https://doi.org/10.33271/nvngu/2026-3/146

M. M. Seniv,
orcid.org/0000-0003-1044-4628,
O. M. Kuzmych*,
orcid.org/0000-0002-9749-6385

Lviv Polytechnic National University, Lviv, Ukraine
* Corresponding author e-mail: oleg.kuzmuch@gmail.com

MODEL FOR IMPROVING THE RELIABILITY OF MICROSERVICE SOFTWARE DURING THE FUNCTIONAL TESTING PHASE

Purpose. The development and justification of a new testing model aimed at improving the reliability of software systems built using microservice architecture. The proposed model systematically takes into account the specifics of distributed systems, the characteristics of microservice architecture, and fault tolerance aspects that are insufficiently covered by classical testing approaches.

Methodology. A six-layer "Pentagon" testing model is proposed. The limitations of existing models ("testing pyramid" and "testing honeycomb") in the context of cloud-native systems were analyzed. A combined validation approach was applied: a survey of IT professionals to identify the importance of each testing layer and comparative load testing of two system versions – the standard one and the one enhanced with the "Pentagon" testing model. Key reliability metrics were evaluated, including Failure Rate, Availability, MTBF, etc.

Findings. The study resulted in the formalization of the "Pentagon" testing model, which consists of six sequential layers. Evaluation criteria were developed to quantitatively assess the contribution of each layer to overall system reliability. Expert feedback confirmed the higher importance of the Fault-Tolerance, End-to-End, and Global Integration Testing layers compared to Unit Testing. Experimental results demonstrated that the system tested using the proposed model exhibited higher reliability under failure conditions: the number of failures decreased by 9.9 %, MTBF increased by 9.5 %, and Availability improved by 2 %.

Originality. For the first time, a conceptual "Pentagon" model has been developed, which, unlike the existing ones, structures the microservice software testing process into six layers. A unique feature of the model is the isolation of fault tolerance verification and resilience to chaotic infrastructure failures into separate mandatory stages, allowing for the formalization of these processes. Based on the developed model, an approach to functional software testing for reliability improvement is proposed for the first time, which involves the systemic integration of the aforementioned testing levels and Chaos Engineering into the Software Development Lifecycle (SDLC). Unlike classical approaches, such integration ensures a comprehensive improvement in software reliability indicators. A method for evaluating the importance of testing layers based on five weighted criteria has been developed, allowing for an objective assessment of the importance and scope of each model layer in the software testing strategy.

Practical value. The model provides a structured foundation for forming a testing strategy for microservice applications, facilitates early detection of architectural flaws, reduces the risks of failures in production, and lowers the costs of their remediation.

Keywords: *software, microservice architecture, functional testing, "Pentagon" testing model, "chaos testing"*

Received: 03 November 2025
Accepted: 18 February 2026
Available online: 27 June 2026

Introduction. The modern era of information technology is characterized by a widespread adoption of distributed architectures, such as microservices and serverless platforms, which has become a response to the growing demands for scalability and flexibility of software systems [1]. However, this architectural paradigm creates new challenges in the field of reliability assurance. A large number of components interacting over a network increases the probability of complex, often unpredictable failures [2].

Traditional software testing strategies prove to be not always adequate for such systems. In particular, the approach known as the "Testing Pyramid," which recommends a predominance of unit tests [3], and the alternative "Testing Honeycomb" model, which emphasizes integration checks [4], do not sufficiently take into account the specifics of distributed environments. These models do not offer a systematic approach to verifying fault tolerance (resilience) and analyzing system behavior under real infrastructure problems, which is critically important for cloud applications.

Underestimating the testing of fault tolerance aspects can lead to situations where a local failure of one service causes cascade degradation or a complete failure of the entire system [1]. Existing methodologies often do not provide clear instructions on verifying risks associated with network interaction or temporary unavailability of external dependencies [5, 6].

The relevance of this problem is confirmed by the growing number of scientific publications and the interest of researchers in new architectures [7], and particularly in methods for increasing their fault tolerance [8, 9]. At the same time, a literature analysis indicates a deficit of comprehensive methodologies that would systematically integrate fault tolerance testing into the development lifecycle of such systems.

The purpose of this study is to develop a model for improving the reliability of serverless microservice software by implementing a comprehensive approach to functional testing.

© Seniv M. M., Kuzmych O. M., 2026

Literature review. Modern software is becoming increasingly complex and distributed, leading to a decrease in its reliability and fault tolerance. Simultaneously, user requirements for the uninterrupted operation of software systems are growing. In the context of microservice architecture, this problem acquires special relevance due to the high fragmentation of the system and the complexity of interaction between individual components. Analysis of recent research on microservice fault tolerance shows a variety of approaches to solving this problem. Zhao, et al. [10] propose a classification of methods for ensuring the fault tolerance of microservice systems into three categories: proactive, reactive, and hybrid ones.

Proactive methods are aimed at preventing errors and include:

1. Static code analysis for identifying potential errors before deployment [11].
2. Modeling and verification of system behavior [12].
3. Comprehensive testing at the development stage [13].

Reactive methods focus on rapid recovery after failures:

1. Use of "Circuit Breaker" patterns to isolate faulty services [14].
2. Retry mechanisms with exponential backoff [15].
3. Backup and automatic recovery.

Hybrid methods combine elements of both approaches:

1. Chaos testing to identify potential points of failure.
2. Adaptive recovery algorithms based on machine learning

Despite the variety of methods, their effectiveness is often limited due to the lack of a systematic approach to fault tolerance testing, especially in the context of serverless architectures.

Functional testing is one of the key approaches to improving software reliability. Traditionally, it is represented by the "Testing Pyramid" model proposed by Mike Cohn [5], which includes three levels:

- unit tests – check individual system components;
- integration tests – check the interaction between components;
- end-to-end tests – check the functionality of the entire system from the user's perspective.

With the development of microservice architecture, an alternative model was proposed – the "Testing Honeycomb" [6], which modifies the pyramid by placing greater emphasis on integration testing rather than unit testing.

