This study investigates the process of verifying the reliability of software with a microservice architecture, developed by several teams in parallel. The task addressed relates to the lack of cost-effective methods for automated integration of testing models into continuous integration and delivery (CI/CD) pipelines adapted to the microservice architecture. This leads to the emergence of "blind spots" in the field of functional and non-functional requirements and excessive time consumption of developers.

This paper reports a method devised for automated integration of the Pentagon testing model into a multi-stage continuous integration and delivery pipeline. The method implements the cumulative principle of progressive distribution of six testing layers between four isolated environments: DEV (development), TEST (test), PRE-PROD (pre-production), and PROD (production). The devised method guarantees coverage of all six layers during each change in the code base.

The key feature of the method is to eliminate the influence of human factor on decisions to launch tests. This makes it possible to achieve increased reliability values, namely Defect Removal Efficiency (DRE), while the manual method detects on average only 27% of defects. The devised method reduces the time when the developer is involved in the process of running tests by 70% compared to the manual approach. For 100 commits (fixing the current state of authentic code and files in the local repository) per month, the savings compared to the manual method are 37%.

The method is suitable for verifying software with a microservice architecture, which is deployed on cloud platforms (Amazon Web Services, USA; Azure, USA; Google Cloud, USA) with support for automated testing and container orchestration

Keywords: microservice architecture, continuous integration and delivery pipeline, software reliability

UDC 004.41

DOI: 10.15587/1729-4061.2026.360693

DEVELOPMENT OF A METHOD FOR AUTOMATED INTEGRATION OF THE "PENTAGON" TESTING MODEL INTO A CONTINUOUS INTEGRATION AND DELIVERY PIPELINE FOR MICROSERVICE SOFTWARE

Oleh Kuzmych
Corresponding author
PhD Student*
E-mail: oleg.kuzmuch@gmail.com
ORCID: https://orcid.org/0000-0002-9749-6385
Maksym Seniv
Candidate of Technical Sciences, Associate Professor*
ORCID: https://orcid.org/0000-0003-1044-4628
*Department of Software
Lviv Polytechnic National University
S. Bandery str., 12, Lviv, Ukraine, 79013

Received 18.02.2026
Received in revised form 01.05.2026
Accepted date 11.05.2026
Published date 30.06.2026

How to Cite: *Kuzmych, O., Seniv, M. (2026). Development of a method for automated integration of the "Pentagon" testing model into a continuous integration and delivery pipeline for microservice software. Eastern-European Journal of Enterprise Technologies, 3 (2 (141)), 56–66. https://doi.org/10.15587/1729-4061.2026.360693*

1. Introduction

Microservice architecture is the dominant paradigm for developing cloud-based enterprise software: it enables independent scaling of components, accelerates development cycles, and underpins most modern cloud platforms. Systems built on this principle serve billions of requests every day, from banking transactions to logistics. Accordingly, the reliability of such software is directly related to the continuity of critical business processes.

Research in the field of verifying the reliability of microservice software in CI/CD pipelines can provide practice with reproducible, cost-effective methods suitable for direct application by development teams. The applied value of such research is to reduce the time to detect defects, reduce the cost of their correction, and increase the availability of systems – indicators with direct financial expression [1, 2].

If research in this area does not evolve, the responsibility for the completeness of test coverage will remain with the human factor. In the context of parallel development by several teams, this means that non-functional requirements – fault tolerance, behavior during degradation of dependent services – will not be systematically verified before going live, which could result in an increase in critical incidents and economic losses.

The area related to the automated integration of complex testing models into CI/CD pipelines requires special attention. The results of such research allow the industry to move from a manual, unpredictable approach to a deterministic verification process independent of the decisions of an individual developer.

Thus, research on the automated integration of testing models into CI/CD pipelines for microservice software is relevant: it is capable of providing practical specific reliability verification methods, the absence of which could have negative consequences for the quality and cost-effectiveness of software development.

2. Literature review and problem statement

It was found in [3] that functional property verification (unit and integration tests) dominates research on microservices testing, while non-functional property testing – fault tolerance, behavior during degradation of dependent services – covered by

a much smaller proportion of studies. At the same time, in [3], no holistic method for mandatory integration of non-functional layers into the CI/CD pipeline as a standardized stage of the development life cycle is proposed. A probable reason: the methodology of the systematic review is aimed at describing existing practice, which itself has not yet devised established methods for testing non-functional properties in pipelines.

Study [4] analyzed 33 primary publications and confirmed that API-level unit and integration tests are dominant approaches, while fault tolerance testing and chaos testing remain rare in the CI/CD context – a lack of specialized tools for these approaches was noted. The study does not propose a specific method that would determine in which pipeline environment and on which trigger these layers should be executed.

In [5], based on the analysis of 37 open Java projects, a statistically significant positive correlation was established between test automation maturity and product quality ($p < 0.01$): projects with a higher level of automation demonstrate a shorter release cycle and higher quality. However, the cited study analyzes the overall level of automated coverage without qualitatively distinguishing between functional and non-functional layers. Therefore, it remains unclear what specific combination of testing methods, and at which point in the pipeline provides optimal defect detection. A possible reason is the lack of structural distinction between test types in the repository metadata.

In [6], 40 problems of continuous delivery implementation in 30 primary publications were identified. Among the technical issues, there is insufficient coverage of non-functional requirements and instability of test environments; among the organizational problems, there is no agreed policy on which testing methods should be used and when. In [6], the problem of incomplete coverage of non-functional requirements is identified but a specific testing method to solve it is not proposed: the research is aimed at diagnostics, not solution synthesis.

In [7], 69 papers on CI/CD practices are analyzed and non-functional requirements testing is identified as an open problem: most studies focus on functional testing, while non-functional is either absent from the pipeline or implemented outside it. Among the identified reasons is the lack of unified criteria for distributing non-functional testing methods between pipeline environments and the conditions for their mandatory launch, as well as the high cost of preparing the appropriate cloud infrastructure.

In [8], 56 papers were analyzed and a set of continuous testing techniques in CD was identified: shift-left testing, smoke tests, contract testing. Verification of non-functional requirements – chaos testing and fault tolerance testing – absent from the recommended mandatory testing methods in the pipeline. The probable reason: at the time of the review (2018), those testing methods had not yet acquired a systematic form in the CI/CD context.

In study [9], based on the analysis of GitHub projects and Docker Hub, it was found that teams use fundamentally different approaches to containerized CD workflows and do not adhere to a single standard for deploying environments. Inconsistency of environment configurations is one of the main reasons for incomplete test coverage before release. However, the work does not propose a standardized testing method with the distribution of testing layers between environments: the study characterizes variability but does not formulate a normative solution – in particular, which environment is responsible for which layers, and on which trigger each testing method is executed.

Paper [10] analyzed 22 failure scenarios reproduced on a benchmark microservice system based on real industrial cases and found that a significant proportion of critical failures are associated with non-functional aspects – timeout errors, cascading failures, incorrect handling of partial unavailability of dependent services. That is, a significant category of production incidents occurs in the class of defects that are not verified by standard functional testing methods. The work did not address the issue of how to proactively verify resistance to such failures in an automated pipeline: the study focuses on post-facto analysis of incidents, rather than on the design of a testing method.

