ISSN: 2633-4828 Vol. 6 No.2, June, 2024

International Journal of Applied Engineering & Technology

SONARQUBE-DRIVEN QUALITY GATES: IMPROVING SOFTWARE INTEGRITY THROUGH AUTOMATED CODE REVIEWS

Pathik Bavadiya
Vice President, Production Services (Independent Researcher) BNY, New York, USA
pathikbavadiya1900@gmail.com
ORCID: 0009-0003-4405-3657

ABSTRACT
The continual monitoring of code quality and the adherence to best practices are both necessary components in modern development environments for the purpose of ensuring the integrity of software. The objective of this study is to reduce technical debt, improve maintainability, and enhance overall software reliability. To achieve these goals, the study analyzes a SonarQube-driven approach to enforcing quality gates as part of automated code reviews. In order to enforce predetermined quality levels that were generated from the SQALE model and ISO 9126 standards, SonarQube, which is an open-source static code analysis platform that supports several programming languages, was included into a continuous integration and continuous delivery (CI/CD) pipeline. After six weeks of quality gate enforcement, an experimental evaluation was carried out on five software projects of varied sizes. The evaluation involved comparing the baseline quality metrics with the outcomes obtained after the implementation of quality gate. Significant advancements have been demonstrated by the findings, which include bug reductions of up to one hundred percent in smaller projects, reductions of up to fifty percent in code smell, and reductions of over forty percent in technical debt throughout all stages of project development. In addition, maintainability indices showed significant improvement, with the highest increases being recorded in projects that were relatively large in scale. These results demonstrate that SonarQube-driven quality gates are capable of efficiently preventing the introduction of code of poor quality, fostering consistent adherence to coding standards, and promoting behaviors that are sustainable for software development.

Keywords*: SonarQube, Quality Gates, Software Integrity, Automated Code Review, Static Code Analysis.*

1. INTRODUCTION

The maintenance of a high code quality is not merely a recommended practice in contemporary software development; rather, it is an absolute requirement for assuring the long-term maintainability, scalability, and security of the product. The development efficiency and product reliability of applications can be greatly hindered by undetected flaws, excessive technical debt, and poor maintainability. This is especially dangerous when programs continue to grow in size and complexity. Although traditional code review procedures are beneficial, they frequently require a significant amount of time, are susceptible to human oversight, and have a limited capacity to enforce quality standards that are consistent across a wide range of teams and projects. As a result of this gap, automated quality assurance technologies that integrate smoothly into the development lifecycle have become increasingly popular.

SonarQube has established itself as a leading platform in this field by providing team members with automated static code analysis and quality gates that can be customized. These features assist teams in identifying and resolving issues at an earlier stage in the development process. To be promoted to production or merged into the main branch, code must first satisfy quality gates in SonarQube, which are specified criteria that must be met before the code can be promoted. The thresholds for code coverage, bug density, code smells, technical debt, and maintainability index are some of the metrics that are frequently included in these criteria. Through the implementation of these gates, organizations have the opportunity to guarantee that only code that satisfies the quality criteria that have been agreed upon will be allowed to proceed, hence lowering the likelihood of introducing vulnerabilities or maintainability difficulties.

1.1. Objectives of the Study

- To evaluate the effectiveness of SonarQube-driven quality gates in improving software integrity through automated code reviews.
- To analyze the impact of quality gate enforcement on code metrics such as bugs, code smells, technical debt, and maintainability index across projects of varying sizes.

2. LITERATURE REVIEW

Kuo, et al. (2020) performed research on automated program quality analysis and correction, with a particular emphasis on the incorporation of static analysis tools into software development workflows along the course of the study. Their research highlighted the importance of automated methods in identifying errors, code smells, and maintainability difficulties at an early point in the development process. It was demonstrated by the authors that automated quality analysis may greatly reduce the amount of time spent on manual inspection while simultaneously enhancing the overall quality of software. This was accomplished by applying a systematic method that utilized established quality measures and correction mechanisms. The findings of this study showed the potential of automation to improve both the efficiency and uniformity of code quality assurance.