Unsolved aspects of the problem. The existing testing models, such as the "Testing Pyramid" and "Testing Honeycomb," do not pay sufficient attention to the specific challenges of such systems, in particular, testing fault tolerance mechanisms and system behavior under cloud infrastructure failure conditions. Traditional functional testing models do not provide a clear answer to the question of how to systematically verify the operation of reactive mechanisms and the system's readiness for chaotic failures. This creates a "blind spot," especially in systems deployed in a serverless environment, and confirms the deficit of complex models that would systematically integrate fault tolerance testing into the software development lifecycle.

Purpose of the article and task formulation. The purpose of this study is to develop a model for improving the reliability of software with microservice architecture by implementing a comprehensive approach to functional testing, which takes into account the features of such systems and ensures a systematic approach to identifying potential points of failure.

To achieve this goal, the following tasks were set:

- to analyze existing models for ensuring the reliability of microservice software and determine their limitations;
- to develop a new functional testing model that eliminates the shortcomings of existing ones and takes into account the features of software with microservice architecture;
- based on the model, to propose an approach to functional software testing for improving its reliability, which will be integrated into the SDLC;
- to develop a method for evaluating the importance of testing layers which allows for an objective assessment of the importance and volume of each layer of the "Pentagon" model;
- to conduct an experimental verification of the effectiveness of the proposed model through a comparative analysis of key characteristics and reliability metrics.

Based on the analysis of existing approaches and models for ensuring reliability and functional testing of microservice software, we formulated requirements for the new model:

Comprehensiveness: coverage of not only standard testing levels, from modular to systemic ones;

Specificity: consideration of the features of serverless architecture;

Integration: possibility of embedding into the CI/CD process;

Proactivity: detection of potential problems before they arise in the production environment;

Efficiency: optimal ratio between testing costs and reliability improvement.

Testing approach for improving software reliability based on the application of the "Pentagon" model in SDLC. Before delving into the structure of the model itself, it is important to define its place in the general Software Development Life Cycle (SDLC). Accordingly, an approach to functional software testing was developed to increase its reliability, which consists of the systemic integration of testing levels: unit, local integration, global integration, fault-tolerance testing, E2E, and chaos testing into the software development lifecycle. Below (Fig. 1) it is shown how the developed approach integrates the "Pentagon" model into the SDLC, deepening the implementation and testing stages compared to other approaches.

Stage 4 (Implement). At this stage, developers do not just write code, but immediately cover it with unit tests. A key difference in our approach is the integration of local integration tests here as well. This allows the developer to get immediate feedback on whether they have violated the "contract" of interaction with other components, without waiting for the full build and deployment.

Stage 5 (Test). This stage transforms from a simple functionality check to a comprehensive reliability vali-

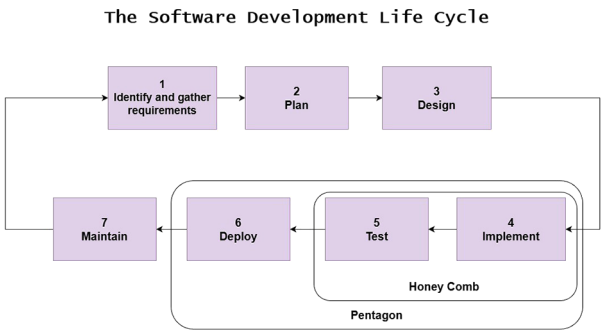

Fig. 1. Place of the "Pentagon" model in SDLC

dation. If in classical approaches E2E tests are predominantly performed here, the "Pentagon" adds two mandatory, formalized stages: fault tolerance testing and chaos testing. This allows shifting from the question "does the functionality work?" to the question "will the system survive a failure?".

Stage 6 (Deploy). The decision to deploy to production is made not only based on successful functional tests but on the basis of a comprehensive reliability report. Successful passage of the "Pentagon" in intermediate environments gives the team well-grounded confidence that the system is capable of withstanding failures.

The developed approach complements the standard SDLC life cycle, deepening testing to achieve higher software quality before its deployment (Deploy, stage 6) and subsequent maintenance (Maintain, stage 7).

According to the requirements defined above, a "Pentagon" testing model was developed, which, unlike the traditional three-level division, consists of six levels [16].

The "Pentagon" Testing Model is a conceptual six-level model of functional testing for microservice architecture, which expands traditional models (in particular, such as the "pyramid" and "honeycomb") by adding specialized layers for validating fault tolerance and system behavior under chaotic failure conditions. Structurally, the model consists of sequential verification levels: from isolated components to system integration under failure conditions.

Description of the "Pentagon" model layers.

Layer 1: Unit testing. Unit testing is the lowest level of testing, performing verification of individual functions or methods in complete isolation from other system components. This type of tests focuses on the correctness of code logic implementation, handling of exceptional situations, boundary cases, and correctness of output results. For implementation, frameworks like Pytest and tools for mocking dependencies are used, allowing errors to be detected at the initial development stage. Unit tests allow for quick localization of defects, simplify code maintenance, and make the system more stable at the low implementation level.

Layer 2: Local integration testing. Local integration testing checks the interaction of various software parts in a local environment, for example, the interaction of AWS Lambda functions with local database emulators (Moto, Localstack). These tests are aimed at detecting errors in the interaction of individual components, including the correctness of CRUD operations execution. Their conduct ensures rapid feedback regarding the correctness of component interaction without the need for deployment to a real environment, allowing problems to be effectively eliminated before subsequent testing stages.

Layer 3: Global integration testing. Global integration testing performs verification of the functioning of the entire system in conditions maximally close to reality, using fully deployed services, such as AWS Lambda, API Gateway, and DynamoDB. Using HTTP clients (Postman, automated Python scripts), the full interaction chain from the client to the final data storage is tested, checking the correctness of HTTP requests, responses, and interaction with real infrastructure. Such an approach allows ensuring the correctness of component integration in an environment that maximally corresponds to operating conditions, reducing risks of unexpected problems in real system operation.

