https://doi.org/10.31891/2219-9365-2026-87-36
UDC 637.5.02

MAKIEIEV Oleksii
Kharkiv National University Of Radioelectronics
https://orcid.org/0009-0002-7909-4793
e-mail: oleksii.makieiev@nure.ua
KRAVETS Natalia
Kharkiv National University Of Radioelectronics
https://orcid.org/0000-0002-6753-3333
e-mail: natalia.kravets@nure.ua

STUDY OF VARIABILITY OF ARCHITECTURAL METRICS DURING THE EVOLUTION OF A SOFTWARE SYSTEM IN CI/CD CONDITIONS

Ensuring the architectural quality of software systems in continuous development is an urgent task of modern software engineering. The work is devoted to the development of a method for quantitatively assessing the metrics of the evolutionary architecture of software systems in continuous integration and delivery (CI/CD). An approach is proposed to construct a generalized fitness function that aggregates six structural quality metrics - SpotBugs violations, critical violations, line coverage by tests, code smells, line duplication density and Checkstyle violations - into a single quantitative assessment that is calculated automatically at each execution of the CI/CD pipeline. The pipeline normalizes each component by comparing the current value of the metric with the base value of the main branch stored in the cloud database. The weighting coefficients of the metrics are determined in accordance with the classification of quality characteristics priorities according to the ISO/IEC 25023 standard, where reliability, testability and maintainability receive relative priorities, respectively. An algorithm for integrating the architectural metrics assessment process into a CI/CD pipeline based on GitHub Actions has been developed, which ensures timely detection of signs of architectural degradation for each pull request and on schedule for the main branch. The approach has been experimentally tested on a real monolithic Java/Spring Boot system within 12 controlled scenarios covering improvement, degradation, and neutral changes in architectural quality. Statistical analysis confirmed the ability of the fitness function to qualitatively distinguish between different types of architectural changes. Comparison with the SonarQube Quality Gate mechanism showed the advantage of the proposed method: unlike the binary "pass/fail" signal, the fitness function quantifies the direction and magnitude of the variability of architectural metrics and avoids false positives in purely structural refactoring.

Keywords: evolutionary architecture, fitness functions, architectural metrics, CI/CD, DevOps, software quality.

МАКЄЄВ Олексій, КРАВЕЦЬ Наталя
Харківський національний університет радіоелектроніки

ДОСЛІДЖЕННЯ ВАРІАБЕЛЬНОСТІ АРХІТЕКТУРНИХ МЕТРИК ПІД ЧАС ЕВОЛЮЦІЇ ПРОГРАМНОЇ СИСТЕМИ В УМОВАХ CI/CD

Забезпечення архітектурної якості програмних систем в умовах безперервного розвитку є актуальним завданням сучасної програмної інженерії. Роботу присвячено розробці методу кількісного оцінювання метрик еволюційної архітектури програмних систем в умовах безперервної інтеграції та доставки (CI/CD). Запропоновано підхід до побудови узагальненої функції придатності, що агрегує шість структурних метрик якості - порушення SpotBugs, критичні порушення, покриття рядків тестами, запахи коду, щільність дублювання рядків та порушення Checkstyle - в єдину кількісну оцінку, що обчислюється автоматично при кожному виконанні CI/CD-конвеєра. Конвеєр нормалізує кожну компоненту шляхом порівняння поточного значення метрики з базовим значенням основної гілки, що зберігається у хмарній базі даних. Вагові коефіцієнти метрик визначено відповідно до класифікації пріоритетів характеристик якості за стандартом ISO/IEC 25023, де надійність, тестованість та супроводжуваність отримують відносні пріоритети відповідно. Розроблено алгоритм інтеграції процесу оцінювання архітектурних метрик у CI/CD-конвеєр на базі GitHub Actions, що забезпечує своєчасне виявлення ознак деградації архітектури при кожному pull request та за розкладом для основної гілки. Проведено експериментальну перевірку підходу на реальній монолітній системі на основі Java/Spring Boot у межах 12 контрольованих сценаріїв, що охоплюють покращення, деградацію та нейтральні зміни архітектурної якості. Статистичний аналіз підтвердив здатність функції придатності розрізняти якісно різні типи архітектурних змін. Порівняння з механізмом SonarQube Quality Gate показало перевагу запропонованого методу: на відміну від бінарного сигналу «пройдено/не пройдено», функція придатності кількісно визначає напрям та величину варіабельності архітектурних метрик і уникає хибнопозитивних результатів при суто структурному рефакторингу.

