

Dmytro Diachenko, Maksym Shostak

Kharkiv National University of Radio Electronics, Kharkiv, Ukraine

METHOD FOR ASSESSING THE RISK OF DEFECT MANIFESTATION IN .NET MICROSERVICES IN DISTRIBUTED COMPUTER SYSTEMS BASED ON OPERATIONAL METRICS

Abstract. Background. Software defects undetected during functional testing may manifest under load, making risk assessment based on operational metrics relevant. **Subject of the study:** methods and tools for assessing the risk of defects in .NET microservices within distributed computer systems, taking operational metrics into account. **Objective:** to develop a method for assessing the risk of defect manifestation in .NET microservices within distributed computer systems based on a comprehensive analysis of operational metrics. **Research tasks.** To achieve the purpose, a set of operational metrics was defined, and a model for forming an integral risk indicator based on their normalized values was developed. **Methods.** Methods of system analysis, normalization and integral assessment, load testing, and statistical processing were used. **Results.** A method for the integral assessment of the risk of defect manifestation in .NET microservices based on normalized operational metrics was proposed. **Conclusions.** The proposed method makes it possible to assess the risk of defect manifestation in software components of distributed computer systems without modifying their program code, based on operational data obtained during load testing.

Keywords: defect manifestation risk; .NET microservices; distributed computer systems; operational metrics; integral risk indicator; load testing; throughput; Kubernetes; Docker; software reliability.

1. Introduction

1.1 Relevance of the Topic

At the present stage of software system development, there is an active transition to cloud deployment models, within which microservice architecture has become one of the leading approaches to building scalable, fault-tolerant, and flexible applications. This approach involves decomposing a software system into a set of relatively independent services, each of which performs a separate functional role, is deployed autonomously, and can be updated or scaled independently of other components. Combined with containerization, the microservice model facilitates application portability across different execution environments, provides a high degree of execution environment isolation, and simplifies the automation of deployment processes. In this regard, container orchestration systems, particularly Kubernetes, have become important tools for managing service lifecycles, load balancing, failure recovery, and horizontal autoscaling.

Under these conditions, ensuring the effective horizontal scaling of microservice applications becomes particularly important. Unlike vertical scaling, which involves increasing the resources of an individual node, horizontal scaling is based on increasing the number of service instances to distribute the load among them. This approach is more natural for cloud and distributed environments; however, its effectiveness is determined not only by infrastructure parameters but also by the implementation characteristics of the service itself. The practical use of container platforms demonstrates that services with identical functional purposes may differ significantly in performance, response time stability, throughput, and resource consumption patterns, which directly affects the operation of autoscaling mechanisms.

This issue is particularly relevant for .NET microservices because the .NET platform provides a wide range of tools for building high-performance web applications, including an asynchronous request-

processing model, memory optimization mechanisms, minimal API approaches, caching, and various methods of organizing parallel execution. However, the availability of these tools alone does not guarantee high efficiency in real-world container deployments. Service performance in Kubernetes is influenced by a combination of factors, including the load-processing model, thread-blocking patterns, container startup time, request-processing latency, CPU and memory utilization profiles, and service behavior when the horizontal autoscaling mechanism is triggered. Consequently, there is a need not only to create a microservice application on the .NET platform but also to purposefully investigate its behavior in an orchestrated environment to identify operational indicators that may suggest an increased risk of software implementation defects manifesting.

The traditional approach to deploying microservices in a container environment is often limited to verifying the functional operability of a service and performing basic configuration of the horizontal autoscaling mechanism. However, under real-world conditions, this is insufficient for a substantiated assessment of the reliability of a software component and the risk of defects in its implementation manifesting. With formally identical scaling parameters, two functionally similar services may demonstrate different sensitivities to load, use computing resources differently, and differ significantly in average response time, tail latencies, throughput, and result stability. Such differences may be associated with thread blocking, inefficient organization of parallel execution, excessive memory utilization, lack of caching, or other software implementation characteristics that are not always detected during traditional functional testing.