Layer 4: Fault-tolerance testing. Fault-tolerance testing is intended to verify the system's ability to function correctly under error and failure conditions. At this stage, retry mechanisms (Retries) and automatic blocking of erroneous requests (Circuit Breaker) are tested, which help the system avoid cascade failures and quickly recover after temporary malfunctions. Testing is conducted by artificially creating errors through special requests, which allows accurately assessing how the system processes and recovers from failures. This layer ensures high system availability and minimizes the impact of errors on end-users.

Layer 5: E2E testing (end-to-end). E2E testing is the highest level of integration verification and performs verification of the entire system from the point of view of the end-user. They are performed using tools that simulate real user behavior (e.g., Postman, Cypress, Selenium), testing full business scenarios, such as user registration, record management, or transactions. This allows obtaining full confidence that the system works correctly as a holistic product, detecting defects at the business process integration level, and also confirms that the user will get exactly the result they expect.

Layer 6: Chaos testing. Chaos testing is the final layer of the "Pentagon" model, which evaluates system behavior in case of unpredictable, random, and critical emergency situations. Using Chaos Engineering methodologies and specialized tools (e.g., AWS Fault Injection Simulator), various types of failures are purposefully simulated: database crashes, network connection breaks, failure of individual components. This allows detecting hidden problems, assessing the real readiness of the system for accidents, and ensuring that even in the most critical situations, the system will maintain its stability and quickly recover from errors.

Such a testing scheme should allow detecting errors at early stages and correcting them before they get into production and lead to a failure, which in turn increases software reliability.

Concept and evaluation of testing layer importance. The classical model, such as the "pyramid," usually interprets the size of layers as the recommended number of tests. In our model, we propose a different interpretation: the size of the layer reflects its *importance* for ensuring the general reliability of the system. In distributed systems, reliability is determined not so much by the flawless operation of each component in isolation, but rather by their ability to stably interact with each other, especially under failure conditions.

To move from the qualitative concept of "importance" to an objective assessment, we propose using a quantitative evaluation method based on developer expert assessment.

Quantitative importance evaluation method. This method allows measuring the *subjective importance* of each layer and the level of trust it provides to the team. It is based on expert assessment according to several key criteria that form the concept of "importance for reliability".

Survey process. To collect data, a specialized anonymous questionnaire form was developed. The survey participants are engineers with experience in microservice architecture. The survey is divided into two stages:

Stage 1: Determination of weight coefficients. At the first stage, the respondents are asked to distribute 100 points (or 100 %) among five importance criteria depending on which of them, in their opinion, makes the greatest contribution to the reliability of microservice systems, taking into account its cost and speed. The average values obtained from all experts form objective weight coefficients (W) for further calculations.

Stage 2: Evaluation of model layers. At the second stage, the respondents are asked to evaluate each of the six layers of the "Pentagon" testing model according to each of the five criteria, using a scale from 1 (least important/effective) to 10 (most important/effective).

Criteria for evaluation. To calculate the final "Importance Index," each criterion was assigned a weight coefficient (W), reflecting its priority in the context of ensuring the reliability of distributed systems. The sum of all coefficients equals 1.0.

Direct Validation of fault-tolerance mechanisms (DVFT) evaluates the ability of the testing layer to purposefully verify the operation of fault tolerance mechanisms (retries, circuit breakers). A high score means that tests validate the system's ability to survive and recover from partial failures, which is the norm for microservices.

Potential to prevent critical failures (PFCF) measures how effectively the testing layer finds serious errors that can lead to cascade failures, data loss, or complete service unavailability. This criterion focuses on the quality of found defects, not their quantity.

Reduction of deployment risks (RDR) evaluates the psychological factor: how much the successful passage of tests of a certain layer increases the team's trust in the stability and safety of the new release. High RDR means that tests imitate real conditions well, allowing for an informed decision on deployment.

Early defect detection (EDD) measures the ability of the layer to find errors at the earliest possible stage of the CI/CD pipeline. A high score means that tests provide quick feedback to the developer, significantly cheapening the cost of fixing defects.

Cost-effectiveness (CE) evaluates the ratio between the cost of the layer (time for development and execution, complexity of support) and the benefit it brings. A high score means that the testing layer finds important errors at relatively low resource costs.

The evaluation process consists in each criterion receiving its weight coefficient (W), and a group of experts evaluating each layer according to each criterion. The final "Importance Index" is calculated by the weighted sum formula.

$$ImportanceIndex_{LT} = (Score_{DVFT} \times W_{DVFT}) + \\ + (Score_{PFCF} \cdot W_{PFCF}) + (Score_{RDR} \cdot W_{RDR}) + \\ + (Score_{EDD} \times W_{EDD}) + (Score_{CE} \cdot W_{CE}). \quad (1)$$

Results of expert survey regarding testing layer importance. Let us analyze the results obtained during the survey of 23 respondents regarding the importance of testing layers using a Google form. Analysis of the obtained data will allow obtaining a quantitative assessment of the importance of each testing layer.

Determination of weight coefficients. At the first stage of the study, experts were asked to distribute 100 points among five importance criteria. The averaged results, normalized to a sum of 1.0, formed the objective weight coefficients (W) used in further calculations:

- *W_DVFT*: 0.241 (24.1 %);
- *W_PFCF*: 0.225 (22.5 %);
- *W_RDR*: 0.202 (20.2 %);
- *W_EDD*: 0.174 (17.4 %);
- *W_CE*: 0.158 (15.8 %).

At the second stage, the experts evaluated each of the six testing layers of the model according to each of the five criteria. Below in Table 1, I present the average estimates of each criterion for each testing layer.

Also, for greater clarity, let us present these results in the form of a bar chart.

