UDC 004.415:378.147
DOI: 10.15587/2706-5448.2026.366657

Valerii Kalashnyk,
Dmytro Statsenko,
Oksana Kolysko,
Tetiana Astistova

DEVELOPMENT OF AN EDUCATIONAL AND PRACTICAL ENVIRONMENT BASED ON THE "QUALITY-AS-CODE" CONCEPT AND AUTOMATED QUALITY GATES FOR TRAINING COMPUTER SCIENCE UNDERGRADUATES

The object of research is the process of quality control of student software projects in the training of computer science undergraduates.

In many educational projects, code review relies on manual review by the instructor. This complicates the rapid identification of technical debt, complex methods, access control errors, and performance issues. The student often sees the problem not at the moment of writing the code, but only after manual review. Such feedback is delayed.

In this work, a model was developed, on the basis of which an educational and practical environment was created for automated verification of the codebase of the Simple Booking System (SBS), an educational software system. Verification was divided into four stages: analysis of code structure and cyclomatic complexity, unit tests and negative scenarios, verification of application programming interface (API) contracts, and load testing. ESLint, SonarQube, Jest, Postman, Cypress, and k6 were used. All stages were run in a Docker environment as part of a Continuous Integration/Continuous Deployment (CI/CD) pipeline.

The reduction in code complexity was achieved after refactoring methods with an excessive number of branches that failed static analysis. Verification of Insecure Direct Object Reference (IDOR) scenarios was performed using negative tests. The 95th percentile response time (p95) decreased after changing the PostgreSQL connection pool settings and verifying system behavior under load.

The developed model was verified by comparing two states of the same educational system: the baseline implementation before automated quality gates were applied and the state after the implementation of Quality-as-Code and the completion of corrective changes. In the second state, the average cyclomatic complexity decreased from 14.5 to 4.2, Jest test coverage was 86.4%, and the p95 response time under a load of 10000 virtual users was 280 ms. The number of critical defects in the scenarios under study decreased from 4 to 0. The created environment can be used in higher education institutions for courses related to software testing, DevOps practices, and CI/CD, as well as for preparing students to perform tasks typical of a Software Development Engineer in Test (SDET).

Keywords: *Quality-as-Code, quality gates, DevOps, CI/CD, automated testing, technical debt, cyclomatic complexity.*

Received: 26.04.2026
Received in revised form: 18.06.2026
Accepted: 29.06.2026
Published: 31.10.2026

© The Author(s) 2026
This is an open access article under the Creative Commons CC BY license https://creativecommons.org/licenses/by/4.0/

How to cite
Kalashnyk, V., Statsenko, D., Kolysko, O., Astistova, T. (2026). Development of an educational and practical environment based on the "Quality-as-Code" concept and automated quality gates for training computer science undergraduates. Technology Audit and Production Reserves, 5 (2 (91)), 64–70. https://doi.org/10.15587/2706-5448.2026.366657

1. Introduction

Modern changes in software system architecture, particularly the spread of cloud computing, the transition to microservices, and the dominance of DevOps culture, are changing software development, testing, and operation. The impact of DevOps processes on quality assurance and software architecture design is comprehensively analyzed in [1]. The principles of automated development pipelines, in which code changes go through automated stages of building, testing, and preparation for deployment, are presented in [2].

It was shown in [3] that continuous integration, delivery, and deployment combine frequent code changes with automated checks. Such processes have become essential practices of the modern software lifecycle. In [4], these practices are considered as a component of continuous software engineering, where quality control is performed throughout the software product lifecycle. This organization corresponds to the Shift-Left principle, according to which defects should be detected before development is completed, rather than during the final inspection of the finished system.

In this paper, "Quality-as-Code" is understood as an approach in which rules for static analysis, testing, and acceptable metric values are formalized in a Continuous Integration and Deployment (CI/CD) pipeline.

For the educational process, the problem lies not in the absence of separate testing tools, but in the complexity of combining them into a single scenario of working on a software project. The specifics of teaching DevOps methodology in academic and industrial sectors were investigated in [5]. The research proved that the training of specialists should cover not only software tools but also process automation, participant interaction, and the overall organization of the development lifecycle. The experience of implementing DevOps technologies in

the training of computer science undergraduates is described in [6]. The main attention is paid to the content of training and familiarizing students with DevOps practices. The control of technical debt, security, API contracts, and system behavior under load remained outside the subject area of these works.

The criteria for selecting types of checks and the feasibility of their automation are considered in [7]. It is shown that the decision on automation depends on the type of test, the cost of its execution, and the possibility of reuse. In educational systems, automation is most often used to check individual tasks. For example, the ArTEMiS system [8] automatically evaluates programming exercises and quickly gives feedback to the student. Such an approach is useful for checking functional correctness but does not cover the full range of engineering characteristics of a software project.

Separate researches are devoted to the application of static analysis in education. In [9], static analysis tools were used as an auxiliary tool for evaluating student projects. The effectiveness of using the PMD static analysis tool to engage students in refactoring and improving code quality was investigated in [10]. The change in quality indicators of student course projects after their refinement was analyzed in [11]. These researches confirm the benefits of automated analysis, but mainly focus on individual indicators or one class of defects. They do not use metrics as interconnected conditions for passing several stages of a single CI/CD pipeline.