Study [11] reports a designed framework for implementing chaos engineering at an industrial enterprise and finds that organizations do not have established practices for its systematic application: interviews with engineers confirmed the lack of a standard method for integrating chaos testing into the pipeline. Among the identified obstacles are the lack of formalized criteria for completing chaos tests and significant infrastructure requirements.

Chaos engineering and fault injection tools [12–14] demonstrate the practical maturity of methods for detecting non-functional defects: paper [12] formulates the principles of chaos engineering based on Netflix industrial practice, study [13] implements systematic verification of microservices stability through inter-service message manipulation, the authors of [14] – fault injection at the RPC call level to identify missing partial fault handling. However, none of these papers considers the automated launch of such checks as a mandatory step of the pipeline: the methods are positioned either for a productive environment [12], or as target tools outside the pipeline logic [13, 14]. The common reason is the lack of formalized criteria for the mandatory launch of non-functional checks in the pipeline and established practices for their integration.

In [15], the effectiveness of query replication as a fault tolerance mechanism on the FaaS platform was investigated: it was found that replication reduces the proportion of lost queries when computing nodes fail but requires an increase in resources. In [15], one specific mechanism was considered without analyzing testing methods for its systematic verification in an automated pipeline.

The systematization of the identified problem components allows us to identify four interrelated problematic components, each of which defines a specific characteristic that the solution method should have:

– systematic exclusion of non-functional testing from CI/CD pipelines [4, 7, 8, 16]. The six-layer Pentagon testing model [16] expanded the traditional testing pyramid [17], clearly distinguishing non-functional layers: fault tolerance testing, E2E and chaos testing. However, work [16] does not specify the order of integration of these layers into an automated CI/CD pipeline. This gap is also confirmed by broader studies [4, 7, 8]: non-functional layers are not implemented as mandatory pipeline steps, as a result of which a significant proportion of critical failures in microservice systems of a non-functional nature [10] are not verified before release to PROD. The solution method should determine how all six layers – in particular, the non-functional layers of fault tolerance testing, E2E, and chaos testing – are integrated into the pipeline as mandatory steps;

– chaos testing as an isolated process, not a gate step of the pipeline [11–13]. Chaos engineering tools exist and are proven in practice, but none of the considered testing methods formalizes their launch as an automated and reproducible mandatory step before each release. The solution method should define chaos testing as a mandatory pipeline step with clear passing conditions, not as a situational process outside the pipeline;

– lack of a standardized method for distributing testing layers between pipeline environments [6, 7, 9]. The existing practice of testing methods is variable and unregulated: which layers, in which environment and on which trigger are executed – is determined situationally, which leads to uneven coverage. The solution method should formalize the distribution of layers between isolated environments (DEV, TEST, PRE-PROD, PROD) on a cumulative basis, defining specific layers and activation triggers for each environment;

– dependence of the completeness of test coverage on the human factor [5, 6, 11, 18]. According to [18], unit tests are used by 78% of developers, while chaos testing is systematically used by only 6% [19]. In the absence of automated launch, this imbalance is reproduced at each commit: the developer decides situationally which layers to launch. The solution method should exclude the dependence of completeness of coverage on the developer's decisions – the pipeline launches all the provided layers automatically on the corresponding git push.

Among the reasons for this, the following components were made impossible to resolve:

1) chaos engineering, fault tolerance testing methods, and functional testing developed as independent disciplines with separate tools;

2) the lack of formalized criteria for mandatory launch of non-functional tests in the pipeline, which made their automated execution impossible;

3) the high cost of preparing cloud infrastructure for full-fledged non-functional testing, which deterred researchers from considering this issue in a practical context.

Thus, the unsolved issue is the lack of a method for automated integration of all testing layers – including non-functional ones – into a multi-stage CI/CD pipeline that implements the cumulative principle of distribution between isolated environments; formalizes the order of distribution of layers and the conditions for their activation in each environment; eliminates the dependence of completeness of coverage on the developer's decisions.

3. The study materials and methods

The purpose of our research is to devise a method for automated integration of the Pentagon testing model into the CI/CD pipeline for microservice software, which will ensure complete and reproducible coverage of non-functional requirements with minimal developer time. This will make it possible to increase the reliability of microservice systems and reduce the cost of fixing defects by early detection in the pipeline.

To achieve the goal, the following tasks were set:

– to propose a structure for a method for automated integration of testing layers of the Pentagon testing model into a multi-stage CI/CD pipeline with formalized criteria for the distribution of layers between environments and the transition between them;

– to design two versions of the microservice system – standard (with coverage according to the "testing pyramid" model) and improved (with coverage according to the "Pentagon" model using the proposed method) – as objects of comparative research;

– to measure the real-time characteristics of all six testing layers on a cloud infrastructure to determine the practical execution time of the pipeline;

– to compare the defect detection efficiency of three testing methods (automated, manual with full coverage, and manual ad-hoc) with a quantitative assessment of variability using Monte Carlo defect detection simulation;

– to conduct a comparative assessment of the proposed method: analysis of cost-effectiveness relative to the manual testing method according to the criteria of developer time and execution cost, as well as validation of the reliability of the designed systems through availability simulation at different levels of infrastructure failures.

4. The study materials and methods

The object of our study is the process of verifying the reliability of software with a microservice architecture, which is developed in parallel by several teams.

Research hypothesis assumes that automated integration of all six layers of the Pentagon testing model into a CI/CD pipeline with a cumulative distribution principle provides a higher DRE indicator with less developer active time compared to manual testing methods.

Assumptions: adoption rates from industry surveys [18–20] are correct approximations of the probability of running tests on a specific commit for method B; the cost of developer time (USD50/hour) is representative for the regions of the USA and Western Europe.

Simplifications: measurement of local layers (L1, L2, L4) was carried out on one platform (Windows 11, Python 3.12) – results may vary on other configurations; L6 layer measured in one FIS experiment run; reliability simulation limited to three failure levels (10%, 20%, 50%).

The Pentagon testing model [16] defines six layers $L = \{L1, L2, L3, L4, L5, L6\}$:

– L1 (Unit Testing) – testing of isolated business logic with mocked dependences;

– L2 (Local Integration Testing) – testing of component interaction with emulated AWS services (moto);

– L3 (Global Integration Testing) – testing in a real cloud environment;

– L4 (Fault-Tolerance Testing) – error injection (timeout, 503, limit exceeded) for retry and circuit breaker verification;

– L5 (End-to-End Testing) – end-to-end tests of the complete business scenario through all microservices;

– L6 (Chaos Testing) – simulation of infrastructure failures via AWS Fault Injection Simulator (FIS).

The ci_timing.py tool was used for the measurement, which runs each layer as a subprocess and records the execution time. Local layers (L1, L2, L4) are measured as the average of 5 independent runs on Windows 11, Python 3.12. Cloud layers (L3, L5) are measured as the average of 3–4 runs against deployed AWS stacks in the eu-central-1 region. Layer L6 is measured on one full run of a real FIS experiment.