Ключові слова: еволюційна архітектура, функції придатності, архітектурні метрики, CI/CD, DevOps, якість програмного забезпечення.

Стаття надійшла до редакції / Received 11.06.2026
Прийнята до друку / Accepted 04.08.2026
Опубліковано / Published 10.09.2026

FORMULATION OF THE PROBLEM

The evolution of software systems is inevitable in the conditions of modern development, based on agile methodologies and DevOps practices. Intensive code changes, frequent releases and constant expansion of functionality can lead to architectural degradation - a gradual decrease in the structural quality of the system. In such

conditions, it is especially important to timely detect changes that negatively affect the architecture and make decisions about its maintenance or refactoring.

However, traditional approaches to assessing architectural integrity are manual or episodic and are not integrated into CI/CD[1] processes. This reduces the effectiveness of responding to potential architectural risks. The presence of formal, quantitative metrics that allow automatically assessing the state of the architecture is a critical condition for ensuring long-term stability and quality of software.

Thus, the task of developing a methodology for continuous monitoring of the architectural characteristics of a software system based on structural metrics that are integrated into the CI/CD pipeline and allow for rapid tracking of variations and prevention of architectural defects is relevant. This approach allows solving a number of important scientific and practical problems in maintaining software quality in conditions of dynamic development.

ANALYSIS OF RESEARCH AND PUBLICATIONS

Researchers have examined the problem of continuous architectural quality monitoring in evolving software systems from several complementary directions: empirical analysis of structural anti-patterns, systematic study of architectural technical debt, automation of architectural governance in CI/CD pipelines, and the limitations of existing binary quality control mechanisms.

Jolak et al. [2] conducted a large-scale empirical study of the impact of architectural smells on software maintainability across 378 versions of eight open-source projects. It was found that architectural smells negatively correlate with quality attributes, particularly testability and modularity. This finding motivates the need for continuous, iteration-level structural monitoring, since degradation accumulates incrementally and must be detected early rather than through periodic reviews.

Sousa et al. [3] conducted a systematic review of 70 works on architectural technical debt, identifying its main types and existing measurement approaches. The authors conclude that architectural debt significantly impacts the long-term maintainability of evolving systems, yet unified methods for assessing its cost and defining threshold-based repayment strategies remain absent. This gap underscores the need for automated, pipeline-integrated mechanisms that track architectural quality changes at each development iteration.

Kirschner et al. [4] present the Retriever tool, which integrates automated architectural model generation and verification directly into the CI/CD pipeline, enabling deviations to be detected automatically at each integration cycle. This provides a practical precedent for embedding architectural governance into continuous development workflows.

SonarQube Quality Gate [5] is one of the most widely adopted binary quality control mechanisms in CI/CD pipelines, providing automated pass/fail verdicts based on configurable metric thresholds. However, as a binary decision mechanism, it cannot quantify the magnitude or direction of quality changes, making it impossible to distinguish between marginal and critical regressions.

Khatsko et al. [6] present a modified software build algorithm for AWS CodeBuild that uses multithreading, formally modelled using timed automata and Petri nets. The study demonstrates that standard parallelization mechanisms in cloud-based build environments can lead to suboptimal resource utilization - increasing build time from 211 s to 765 s under naive parallel execution. In contrast, the proposed Python-based multithreading approach reduced it to 73 s. This work confirms that metric measurement within CI/CD pipelines must account for the inherent variability introduced by the build process itself, which is directly relevant to the present study.