Mendes (2023) Sonarch, a static analysis plugin that was developed exclusively for Spring-based apps, was incorporated into the SonarQube platform when it was developed. The tool improved SonarQube's analysis capabilities by focusing on framework-specific code patterns. This made it possible to identify problems that are specific to Spring projects. Sonarch was evaluated on a number of different case studies, and the results demonstrated that it was able to effectively identify vulnerabilities and code quality issues that generic analyzers frequently fail to notice. The study that was done led to the improvement of framework-specific code quality assessment within ecosystems that are used for static analysis.

Abeysinghe, et al. (2021) Secure CodeCity is a visualization framework that was created for the purpose of finding and controlling security vulnerabilities within software systems. The framework offered developers an easy-to-understand method for locating security flaws in big codebases by the utilization of metaphors based on three-dimensional code cities. According to the findings of the study, providing developers with visual representations of vulnerabilities increased both their comprehension and their response times when compared to typical text-based reports. The findings of this study highlighted the importance of security-focused and static code analysis methodologies that utilize visualization techniques.

Vourou (2023) the strengthening of application security was investigated via the lens of DevSecOps, with a specific emphasis placed on vulnerability identification and management within pipelines for continuous integration and continuous delivery (CI/CD). In the course of the research, a number of different security scanning technologies were incorporated into automated build processes. This made it possible to conduct vulnerability assessments and implement security policies in real time. The findings suggested that including security checks into continuous integration and continuous delivery pipelines helped to build a proactive security culture among development teams and reduced the amount of vulnerable code that was introduced into production environments. This effort was undertaken in accordance with the growing trend in the industry toward incorporating security measures into each and every stage of the software development lifecycle.

3. RESEARCH METHODOLOGY

This section provides an overview of the methodical approach that was utilized to evaluate the efficacy of SonarQube-driven quality gates. It provides specifics regarding the experimental setup, data collection methods, evaluation metrics, and analysis techniques that were utilized to measure improvements in software quality across a variety of project scales.

3.1. Research Design

For the purpose of determining whether or whether SonarQube-driven quality gates are efficient in enhancing software integrity through automated code reviews, this study utilized an experimental research approach. Establishing baseline quality measures, creating SonarQube quality gates based on established software quality models, and monitoring the improvement in code quality after implementation over a predetermined amount of time were all components of the approach.

3.2. Study Environment and Tools

The experiments were conducted within a controlled development environment using the following tools and technologies:

- **SonarQube (v.9.x) –** For static code analysis, quality gate configuration, and continuous monitoring.
- **SonarQube Scanner –** To integrate static analysis into the CI/CD pipeline.
- **Jenkins –** As the automation server for executing build and analysis tasks.
- **Programming Languages –** Java, Python, and JavaScript, to ensure multi-language applicability.
- **Quality Models –** SQALE model and ISO 9126 guidelines to define and evaluate quality attributes such as maintainability, reliability, and efficiency.

3.3. Project Selection

A total of five projects were selected, representing different codebase sizes and domains:

- **Small-scale:** LOC < 10,000
- **Medium-scale:** LOC 10,000–50,000
- **Large-scale:** LOC > 50,000

Three projects were open-source repositories, while two were in-house enterprise applications.

Phase 1: Baseline Measurement

Before quality gate enforcement, static analysis was performed on all projects to collect baseline metrics, including:

- Number of Bugs
- Code Smells
- Technical Debt (hours)
- Maintainability Index

These measurements established a reference point for subsequent comparison.

Phase 2: Quality Gate Configuration

SonarQube quality gates were configured using a combination of SQALE recommendations and ISO 9126 criteria. The predefined thresholds included:

- Bugs ≤ 0 on new code
- Code Smells $\leq 5\%$ of total LOC
- Test Coverage $\geq 80\%$ on new code
- Duplicated Lines $\leq 3\%$
- Maintainability Index ≥ 70

Quality gates were set to automatically fail builds that did not meet these thresholds.

Phase 3: Integration into CI/CD Pipeline

Each project's Jenkins continuous integration and continuous delivery pipeline was modified to include the SonarQube analysis process. As soon as a commit was made, an automated build and static analysis were initiated. Modifications to the code that were not approved by quality gate checks were not implemented until the problems were fixed. Developers were provided with feedback in real time through the use of SonarQube dashboards.