Control of technical debt is important for student projects, as compromised engineering decisions significantly reduce the maintainability of software systems [12]. SonarQube rules and their relationship to the detection of code defects are analyzed in [13]. For the quantitative assessment of complexity in this research, the McCabe metric was used [14]. Access control violation scenarios are checked according to the OWASP classification [15]. At the same time, the results of [16] show a connection between the maturity of test automation and the quality of the software product. This justifies the need to evaluate not just a single metric, but a set of characteristics of the code and system behavior.

Thus, the analyzed works cover DevOps education, automated assessment, static analysis, technical debt, and software testing. However, these components are mainly considered in isolation from each other. The reviewed sources do not propose an educational model in which static analysis, unit and security testing, API contract verification, and load auditing are integrated into sequential quality gates of a single CI/CD pipeline. The question of how such integration affects code complexity indicators, test coverage, the elimination of critical defects, and the stability of an educational software project under load remains unresolved. The identified scientific problem determined the aim and tasks of this research.

The object of research is the process of quality control of student software projects in the training of computer science undergraduates.

The aim of research is to develop a model of an educational and practical environment based on the "Quality-as-Code" concept and automated quality gates for quality control of student software projects. The model involves controlling indicators of technical debt, cyclomatic complexity, and system stability under load. This will overcome the shortcomings of manual assessment, increase the speed of feedback, reduce code degradation in real time, and create a system for automated assessment of student software projects. Students receive check results immediately after saving code or committing, allowing for timely correction of structural errors.

To achieve the aim, the following tasks were set:

1. Develop the structure of an educational and practical environment model that includes four stages of automated code verification via quality gates in isolated Docker containers as part of a CI/CD pipeline.
2. Integrate into the model structure the tools for automated control of McCabe's cyclomatic complexity, code coverage with Jest unit tests, API and security check results, as well as k6 load testing indicators.
3. Conduct a comparative experiment to evaluate the quality indicators of two engineering states of the SBS educational software project: the baseline implementation before applying automated quality gates (State 1) and the codebase after passing the quality gates and executing necessary corrective changes (State 2).

2. Materials and Methods

The main hypothesis of research is that the formalization of quality indicators and the automation of their control in the CI/CD pipeline contribute to minimizing technical debt. It was assumed that this approach increases the engineering maturity of projects due to early detection, blocking, and elimination of structural defects, security vulnerabilities, and performance issues. Control and interaction with the verification system occur at the level of each commit. Configuration files of the CI/CD pipeline are used for this.

A specially developed educational system, Simple Booking System (SBS), implemented based on NestJS, served as an experimental platform for students. Its modular structure allowed for separate verification of program logic, access control, API interaction, and system operation under load. Such organization and the defined order of running checks ensured the reproducibility of the experiment.

During theoretical classes, issues of software testing, CI/CD, and technical debt were considered. In laboratory work, students applied this knowledge when checking the SBS code using automated quality gates.

The educational environment was piloted in the spring semester of 2026 as part of the "Software Testing Technologies" course. The pilot involved 72 students from four academic groups: BIT-1-22 – 20 students, BIT-2-22 – 25 students, BDIT-22 – 7 students, and BZIT-22 – 20 students. During twelve laboratory sessions, students performed static code analysis, designed unit tests using test doubles, checked API contracts, implemented end-to-end scenarios, conducted load and security testing, and prepared a final Test Summary Report.

The participation of 72 students reflects the scale of the educational pilot rather than a statistical sample for calculating technical indicators. Within the laboratory work, students worked with automated feedback and made corrective changes to the code.

Quantitative research results were obtained by comparing two engineering states of the same SBS codebase. The students' pedagogical outcomes were not the subject of a separate quantitative analysis.

To verify the quality gates' ability to detect typical defects, controlled defects were intentionally integrated into the system architecture. Among them were Insecure Direct Object Reference (IDOR) vulnerabilities, performance anomalies, memory leaks, and inefficient management of the PostgreSQL connection pool. Such defects were used for a separate check of the security and infrastructure levels of the proposed structure. To justify the choice of a technological solution, the comparative analysis method of existing automation tools was applied according to their capabilities for technical debt control, OWASP verification, load auditing, and pedagogical potential.

To ensure the reproducibility of the experiment, the following software was used in the research: Node.js (OpenJS Foundation, USA), NestJS (NestJS core team, international open-source project), PostgreSQL (PostgreSQL Global Development Group, international open-source project), Docker (Docker Inc., USA), GitLab CI (GitLab Inc., USA), Jest (Meta Platforms Inc., USA), Cypress (Cypress.io Inc., USA), Postman (Postman Inc., USA), k6 (Grafana Labs, USA), ESLint (OpenJS Foundation, USA), SonarLint and SonarQube (SonarSource SA, Switzerland), Ubuntu 22.04 LTS (Canonical Ltd., United Kingdom).

The process of evaluating each change in the educational project codebase was divided into four sequential stages of automated verification in the CI/CD pipeline.