Under intensive load, latent deficiencies in a software component may manifest as violations of acceptable response time, an increased percentage of failed requests, unstable resource utilization, reduced throughput, and an unpredictable response of the autoscaling mechanism. At the same time, the individual use of each of these metrics

does not provide a sufficiently comprehensive characterization of the state of a software component. For example, an increase in the number of replicas may be accompanied by persistently high latencies, while an acceptable average response time may conceal a significant increase in its 95th percentile. Therefore, the risk of defect manifestation should be assessed based on the comprehensive consideration of operational indicators characterizing the performance, stability, resource efficiency, and scalability of the component.

A promising approach to addressing this task is a comparative study of basic and optimized .NET microservice implementations in a container orchestration environment using load testing. This approach makes it possible to determine how software implementation characteristics affect average response time, the 95th percentile of response time, result variability, throughput, the percentage of failed requests, CPU and memory consumption, and changes in the number of service instances resulting from the activation of the autoscaling mechanism. The combination of these characteristics can be used to form an integral indicator of the risk of defect manifestation in a software component under varying load conditions.

The relevance of the study is further reinforced by the fact that, in cloud service development practice, performance and scaling indicators are primarily used for infrastructure configuration, whereas the possibility of applying them to assess the risk of defect manifestation in software components remains insufficiently investigated. Autoscaling is often regarded as a universal means of resolving performance problems, although the actual behavior of a distributed system is determined by the interaction among the orchestrator, load characteristics, and the internal implementation of the service. Under these conditions, a comprehensive analysis of operational metrics makes it possible not only to assess scaling efficiency but also to identify software components whose behavior is characterized by increased instability and a risk of defect manifestation under load.

Therefore, it is advisable to develop a method for assessing the risk of defect manifestation in .NET microservices within distributed computer systems that provide for the formation of an integral assessment based on timing characteristics, throughput, the percentage of failed requests, computing resource utilization, and component behavior during horizontal scaling. An experimental comparison of the basic and optimized implementations of the same functional service under identical conditions makes it possible to establish the relationship between software implementation characteristics and the level of risk associated with its unstable operation.

1.2 Literature Review and Problem Definition

Assessing the risk of defect manifestation in software components of distributed computer systems requires consideration not only of functional testing results but also of operational indicators characterizing component behavior under varying load conditions. For microservice applications, such indicators include request-processing timing characteristics, throughput, the percentage of failed requests, computing resource consumption, and the

response of autoscaling mechanisms. In this regard, it is advisable to analyze current studies devoted to microservice scaling, the selection of operational metrics, the prediction of resource requirements, and the verification of component operational stability in container environments.

In article [1], the authors consider the problem of microservice autoscaling under the need to ensure optimal resource allocation while meeting quality-of-service requirements and service-level agreements. The paper summarizes current approaches to microservice autoscaling, with an emphasis on verifying the correctness of the proposed algorithms. The authors systematize the publication results according to three interrelated areas: microservices, verification, and autoscaling. Studies on scaling verification in cloud systems are also analyzed. The article demonstrates that formal methods have long been used to verify autoscaling policies in cloud environments; however, such approaches are only beginning to become widespread in microservice systems. The paper also identifies the main factors affecting microservice scaling and the corresponding assessment metrics and establishes that probabilistic model checking is the most widely used formal method in this subject area. Open problems are separately outlined, including the verification of existing microservice scaling mechanisms and the validation of approaches based on machine learning.