Thus, the reviewed works confirm a clear trend toward integrating architectural quality monitoring into CI/CD processes. However, three significant gaps persist. First, there is a lack of generally accepted criteria for the comprehensive automation of architectural control that applies uniformly across different system types within a single pipeline. Second, the field lacks established threshold values for pipeline-integrated quality gates - existing tools either produce binary pass/fail signals or rely on manually configured, system-specific rules. Third, empirical verification of continuous architectural monitoring methods remains limited to isolated cases. The present study addresses all three gaps by proposing a fitness function with formally defined weighted metrics, normalized threshold values anchored to the main branch baseline, integrated into a CI/CD pipeline, and validated empirically across a controlled set of experimental scenarios on a real monolithic software system.

PURPOSE OF THE STUDY

The purpose of this study is to develop a method for quantitative assessment of evolutionary architecture metrics of software systems within a CI/CD pipeline. The method relies on a generalized fitness function that aggregates six structural quality metrics - SpotBugs violations, critical violations, line coverage, code smells, duplicated lines density, and Checkstyle violations - into a single score, computed automatically at each pipeline execution. Metric weights follow the priority classification defined by ISO/IEC 25023[7]. The proposed method enables automated monitoring of architectural metric variability and supports architectural integrity during system evolution. The proposed method is compared against the SonarQube Quality Gate to demonstrate its advantage over binary pass/fail quality control in quantifying the magnitude and direction of architectural metric variability.

PRESENTATION OF THE MAIN MATERIAL

Architectural metric variability relates to measurable changes in structural quality indicators - such as coupling, complexity, test coverage, and code duplication - that occur throughout successive iterations of a software system's evolution. To continuously monitor such variability and identify architectural degradation before it becomes critical, a fitness function-based monitoring approach integrated into the CI/CD pipeline was developed.

The generalized fitness function F aggregates six structural metrics into a single quantitative score computed automatically at each pipeline execution and is defined as follows:

$$F(x) = a_1 \cdot F_1(x) + a_2 \cdot F_2(x) + a_3 \cdot F_3(x) + a_4 \cdot F_4(x) + a_5 \cdot F_5(x) + a_6 \cdot F_6(x), \quad (1)$$

where $F(x)$ - generalized fitness function, $F_1(x)$ – normalized SpotBugs[8] violations score, $F_2(x)$ – normalized critical violations score (SonarCloud[9]), $F_3(x)$ – normalized line coverage score (JaCoCo[10]), $F_4(x)$ – normalized code smells score (SonarCloud), $F_5(x)$ – normalized duplicated lines density score (SonarCloud), $F_6(x)$ – normalized Checkstyle violations score, $a=(a_1, a_2,..., a_6)$, a_i can only take values from (0,1) – weighting coefficients.

The method does not prescribe fixed values for the weighting coefficients a_i. In this study, the coefficients are selected to reflect the relative importance of software quality characteristics for the specific system under study. This structure supports recalibration for different system types or stakeholder priorities without altering the overall method. In this work, the weights assigned to each component of the fitness function reflect the priority classification of software quality characteristics defined by ISO/IEC 25023. SpotBugs violations $F_1(x)$ and critical violations $F_2(x)$ directly affect system reliability and together account for a combined weight of 0.50. Line coverage $F_3(x)$ is the primary indicator of testability and carries the highest individual weight of 0.30. The remaining three metrics — code smells $F_4(x)$, duplicated lines density $F_5(x)$, and Checkstyle[11] violations $F_6(x)$ — reflect maintainability concerns and are distributed with weights of 0.07, 0.07, and 0.06, respectively, with a combined weight of 0.20. Thus we get $a=(0,25;\ 0,25;\ 0,3;\ 0,07;\ 0,07;\ 0,06)$.

