, 2717–2739. doi:10.18653/v1/2023.acl-long.153

[8] Danilo Nikolić et al. 2021. Analysis of the tools for static code analysis. In *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE, 1–6.

[9] Greta Dolcetti et al. 2024. Helping LLMs Improve Code Generation Using Feedback from Testing and Static Analysis. *arXiv preprint arXiv:2412.14841* (2024).

[10] Junyi Lu et al. 2023. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 647–658.

[11] Khashayar Etemadi et al. 2022. Sorald: Automatic patch suggestions for sonarqube static analysis violations. *IEEE Transactions on Dependable and Secure Computing* 20, 4 (2022), 2794–2810.

[12] Lakshya A. Agrawal et al. 2023. Monitor-guided decoding of code lms with static analysis of repository context. *Advances in Neural Information Processing Systems* 36 (2023), 32270–32298.

[13] Lishui Fan et al. 2024. Exploring the capabilities of llms for code change related tasks. *ACM Transactions on Software Engineering and Methodology* (2024).

[14] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Mohammad Mahdi Mohajer et al. 2024. Effectiveness of chatgpt for static analysis: How far are we?. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*. 151–160.

[16] Maosheng Zhong et al. 2023. Codegen-test: An automatic code generation model integrating program test information. In *2023 2nd International Conference on Cloud Computing, Big Data Application and Software Engineering (CBASE)*. IEEE, 341–344.

[17] Qianou Ma et al. 2024. How to teach programming in the ai era? using llms as a teachable agent for debugging. In *International Conference on Artificial Intelligence in Education*. Springer, 265–279.

[18] Quanjun Zhang et al. 2024. A systematic literature review on large language models for automated program repair. *arXiv preprint arXiv:2405.01466* (2024).

[19] Shraddha Barke et al. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Commun. ACM* 66, 6 (2023), 106–115.

[20] Steven I Ross et al. 2023. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 491–514.

[21] Toufique Ahmed et al. 2023. Improving few-shot prompts with relevant static analysis products. *arXiv preprint arXiv:2304.06815* (2023).

[22] Xinyun Chen et al. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[23] Xinyi Hou et al. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[24] Yufan Cai et al. 2025. Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2057–2089.

[25] Yu Hao et al. 2023. E&v: Prompting large language models to perform static analysis by pseudo-code execution and verification. *arXiv preprint arXiv:2312.08477* (2023).

[26] Yue Wang et al. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 8696–8708.

[27] Zhangyin Feng et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1536–1547.

[28] Maryam Al-Hitmi Fida Zubair and Cagatay Catal. 2024. The use of large language models for program repair. *Computer Standards & Interfaces* (2024), 103951.

[29] GitHub. 2024. GitHub Copilot. https://github.com/features/copilot. Accessed: 2024-04-18.

[30] João Carlos Gonçalves and Marcelo de Almeida Maia. 2025. Data of An Empirical Study on the Effectiveness of Iterative LLM-Based Improvements for Static Analysis Issues. doi:10.5281/zenodo.15278368

[31] Hugging Face. 2023. StarCoder and StarCoderBase: The next generation of code LLMs. https://huggingface.co/blog/starcoder. Accessed: 2024-04-18.

[32] Michael W. Hicks Jeffrey S. Foster and William Pugh. 2007. Improving software quality with static analysis. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 83–84.

[33] Aman Madaan. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2023), 46534–46594.

[34] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139.

[35] Meta. 2023. Code Llama: An Open Reproduction of Code-Related Large Language Models. https://ai.meta.com/blog/code-llama-large-language-model-coding/. Accessed: 2024-04-18.

[36] Douglas C. Montgomery. 2017. *Design and analysis of experiments*. John wiley & sons.

[37] OpenAI. 2021. OpenAI Codex. https://openai.com/index/openai-codex/. Accessed: 2024-04-18.

[38] OpenAI. 2023. GPT-4 Technical Report. OpenAI Research. https://openai.com/research/gpt-4.

[39] OpenAI. 2024. GPT-3 Apps: Applications Powered by OpenAI's GPT-3. https://openai.com/index/gpt-3-apps/. Accessed: 2024-04-18.

[40] OpenAI. 2024. GPT-4 by OpenAI. https://openai.com/index/gpt-4/. Accessed: 2024-04-18.

[41] Yuhao Wang Shuyang Jiang and Yu Wang. 2023. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907* (2023).

[42] SonarSource. 2024. SonarSource - Continuous Code Quality. https://www.sonarsource.com/ Accessed: 2024-04-18.

[43] Jones Yeboah and Saheed Popoola. 2023. Efficacy of static analysis tools for software defect detection on open-source projects. In *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 1588–1593.