Stage 1. Static analysis and audit of algorithm complexity: At the first stage, the code was checked before the program was run according

ISSN-L 2664-9969; E-ISSN 2706-5448

to the Shift-Left principle. ESLint and SonarLint were used for local static analysis and verification of established code rules. SonarQube was used for centralized analysis of the codebase and recording complexity indicators. These tools analyzed the source code abstract syntax tree and identified overly complex methods. During the experiment, threshold values were set for cognitive complexity and cyclomatic complexity according to Thomas McCabe's method [14]

$$M = E - N + 2 \cdot P, \quad (1)$$

where M – the cyclomatic complexity; E – the number of edges in the control flow graph; N – the number of nodes; P – the number of connected components. In case the set limit was exceeded, the CI/CD system was configured to automatically block the build.

Stage 2. Isolated unit testing and security analysis: At this stage, the Jest framework was used. The research conditions required students to use mock objects, stubs, and spies for deep isolation of components. The TestingModule tools and the overrideProvider method in NestJS were used to isolate databases and external APIs. Negative testing scenarios were designed separately. These scenarios verified the system's resilience to Broken Access Control vulnerabilities (in particular, IDOR) according to the OWASP Top 10 standards [15].

Stage 3. Audit of API contracts and system integration: At the third stage, API contracts and the interaction between system components were checked. Postman was used for this. The API was described according to the OpenAPI specification using the Swagger tooling. The JSON Schema format was applied for automated validation of data types and verification of HTTP status codes. For each API module, at least four basic scenarios were envisaged: a successful request, checking invalid input data, checking access control, and analyzing network characteristics. Subsequently, these scenarios were supplemented with automated checks of HTTP responses and data structure validation using JSON Schema. This check allowed errors in data types, HTTP statuses, and the server response structure to be seen immediately. Cypress tests were run separately to test the user interface. To make these scenarios easier to maintain, the Page Object Model pattern was used in the tests to simulate end users' behavior in the browser.

Stage 4. Release management and residual risk analysis: Using the k6 platform, system behavior under load was verified. k6 scenarios were run in the Docker Compose environment. Concurrent requests, server response times, and the operation of the PostgreSQL connection pool were checked. These indicators were crucial for detecting problems that are not visible during regular unit or integration testing.

After testing, students compiled a Test Summary Report. It recorded the main metrics and residual risks before a conditional release. The decision was made in three variants: Go, No-Go, or Conditional Go. The following formula was used to assess risk

$$Risk = Likelihood \cdot Impact, \quad (2)$$

where *Risk* – the risk of releasing defective code; *Likelihood* – the probability of a failure occurring; *Impact* – the degree of its impact on the business logic of the system. The assessment was performed by students separately for the defects and residual risks of their individual variants and was used to justify the Go, No-Go, or Conditional Go decision in the Test Summary Report. These assessments were not aggregated as a quantitative result of the comparative experiment of the two SBS states.

In the experiment, two engineering states of a single SBS codebase were compared under identical execution conditions. The system was implemented based on the NestJS framework:

– *State 1 (Baseline):* the baseline implementation prior to applying automated quality gates;
– *State 2 (Quality-as-Code):* the codebase after passing four gates in the CI/CD pipeline and making necessary changes to meet quality conditions.

Node.js v20.11 and PostgreSQL v16.2 were used in the work. All tests were run on a dedicated server with Ubuntu 22.04 LTS, Intel Xeon E-2336 2.9 GHz, 6 cores, 32 GB RAM DDR4, and SSD 512 GB.

The same metrics were taken for comparison: cyclomatic complexity, code coverage with Jest tests, expected HTTP exceptions, server response time, and PostgreSQL behavior under load. The maximum load in k6 was 10000 virtual users for each state of the codebase.

3. Results and Discussion

3.1. Development of the structure of the educational and practical environment model based on automated quality gates

As a result of the design process, a structure of the educational and practical environment model was developed. It includes four sequential quality gates in isolated Docker containers as part of a single CI/CD pipeline. Each gate is responsible for a separate verification level, and the transition to the next stage is possible only after fulfilling the conditions set by the previous gate. The results of the comparative analysis of educational and industrial solutions for evaluating IT projects are shown in Table 1.

Moodle, as a learning management system, is designed to organize the educational process. It does not check technical debt, API security, or system behavior under load, performing a primarily organizational function. Platforms like LeetCode check isolated algorithms well, but do not provide a complete picture and do not support testing the architecture of a software solution. Cloud solutions like SonarQube perform only static analysis of the program text and are incapable of checking the system dynamically under load, nor do they cover unit tests or API contracts.

The comparison results show that none of the solutions considered alone covers all the identified control areas. Therefore, a stack based on NestJS, Docker, PostgreSQL, GitLab CI, and k6 was chosen for the technological implementation of the proposed model structure. It has sufficient functionality to implement pedagogical and engineering automation. Its key advantage is the ability to manage the automated verification process through formalized Quality Gates rules, unite all checks in a single CI/CD pipeline, and define uniform conditions for passing each gate.