Each component $F_n(x)$ is computed using metric values collected automatically at each CI/CD pipeline execution, comparing the current branch against the main branch baseline and normalizing the result to the range [-1, 1]:

$$F_n(x) = \begin{cases} \dfrac{b-c}{b}, if\ -1 \le \dfrac{b-c}{b} \le 1; \\ -1, if\ \dfrac{b-c}{b} < -1; \\ 1, if\ \dfrac{b-c}{b} > 1; \\ 0, if b = 0 \end{cases}, \quad (2)$$

where b – baseline value (main branch), c – current metric value.

For metrics where lower values indicate better quality – such as violations and code smells – an improvement produces a positive score and a regression produces a negative one. For line coverage, where a higher value is desirable, the direction is inverted accordingly. Three boundary conditions are defined:

1. if both baseline and current values are zero, $F_n(x) = 0$;
2. if the baseline is zero but the current value is positive, the score is set to +1 or -1 depending on the metric direction;
3. if the normalized delta exceeds the range [-1, 1], it is clamped to the nearest boundary value.

The proposed methodology is implemented as an automated GitHub Actions[12] pipeline triggered on every pull request and on a daily scheduled basis for the main branch.

Figure 1 illustrates the main steps of the CI/CD pipeline for computing the generalized fitness function. The first two steps execute static analysis tools — Checkstyle, SpotBugs, and unit tests — and SonarCloud respectively, producing structured reports as output. The third step runs the metric collector script that collects data from reports that were generated by static analyzers. In the final step, the computed fitness score and raw metric values are saved to CosmosDB for baseline comparison in subsequent pipeline runs.

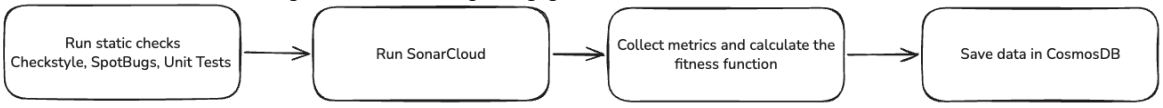

Fig. 1. Main steps in GitHub Actions

The metrics used in the fitness function are collected from two distinct sources during pipeline execution. SonarCloud metrics - critical violations, code smells, and duplicated lines density - are fetched live via the SonarCloud API during each run, as SonarCloud maintains its own historical analysis. SpotBugs violations, Checkstyle violations, and line coverage are read from XML reports generated in the current run and compared against the latest main branch snapshot stored in Cosmos DB[13]. This two-source design ensures that all six metrics are always available regardless of which branch is being analyzed.

For pull request runs, the fitness score is computed by comparing against the latest main branch snapshot. For scheduled main branch runs, the comparison is made against the previous main branch snapshot, allowing continuous tracking of architectural metric variability over time.

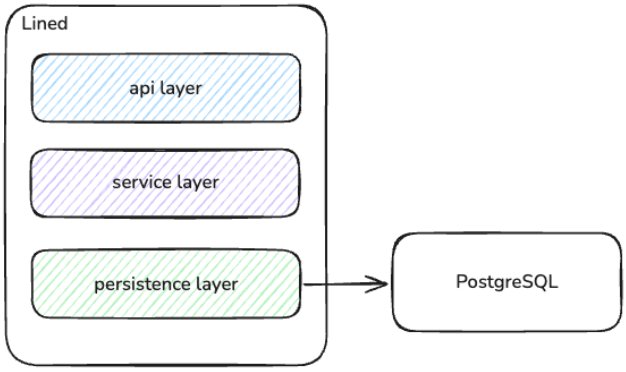

Fig. 2. High-level architecture of the "Lined" experimental system

The system follows a three-tier layered architecture consisting of an API layer, a service layer, and a persistence layer backed by a PostgreSQL[14] database, as illustrated in Figure 2. This layered structure ensures a clear separation of concerns, making the impact of code changes on individual architectural metrics traceable and unambiguous across experiments. To verify the proposed methodology, a real-world monolithic backend system named "Lined" was selected as the experimental subject. The system is implemented using Java 17 and Spring Boot, and consists of eight domain modules: user, role, plan, subscription, lobby, event, task, and application service. A monolithic architecture was chosen over a microservices approach to simplify metric collection and eliminate inter-service communication noise in the measurements. The codebase contains 136 unit tests covering all service layer components, written using JUnit 5 and Mockito. Static analysis is performed using Checkstyle and SpotBugs, test coverage is measured with JaCoCo, and code quality metrics are collected via SonarCloud.