Phase 4: Post-Implementation Measurement

Following a continuous enforcement period of six weeks, the same collection of metrics was gathered from each and every construction project. The effectiveness of the quality gates was evaluated by calculating the amounts of change in the number of bugs, the smell of code, the amount of technical debt, and the maintainability index.

3.4. Data Analysis

A comparison was made between the metrics that were used before and after the implementation in order to determine the percentage of improvement for each project size. Quantifying reductions in defects, code smells, and technical debt, as well as gains in maintainability, was the primary emphasis of the investigation. For the purpose of determining the total influence that SonarQube-driven quality gates have on software quality, this data was summarized in tabular form and interpretation was performed.

4. RESULTS AND DISCUSSION

This section presents the outcomes of implementing SonarQube-driven quality gates across software projects of varying sizes, comparing pre- and post-enforcement metrics to evaluate their impact on code quality.

Table 1: Baseline Metrics before Quality Gate Enforcement

Project Size	LOC	Bugs	Code Smells	Technical Debt (hrs)	Maintainability Index
Small	8,560	12	248	38	65
Medium	32,450	38	1,024	115	58
Large	76,890	92	2,846	327	52

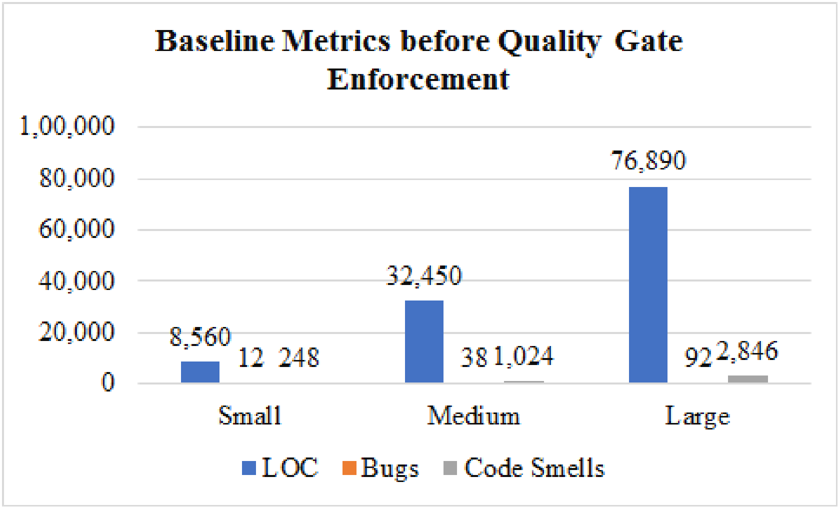

Figure 1: Baseline Metrics before Quality Gate Enforcement

The software quality metrics that were considered baseline prior to the implementation of SonarQube-driven quality gates are presented above in Table 1. The findings reveal that larger projects had much higher numbers of errors, code smells, and technical debt in comparison to smaller projects. This is a reflection of the increased complexity and maintenance issues that are associated with larger codebases. To be more specific, the large-scale project recorded 92 defects, 2,846 code smells, and 327 hours of technical debt. Additionally, it had the lowest maintainability score of 52, which indicates that the code structure was bad and that the long-term maintenance costs were higher. Small-scale projects had the fewest problems, recording 12 bugs, 248 code smells, and 38 hours of technical debt, coupled with a considerably higher maintainability index of 65. Medium-scale projects showed moderate defect levels, with 38 bugs, 1,024 code smells, and 115 hours of technical debt. On the other hand, small-scale projects had the fewest troubles. These findings from the baseline demonstrate that there is a direct correlation between the scale of the project and the difficulties associated with the quality of the code. Furthermore, they highlight the necessity of automated quality enforcement systems in order to control the rise of defects and maintain maintainability.