The proposed model structure includes four sequential quality gates. The first gate is a component for static analysis and code complexity control. Its function is to detect structural deviations, an excessive number of branches and violations of the established complexity thresholds before transitioning to subsequent verification levels.

Table 1

Comparative characteristics of educational and industrial solutions for evaluating IT projects

Platform/Stack	System type	Automation level	Technical debt control	OWASP validation	Load audit	Pedagogical potential
Moodle/LMS	Educational environment	Low (manual)	No	No	No	High
LeetCode/QA platforms	Algorithmic tests	High	Speed/memory only	No	No	Limited
SonarQube Cloud	Industrial static analysis	High	High	Partially	No	Medium
Proposed stack (NestJS + Docker + k6)	Comprehensive CI/CD pipeline	Very high	High (automated gates)	Yes	Yes	Very high

The second gate is a component for isolated unit and security testing. Its function is to verify program logic, isolate dependencies, and verify negative access-control scenarios.

The third gate is a component for checking API contracts and integration interactions. It ensures verification of the structure of requests and responses, data types, HTTP statuses, and the correctness of interaction between system modules.

The fourth gate is a component for load control. Its function is to evaluate response time, system stability, resource utilization, and the operation of infrastructure components under a set load level.

Checks within the proposed structure are performed in a defined sequence. If the set conditions of one of the gates are not met, the pipeline stops at the corresponding stage. The next check is launched only after the detected problem is fixed. This order makes it possible to determine at what level the discrepancy arose: in the code structure, program logic, access control, API interaction, or system behavior under load.

Consequently, a specific feature of the proposed model structure is the integration of static analysis, unit and security testing, API contract verification, and load testing into four sequential quality gates of a single CI/CD pipeline. Such integration is implemented through formalized rules for passing the pipeline, and not through the separate use of ESLint, Jest, Postman, Cypress, or k6. Failure to fulfill a set condition at any stage blocks further progress through the pipeline until the detected problem is resolved.

The proposed structure involves transferring a part of repetitive technical verification to an automated environment, while the instructor retains the ability to evaluate architectural decisions, project logic, and the validity of the applied changes. The obtained result concerns a single educational backend and the technological stack of Node.js, NestJS, PostgreSQL, and Docker. Adapting the proposed structure to other programming languages and architectural solutions will require changing gate configurations, analysis tools, and threshold values, but the overall sequence of checks can be preserved.

3.2. Integration of automated quality indicator control into the model structure

During the integration of automated quality indicator control into the proposed model structure, at the first stage, structural code complexity was evaluated as one of the manifestations of technical debt. At this stage, via SonarQube, the cyclomatic complexity of critical methods, in which the core business logic of SBS was concentrated, was assessed. In the baseline state of the system, the average cyclomatic complexity indicator (McCabe) was $M = 14.5$. This meant that a portion of the methods had an excessive number of branches and were poorly amenable to subsequent testing. Such a state indicated the structural degradation of the codebase and a violation of modularity principles.

Refactoring and decomposition driven by the CI/CD pipeline requirements (State 2, Quality-as-Code) made it possible to simplify the control structure and reduce this indicator to $M = 4.2$ (Fig. 1).

At the second stage, the *TestingModule* toolkit and the *overrideProvider* method in the NestJS framework were applied. They enabled architectural isolation, i. e., the ability to test services separately without connecting to a real database or external APIs. Code coverage by Jest tests exceeded the target value of 80% and reached 86.4%. At this same stage, a negative IDOR scenario test was performed. In the baseline state, a request for a resource using a modified identifier was not blocked by proper access control. In the Quality-as-Code state, the test verified that an HTTP 403 response was returned through ForbiddenException. Failure to meet this condition caused the build to stop and required fixing the authorization logic.

At the third stage, API contracts and integration interactions between system components were checked. API contracts were controlled via Postman, OpenAPI, and JSON Schema, while Cypress was used for end-to-end user interaction scenarios with the system. The task of this gate was to verify the correct interaction between system components and the proper processing of HTTP requests.

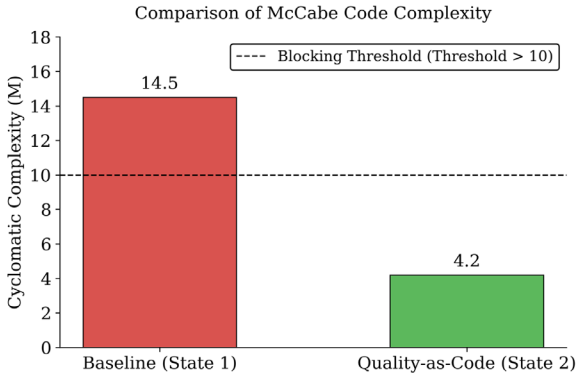

Fig. 1. Comparative analysis of the cyclomatic complexity (McCabe index) of the source code in student projects

At the fourth stage, the infrastructure level was investigated using the k6 platform. This gate made it possible to evaluate not only the presence of an error but also the system's behavior under load. It checked whether problems arose that are usually visible during system operation in real conditions. The automated gate helped detect architectural bottlenecks. In the Baseline state, under a load of up to 10000 virtual users, the exhaustion of the PostgreSQL connection pool was recorded. Blocking of the event loop by heavy synchronous operations also manifested itself. Because of this, the system started returning 504 Gateway Timeout errors, and the server response time increased linearly to 12 seconds. In the Quality-as-Code state, the k6 gate blocked the build because it failed to meet the defined quality thresholds. After this, refactoring was performed. Some operations were moved to asynchronous mode. Inter-component queries were optimized, and PostgreSQL connection pool parameters were adjusted in the Docker Compose environment. A repeated run of the load test produced a stable response-time profile (Fig. 2). After these changes, the p95 response time was 280 ms.