A total of 12 controlled experiments were designed to verify the sensitivity of the fitness function to different types of architectural changes. Each scenario introduces exactly one type of change to isolate its effect on the fitness score. The scenarios are grouped into three categories: improvement (expected $F > 0$), degradation (expected $F < 0$), and neutral (expected $F \approx 0$). Table 1 presents the full list of experimental scenarios.

Table 1

Experimental scenarios for fitness function verification

№	Branch	Scenario	Category	Expected F
1	experiment/improve-coverage	Add 136 unit tests	Improvement	$F > 0$
2	experiment/improve-spotbugs	Fix SpotBugs violations	Improvement	$F > 0$
3	experiment/reduce-duplication	Reduce code duplication via refactoring	Improvement	$F > 0$
4	experiment/fix-checkstyle	Fix Checkstyle violations	Improvement	$F > 0$
5	experiment/degrade-spotbugs	Introduce deliberate SpotBugs patterns	Degradation	$F < 0$
6	experiment/degrade-checkstyle	Introduce Checkstyle violations	Degradation	$F < 0$
7	experiment/degrade-duplication	Increase duplication by copy-paste	Degradation	$F < 0$
8	experiment/degrade-coverage	Remove unit tests	Degradation	$F < 0$
9	experiment/neutral-rename	Pure refactor - rename/extract method	Neutral	$F \approx 0$
10	experiment/neutral-move-packages	Move code between packages	Neutral	$F \approx 0$
11	experiment/neutral-documentation	JavaDoc changes only	Neutral	$F \approx 0$
12	experiment/neutral-test-refactor	Rename test helpers, no coverage change	Neutral	$F \approx 0$

The results of all 12 controlled experiments are summarized in Table 2. The fitness function produced positive scores for all four improvement scenarios, with a mean F of 0.4379 and individual values ranging from +0.300 to +0.570.

The SonarQube Quality Gate was selected as the baseline comparison method because it represents the most widely adopted automated quality control mechanism in CI/CD pipelines, providing a standardized binary pass/fail verdict based on configurable metric thresholds. Unlike the proposed fitness function, which produces a continuous score reflecting the magnitude and direction of metric variability, the Quality Gate cannot distinguish between marginal and significant architectural changes. This fundamental difference makes it a suitable baseline for evaluating the diagnostic granularity of the proposed method.

All improvement scenarios received a SonarQube Quality Gate status of OK. Among the four degradation scenarios, only one - experiment/degrade-duplication - produced the expected negative score (F = −0.307), while the remaining three produced small positive values (0.020-0.142), reflecting mild degradations that high line coverage on newly added code in the same pull request partially offset. All four degradation scenarios were correctly flagged as ERROR by the SonarQube Quality Gate.

Neutral scenarios produced a mean F score of 0.2423, which exceeds the expected near-zero range. SonarCloud PR-scoping explains this systematic positive bias: critical violations, code smells, and duplicated lines density are evaluated only on changed files within the pull request, not across the full project, introducing a positive offset for pull requests that touch clean code. The exception was experiment/neutral-move-packages, which produced F = 0.009 - correctly near zero - while the SonarQube Quality Gate incorrectly flagged it as ERROR, demonstrating a false positive that the fitness function avoided.