The effectiveness of the method was tested by comparing three methods:

– Method A (Automated): all six layers are executed automatically by the pipeline for each relevant commit without developer involvement. This is the method that demonstrates our devised method for automated integration of the Pentagon testing model;

– Method B (Manual ad-hoc): the developer runs tests manually according to his/her own decision. The probability of execution of each layer is determined based on published industry studies: JetBrains Developer Ecosystem 2024 [18] (78% unit, 63% integration, 48% E2E among 23,262 developers), CNCF/SlashData Q3 2025 [19] (6% chaos engineering among backend developers), Gremlin 2021 [20] (40% have never conducted a chaos experiment). Note that adoption rates reflect general practice (do you use this type of testing at all), not the frequency of runs per specific commit – so the transition from adoption rate to per-commit probability is a rough estimate;

– Method C (Manual Full): the developer runs all six layers manually with the same coverage as Method A. Execution time \= test time + 30 s of overhead per layer (entering a command, reading the output, making a decision) + 60 s for preparing the environment.

The introduction of method C is key to the purity of the experiment: it allows us to distinguish between the benefits of complete coverage and the benefits of automation over the developer's efforts.

Monte Carlo defect detection simulation. To quantify the variability of Method B, the Monte Carlo method [21] was used, a stochastic modeling method based on the generation of a large number of random iterations. 1000 iterations were performed (seed = 42 for reproducibility): in each iteration, for each of the 12 identified defects, it is stochastically determined whether it will be detected, based on the probabilities of Method B.

The cost of AWS infrastructure is calculated from the actual tariffs of the eu-central-1 region (Lambda 128 MB: USD 0.0000166667/GB s, API Gateway: USD 0.0000037/request, DynamoDB on-demand: less than USD 0.000001 per operation) [22]. The cost of developer time of USD 50/hour is calculated according to an industry survey of developers for the US and Western Europe regions, which corresponds to the median annual compensation of a backend developer in the range of USD 67–104 thousand [23].

To compare the reliability of the two systems, a simulation of 4800 requests in 600 seconds was conducted at three failure rates (10%, 20%, 50%) with a fixed percentage of successful responses.

5. Results of research on the method of automated integration of the Pentagon testing model into the CI/CD pipeline

5. 1. Structure of the method of automated integration of the Pentagon model into the CI/CD pipeline

The proposed method is based on the cumulative principle of distributing testing layers between CI/CD pipeline environments: each subsequent environment inherits all checks of the previous ones and adds new layers specific to its level of readiness. This approach implements the principle of "Fail Fast, Fail Cheap" [24].

The distribution of layers across environments is determined by the balance between the cost of execution, the speed of feedback, and the required depth of testing (Table 1).

Table 1

Distribution of testing layers across pipeline environments

Environment	Trigger	Executed layers	New layer
DEV	Push to dev	L1, L2, L3	L3: Global integration
TEST	Push to test	L1, L2, L3, L4	L4: Fault tolerance testing
PRE-PROD	Push to pre-prod	L1, L2, L3, L4, L5, L6	L5: E2E, L6: Chaos
PROD	Push to prod	L1, L2 → Deploy → Monitoring	–

The scheme of the proposed method is shown in Fig. 1. It covers four main environments and demonstrates the cumulative principle of increasing the depth of checks.

The method includes 10 formalized Steps:

Step 1. Change the code base.

The developer makes changes to the code base and performs git commit and git push operations, sending their local commits to a remote repository of the version control system (e.g., GitLab, GitHub, Bitbucket). The push operation automatically starts the CI pipeline. The pipeline logic can be defined in a configuration file (e.g. .gitlab-ci.yml, Jenkinsfile) and can have different branches or stages depending on the settings and the environment in which the changes are made. The devised method provides 4 branches dev, test, pre-prod, prod, which correspond to the environments of the same name in the cloud.

Step 2. Build the project.

In this block, dependences are established, code is compiled (if necessary), and artifacts are generated, for example, keys required for subsequent testing or deployment steps.

Step 3. Check the code base.

Static analysis, linting, formatting checks, and vulnerability scanning are performed. Tools like SonarQube can be used for this purpose. The result is a code quality report.

Step 4. Unit testing.

Unit tests are run to test the operation of individual functions, classes, or methods. Frameworks such as Pytest, JUnit, or Jest are usually used. At this stage, it is guaranteed that the changes made do not break the basic functionality.

Step 5. Local integration testing.

Integration testing of microservices is performed in a local environment using emulators (Moto, LocalStack, SQLite). This allows one to quickly identify problems with component interaction without wasting resources on full deployment.

Step 6. Branching (dev branch, test branch, pre-prod branch, prod branch).

After successful previous steps, the system analyzes which branch the changes have been made to:

– if prod branch, then after manual approval by the responsible person, deployment is performed in PROD, where minimal intervention is performed before release;

– if dev branch, test branch or pre-prod branch, the project is deployed to the DEV, TEST or PRE-PROD environment, respectively.

Step 7. Global integration testing.

In the DEV, TEST, and PRE-PROD environments, global integration tests are automatically launched, which check the operation of microservices under more real-world conditions (AWS Lambda, API Gateway, DynamoDB databases, etc.). Postman, Newman, or Python scripts with the requests library can be used for this purpose.