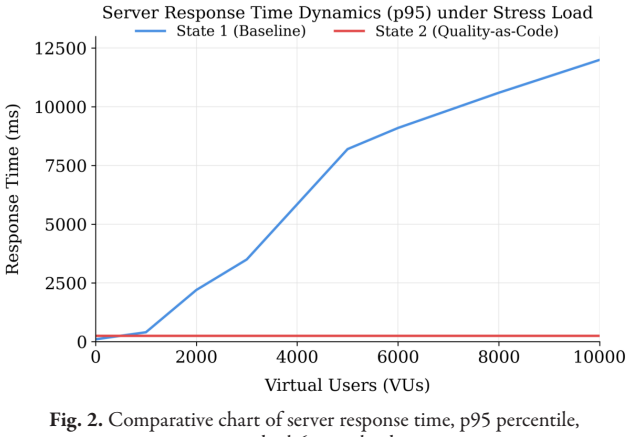

Fig. 2. Comparative chart of server response time, p95 percentile, under k6 stress load

Thus, the load test result was used as a condition to block a build with unacceptable performance indicators. The fourth gate, with defined quality thresholds, acts as a technical constraint and prevents such code from entering the production environment. For the educational process, this is important because the student sees not only the error in the code but also the consequences of an architectural decision for the operation of the entire system.

ISSN-L 2664-9969; E-ISSN 2706-5448

The visualization of temporal characteristics in Fig. 2 demonstrates a pronounced discrepancy between the two configurations:

– in the Baseline state, an uncontrolled linear increase in delays is observed;

– in the Quality-as-Code state, the chart records a stable horizontal plateau without infrastructure failures.

The obtained indicators were used not only for the final assessment of the two system states but also as conditions for passing the respective gates. Exceeding the allowable complexity required code refactoring; insufficient test coverage required expanding the Jest test suite. Failure to pass a security or API scenario necessitated correcting the program logic, and exceeding k6 thresholds required eliminating performance and infrastructure configuration problems.

The integrated quality control process includes McCabe's cyclomatic complexity, code coverage with Jest tests, the results of API and security checks, and k6 load-testing indicators; all of these are used as conditions for passing a single CI/CD pipeline. The established thresholds were determined for the SBS educational system and the corresponding level of task complexity. Therefore, the obtained numerical limits should not be considered as universal standards for all software systems.

During the adaptation of the proposed solution to other disciplines, architectures, or technological stacks, threshold values will require separate configuration, whereas the principle of using metrics as gate-passing conditions can be preserved.

3.3. Comparative experiment for Baseline and Quality-as-Code states

The experiment consisted of a comparative analysis of two engineering states of the codebase:

– *State 1 (Baseline):* the codebase prior to the application of automated quality gates;

– *State 2 (Quality-as-Code):* the codebase after passing four quality gates in the CI/CD pipeline and executing corrective changes.

Corrective changes included refactoring, decomposition of complex methods, expansion of the Jest test suite, correction of program logic and access control, optimization of requests between components, and adjustment of the PostgreSQL connection pool parameters. Consequently, quantitative indicators characterize the full Quality-as-Code cycle: detection of non-compliance, blocking of the corresponding stage, refinement of the codebase, and re-verification.

Summary results of the comparison of quality metrics for the two engineering states are presented in Table 2.

Table 2

Comparison of code quality metrics and defectiveness of the educational project

Quality metric	State 1 (Baseline)	State 2 (Quality-as-Code)	Change
Average cyclomatic complexity (M)	14.5	4.2	–71.0%
Code coverage by unit tests (%)	24.5%	86.4%	+61.9 pp
System's ability to block IDOR	No	Yes	Achieved in the studied scenario
Critical defects	4	0	Eliminated within the experiment

Aggregated data in Table 2 shows an improvement in key indicators. After implementing the quality gates, no critical defects were recorded in the studied scenarios, and code coverage with Jest tests increased to 86.4%. The four critical defects recorded in the baseline state of SBS included: an abnormal delay in the search module due to inefficient resource usage, an Off-by-one error in calculating the booking cost, returning an erroneous 200 OK status in the administration module, and an error processing boundary values in the financial module. The IDOR scenario was evaluated separately as an access control violation.

The results shown in Table 2 demonstrate that after introducing Quality-as-Code, a portion of repetitive technical checks was transferred from the instructor's manual review to the automated CI/CD pipeline. In the baseline state, metrics were recorded after completing the work on the corresponding functionality and did not affect the further progress of the build. Exceeding the set complexity threshold, failing unit or security tests, violating an API contract, or exceeding allowable load indicators caused the corresponding stage to stop. Therefore, the metrics were not just final indicators but also conditions that demanded code refinement.