Article [2] considers the problem of predictive scaling of microservice applications on Kubernetes platforms using machine learning methods. The paper proposes an approach to estimating the future resource requirements of microservices based on data obtained during the operation of the standard horizontal autoscaling mechanism, followed by using machine learning models to predict the resources required as the number of users increases. The article describes an experimental setup that includes Kubernetes, Google Online Boutique as a microservice application, Locust for load simulation, and Prometheus for metric collection. It also presents a procedure for selecting horizontal scaling parameters, forming datasets, and training several models, including linear regression, SVM, and MLP. The obtained results demonstrated that machine learning models can predict future computing resource requirements with high accuracy for both individual pods and the cluster, with the MLP model showing the best results in generalized prediction. At the same time, the paper emphasizes limitations associated with extrapolating predictions beyond the available data, the dependence of results on the size of the collected observation dataset, and the need for further model improvement before their use in production environments.

Article [3] considers the research and applied problem of microservice autoscaling while accounting for quality-of-service requirements and efficient resource utilization in a Kubernetes environment. The paper proposes the MS-RA approach, which is based on service-level objectives and implemented using the MAPE-K loop. The proposed solution makes it possible to make real-time scaling decisions while considering not only the need to improve performance but also the need to prevent excessive use of computing resources. The article describes the architecture of the approach and its input parameters, including service

objectives, server types, and scaling parameters, and presents the results of an experimental comparison with horizontal scaling in Kubernetes. The obtained results demonstrated that the proposed approach ensures compliance with the established SLOs while using significantly less CPU, memory, and fewer replicas compared with standard horizontal scaling. The paper also emphasizes the importance of aligning scaling with service requirements, while future research directions include extending the approach to other types of microservice systems and combining reactive mechanisms with machine learning methods.

Article [4] is devoted to improving the efficiency of automatic resource scaling in cloud environments using machine learning methods. The paper proposes an approach based on the statistical selection of the metrics most relevant to a particular application to improve the accuracy of scaling decisions. The authors proceed from the assumption that different applications require different indicators for the correct estimation of required resources; therefore, universal metric sets do not provide sufficient efficiency. The article demonstrates that using a metric-selection mechanism makes it possible to improve the system response speed to load changes, reduce resource utilization, and stabilize quality-of-service indicators. Attention is paid to the limitations of existing approaches, including their dependence on fixed metrics, the difficulty of adapting to dynamic load scenarios, and the need for more flexible resource orchestration tools.

Article [5] considers the problem of automatic resource scaling in cloud computing environments to improve the efficiency of computing infrastructure utilization, ensure quality of service, and reduce costs. The paper systematizes current autoscaling methods, including reactive and proactive approaches such as threshold-based rules, queuing theory, reinforcement learning, fuzzy logic, machine learning, and time-series analysis. The authors analyze the practical application areas of these approaches in cloud services, telecommunications infrastructure, big data processing systems, healthcare services, and smart city platforms. The article demonstrates that autoscaling plays a key role in ensuring the elasticity of cloud systems; however, its effectiveness is limited by the difficulty of accurate load prediction, environmental dynamics, and issues related to energy efficiency, cost, security, and compatibility with existing infrastructure.

In article [6], the authors analyze the impact of microservice autoscaling policies on host energy consumption and the risk of violating energy constraints in cloud environments. The paper proposes an approach based on probabilistic model checking that makes it possible to formally assess the correctness of scaling decisions as early as the design stage. The authors constructed four variants of an MDP model that account for different scaling constraints based on Kubernetes horizontal scaling and used them to analyze autoscaling policies from an energy-efficiency perspective. The obtained results demonstrated that policies simultaneously accounting for latency and energy consumption are the most appropriate, as this approach provides a better balance between quality of service and energy costs. The paper also establishes that the number of pods, the scale of replica-increase operations, and the

selection of scaling thresholds have a significant impact on energy consumption.

Article [7] considers the scalability of microservice architecture. The paper summarizes the results of 55 studies devoted to microservice scaling approaches, the metrics used, and autoscaling tools and mechanisms. The authors demonstrate that existing studies focus primarily on autoscaling, particularly on machine-learning-based approaches, while the main challenges include collecting valid metrics, making dynamic scaling decisions, balancing performance, cost, and security, and addressing the risk of cascading degradation in quality of service. It is separately emphasized that, despite significant progress in this field, substantial research gaps remain due to the lack of standardized autoscaling metrics and the need to develop more adaptive and resilient solutions for real-world dynamic environments.