Analysis of this data allows drawing intermediate conclusions. Each layer has its clear strengths. Unit and Local Integration tests are undisputed leaders by the Cost/Efficiency (CE) criterion; Global Integration does not lag significantly behind by these indicators. Direct Validation of Fault-Tolerance Mechanisms (DVFT) and Potential for Critical Failures (PFCF). E2E and Global Integration tests received the highest scores for

Table 1

Results of expert survey regarding testing layer importance

Testing Layer	Average PFCF	Average DVFT	Average RDR	Average EDD	Average CE
Unit Testing	5	4.39	4.13	8.65	7.82
Local Integration Testing	6.3	5.65	5.56	7.69	7
Global Integration Testing	7.08	6.56	6.82	6.39	6
Fault-Tolerance Testing	7.60	7.95	7.60	5.47	4.91
End-to-End Testing	8	7.65	7.08	4.95	5.13
Chaos Testing	7.52	7.52	7.73	1.82	3.26

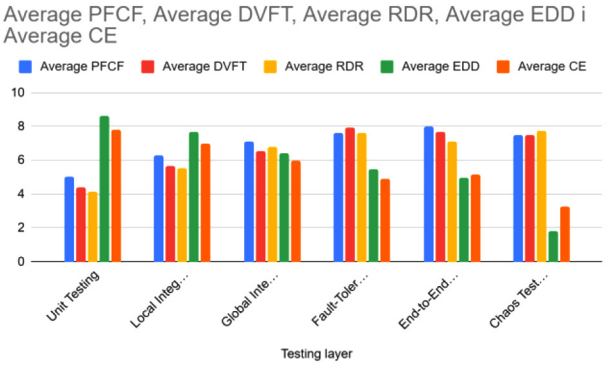

Fig. 2. Diagram of expert survey results regarding testing layer importance

Table 2

Results of calculating final importance indices of layers

Testing Layer	Importance Index (out of 10)
Chaos Testing	5.9
End-to-End Testing	6.7
Fault-Tolerance	6.8
Global Integration Testing	6.6
Local Integration Testing	6.3
Unit Testing	5.7

Reduction of Deployment Risks (RDR). At the same time, Unit and Local Integration tests are practically incapable of validating fault tolerance, confirming their limitation for distributed systems. One can also see low scores for Chaos tests by Early Defect Detection (EDD) and Cost/Efficiency indicators.

Now let us calculate the final "Importance Index" for each layer using formula (1) of the weighted sum and format this as Table 2.

Summarizing all the above survey results, one can come to the following conclusions:

1. Prioritization of interaction and fault tolerance. The highest "Importance Indices" were received by the central layers of the model: Fault-Tolerance Testing (6.8), End-to-End Testing (6.7), and Global Integration Testing (6.6). This indicates that the expert community considers the validation of systemic interaction and resilience to failures more significant for microservice reliability than testing the isolated logic of individual components.

2. Pragmatic assessment of chaos testing. Despite the highest scores for fault tolerance validation (DVFT) and critical defect detection (PFCF) criteria, the final index of Chaos Testing (5.9) is relatively low. This indicates a pragmatic paradox: although experts recognize the theoretical power of this approach, its high cost and implementation complexity (reflected in the low score for the "Cost/Efficiency" criterion) significantly reduce its overall perceived importance compared to more deterministic types of testing.

3. Empirical confirmation of the "Pentagon" model. The obtained "form of importance" radically differs from classic "pyramid" and "honeycomb" models. It has a narrow base, represented by Unit Testing (5.7), with gradual expansion through integration layers to a peak at the Fault-Tolerance Testing level, after which a slight narrowing occurs. Such a structure, obtained based on real data, is weighty empirical confirmation of the feasibility of the proposed "Pentagon" model, which shifts the focus from the quantity of tests to their ability to ensure the comprehensive reliability of a distributed system.

Investigation of the impact of the "Pentagon" testing model on software reliability metrics. Before moving to the proof of effectiveness itself and software development, let us define the metrics by which we will measure reliability for comparing software using the "Pentagon" and the testing honeycomb. To compare the effectiveness of the "Pentagon" with the analog, the following metrics will be measured:

MTBF (Mean Time Between Failures): average time between failures [17].

$$MTBF = \frac{\text{Test duration}}{\text{Number of failures during this period}}.$$

Failure Rate: frequency of failures [18].

$$\text{Failure Rate} = \frac{\text{Number of failures}}{\text{Test duration}}.$$

System Availability: availability of the system [19].

$$\text{System Availability} = \frac{\text{Number of requests} - \text{Number of failures}}{\text{Number of requests}} \cdot 100\,\%.$$

Failure Avoidance: the number of prevented failures [20].

Failure Avoidance = Number of failures that should have resulted in failures – Number of failures that failed to result in failures.

Measurement of these metrics will allow evaluating the effectiveness of the "Pentagon" compared to the analog and drawing conclusions about its advantages and disadvantages, as well as allowing measuring the importance described above.

Testing Plan. With the aim of verifying the developed model for increasing the fault tolerance of microservice software, it is proposed to conduct an experimental study which will include the following stages:

1. Creation of the base system.

The base system is a typical implementation of a student data management system with CRUD functionality using the testing honeycomb.

2. Creation of the enhanced system.

The enhanced system is the implementation of the same functionality, but with the introduction of the proposed "Pentagon" testing model.

3. Conducting stress testing:

- creation of realistic load scenarios simulating system operation in real conditions under load and with error injection;
- application of created scenarios to both systems over a specified time;
- collection of indicators for metric calculation.

4. Analysis of results:

- calculation and comparison of metric values for the base and experimental systems obtained during stress testing and fault injection testing;
- conclusions about the effectiveness of the "pentagon" testing model compared to the base system, taking into account the results of both types of testing.

Description of the software complex for testing. A software complex simulating a real microservice system was developed. The architecture consists of three independent serverless services deployed in AWS (Fig. 4). The first is the Student Service (student-service): responsible exclusively for student profiles (creation, editing, retrieving data). The second is the Enrollment Service (enrollment-service): responsible for enrolling students in courses and viewing their enrollments. The third is the Notification Service (notification-service): respon-

sible for asynchronous sending of notifications (e.g., email) about successful enrollment.

Key element: Interaction between services.