Table 2

Experimental results – fitness function scores and Quality Gate status

№	Branch	Category	Fitness Function Score	SonarQube QG	Key metric change
1	experiment/improve-coverage	Improvement	+0.562	OK	Coverage 0% → 63.3%
2	experiment/improve-spotbugs	Improvement	+0.570	OK	SpotBugs 73 → 0
3	experiment/reduce-duplication	Improvement	+0.320	OK	Code smells 9 → 0
4	experiment/fix-checkstyle	Improvement	+0.300	OK	Checkstyle 8 → 0
5	experiment/degrade-spotbugs	Degradation	+0.054	ERROR	SpotBugs 0 → 3
6	experiment/degrade-checkstyle	Degradation	+0.142	ERROR	Checkstyle 0 → 8
7	experiment/degrade-duplication	Degradation	-0.307	ERROR	Duplication increased
8	experiment/degrade-coverage	Degradation	+0.020	ERROR	Coverage reduced
9	experiment/neutral-rename	Neutral	+0.320	OK	No metric change
10	experiment/neutral-move-packages	Neutral	+0.009	ERROR	No metric change
11	experiment/neutral-documentation	Neutral	+0.320	OK	No metric change
12	experiment/neutral-test-refactor	Neutral	+0.320	OK	No metric change

Figure 3 presents the fitness function scores for all 12 experiments sorted in descending order, with color indicating the experiment category.

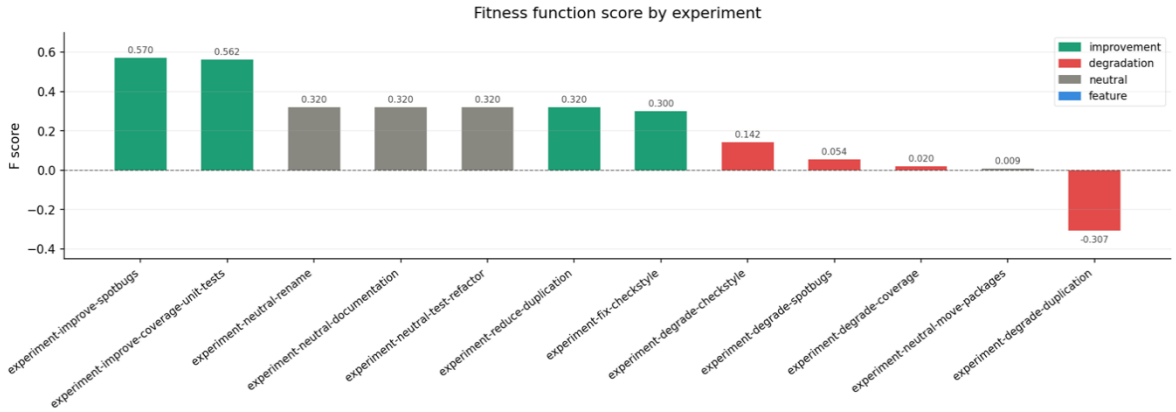

Fig. 3. Fitness function scores across all 12 experimental scenarios, sorted by score value

Improvement scenarios (green) cluster at the top of the chart with scores ranging from +0.300 to +0.570, confirming consistent positive detection. Degradation scenarios (red) occupy the lower range, with one scenario (experiment/degrade-duplication) producing the only negative score of -0.307. Neutral scenarios (grey) appear in the middle range, reflecting the coverage-driven elevation of F scores discussed above.

Table 3 presents descriptive statistics of the fitness function score grouped by experiment category.

The improvement category demonstrates the most consistent behavior, with a mean F score of 0.4379 and a median of 0.4408, indicating a symmetric distribution without extreme outliers.

The narrow standard deviation of 0.1480 and the small gap between P5 (0.3030) and P95 (0.5688) confirm that all four improvement scenarios produced reliably positive scores regardless of which specific metric was improved.

Table 3

Descriptive statistics of fitness function score by experiment category

Category	N	Mean	Median	P5	P95	Std	Min	Max
Improvement	4	0.4379	0.4408	0.3030	0.5688	0.1480	0.3	0.57
Degradation	4	-0.0226	0.0372	-0.2577	0.1286	0.1962	-0.3067	0.1417
Neutral	4	0.2423	0.3200	0.0558	0.3200	0.1554	0.0092	0.32

The degradation category shows a near-zero mean of -0.0226 with the highest variance (0.0385) among all three groups, reflecting significant internal variability - scores range from -0.3067 to +0.1417. This spread indicates that the fitness function's sensitivity to degradation depends heavily on the type and magnitude of the introduced quality regression. The neutral category produces an unexpectedly high mean of 0.2423, driven by three scenarios that scored 0.320.