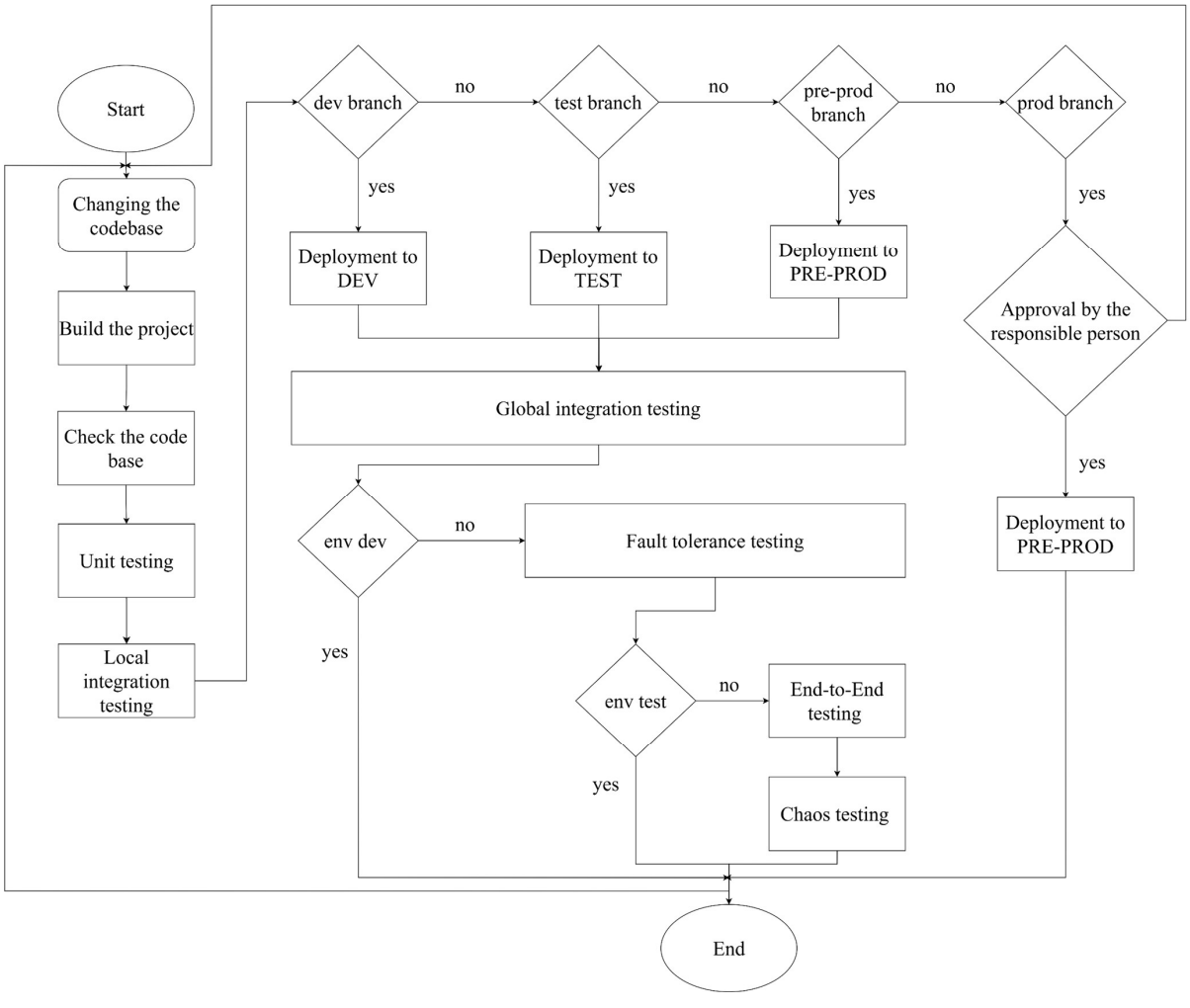

Fig. 1. Scheme of the proposed method for automated integration of the Pentagon model into the CI/CD pipeline for four environments

Completion for DEV.

After global integration tests, the DEV pipeline is completed. Its main purpose is to provide developers with quick feedback on the logic and correct interaction of services.

Step 8. Fault tolerance testing.

In the TEST and PRE-PROD environments, fault tolerance testing is then performed. This stage involves error injection (timeout, service, or database unavailability, etc.) to test the operation of retries and circuit breakers.

Completion for TEST.

After successful fault tolerance testing in the TEST environment, the pipeline is completed, as this environment is intended for thorough integration and stability testing of the basic functions.

Step 9. End-to-End testing (PRE-PROD).

For the PRE-PROD environment, the testing stage continues with end-to-end (E2E) tests. Using Cypress, Selenium, or Postman, end-user actions are simulated and the correct operation of the entire system is verified.

Step 10. Chaos testing (PRE-PROD).

The final stage is chaos testing, which simulates various unforeseen failures: network delays, connection drops, service failures, etc. AWS Fault Injection Simulator, Chaos Monkey or Chaos Mesh are used to check how the system reacts to critical failures and whether it is able to recover with minimal losses.

Completion for PRE-PROD.

After testing the system for stability under non-standard conditions, the PRE-PROD pipeline is completed. The test results are analyzed, and, if necessary, developers make corrections.

The "End" element in the diagram marks the formal completion of the CI/CD pipeline. The process is ready for a new cycle when the next changes will be made to the code base.

Thus, the diagram in Fig. 1 demonstrates how the "Pentagon" model is integrated into the classic CI/CD process, providing step-by-step and multi-level testing at different stages of development and in different environments. Thanks to this method, it is expected that reliability will increase, namely:

– the number of failures in PROD will decrease;

– the Defect Removal Efficiency (DRE) indicator will improve;

– controlled preparation for production release will be ensured;

– the time spent on testing will decrease;

– the total cost of testing will decrease.

A method has been devised that implements the Continuous Delivery approach: deployment to test environments is performed automatically, while the final transition to PROD is made by the team's decision. This allows for a more careful assessment of risks and avoids uncontrolled release of poor-quality code into the production environment.

5. 2. Microservice systems for researching the effectiveness of the method

A microservice student administration system on the AWS SAM (Serverless Application Model) platform is proposed, consisting of three services: student-service (AWS Lambda + DynamoDB + + API Gateway), enrollment-service, and notification-service (AWS SQS), deployed in the eu-central-1 region.

Two variants of the system were prepared for comparative research:

– standard system – implemented with coverage according to the traditional "testing pyramid" model (layers L1, L2, L5); non-functional requirements are not verified, there are no fault tolerance mechanisms;

– improved system – implemented with full coverage of all six layers of the "Pentagon" model (L1–L6), which includes verification of fault tolerance patterns: retry_with_backoff, CircuitBreaker, validate_student_data, graceful degradation.

Both options have the same business logic and API, which ensures a clean comparison: the difference is determined solely by the applied testing method and the presence of fault tolerance mechanisms.

5. 3. Time characteristics of testing layers

The results of the execution time measurements of all six layers are given in Table 2.

Fig. 2 clearly shows how the total pipeline time increases cumulatively from environment to environment. The largest increase is in PRE-PROD due to chaos testing (L6: 143.4 s), but these costs are justified: they are incurred only for tagged releases, not for each commit.

Total pipeline execution time for each environment (Build 5 s + Lint 3 s + Deploy 60 s + layers):

– DEV (L1, L2, L3): 88.8 s (1.5 min);
– TEST (L1, L2, L3, L4): 99.6 s (1.7 min);
– PRE-PROD (all 6 layers): 256.8 s (4.3 min);
– PROD (L1, L2 + deploy): 81.5 s (1.4 min).

Table 2

Characteristics of layers in the Pentagon model

Layer	Number of tests	Execution time	Coverage	Need to deploy on AWS	First environment
L1: Unit testing	5	8.1 c ($\sigma = 2.0$ c, $n = 5$)	58%	No	All
L2: Local integration	2	5.4 c ($\sigma = 1.1$ c, $n = 5$)	59%	No	All
L3: Global integration	1	7.3 c (avg, $n = 4$)	~68%	Yes	DEV
L4: Fault tolerance testing	8	10.8 c ($\sigma = 1.4$ c, $n = 5$)	70%	No	TEST
L5: E2E	1	13.8 c (avg, n = 3)	~85%	Yes	PRE-PROD
L6: Chaos (AWS FIS)	1	143.4 c ($n = 1$)	~92%	Yes	PRE-PROD

5. 4. Comparison of defect detection efficiency

The probabilities of layer execution in Method B (manual ad-hoc) are given in Table 3.

As can be seen from Fig. 3, Method A and Method C provide 100% execution of all layers, while Method B has a critical drop for complex layers (L4 = 25%, L6 = 6%).