We implement two types of interaction often found in microservice architecture in serverless implementation:

1. Synchronous interaction (direct API call):

- scenario: Get full information about a student along with their courses;

- implementation: student-service upon a request to GET/students/{id}/details will:

- retrieve student data from its StudentsTable;

- make a direct synchronous HTTP request to enrollment-service (GET/students/{id}/enrollments) to get the list of courses;

- combine results and return to the client.

2. Asynchronous interaction (via message queue):

- scenario: when a student is successfully enrolled in a course, an email confirmation needs to be sent;

- implementation:

1) enrollment-service after successful creation of a record in EnrollmentsTable does not call notification-service directly;

2) instead, it sends a message (event) StudentEnrolled to the Amazon SQS (Simple Queue Service) queue;

3) the Lambda function of notification-service is automatically activated when a new message appears in the queue. It processes it and sends an email.

Conducting Stress Testing. For simulation of real load and fault injection, the *Artillery* platform was used. Stress testing was performed according to an identical scenario for both systems:

1. During ***half of the test time***, the system worked in normal mode without error injection, just under load.

2. During the ***second half***, intentional failures were launched (timeouts, DB unavailability, service crashes). Each testing session lasted ***5 minutes***, during which.

3. Parallel HTTP requests were generated to the API Gateway.

4. Using ***CloudWatch*** and ***Artillery***, metrics were collected: response time, the number of errors, recovery duration.

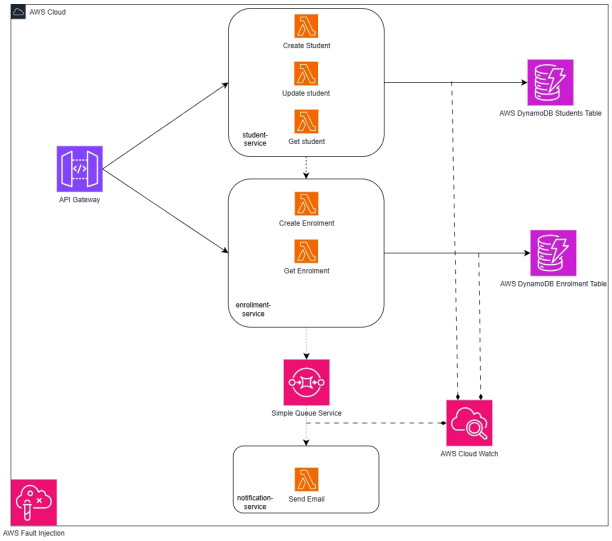

Fig. 3. Architectural scheme of the system on which testing will take place

5. All events were registered: the number of successful/failed requests, exception logging, retries/circuit breakers triggering.

The collected data will be used to calculate the metrics we indicated earlier.

Description and analysis of obtained results. In this section, the results of the experimental study of the influence of the proposed "Pentagon" model on the reliability of microservice software are presented. For quantitative assessment of effectiveness, comparative load testing of two versions of the system was conducted: "standard" (using the testing honeycomb) and "enhanced" (using the "Pentagon" model).

Both systems were subjected to identical stress testing lasting 600 seconds (10 minutes), generating 4,800 requests (8 requests/s). The testing scenario simulated real operating conditions: for the first 300 seconds, the system worked under stable load, for the next 300 seconds, periodic failures were introduced into the service operation.

As seen from Table 3 and Fig. 4, the "standard system" demonstrates stable operation during the first 5 minutes of the test (100 % Availability). However, with the start of dependency failure simulation (after the 5^{th} minute), the number of failures begins to grow rapidly (reaching 960 by the end of the test). This leads to a significant deterioration in reliability indicators: avail-

Table 3

Results of testing the standard system lasting 600 seconds and 4,800 requests

Time, s	Total Requests	Total Failures	MTB, s	Failure Rate, fail/s	Availability, %	Failure Avoidance
60	480	0	–	0	100	0
120	960	0	–	0	100	0
180	1,440	0	–	0	100	0
240	1,920	0	–	0	100	0
300	2,400	0	–	0	100	0
360	2,880	192	1.87	0.53	93.33	0
420	3,360	384	1.09	0.91	88.57	0
480	3,840	576	0.83	1.2	85	0
540	4,320	768	0.70	1.42	82.22	0
600	4,800	960	0.625	1.6	80	0

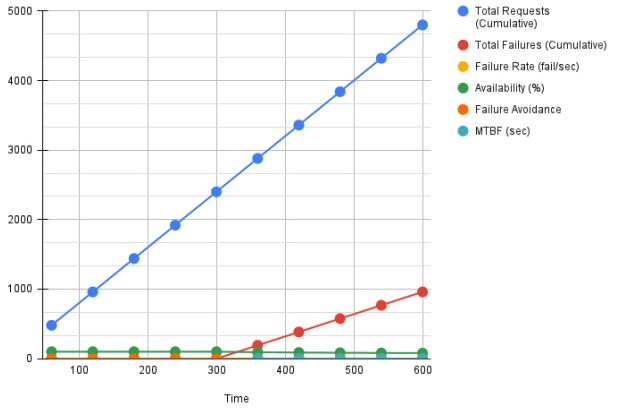

Fig. 4. Graph of testing results of the system using the testing honeycomb

Table 4

Results of testing the system using “Pentagon” testing lasting 600 seconds and 4,800 requests

Time, s	Total Requests	Total Failures	MTBF, s	Failure Rate, fail/s	Availability, %	Failure Avoidance
60	480	0	–	0	100	0
120	960	0	–	0	100	0
180	1,440	0	–	0	100	0
240	1,920	0	–	0	100	0
300	2,400	0	–	0	100	0
360	2,880	173	2.08	0.48	93.99	19
420	3,360	346	1.21	0.82	89.70	38
480	3,840	519	0.92	1.08	86.48	57
540	4,320	692	0.78	1.28	83.98	76
600	4,800	865	0.69	1.44	81.97	95

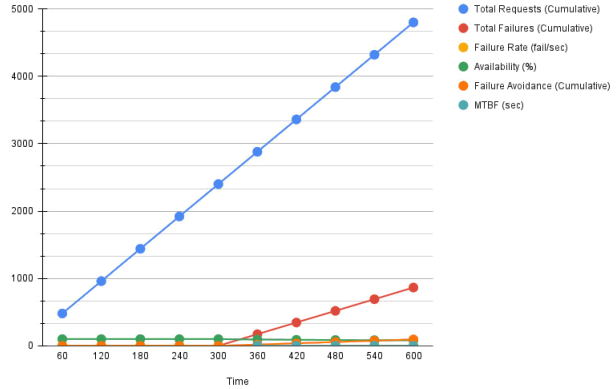

Fig. 5. Graph of testing results of the system using “Pentagon” testing

ability falls to 80 %, and the mean time between failures shortens to 0.63 seconds, indicating high system vulnerability to problems in dependent components.