This behavior is attributed to the normalization mechanism: branches with small numbers of new lines and full unit test coverage receive a high coverage component score, which elevates F even when no intentional quality change was made. The single neutral scenario with near-zero F - experiment/neutral-move-packages (F = 0.009) - confirms that when no coverage advantage exists, the function correctly returns a score close to zero.

The fitness function and the SonarQube Quality Gate approach the same problem in fundamentally different ways, as shown in Figure 4.

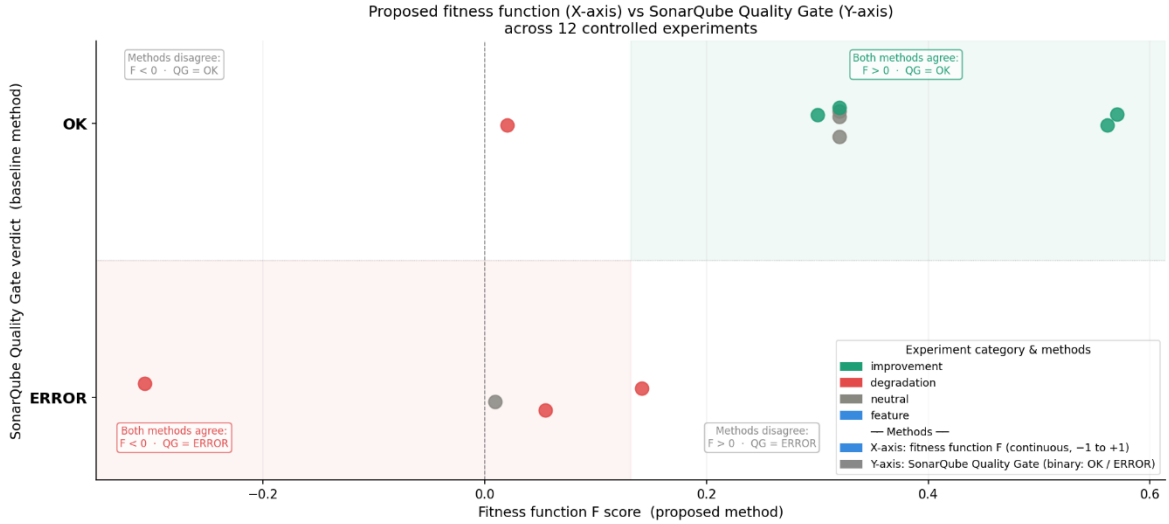

Fig. 4. SonarQube Quality Gate result vs Fitness function score for all experiments

The Quality Gate provides only a binary pass/fail signal, making it impossible to distinguish between a mild degradation (F = +0.02) and a severe one (F = -0.307) - both are labelled ERROR with no indication of magnitude. Conversely, the fitness function quantifies the degree of architectural metric variability continuously, allowing the severity of each change to be assessed and compared across experiments. The most notable anomaly is the experiment/neutral-move-packages scenario, where the Quality Gate incorrectly returned ERROR for a purely structural refactoring with no impact on any quality metric, while the fitness function correctly assigned F = +0.009, indicating a near-zero effect.

Statistical analysis confirms the discriminative power of the proposed methodology. The Mann-Whitney U test showed a statistically significant difference between improvement and degradation categories (U = 16.0, p = 0.014), with a large effect size (Cohen's d = 2.651), supporting the hypothesis that the fitness function reliably distinguishes between qualitatively different types of architectural changes. The large effect size between improvement and neutral categories (d = 1.289) suggests a meaningful difference that could reach statistical significance with a larger experiment set. Spearman correlation analysis revealed no statistically significant relationship between any individual metric and the fitness function score (coverage: p = 0.528, p = 0.078; SpotBugs: p = -0.131, p = 0.685; Checkstyle: p = −0.133, p = 0.680), confirming that F captures a composite signal that cannot be reduced to any single metric alone.