Table 3

Probability of performing layers in manual testing (Method B)

Layer	Probability	Justification (source)
L1: Unit tests	100%	78% adoption, 100% during testing session [18]
L2: Local integration	65%	63% adoption for integration tests [18]
L3: Global integration	40%	Between 63% and 48%; reduced due to cloud deployment barrier [18]
L4: Fault tolerance testing	25%	Below E2E; specialized fault injection tools are absent in most teams [18]
L5: E2E	20%	48% adoption [18], but cloud E2E requires a large infrastructure
L6: Chaos	6%	6% of backend developers practice [19]; 40% have never tried [20]

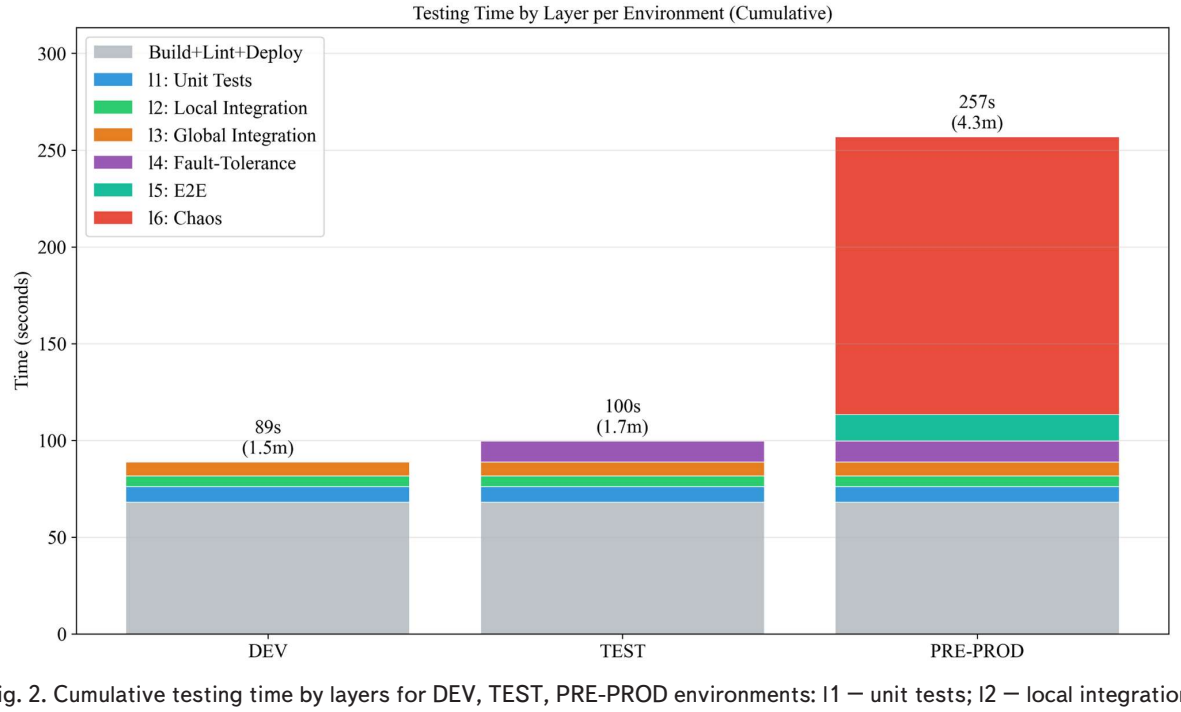

Fig. 2. Cumulative testing time by layers for DEV, TEST, PRE-PROD environments: l1 — unit tests; l2 — local integration; l3 — global integration; l4 — fault tolerance; l5 — E2E; l6 — chaos testing

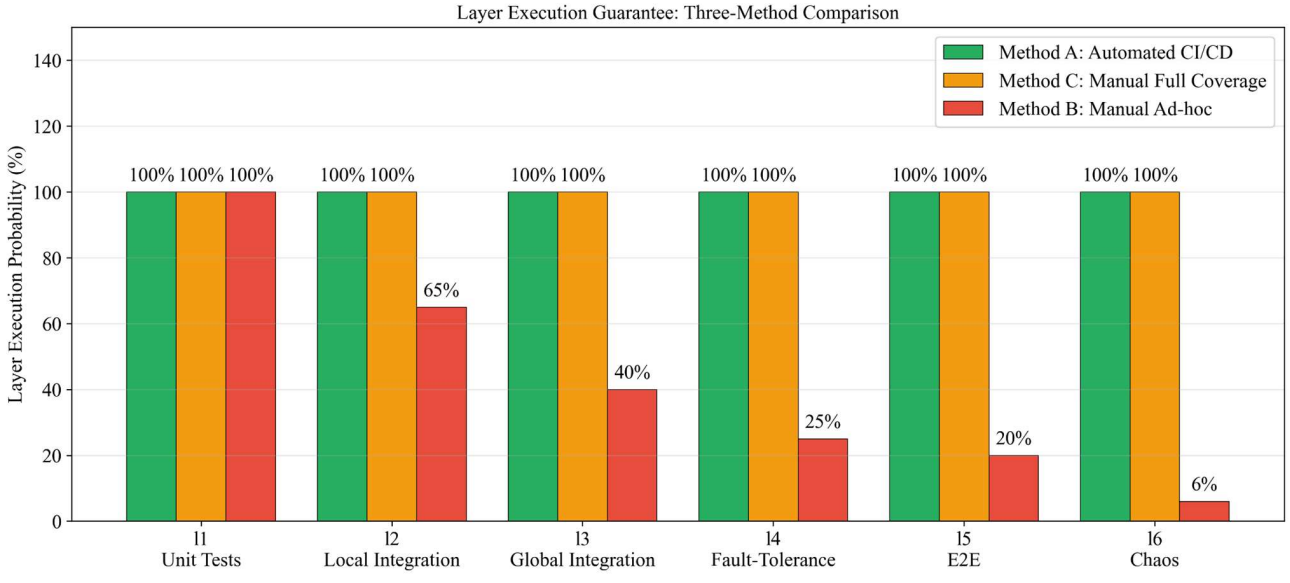

Fig. 3. Comparison of the probability of layer execution: Method A (automated), Method C (manual full), Method B (manual ad-hoc)

For comparison, 12 error injection scenarios were performed: 8 fault tolerance defects (L4: transient DynamoDB errors, retry logic, circuit breaker, input validation, graceful degradation), 3 cloud integration defects (L3: IAM auth, malformed JSON, cross-service communication), and 1 chaos tolerance defect (L6: FIS Lambda delay injection). The results are shown in Table 4, Fig. 4.

To quantify the variability of Method B, a Monte Carlo simulation (1000 iterations, seed = 42) was performed. The results are given in Table 5, Fig. 5.

Table 4

Defect detection: three methods compared

Layer	Defect	Method A (Auto)	Method C (Manual full)	Method B (Manual ad-hoc)
L3: Global integration	3	3/3 (100%)	3/3 (100%)	~1.2/3 (40%)
L4: Fault tolerance testing	8	8/8 (100%)	8/8 (100%)	~2.0/8 (25%)
L6: Chaos	1	1/1 (100%)	1/1 (100%)	~0.1/1 (6%)
TOTAL	12	12/12 (100%)	12/12 (100%)	~3.3/12 (27%)
DRE	–	100%	100%	~27%