Article [8] considers the problem of improving the scalability and performance of .NET microservices deployed in an Azure Kubernetes Service environment. The paper analyzes the impact of various configurations and optimization approaches, including the configuration of node pools, the Kubernetes scheduler, horizontal autoscaling mechanisms, VPA, and the use of a service mesh, on the key operational indicators of a microservice application. The authors conducted an experimental evaluation using metrics such as CPU utilization, memory utilization, response time, throughput, and pod-scaling efficiency. The obtained results demonstrated that optimized configurations make it possible to reduce resource consumption, shorten response time, and increase throughput compared with the standard Azure Kubernetes Service deployment configuration. The authors note that the greatest effect is achieved by combining optimized resource allocation, specialized scheduling, and autoscaling mechanisms, although implementing such solutions increases configuration complexity and the requirements for system monitoring and administration.

Article [9] considers software defect prediction based on optimized machine learning models using NASA MDP datasets. The authors compared the k-nearest neighbors method, decision trees, logistic regression, linear discriminant analysis, a multilayer perceptron, and the support vector machine method by applying PCA, SMOTE, and random hyperparameter search; the k-nearest neighbors method demonstrated the best results for most datasets. At the same time, the research focuses on defect prediction using software project data and does not provide an integral assessment of the risk of their manifestation based on operational metrics of components underload.

Article [10] compares traditional and ensemble machine learning algorithms for defect prediction using four datasets. The Gradient Boosting model with boosting demonstrated the highest accuracy of 94.5%; however, the assessment of the risk of defect manifestation based on operational metrics of components under load was not considered in the research. Article [11] presents a systematic review of 63 studies and a meta-analysis of the application of deep learning to software defect prediction. Deep learning was found to outperform traditional models in terms of accuracy, F-measure, and AUC; however, operational metrics of components under load were not considered.

Article [12] compares machine learning methods for test case prioritization in the continuous integration process. The effectiveness of the models was found to depend on the size of the test history and the available execution time; however, the integral assessment of the risk of defect manifestation based on operational metrics was not considered. Article [13] compares online and offline learning models for the just-in-time prediction of defect-inducing changes in program code. The proposed BORB approach enables the use of offline models in an online mode but does not consider operational metrics of components under load. Article [14] proposes an ensemble method for predicting defects in software modules based on PROMISE and GitHub data. At the same time, the integral assessment of the risk of defect manifestation based on operational indicators was not considered.

Study [15] proposes a Top-k classification method with label ranking, the effectiveness of which was confirmed using text and image datasets. However, the method is not directly intended for defect prediction or the analysis of operational metrics of software components. Article [16] proposes the APFD-Net model for test case prioritization based on variational learning to rank and a differentiable loss function. The approach is aimed at the early detection of defects during testing but does not assess the risk of their manifestation based on operational metrics. Article [17] presents a systematic analysis of the application of machine learning to defect prediction from the perspective of its practical use in commercial development. At the same time, a comprehensive assessment of the operational risk of software components was not considered. Study [18] presents SQuaD, a multidimensional temporal dataset of quality indicators from 450 software projects, covering more than 700 metrics, version histories, defects, and vulnerabilities. The dataset is intended for software quality research but does not contain operational metrics obtained through microservice load testing.

Article [19] presents the Defectors dataset for defect prediction at the change and code-line levels, containing approximately 213,000 files from 24 Python projects. It supports the training of defect prediction models but does not consider the behavior of components under load. Article [20] systematizes datasets, data validation methods, artificial intelligence models, and defect prediction tools. At the same time, the assessment of the risk of defect manifestation based on operational metrics of components under load was not considered. Article [21] compares online and offline learning models for the just-in-time prediction of defect-inducing changes in program code. The proposed BORB approach enables the use of offline models in an online mode but does not consider operational metrics of components under load. Article [22] proposes a general process for predicting defects in large-scale complex information systems using machine learning and static software characteristics. However, the behavior of components under load and the integral operational risk are not assessed. Article [23] presents a systematic review of the application of deep learning models to software defect prediction. The predominant use of supervised learning, software metrics, and convolutional neural networks was identified; however, the operational indicators of microservices under load were not considered.