CONCLUSIONS FROM THIS RESEARCH

The study presents a method for quantitative assessment of evolutionary architecture metrics of software systems operating under CI/CD conditions. The study proposes a generalized fitness function that aggregates six structural quality metrics - SpotBugs violations, critical violations, line coverage, code smells, duplicated lines density, and Checkstyle violations - into a single quantitative score computed automatically at each pipeline execution. The metric weights follow the priority classification defined by ISO/IEC 25023, with SpotBugs violations and critical violations $a1$=$a2$=0.25, line coverage $a3$=0.30, code smells $a4$=0.07, duplicated lines density $a5$=0.07, and Checkstyle

violations *аб*=0.06.

Comparison with the SonarQube Quality Gate demonstrated a key advantage of the proposed method. While the Quality Gate provides only a binary pass/fail result, the fitness function quantifies the magnitude and direction of architectural metric variability continuously. A particularly notable finding was the false positive produced by the Quality Gate for a purely structural refactoring scenario (experiment/neutral-move-packages), which the fitness function correctly assessed as near-zero (F = 0.009).

The primary limitation of the method is the sensitivity of neutral scenario scores to line coverage in newly added code, which can elevate F above the expected near-zero range when pull requests contain few new lines with full test coverage. Future work should focus on expanding the experiment set to increase statistical power, calibrating metric weights for different system types, and extending the method to microservice architectures where metric collection requires inter-service aggregation.

References

1. A. Mittal and V. Venkatesan, "Practical Integration of Large Language Models into Enterprise CI/CD Pipelines for Security Policy Validation: An Industry-Focused Evaluation," 2025 IEEE International Conference on Service-Oriented System Engineering (SOSE), Tucson, AZ, USA, 2025, pp. 197-203

2. Jolak R. et al. An empirical investigation of the impact of architectural smells on software maintainability // Journal of Systems and Software. - 2025. - Vol. 225. - P. 112382

3. Sousa A. et al. Architectural Technical Debt – A Systematic Mapping Study // Proc. of SBES 2023 (Brazilian Symp. on Software Engineering). - 2023

4. Kirschner Y.R. et al. Retriever: A view-based approach to reverse engineering software architecture models // Journal of Systems and Software. - 2025. - Vol. 220. - P. 112277

5. SonarQube Quality Gates. URL: https://docs.sonarsource.com/sonarqube-server/10.8/instance-administration/analysis-functions/quality-gates

6. Khatsko N., Sliepushkov M., Khatsko K., Shebanov Y. Modified Software Deployment Algorithm Using Multi-Threading // Bulletin of NTU "KhPI". Series: System Analysis, Control and Information Technologies. - 2024. - No. 2 (12)

7. ISO/IEC 25023:2016. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Measurement of system and software product quality. Geneva: ISO/IEC, 2016

8. Spotbug. URL: https://spotbugs.github.io/

9. SonarQube Cloud. URL: https://www.sonarsource.com/products/sonarqube/cloud/

10. JaCoCo. URL: https://www.jacoco.org/jacoco/trunk/doc/index.html

11. Checkstyle. URL: https://checkstyle.sourceforge.io/

12. N. Saavedra, A. Silva and M. Monperrus, "GitBug-Actions: Building Reproducible Bug-Fix Benchmarks with GitHub Actions," 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Lisbon, Portugal, 2024, pp. 1-5

13. Cosmos DB. URL: https://azure.microsoft.com/en-us/products/cosmos-db (application date 18.03.2025)

14. S. Thomas, PostgreSQL 12 High Availability Cookbook: Over 100 recipes to design a highly available server with the advanced features of PostgreSQL, 2020, 462 p.