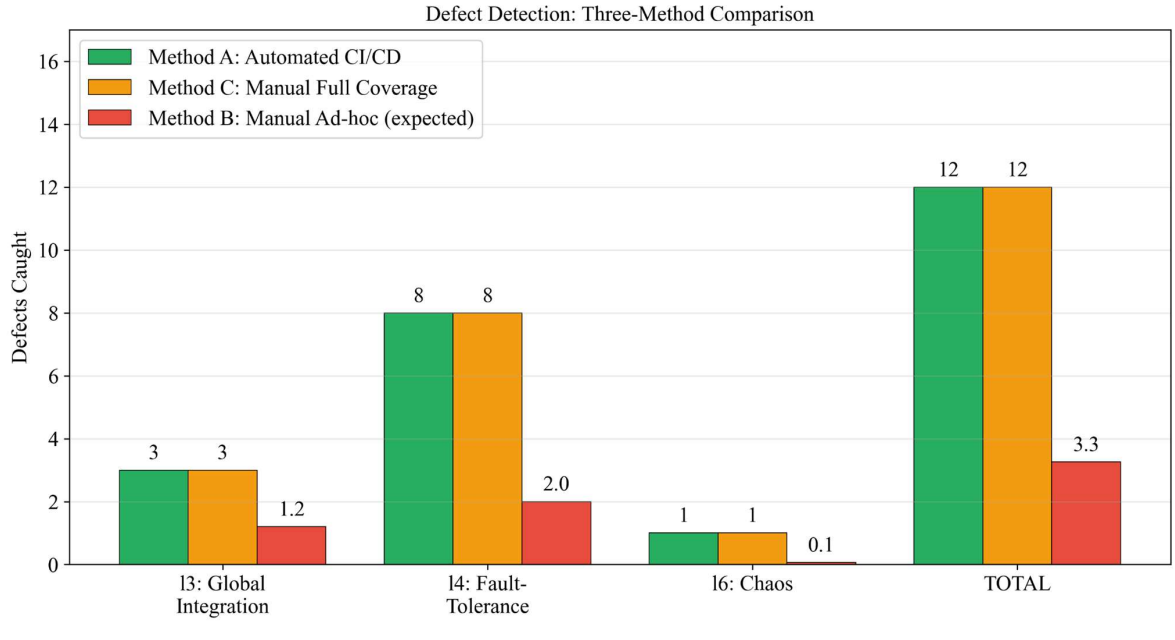

Fig. 4. Defect detection: Method A and Method C find 12/12 equally, Method B only ~3.3/12 on average: l3 — global integration; l4 — fault tolerance; l6 — chaos testing

Table 5

Monte Carlo simulation results (Method B, 1000 Iterations)

Approach	Mean	Minimum	Median	Maximum	σ
Method A (Automated)	12/12	12	12	12	0.0
Method C (Manual full)	12/12	12	12	12	0.0
Method B (Manual ad-hoc, MK)	3.2/12	0	3	8	1.5

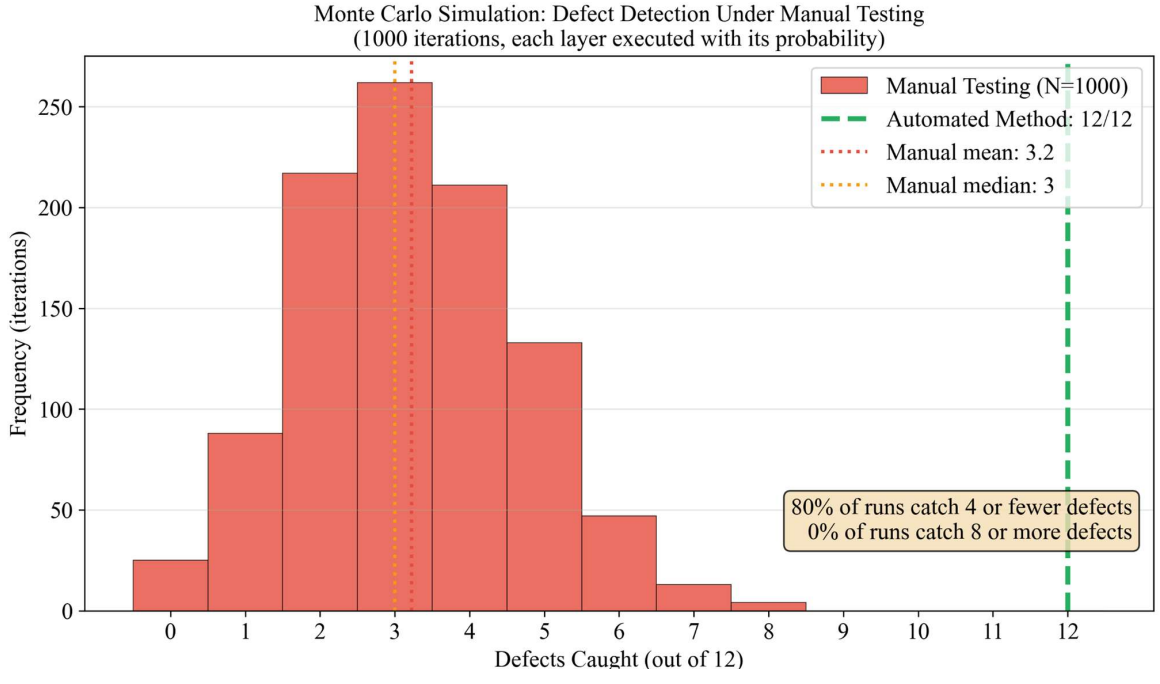

Fig. 5. Monte Carlo simulation histogram for Method B (1000 iterations, seed = 42); green dashed line — guaranteed result of methods A and C (12/12)

The simulation results confirm the fundamental unreliability of Method B as a mechanism for ensuring test coverage. The median value of detected defects is 3 out of 12, and 70% of the iterations detect 3 or fewer defects. None of the 1000 iterations reached the full coverage of 12/12, which is deterministically provided by methods A and C. The high variability ($\sigma = 1.5$) indicates the unpredictability of the ad-hoc approach, which is a critical drawback for systems with increased reliability requirements.

5. 5. Comparative evaluation of the proposed method: cost-effectiveness and reliability validation

The comparative cost of the three methods is given in Table 6 and illustrated in Fig. 6.

Table 6

Comparative cost of three testing methods

Factor	Method A (Auto)	Method C (Manual full)	Method B (Manual ad-hoc)
AWS cost (PRE-PROD run)	~USD 0.00019	~USD 0.00019	~USD 0.00002
Developer uptime	~2.5 min	8.3 min	~4 min
Developer time cost (USD 50/hour)	~USD 2.08	~USD 6.90	~USD 3.33
Cost per 100 commits/month	~USD 208	~USD 690	~USD 333
DRE	100%	100%	~27%

The devised method (A) demonstrates the lowest developer active time (~2.5 min) in all environments – 70% less compared to Method C (8.3 min) at the same DRE = 100% and 37% less compared to Method B (~4 min) at 3.7 times higher DRE. For 100 commits per month, the savings compared to Method C are ~USD 484 (USD 208 vs. USD 692), which confirms the economic feasibility of the automated approach.

To compare the proposed method in terms of reliability, we will use the systems proposed in Section 5. 2, a simulation of 4800 requests in 600 seconds was conducted at different failure levels. The improved system (developed with full Pentagon coverage) is compared with the standard system (with Pyramid coverage). The results are given in Table 7.

Table 7

Comparing the accessibility of standard and improved systems

Failure rate	Standard system	Improved system	Improvement
10%	95.0%	99.3%	+4.5%
20%	90.0%	97.7%	+8.6%
50%	75.0%	92.1%	+22.8%

Our results confirm that the improved system, designed with full Pentagon model coverage, demonstrates consistently higher availability under all failure scenarios. The greatest effect is observed at a critical failure level of 50%: availability increases from 75.0% to 92.1% (+22.8%).