The “Enhanced system” also experiences a significant impact from dependency failures as can be seen from Table 4 and Fig. 5, however, it demonstrates noticeable improvement compared to the standard one. The retry mechanism, despite low effectiveness in this scenario, was able to “save” 95 requests (Failure Avoidance). The total number of failures amounted to 865. This allowed maintaining availability at the level of ~82 %, which is 2 percentage points higher than that of the standard system. The mean time between failures also demonstrates a small improvement, reaching 0.69 seconds at the end of the test.

For clarity of comparison of load testing results of the two systems, let us compile Table 5 and Figs. 6–10.

For clarity of the difference, we will also demonstrate the testing results of the two systems on one graph.

Discussion of stress testing results. Analysis of the obtained results allows making the following conclusions:

1. Moderate, but stable increase in reliability: Implementation of retry and circuit breaker mechanisms allowed reducing the number of failures by approximately 10 % and increasing the mean time between failures by almost 10 %. System availability under failure conditions increased by ~2 percentage points, reaching ~82 %. These indicators demonstrate a small, but measurable positive effect from applying the fault tolerance mechanism, even under the condition of its low effectiveness (10 %) in this scenario of prolonged failure.

2. Confirmation of retry mechanism operation: The Failure Avoidance metric, equal to 95, indicates that the

Table 5

Comparison of testing results of two systems lasting 600 seconds and 4,800 requests

Metrics	“Standard system”	“Enhanced system”	Improvement
Total Failures	960	865	–9.9 %
MTBF	0.63	0.69	+9.5 %
Failure Rate	96.0	86.5	–9.9 %
Availability	80.00 %	81.98 %	+1.98 %
Failure Avoidance	0	95	+95

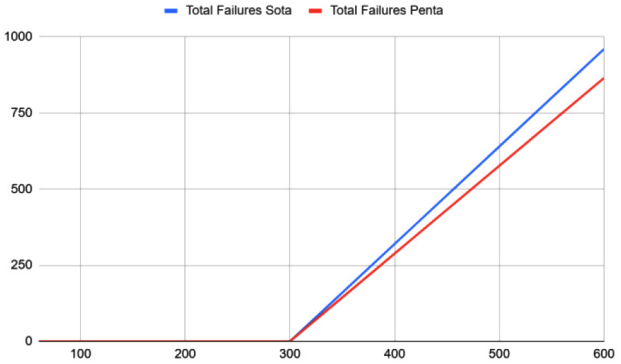

Fig. 6. Comparison of Total Failures of systems using testing honeycomb and “Pentagon” testing

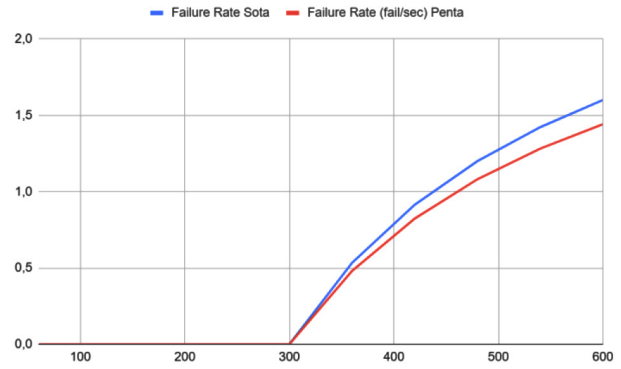

Fig. 7. Comparison of Failure Rate of systems using testing honeycomb and “Pentagon” testing

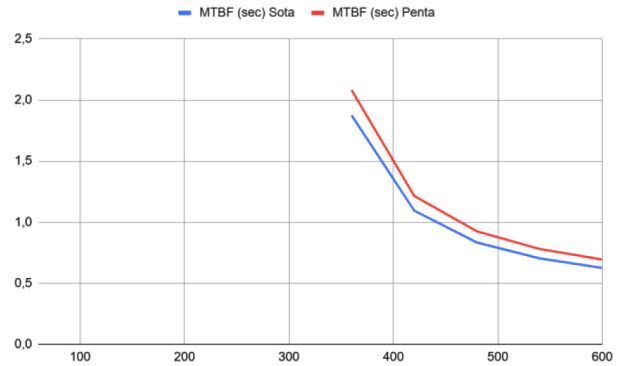

Fig. 8. Comparison of MTBF of systems using testing honeycomb and “Pentagon” testing

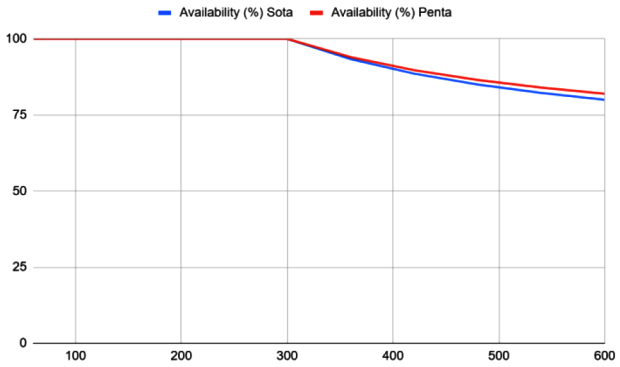

Fig. 9. Comparison of Availability of systems using testing honeycomb and "Pentagon" testing