Table 2: Metrics after Quality Gate Enforcement (6 Weeks)

Project Size	Bugs	Code Smells	Technical Debt (hrs)	Maintainability Index
Small	0	124	21	78
Medium	5	612	69	72
Large	15	1,765	204	66

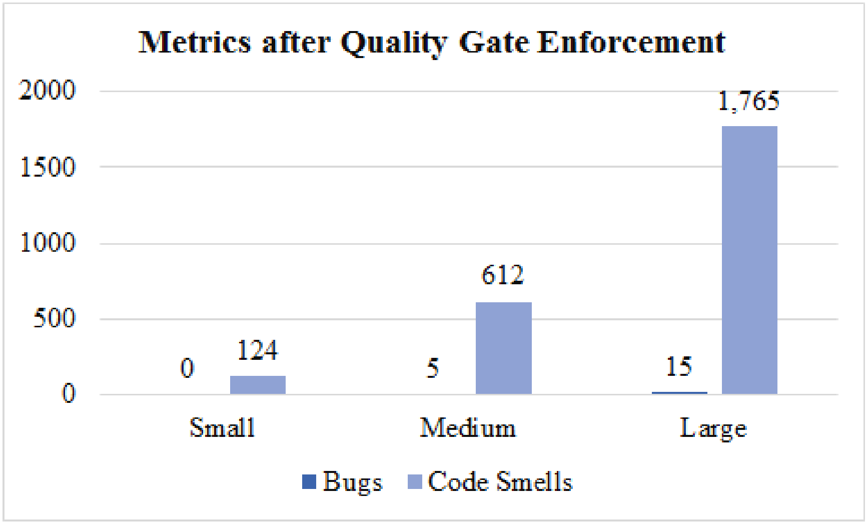

Figure 2: Metrics after Quality Gate Enforcement (6 Weeks)

Following a period of six weeks during which SonarQube-driven quality gates were enforced, the software quality metrics that were recorded are present in Table 2. In all of the metrics that were measured, the results show that there was a significant improvement across all of the project sizes. Because of the small-scale project, all of the defects were removed completely, the number of code smells was cut down by nearly half to 124, and the amount of technical debt was reduced to 21 hours. Additionally, the maintainability index increased to 78, which indicates that the codebase has become more robust and easier to maintain. Moreover, the medium-scale project demonstrated great progress, as evidenced by the reduction of defects to just five, the reduction of code smells to six hundred twelve, the reduction of technical debt to sixty-nine hours, and the major improvement of the maintainability index to seventy-two. These figures, which were obtained after the installation, demonstrate that automated quality gate enforcement is effective in decreasing faults, minimizing maintenance overhead, and enhancing code organization in a reasonably short amount of time.

Table 3: Percentage Improvement

Metric	Small Project	Medium Project	Large Project
Bug Reduction (%)	100%	86.8%	83.7%
Code Smell Reduction (%)	50%	40.2%	38%

Technical Debt Reduction (%)	44.7%	40%	37.6%
Maintainability Index Improvement (%)	20%	24.1%	26.9%