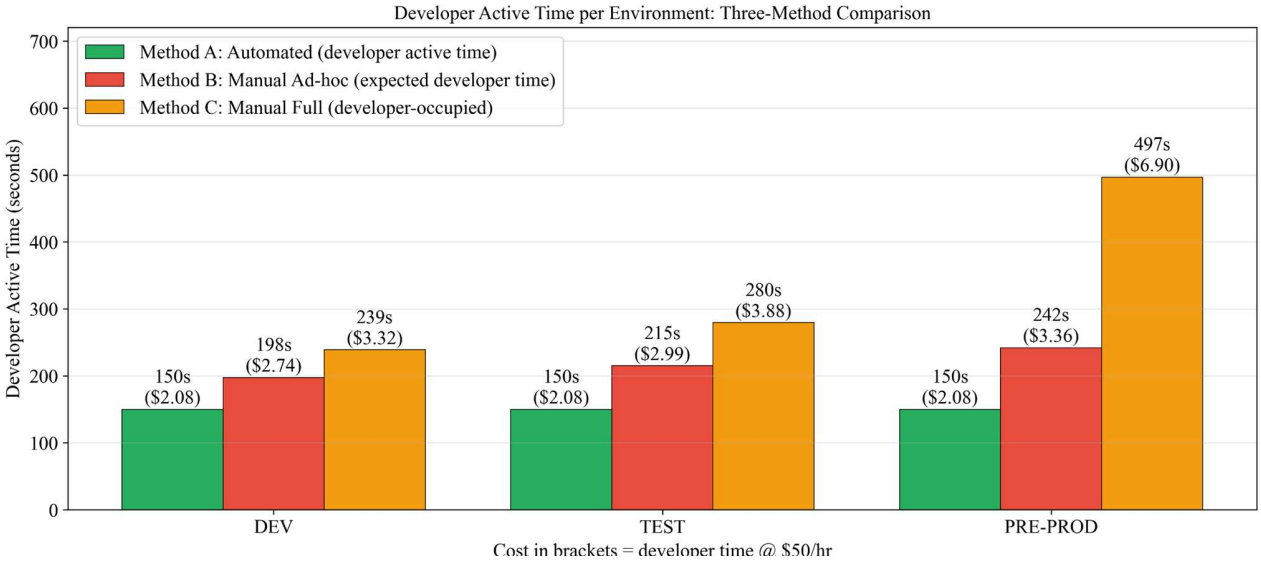

Fig. 6. Comparison of developer active time and cost for three testing methods: A (automated, ~2.5 min), B (ad-hoc, ~3.3–4.0 min), and C (manual full, 4.0–8.3 min) at a rate of USD 50/hour

6. Discussion of results based on investigating the method of automated integration of the Pentagon model into the CI/CD pipeline

The proposed structure of the method is represented in the form of a diagram (Fig. 1) and a table of layer distribution (Table 1). The proposed method formalizes the distribution of six testing layers between four CI/CD pipeline environments based on the cumulative principle: each subsequent environment inherits all the checks of the previous one and adds a new layer according to the level of readiness. Due to this, layers L4 and L6, which are performed with only 25% and 6% probability in the manual ad-hoc approach (Table 3), become mandatory deterministic steps of the pipeline. Unlike [8], in which non-functional testing methods are not among the mandatory ones in the pipeline, and [11], in which chaos testing is implemented outside the pipeline, the proposed method integrates L4 and L6 as automated CI/CD steps with formalized activation conditions for the first time.

Two system variants ensure the purity of the comparative study: the standard system with coverage of layers L1, L2, L5 reflects common industrial practice, while the improved one implements the proposed method with full coverage of L1–L6 and verified fault tolerance patterns. The functional identity of both variants eliminates the influence of business logic on the comparison results.

Time measurements (Table 2, Fig. 2) show that the full PRE-PROD cycle takes 256.8 s (4.3 min). Layer L6 is dominant (143.4 s, 56% of the total time), but it is activated only for PRE-PROD tagged releases, not for every commit. Layers L1–L4 for DEV and TEST together take 31.6 s, which provides fast feedback. The obtained values confirm the practical applicability of the method.

The results of the comparative analysis of defect detection (Table 4, Fig. 4) demonstrate that method A and method C provide DRE = 100% (12/12 defects), while method B achieves only ~27% (~3.2/12). Monte Carlo simulation (1000 iterations, seed = 42; Table 5, Fig. 5) confirms that none of the 1000 iterations of method B reached 12/12 – the result that method A provides deterministically. The high variability of method B ($\sigma = 1.5$) indicates the fundamental unreliability of the human factor as a mechanism for guaranteeing coverage. The "blind zone" is the layers L4 (25%) and L6 (6%), which account for 73% of undetected defects.

Cost-effectiveness analysis (Table 6, Fig. 6) reveals that the proposed method (A) provides the lowest developer active time (~2.5 min) at DRE = 100%. Compared to method C (8.3 min, DRE = 100%) – a 70% reduction in time; compared to method B (~4 min, DRE ~27%) – a 37% reduction at 3.7 times higher DRE. For 100 commits/month, the savings relative to method C are ~ 484 (208 vs. USD 692). Reliability validation (Table 7) confirms that the improved system provides 22.8% higher availability than the standard one at a critical failure rate of 50% (92.1% vs. 75.0%). The greatest effect at critical load is explained by the fact that fault tolerance patterns (retry_with_backoff, CircuitBreaker), verified by the L4 layer, provide controlled degradation instead of complete failure. The obtained increase in reliability is consistent with the empirically established pattern [25]: among the 15 identified factors determining software reliability, the decisions of the initial stage of the Lifecycle have the greatest impact – the proposed method implements this principle as the structure of the test coverage is formalized at the level of the pipeline architecture, before the development of specific components begins.

It is necessary to note a number of limitations in our study. The L6 layer was measured only on the basis of one run of the FIS experiment; for the reliability of the results in the future, testing on a larger number is planned. The method was validated on a small system with 3 microservices.

Among the shortcomings, it should be noted that the implementation of the method involves writing a larger number of automated tests compared to the testing pyramid, which requires an initial investment of time. In addition, the L6 layer requires special IAM rights for AWS FIS, which may be a limitation in organizations with strict policy models.

Promising areas of development include researching the method for systems with 10+ microservices. The second area is dynamic adjustment of the composition of layers depending on the type of code change to reduce feedback time. The

third area is integration with Kubernetes-based chaos tools (Litmus, Chaos Mesh) for environments outside AWS.

7. Conclusions

1. A structure of the method for automated integration of six layers of the Pentagon model into a multi-stage CI\CD pipeline with progressive distribution across four environments (DEV, TEST, PRE-PROD, PROD) has been proposed. The method is formalized in the form of 10 steps with clear triggers and transition criteria, which allows the team to uniquely reproduce the method in its own infrastructure.

2. Two versions of the microservice student administration system on the AWS SAM platform have been implemented – standard (layers L1, L2, L5) and improved (layers L1–L6) – as functionally identical objects of comparative research.

3. The real-time of all six layers on the AWS cloud infrastructure (eu-central-1) was measured: L1 = 8.1 s (σ = 2.0 s), L2 = 5.4 s (σ = 1.1 s), L3 = 7.3 s, L4 = 10.8 s (σ = 1.4 s), L5 = 13.8 s, L6 = 143.4 s. The full PRE-PROD cycle takes 256.8 s (4.3 min), which corresponds to an acceptable feedback time for the CI/CD pipeline.