The average cyclomatic complexity decreased from 14.5 to 4.2 after method refactoring. Static analysis tools detected excessive code complexity, after which refactoring was performed. A 71.0% reduction in the indicator was the mandatory code refinement result to pass the first gate.

With the suite of unit tests expanded, the test coverage by Jest tests was increased from 24.5% to 86.4%. Dependency isolation in NestJS allowed services to be tested without connecting to a real database and external APIs, as well as including negative scenarios in the automatic run. The established minimum test coverage threshold of 80% did not allow passing the second quality gate without adequate verification of the program logic. At this same stage, the IDOR scenario was checked. After adding an authorization check, the negative test demanded returning an HTTP 403 response via ForbiddenException, and failure to meet this condition stopped further build progress.

The third gate checked API contracts and component integration. Verifying response structures, data types, and HTTP statuses via OpenAPI, JSON Schema, and Postman made it possible to stop the pipeline in case of a violation. This expanded verification beyond the unit level and provided control over interaction between components.

The fourth gate made it possible to link the change in performance indicators with specific changes in the code and system configuration. The change in p95 response time from 12 seconds to 280 ms was obtained after analyzing k6 results and refining the code and system configuration. In the baseline state, under a load of up to 10000 virtual users, problems emerged with event loop blocking and PostgreSQL connection pool exhaustion. During refactoring, a portion of operations was moved to asynchronous mode, requests between system components were optimized, and pool parameters in Docker Compose were changed. Rerunning the load scenario showed a stabilization of p95 at 280 ms.

The obtained results show that each gate controlled a separate class of characteristics. The first gate controlled code complexity, the second – test coverage, program logic, and security. The third was responsible for API contracts and integration interaction, and the fourth – for performance and resource usage. Such a sequence made it possible not only to identify problems but also to require their elimination before further progress through the CI/CD pipeline. The system contributed to reducing technical debt and providing students with experience working with engineering quality control tools.

Comparison with known results shows that automated code analysis is already used for evaluating and improving student software assignments. In research [9], static analysis tools were applied as an auxiliary tool for evaluating the quality of student projects. In work [10], PMD was used to engage students in analyzing and fixing code quality problems. In work [11], changes in the quality indicators of student course projects were evaluated after their refinement.

A distinctive feature of the developed model is that static analysis is not used in isolation. It is combined with unit testing, access control verification, API contract validation, and load testing. Each type of check is implemented as a separate gate of a single CI/CD pipeline. If the established quality conditions are not met, further progression of the codebase is stopped until the detected problem is eliminated. Unlike works [9] and [10], which focus primarily on static analysis, the educational and

practical environment created based on the developed model also covers the functional, security, and infrastructural characteristics of a software system. Compared to the research [11], where quality changes were evaluated after refining student projects, a different approach is applied in this paper. Metrics are used as active conditions for passing the automated pipeline, which makes it possible not only to record the indicator values but also to link them to the decision on whether the build should proceed.

The developed model expands the conceptual foundations of continuous software engineering. In [4], continuous software engineering is considered primarily for industrial systems. In this article, the developed continuous quality control tooling is adapted to the specifics of the academic environment.

The results align with the general direction identified in [16]: the development of automated control in CI is associated with improving software product quality. In the developed model, this connection is realized through automatic blocking of code changes that do not meet established criteria for complexity, test coverage, security, and performance.

To verify the developed model, a comparison of two states of the same codebase before and after passing four quality gates was conducted. The average cyclomatic complexity decreased from 14.5 to 4.2, and coverage with Jest tests increased from 24.5% to 86.4%, as shown in Table 2. During the repeated load testing, the p95 response time was 280 ms, whereas in the baseline state, it increased to 12 seconds. Within the scenarios studied, quality gates blocked four critical functional and architectural defects. An IDOR access control violation scenario was checked with a separate negative test. This demonstrates the technical feasibility of transferring a portion of the repetitive checking of student code to a CI/CD pipeline without excluding the instructor from the assessment process.

The practical significance of research lies in improving the quality control process for student software projects. A significant portion of repetitive checks of student code can be transferred from the instructor's manual review to a CI/CD pipeline based on Quality-as-Code. The obtained results can be used directly in the educational process of higher education institutions when training bachelor's students in software engineering and computer science. It is most expedient to apply the model in disciplines where students work with course or qualification software projects. This makes it possible to automate a portion of repetitive checks of syntax, code structure, API contracts, and system behavior under load. Automating repetitive checks allows the instructor to focus on solution architecture, design quality, and the analysis of complex business scenarios. Unlike traditional manual assessment, in the educational and practical environment created based on the developed model, a portion of the technical verification is transferred to an automated CI/CD pipeline. Compared to separate task-checking platforms, such an environment covers several interrelated levels of checking a student software project.