The analysis of current scientific publications has demonstrated that ensuring the stable operation of microservices in container environments is one of the key problems in cloud computing and distributed software systems. Existing studies investigate formal methods for verifying autoscaling policies, predicting resource requirements, meeting service-level objectives, selecting informative metrics, ensuring energy efficiency, and improving the performance of microservice applications. The results obtained by the authors confirm that response time, throughput, CPU and memory utilization, the number of replicas, and the stability of indicators under load are important characteristics of the state of microservice components.

At the same time, most of the analyzed studies focus primarily on the infrastructure level of the problem, namely scaling policies, load prediction, decision-making algorithms, and resource orchestration mechanisms. Operational metrics in such studies are used primarily for scaling control and performance assessment but are not considered as a set of indicators that can be used to determine the risk of defect manifestation in a software component. The relationship between the internal implementation characteristics of a microservice and the emergence of such adverse indicators under load as increased average response time and its 95th percentile, significant result variability, reduced throughput, an increased percentage of failed requests, unstable resource utilization, and a disproportionate response of the autoscaling mechanism remain insufficiently investigated.

Studies analyzing the impact of .NET platform tools, asynchronous request processing, thread blocking, caching, and the organization of parallel execution on the risk of unstable operation of a software component in a Kubernetes environment are particularly limited. Existing approaches also do not provide a unified integral assessment that comprehensively accounts for timing characteristics, throughput, the percentage of failed requests, resource consumption, and component behavior during horizontal scaling. In this regard, it is advisable to develop a method for assessing the risk of defect manifestation in .NET microservices based on a comprehensive analysis of operational metrics obtained during load testing in a container orchestration system. Comparing the basic and optimized implementations of a functionally similar microservice under identical conditions makes it possible to establish the relationship among software implementation characteristics, the stability of operational indicators, and the level of risk of defect manifestation in a distributed computer system.

1.3 Purpose and Objectives of the Study

The purpose of the study is to develop a method for assessing the risk of defect manifestation in .NET microservices within distributed computer systems based on a comprehensive analysis of operational metrics. This will make it possible to identify software components with potentially unstable behavior under varying load conditions by considering request-processing timing characteristics, throughput, the percentage of failed requests, computing resource utilization, and the response of the horizontal scaling mechanism.

To achieve this purpose, the following objectives were set:

- to identify a set of operational metrics that characterize the operational stability of .NET microservices and can be used as indicators of the risk of software implementation defects manifesting;
- to develop a mathematical model for forming an integral assessment of the risk of defect manifestation in a software component;
- to define the criterion and risk assessment indicators, considering timing characteristics, throughput, request-processing errors, CPU and memory utilization, and component behavior during horizontal scaling;
- to develop a method for assessing the risk of defect manifestation in .NET microservices based on the normalization and comprehensive processing of operational metrics;
- to conduct an experimental study of the basic and optimized .NET microservice implementations under identical container orchestration conditions and compare the risk levels formed for them.

2. Materials and Research Methods

The object of the study is the process of assessing the risk of defect manifestation in software components of distributed computer systems during their operation under varying load conditions. The paper considers .NET microservices containerized using Docker and deployed in a Kubernetes environment. The software components process streams of HTTP requests, use the computing resources of cluster nodes, and interact with the horizontal autoscaling mechanism.

Within this study, defect manifestation is understood as the occurrence of an undesirable operational state of a software component that may be associated with the characteristics of its internal implementation and is characterized by exceeding the acceptable response time, significant tail latencies, failed requests, reduced throughput, unstable resource consumption, or inefficient behavior during scaling. The proposed approach is not intended to directly detect defects in the source code but provides an assessment of the risk of their manifestation based on observations of component behavior.