4. Comparative analysis of defect detection efficiency revealed that the proposed method and Method C achieve DRE = 100% (12/12 defects), while Method B on average detects only ~27% (3.2/12). Monte Carlo simulation (1000 iterations) confirms zero variability for the automated approach (σ = 0.0) and significant variability for the manual one (Method B (ad-hoc) σ = 1.5), making the latter unreliable for critical systems.

5. Our cost-effectiveness analysis revealed that the proposed method reduces developer active time by 70% compared to Method C (2.5 min vs. 8.3 min) with the same DRE = 100%. Compared to Method B, the devised method provides 3.7 times higher DRE with 37% less time (2.5 min vs. ~4 min). For 100 commits/month, the proposed method provides 70% cost savings compared to Method C (~USD 484/month: USD 208 vs. USD 692) and 37% compared to Method B (USD 208 vs. ~USD 333) with insignificant AWS infrastructure costs (~USD 0.00019/launch). A comparative evaluation of the proposed method confirmed its effectiveness in two areas: in terms of cost-effectiveness – a 70% reduction in developer active time compared to method C (2.5 min vs. 8.3 min) with the same DRE = 100% and savings of ~USD 484/month for 100 commits; in terms of reliability – the improved system provides availability 22.8% higher than the standard one with a critical failure rate of 50% (92.1% vs. 75.0%), which is a direct consequence of the verification of fault tolerance patterns by layers L4 and L6 in the CI/CD pipeline.

Conflicts of interest

The authors declare that they have no conflicts of interest in relation to the current study, including financial, personal, authorship, or any other, that could affect the study and the results reported in this paper.

Funding

The study was conducted without financial support.

Data availability

The data will be provided upon reasonable request.

Use of artificial intelligence

In the process of conducting the research and writing this paper, the authors used generative artificial intelligence tools (Claude, Anthropic) for text formatting. The authors confirm that all scientific results, measurements, and analysis were performed directly by the authors and checked by them for correctness.

Authors' contributions

Oleh Kuzmych: Conceptualization, Methodology, Software, Formal analysis, Investigation, Writing – original draft; **Maksym Seniv**: Conceptualization, Supervision, Writing – review & editing.

References

1. Humble, J., Forsgren, N., Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. Portland: IT Revolution Press, 286. Available at: https://ebooks.karbust.me/Technology/Accelerate%20The%20Science%20of%20Lean%20Software%20and%20DevOps%20Building%20and%20Scaling%20High%20Performing%20Technology%20Organizations%20by%20Nicole%20Forsgren%20Jez%20Humble%20Gene%20Kim.pdf
2. Boehm, B., Basili, V. R. (2001). Top 10 list [software development]. Computer, 34 (1), 135–137. https://doi.org/10.1109/2.962984
3. Ponce, F., Verdecchia, R., Miranda, B., Soldani, J. (2025). Microservices testing: A systematic literature review. Information and Software Technology, 188, 107870. https://doi.org/10.1016/j.infsof.2025.107870
4. Waseem, M., Liang, P., Marquez, G., Salle, A. D. (2020). Testing Microservices Architecture-Based Applications: A Systematic Mapping Study. 2020 27th Asia-Pacific Software Engineering Conference (APSEC), 119–128. https://doi.org/10.1109/apsec51365.2020.00020
5. 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
6. Laukkanen, E., Itkonen, J., Lassenius, C. (2017). Problems, causes and solutions when adopting continuous delivery – A systematic literature review. Information and Software Technology, 82, 55–79. https://doi.org/10.1016/j.infsof.2016.10.001
7. 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
8. Mascheroni, M. A., Irrazábal, E. (2018). Continuous Testing and Solutions for Testing Problems in Continuous Delivery: A Systematic Literature Review. Computación Y Sistemas, 22 (3). https://doi.org/10.13053/cys-22-3-2794

9. Zhang, Y., Vasilescu, B., Wang, H., Filkov, V. (2018). One size does not fit all: an empirical study of containerized continuous deployment workflows. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 295–306. https://doi.org/10.1145/3236024.3236033
10. Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., Ding, D. (2021). Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. IEEE Transactions on Software Engineering, 47 (2), 243–260. https://doi.org/10.1109/tse.2018.2887384
11. Jernberg, H., Runeson, P., Engström, E. (2020). Getting Started with Chaos Engineering - design of an implementation framework in practice. Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 1–10. https://doi.org/10.1145/3382494.3421464
12. Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C. (2016). Chaos Engineering. IEEE Software, 33 (3), 35–41. https://doi.org/10.1109/ms.2016.60
13. Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K., Sekar, V. (2016). Gremlin: Systematic Resilience Testing of Microservices. 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), 57–66. https://doi.org/10.1109/icdcs.2016.11
14. Meiklejohn, C. S., Estrada, A., Song, Y., Miller, H., Padhye, R. (2021). Service-Level Fault Injection Testing. Proceedings of the ACM Symposium on Cloud Computing, 388–402. https://doi.org/10.1145/3472883.3487005
15. Bouizem, Y., Dib, D., Parlavantzas, N., Morin, C. (2023). Integrating request replication into FaaS platforms: an experimental evaluation. Journal of Cloud Computing, 12 (1). https://doi.org/10.1186/s13677-023-00457-z
16. Kuzmych, O., Seniv, M. (2024). An Improved Approach to Increase the Fault Tolerance of Microservice Software Through Automated Functional and Fault Tolerance Testing. 2024 IEEE 19th International Conference on Computer Science and Information Technologies (CSIT), 1–4. https://doi.org/10.1109/csit65290.2024.10982594
17. Cohn, M. (2009). Succeeding with Agile: Software Development Using Scrum. Boston: Addison-Wesley Professional, 504. Available at: https://www.researchgate.net/publication/234803335_Succeeding_with_Agile_Software_Development_Using_Scrum
18. Welcome to the State of Developer Ecosystem Report 2024. JetBrains. Available at: https://www.jetbrains.com/lp/devecosystem-2024/
19. State of Cloud Native Development Q3 2025. CNCF. Available at: https://www.cncf.io/wp-content/uploads/2025/11/cncf_report_stateofcloud_111025a.pdf
20. The State of Chaos Engineering in 2021. Gremlin Inc. Available at: https://www.gremlin.com/blog/the-state-of-chaos-engineering-in-2021
21. Rubinstein, R. Y., Kroese, D. P. (2016). Simulation and the Monte Carlo Method. Wiley Series in Probability and Statistics. https://doi.org/10.1002/9781118631980
22. AWS Pricing. Amazon Web Services. Available at: https://aws.amazon.com/pricing/
23. Developer Survey 2024. Stack Overflow. Available at: https://survey.stackoverflow.co/2024/
24. The Economic Impacts of Inadequate Infrastructure for Software Testing (Planning Report 02-3). National Institute of Standards and Technology (NIST). Available at: https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf
25. Yakovyna, V., Seniv, M., Symets, I. (2020). The Relation between Software Development Methodologies and Factors Affecting Software Reliability. 2020 IEEE 15th International Conference on Computer Sciences and Information Technologies (CSIT), 377–381. https://doi.org/10.1109/csit49958.2020.9321937