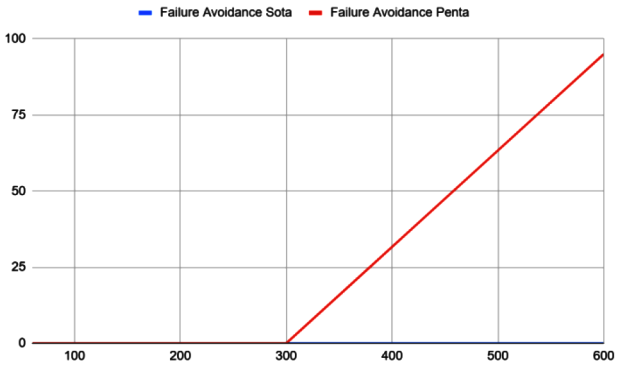

Fig. 10. Comparison of Failure Avoidance of systems using testing honeycomb and "Pentagon" testing

retry mechanism functioned and successfully processed part of the potential errors that arose due to temporary unavailability of the dependency.

3. Justification of the "Pentagon" model: Such improvement in reliability indicators confirms the value of developing a system taking into account fault tolerance and its testing according to the extended model. The Fault-Tolerance Testing and Chaos Testing layers in the "Pentagon" model are aimed precisely at detecting and verifying such mechanisms, which, as the experiment shows, are capable of improving system behavior even under system failure conditions.

Conclusions and prospects for further development. In the course of this study, an urgent scientific and practical task of increasing the reliability of software with microservice architecture was solved by developing and validating a new testing model.

1. Analysis of limitations of existing software testing and verification models ("pyramid", "honeycomb") was conducted, and their insufficient correspondence to the challenges of modern distributed systems with microservice architecture was substantiated. Instead, a new "Pentagon" model was proposed, consisting of six layers (Unit, Local Integration, Global Integration, End-to-End, Fault-Tolerance, Chaos Testing) and prioritizing testing of inter-service interaction and fault tolerance.

2. Based on the developed model, an approach to functional software testing for increasing software reliability was proposed for the first time, which consists of the systemic integration of the "Pentagon" model into the software development lifecycle.

3. A *method for evaluating the importance of testing layers* based on five weighted criteria (PFCF, DVFT, RDR, EDD, CE) was developed and validated. The conducted expert survey showed that the highest importance, in the opinion of specialists, is held by the layers *Fault-Tolerance Testing (index 6.8), End-to-End Testing (6.7)*, and *Global Integration Testing (6.6), while Unit Testing (5.7)* received the lowest score. This empirically confirms the shift of priorities in microservice testing.

4. The effectiveness of the proposed model and approach with its use on simulated data was *experimentally proven*. Comparative load testing showed that the system developed and tested according to the "Pentagon" model (with the *retry* mechanism) demonstrates higher reliability under conditions of simulated failures. In particular, *the number of failures decreased by 9.9 %, MTBF increased by 9.5 %, and system availability increased by ~2 percentage points* compared to the standard model.

Scientific novelty of the work lies in the fact that for the first time, a new conceptual testing model for software with microservice architecture has been developed, which, unlike classical models, divides the testing process into six levels with the aim of prioritizing testing of inter-service interaction and fault tolerance. For the first time, an approach to functional software testing for increasing its reliability has been defined, which consists of the systemic integration of the "Pentagon" model into the software development lifecycle and the application of which allows increasing software reliability indicators and formalizing the testing of fault tolerance mechanisms and resilience to chaotic infrastructure failures as separate, mandatory verification stages. Also, a method for evaluating the importance of testing layers based on five weighted criteria has been developed, which allows objectively assessing the importance and volume of each layer of the model in the software testing strategy.

Practical significance lies in the fact that the proposed model and approach with its use can serve as a practical guide for engineering teams in building effective and balanced testing strategies, which allows reducing risks associated with the operation of complex distributed software systems.

Prospects for Further Research are seen in the development of methods for *automating the implementation of the "Pentagon" model* into continuous integration and delivery (CI/CD) pipelines, in particular for projects with a large number of test environments, which will allow making the process of ensuring reliability even more effective and scalable.

References.

1. ISO/IEC (2023). *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model (ISO/IEC 25010:2023)*. Retrieved from https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-2:v1:en

2. Anand, V., Garg, D., Kaufmann, A., & Mace, J. (2023). Blueprint: A Toolchain for Highly-Reconfigurable Microservice Applications. *SOSP '23: 29th Symposium on Operating Systems Principles*. ACM. https://doi.org/10.1145/3600006.3613138

3. Baboi, M., Iftene, A., & Gîfu, D. (2019). Dynamic Microservices to Create Scalable and Fault Tolerance Architecture. *Procedia Computer Science*, *159*, 1035-1044. https://doi.org/10.1016/j.procs.2019.09.271

4. Waseem, M., Liang, P., Shahin, M., Di Salle, A., & Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, *182*, 111061. https://doi.org/10.1016/j.jss.2021.111061

5. Waseem, P., & Kaur, P. (2018, December). A survey of fault tolerance in cloud computing. Journal of King Saud University – Com-