The main research hypothesis is that the comprehensive consideration of timing, resource, and scaling characteristics provides a more substantiated assessment of the risk of defect manifestation in a software component compared with the use of individual performance indicators. A software implementation characterized by a shorter average response time, a lower 95th percentile value, lower result variability, higher throughput, and more stable resource utilization is expected to have a lower integral risk level. The following conditions were used in developing the risk assessment method.

Condition 1. The basic and optimized microservice implementations have the same functional purpose, support identical HTTP interfaces, and process requests with the same structure.

Condition 2. Both implementations are deployed in the same Kubernetes cluster with identical resource request and limit values, ensuring the comparability of the experimental results.

Condition 3. The horizontal autoscaling mechanism is configured identically for both services: the minimum number of replicas is one, the maximum is five, and the target CPU utilization is 50%.

Condition 4. The load is generated at three intensity levels corresponding to 10, 20, and 50 virtual users. Each test lasts 60 seconds, and each experimental scenario is repeated three times.

Condition 5. The operational metrics of the basic and optimized implementations are recorded according to the same rules and at the same points during the experiment.

Condition 6. The risk assessment is comparative and reflects the relative level of risk of defect manifestation within the investigated load scenarios. The obtained value is not considered the probability of a defect being present in the source code.

During the development of the defect manifestation risk assessment method, a decomposition method was applied, according to which the assessment process is divided into three interrelated components: performance assessment, resource utilization stability assessment, and component behavior assessment during horizontal scaling. The relationship among the components is maintained through a common set of operational metrics and an integral risk indicator.

3. Results of the Development and Study of the Method for Assessing the Risk of Defect Manifestation in .NET Microservices

3.1 Mathematical Model for Assessing the Risk of Defect Manifestation in .NET Microservices

To formally represent the results of each experimental run, the following vector of operational characteristics is used:

$$X_{s,l,j} = [t_{avg}, t_{95}, q, e, c, m, n]^T, \quad (1)$$

where s is the software component variant, with $s=B$ corresponding to the basic implementation and $s=O$ to the optimized implementation; l is the load level specified by the number of virtual users; j is the repeated run number; t_{avg} is the average response time, ms; t_{95} is the 95th percentile of response time, ms; q is the throughput, requests per second; e is the percentage of failed requests; c is the CPU utilization; m is the memory utilization; n is the number of service replicas after completion of the load test; T is the vector transposition symbol.

For each software component variant and each load level, the mean value of the metric under study is determined as follows:

$$\bar{x}_{s,l} = \frac{1}{J} \sum_{j=1}^J x_{s,l,j}, \quad (2)$$

where $\bar{x}_{s,l}$ is the value of the metric under study during the j -th run; J is the total number of repeated runs. In the conducted experiments, $J=3$ for each scenario.

To assess the dispersion of the obtained results, the standard deviation is calculated as follows:

$$\sigma_{s,l} = \sqrt{\frac{1}{J-1} \sum_{j=1}^J (x_{s,l,j} - \bar{x}_{s,l})^2}, \quad (3)$$

where $\sigma_{s,l}$ is the standard deviation of the metric values for the corresponding service variant and load level.

To compare the variability of indicators with different units of measurement and value scales, the coefficient of variation is used:

$$v_{s,l} = \sigma_{s,l} / \bar{x}_{s,l}. \quad (4)$$

An increase in the coefficient of variation indicates reduced predictability of software component behavior. Therefore, increased variability in response time, its 95th percentile, and throughput is considered one of the indicators of the risk of defect manifestation under load.

3.2 Criterion and Indicators for Assessing the Risk of Defect Manifestation in .NET Microservices

Since operational metrics have different units of measurement, they are normalized before being combined. For indicators whose increase signifies an increase in risk, the following formula is used:

$$r_i = \frac{x_i - x_i^{\min}}{x_i^{\max} - x_i^{\min}}, \quad (5)$$

where x_i is the current value of the indicator; $x_i^{\min}$ and $x_i^{\max}$ are its minimum and maximum values in the experimental dataset, respectively; r_i is the normalized risk component.