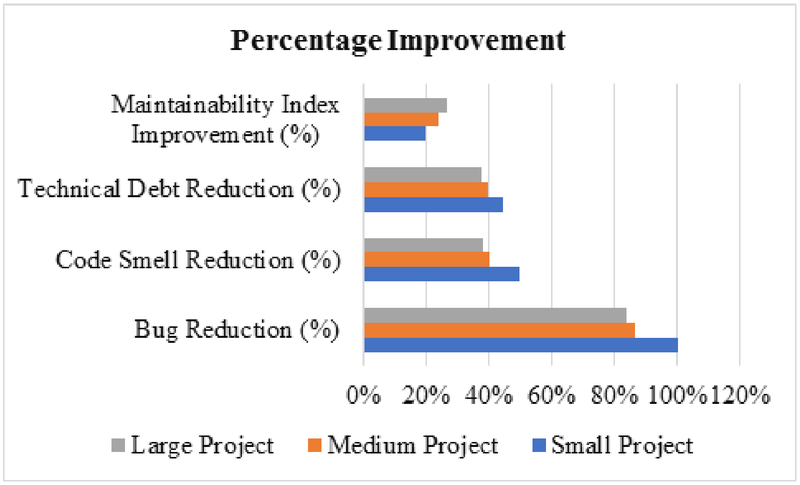

Figure 3: Percentage Improvement

Following the introduction of SonarQube-driven quality gates, the percentage improvement in software quality measures is displayed in Table 3, which may be seen here. The findings indicate that the project with the smallest scale achieved the most significant decrease in the number of bugs, which includes the entire elimination of all flaws that were reported (100 percent). Additionally, the medium-sized and large-scale projects displayed noteworthy reductions of 86.8 percent and 83.7%, respectively. The reduction in code smell was most obvious in the small project, which was at 50%. This was in contrast to the medium project, which had a reduction of 40.2%, and the large project, which had a reduction of 38%. This suggests that smaller codebases respond more successfully to quality enforcement. The reduction of technical debt followed a similar pattern, with drops of 44.7% occurring in small projects, 40% occurring in medium projects, and 37.6% occurring in large projects, respectively. It is interesting to note that the maintainability index improvement was highest in the large-scale project, which was 26.9%. This suggests that even modest defect reductions in large systems can greatly enhance overall maintainability. SonarQube quality gates have the capacity to deliver measurable quality increases across projects of varying sizes, with notably strong outcomes in smaller codebases. These improvements collectively illustrate the capability of SonarQube quality gates to do so.

5. CONCLUSION

According to the findings of this study, incorporating SonarQube-driven quality gates into the workflow of software development can considerably improve the quality of code across a wide range of projects of varied sizes. The strategy guaranteed that only code that complied with the criteria that were agreed upon was merged into the main branch by imposing predefined thresholds for bugs, code smells, technical debt, and maintainability index. The findings demonstrated that all bugs were eradicated entirely in smaller projects, that there was a significant reduction in code smells and technical debt, and that there were noteworthy gains in maintainability. These advantages were especially significant in large-scale systems, where even moderate defect reductions are significant. These findings shed light on the significance of automated static analysis tools in contemporary continuous integration and continuous delivery environments, which help to cultivate a culture of continuous quality improvement and early problem detection. Significant improvements were made to larger projects as well, despite the fact that smaller projects gained the most from the reduction in complexity. In the future, research may investigate the long-term benefits of quality gate enforcement on development pace, defect recurrence, and developer adoption. This would confirm that SonarQube-driven quality gates offer a solution that is both scalable and successful for enhancing software integrity.

REFERENCES

1. J. Y. Kuo, H. C. Lin, and H. S. Yu, "Study of Automatic Programs Quality Analysis and Correction," in *Proc. 2020 Int. Comput. Symp. (ICS)*, Dec. 2020, pp. 503–507.
2. R. L. M. C. Mendes, *Sonarch-A Static Spring Code Analysis Plugin for SonarQube*, M.S. thesis, Universidade NOVA de Lisboa, Portugal, 2023.
3. A. T. G. Abeysinghe, M. A. S. Shalika, M. S. N. Ahamed, and S. M. Mufarrij, *Secure CodeCity: A Framework for Security Vulnerability Visualization*, Doctoral dissertation, 2021.
4. P. Vourou, *Enhancing Application Security through DevSecOps: A Comprehensive Study on Vulnerability Detection and Management in Continuous Integration and Continuous Delivery Pipelines*, M.S. thesis, Πανεπιστήμιο Πειραιώς, 2023.
5. E. Viitasuo, *Adding Security Testing in DevOps Software Development with Continuous Integration and Continuous Delivery Practices*, 2020.
6. R. Manchana, "The DevOps Automation Imperative: Enhancing Software Lifecycle Efficiency and Collaboration," *Eur. J. Adv. Eng. Technol.*, vol. 8, no. 7, pp. 100–112, 2021.
7. E. A. AlOmar, M. Chouchen, M. W. Mkaouer, and A. Ouni, "Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack," in *Proc. 19th Int. Conf. Mining Softw. Repositories*, May 2022, pp. 689–701.
8. P. Haindl and R. Plösch, "Value-Oriented Quality Metrics in Software Development: Practical Relevance from a Software Engineering Perspective," *IET Softw.*, vol. 16, no. 2, pp. 167–184, 2022.
9. Y. Ramaswamy, "DevSecOps in Practice: Embedding Security Automation into Agile Software Delivery Pipelines," *J. Comput. Anal. Appl.*, vol. 31, no. 4, 2023.
10. V. Tortoriello, *Definition of a DevSecOps Operating Model for Software Development in a Large Enterprise*, Doctoral dissertation, Politecnico di Torino, 2022.