puter and Information Sciences, *33*(10), 1159-1176. https://doi.org/10.1016/j.jksuci.2018.09.021
6. Molchanov, H., & Zhmaiev, A. (2018). Circuit breaker in systems based on microservices architecture. *Advanced Information Systems, 2*(4), 74-77. https://doi.org/10.20998/2522-9052.2018.4.13
7. Sambir, A., Yakovyna, V., & Seniv, M. (2017). Recruiting software architecture using user generated data. *2017 XIIIth International Conference on Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. IEEE. https://doi.org/10.1109/memstech.2017.7937557
8. Falahah, Surendro, K., & Sunindyo, W. D. (2021). Circuit Breaker in Microservices: State of the Art and Future Prospects. *IOP Conference Series: Materials Science and Engineering, 1077*(1), 012065. https://doi.org/10.1088/1757-899x/1077/1/012065
9. Mohammadian, V., Navimipour, N. J., Hosseinzadeh, M., & Darwesh, A. (2022). Fault-Tolerant Load Balancing in Cloud Computing: A Systematic Literature Review. *IEEE Access*, *10*, 12714-12731. https://doi.org/10.1109/access.2021.3139730
10. Bouizem, Y., Bouanani, N., & Reddad, A. (2023). Integrating request replication into FaaS platforms: an experimental evaluation. *Journal of Cloud Computing, 12*, 45. https://doi.org/10.1186/s13677-023-00457-z
11. Kuzmych, O. M., Lakhai, V. Y., & Seniv, M. M. (2023). Improved approach to automated software integration testing under the conditions of serverless architecture. *Scientific Bulletin of UNFU, 33*(3), 97-101. https://doi.org/10.36930/40330314
12. Simonsson, J., Zhang, L., Morin, B., Baudry, B., & Monperrus, M. (2021). Observability and chaos engineering on system calls for containerized applications in Docker. *Future Generation Computer Systems*, *122*, 117-129. https://doi.org/10.1016/j.future.2021.04.001
13. Lakhai, V., Kuzmych, O., & Seniv, M. (2023). An improved method for increasing maintainability in terms of serverless architecture application. *2023 IEEE 18th International Conference on Computer Science and Information Technologies (CSIT). IEEE.* https://doi.org/10.1109/csit61576.2023.10324273
14. 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
15. Reghenzani, F., Guo, Z., & Fornaciari, W. (2023). Software Fault Tolerance in Real-Time Systems: Identifying the Future Research Questions. *ACM Computing Surveys*. https://doi.org/10.1145/3589950
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)*, (pp. 1-4). IEEE. https://doi.org/10.1109/csit65290.2024.10982594
17. Patil, P. S. N. (2022). Estimation of Mean Time between Failures of a Two Tandem Mill of a Sugar Plant. *International Journal for Research in Applied Science and Engineering Technology, 10*(1), 474-478. https://doi.org/10.22214/ijraset.2022.39823
18. Wang, J., & Yin, H. (2019). Failure Rate Prediction Model of Substation Equipment Based on Weibull Distribution and Time Series Analysis. *IEEE Access*, *7*, 85298-85309. https://doi.org/10.1109/access.2019.2926159
19. Bartok, T., Hejja, F., & Kocsis, G. (2024). Availability Modeling in Software Architecture. *2024 IEEE 3rd Conference on Information Technology and Data Science (CITDS)*, (pp. 1-6). IEEE. https://doi.org/10.1109/citds62610.2024.10791390
20. Vogel, M., Knapik, P., Cohrs, M., Szyperrek, B., Pueschel, W., Etzel, H., Fiebig, D., Rausch, A., & Kuhrmann, M. (2020). Metrics in automotive software development: A systematic literature review. *Journal of Software: Evolution and Process*, e2296. https://doi.org/10.1002/smr.2296

Модель підвищення надійності мікросервісного програмного забезпечення на етапі функціонального тестування

М. М. Сенів, О. М. Кузьмич[*]

Національний університет «Львівська політехніка», м. Львів, Україна
[*] Автор-кореспондент e-mail: oleg.kuzmuch@gmail.com

Мета. Розробка й обґрунтування нової моделі тестування для підвищення надійності програмного забезпечення (ПЗ) з мікросервісною архітектурою, що системно враховує специфіку розподілених систем, особливості мікросервісної архітектури й аспекти відмовостійкості, які недостатньо покриваються класичними підходами.

Методика. Запропонована шестирівнева модель тестування «Пентагон». Проведено аналіз обмежень існуючих моделей («піраміда» й «сота тестування») у контексті хмарних архітектур. Для перевірки моделі застосовано комбінований підхід: опитування ІТ-фахівців для визначення важливості шарів і порівняльне навантажувальне тестування двох версій системи – стандартної та з інтегрованою моделлю тестування «Пентагон». Оцінювались ключові метрики надійності, зокрема Failure Rate, Availability та MTBF, тощо.

Результати. Результатом дослідження є формалізована модель тестування «Пентагон», що складається із шести послідовних шарів. Розроблено критерії для кількісної оцінки внеску кожного шару в загальну надійність системи. Експертне опитування підтвердило вищу важливість шарів Fault-Tolerance, End-to-End і Global Integration Testing порівняно із Unit Testing. Експериментально доведено, що система, протестована із використанням запропонованої моделі, демонструє вищу надійність в умовах збоїв: кількість відмов зменшилась на 9,9 %, MTBF зріс на 9,5 %, а Availability збільшилася на 2 %.

Наукова новизна. Уперше розроблена концептуальна модель «Пентагон», яка, на відміну від існуючих, структурує процес тестування мікросервісного ПЗ на шість шарів. Особливістю моделі є виділення перевірки механізмів відмовостійкості та стійкості до хаотичних збоїв інфраструктури в окремі обов'язкові етапи, що дозволяє формалізувати ці процеси. На основі розробленої моделі вперше запропоновано підхід до функціонального тестування ПЗ для підвищення його надійності, що полягає у системній інтеграції вищезазначених рівнів тестування й хаос-інжинірингу у життєвий цикл розробки (Software Development Lifecycle SDLC). На відміну від класичних підходів, така інтеграція забезпечує комплексне підвищення показників надійності ПЗ. Розроблено метод оцінки важливості шарів тестування на основі п'яти зважених критеріїв, який дозволяє об'єктивно оцінити важливість й обсяг кожного шару моделі у стратегії тестування ПЗ.

Практична значимість. Модель забезпечує структуровану основу для формування стратегії тестування мікросервісних додатків, сприяє раннього виявленню архітектурних недоліків, зменшує ризики збоїв у продакшені й знижує витрати на їх усунення.

Ключові слова: *програмне забезпечення, мікросервісна архітектура, функціональне тестування, модель тестування «Пентагон», «хаос-тестування»*