Under martial law conditions, this approach helps preserve educational quality by implementing cloud-based automated quality gates. Due to blackouts, an unstable class schedule, and limited access to classrooms, a student cannot always get explanations from the instructor immediately after completing the work. Provided CI/CD pipeline is hosted on an accessible server or cloud infrastructure, the student can submit code whenever they have access to electricity and the Internet, specifically outside class hours. The system automatically checks their work and immediately provides an error description. This does not replace consultation but supports students' independent work between classes. For disciplines on software testing, DevOps practices, and architectural design, this is of practical significance, as code errors need to be detected not at the end of the semester, but during the gradual execution of the project.

3.4. Limitations and directions for future research

The research limitations are primarily related to the fact that quality gates need to be pre-configured for a specific technology stack. The operation of the gates completely depends on the quality of configuration files, the definition of the quality gate structure, and the established numerical thresholds. In this work, the verification was built around Node.js, NestJS, PostgreSQL, Docker, and k6. When transitioning to other ecosystems (e. g., .NET, Java, or Python), linter rules and Docker container configurations require complete re-engineering. A separate limitation concerns threshold values. For example, the acceptable level of code complexity or the minimum percentage of test coverage is established based on expert judgment; thus, they may differ for different disciplines and student training levels. Also, running heavy k6 tests for 10,000 users requires a powerful server. For universities with limited local infrastructure and outdated equipment, this can be a practical problem.

Prospects for further research should be directed towards adapting the developed model to other educational courses. First, this concerns disciplines of cloud systems, DevOps engineering, and distributed development. It is also advisable to separately investigate how generative artificial intelligence affects code-writing processes. An important issue is also code created using large language models. It needs to be determined whether quality gates are capable of detecting vulnerable or poorly structured fragments within them. Another direction concerns students' behavior when working with strict technical constraints. It is important to separately investigate how frequently failed builds affect motivation, learning pace, and the development of engineering discipline. These results may be useful for improving the methodology of teaching complex technical disciplines, particularly in master's programs in computer science.

4. Conclusions

1. A structure for four-stage automated verification of an educational software project was developed, in which each of the four gates controls a separate class of characteristics: structural complexity, program logic and security, API interaction, and system performance. A specific feature of the developed structure is that different types of checks are combined within a single CI/CD pipeline. The four gates are interrelated automatic blocking stages in an isolated Docker environment. The next stage is executed only after the successful completion of the previous one, which makes it possible to link the detected non-compliance with a specific verification level. Such a structure can be used for the automated control of educational software projects. Adaptation to other disciplines is also possible after changing tools and configurations.

2. Automated collection and verification of engineering metrics were integrated: McCabe's cyclomatic complexity index, code coverage with Jest tests, results of the negative IDOR scenario, compliance with API contracts, and k6 load testing indicators. The results are directly explained by the transition from manual semantic control to automated code quality control. The metrics were used not only to form a final report but also as active conditions for passing the gates. Exceeding the complexity threshold forced the application of method decomposition. Insufficient test coverage led to the need to expand the set of tests. Failure of an API scenario required correction of program logic, and exceeding k6 load testing thresholds indicated the need to identify and eliminate performance and configuration problems. This approach provides early technical feedback during development. The student gets the opportunity to correct detected problems directly while working with the code, and not only after the final check by the instructor. In the Quality-as-Code state, Jest test coverage was 86.4% against a set minimum threshold of 80%. The negative IDOR test verified the return of an HTTP 403 response. For each API module, at least four basic scenarios were used, and the load test was performed for 10,000 virtual users.

3. A comparative experiment was conducted for two states of the same SBS codebase: the baseline state before applying automated quality gates and the state after passing the gates and implementing corrective changes. A specific feature of the experiment was the comparison of the same software system in the same server environment. This made it possible to link the change in indicators with the decomposition of complex methods, expansion of unit tests, transitioning some operations

ISSN-L 2664-9969; E-ISSN 2706-5448

to asynchronous mode, optimizing requests between system components, and adjusting the PostgreSQL connection pool. The obtained results confirm the practical feasibility of using Quality-as-Code. The results of each gate can be used as a guideline for further refinement of the codebase. In the educational process, this allows transferring part of the repetitive technical verification to the CI/CD pipeline and providing students with feedback even before the final evaluation by the instructor. Passing the gates led to refactoring, and after refactoring, the average cyclomatic complexity decreased from 14.5 to 4.2. Coverage with Jest tests increased from 24.5% to 86.4%, i. e., by 61.9 percentage points, and exceeded the established threshold of 80%. Within the studied scenarios, the number of critical defects decreased from 4 to 0, and the negative IDOR test ensured blocking of the build in the absence of proper access control. During the repeated k6 load testing, the p95 value was 280 ms, whereas in the baseline state at maximum load, it increased to 12 seconds.

Acknowledgments

The authors express their gratitude to the staff of the Department of Computer Engineering and Electromechanics of the Kyiv National University of Technologies and Design for organizational assistance, provision of material and technical facilities for conducting the experimental research, and expert support during the validation of the methodology.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship, or otherwise, that could affect the research and its results presented in this paper.

Financing

The research was performed without financial support.

Data availability

Manuscript has no associated data.

Use of artificial intelligence

Google Gemini Pro (Google LLC, USA) was used exclusively to perform a grammar, spelling, and punctuation check of the final manuscript without altering the scientific or technical content. Each suggested grammar improvement was manually checked by the authors for applicability and content correctness. The results provided by the AI tool do not affect the research methodology or the final conclusions of the research. The authors bear full responsibility for the final manuscript.

Declaration submitted by Valerii Kalashnyk.

Authors' contributions

Valerii Kalashnyk: Conceptualization, Methodology, Writing – original draft, Writing – review and editing; ***Dmytro Statsenko***: Supervision, Resources, Project administration, Validation; ***Oksana Kolysko***: Software, Validation, Investigation, Visualization; ***Tetiana Astistova***: Data curation, Formal analysis, Writing – review and editing.

All authors have read and agreed to the published version of the article and are jointly responsible for the reliability, accuracy, and integrity of the data and research results presented in the article.

References

1. Bass, L., Weber, I., Zhu, L. (2015). *DevOps: A Software Architect's Perspective.* Boston: Addison-Wesley, 352. Available at: https://dl.acm.org/doi/book/10.5555/2810087
2. Humble, J., Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 512. Available at: https://www.informit.com/store/continuous-delivery-reliable-software-releases-through-9780321601919
3. Shahin, M., Ali Babar, M., Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access, 5,* 3909–3943. https://doi.org/10.1109/access.2017.2685629
4. Fitzgerald, B., Stol, K.-J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software, 123,* 176–189. https://doi.org/10.1016/j.jss.2015.06.063
5. Pang, C., Hindle, A., Barbosa, D. (2020). Understanding devops education with grounded theory. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training,* 107–118. https://doi.org/10.1145/3377814.3381711
6. Sarmiento-Calisaya, E., Mamani-Aliaga, A., Leite, J. C. S. D. P. (2024). Introducing Computer Science Undergraduate Students to DevOps Technologies from Software Engineering Fundamentals. *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training,* 348–358. https://doi.org/10.1145/3639474.3640071
7. Garousi, V., Mäntylä, M. V. (2016). When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology, 76,* 92–117. https://doi.org/10.1016/j.infsof.2016.04.015
8. Krusche, S., Seitz, A. (2018). ArTEMiS: An Automatic Assessment Management System for Interactive Learning. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 284–289. https://doi.org/10.1145/3159450.3159602
9. Molnar, A.-J., Motogna, S., Vlad, C. (2020). Using static analysis tools to assist student project evaluation. *Proceedings of the 2nd ACM SIGSOFT International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*, 7–12. https://doi.org/10.1145/3412453.3423195
10. AlOmar, E. A., AlOmar, S. A., Mkaouer, M. W. (2023). On the use of static analysis to engage students with software quality improvement: An experience with PMD. *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET),* 179–191. https://doi.org/10.1109/icse-seet58685.2023.00023
11. Chren, S., Macák, M., Rossi, B., Buhnova, B. (2022). Evaluating Code Improvements in Software Quality Course Projects. *The International Conference on Evaluation and Assessment in Software Engineering 2022,* 160–169. https://doi.org/10.1145/3530019.3530036
12. Kruchten, P., Nord, R. L., Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. *IEEE Software, 29 (6),* 18–21. https://doi.org/10.1109/ms.2012.167
13. Lenarduzzi, V., Lomio, F., Huttunen, H., Taibi, D. (2020). Are SonarQube Rules Inducing Bugs? *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 501–511. https://doi.org/10.1109/saner48275.2020.9054821
14. McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering, SE-2 (4),* 308–320. https://doi.org/10.1109/tse.1976.233837
15. *OWASP Top 10:2025* (2025). OWASP Foundation. Available at: https://owasp.org/Top10/2025/
16. Wang, Y., Mäntylä, M. V., Liu, Z., Markkula, J. (2022). Test automation maturity improves product quality – Quantitative study of open source projects using continuous integration. *Journal of Systems and Software, 188,* 111259. https://doi.org/10.1016/j.jss.2022.111259

✉***Valerii Kalashnyk,*** *Candidate of Technical Sciences, Senior Lecturer, Department of Computer Engineering and Electromechanics, Kyiv National University of Technologies and Design, Kyiv, Ukraine, e-mail: kalashnyk.vy@knutd.edu.ua, ORCID: https://orcid.org/0000-0002-2345-4377*

Dmytro Statsenko, *Candidate of Technical Sciences, Associate Professor, Department of Computer Engineering and Electromechanics, Kyiv National University of Technologies and Design, Kyiv, Ukraine, ORCID: https://orcid.org/0000-0002-3064-3109*

Oksana Kolysko, *Candidate of Technical Sciences, Associate Professor, Department of Computer Engineering and Electromechanics, Kyiv National University of Technologies and Design, Kyiv, Ukraine, ORCID: https://orcid.org/0000-0003-4043-1238*

Tetiana Astistova, *Candidate of Technical Sciences, Associate Professor, Department of Computer Engineering and Electromechanics, Kyiv National University of Technologies and Design, Kyiv, Ukraine, ORCID: https://orcid.org/0000-0002-8452-4797*

✉*Corresponding author*