According to formula (5), the average response time, the 95th percentile of response time, the percentage of failed requests, CPU utilization, memory utilization, and variability indicators are normalized. Throughput has the opposite direction of influence: its decrease indicates deterioration in the component state. Therefore, the normalized risk component associated with insufficient throughput is determined using the following formula:

$$r_q = (q^{\max} - q) / (q^{\max} - q^{\min}). \quad (6)$$

As a result of normalization, all partial assessments fall within the interval from 0 to 1. A value of 0 corresponds to the lowest recorded adverse impact, while a value of 1 corresponds to the highest. If the maximum and minimum values of a particular metric are equal, the corresponding component is set to zero because this metric does not make it possible to distinguish between the investigated variants. The normalized assessment of the risk associated with timing characteristics is formed based on the average response time and its 95th percentile:

$$r_t = \alpha r_{avg} + (1 - \alpha) r_{95}, \quad (7)$$

where r_{avg} is the normalized assessment of the average response time; r_{95} is the normalized assessment of the 95th percentile; α is the coefficient determining the ratio of their influence, $\alpha \in [0; 1]$.

Within the experimental study, it is advisable to set $\alpha=0.4$, resulting in a weight of 0.4 for the average response time and 0.6 for the 95th percentile. The greater weight of the 95th percentile is explained by the fact that it better characterizes adverse request-processing conditions. The risk component associated with computing resource utilization is determined based on the normalized CPU and memory utilization indicators:

$$r_r = \beta r_c + (1 - \beta) r_m, \quad (8)$$

where r_c – is the normalized assessment of CPU utilization; r_m – is the normalized assessment of memory utilization; β – is the coefficient determining the ratio of their influence. If CPU and memory are considered equally significant, $\beta=0.5$ is used.

The risk component characterizing the instability of software component behavior is determined based on the normalized coefficients of variation:

$$r_v = \frac{r_{v,avg} + r_{v,95} + r_{v,q}}{3}, \quad (9)$$

where $r_{v,avg}$ is the normalized variability of the average response time; $r_{v,95}$ is the normalized variability of the 95th percentile; $r_{v,q}$ is the normalized variability of throughput.

To account for component behavior during horizontal scaling, a scaling inefficiency indicator is used:

$$r_s = \frac{n}{n_{max}} \cdot \frac{r_t + r_q}{2}, \quad (10)$$

where n is the actual number of replicas after completion of the test; n_{max} is the maximum number of replicas specified in the autoscaling settings; r_t is the risk component associated with timing characteristics; r_q is the insufficient-throughput component. The r_s indicator increases when an increase in the number of replicas is not accompanied by a sufficient reduction in response time and an increase in throughput. Therefore, the number of replicas itself is not considered an indicator of a defect but is considered only in conjunction with scaling effectiveness. To reduce assessment subjectivity, it is advisable to determine the weighting coefficients of the partial criteria using the entropy method. First, the relative normalized value is calculated for each experimental scenario and each criterion:

$$p_{ak} = r_{ak} / \sum_{a=1}^M r_{ak}, \quad (11)$$

where a is the experimental scenario number; k is the partial criterion number; M is the number of compared scenarios; r_{ak} is the normalized value of the k -th criterion for the a -th scenario. For two service implementations and three load levels, $M=6$. The entropy of an individual criterion is determined using the following formula:

$$E_k = -\frac{1}{\ln M} \sum_{a=1}^M p_{ak} \ln p_{ak}, \quad (12)$$

where E_k characterizes the homogeneity of the values of the corresponding criterion. The more strongly the indicator differs among the investigated scenarios, the greater its informational significance. The weighting coefficient of the criterion is determined using the formula:

$$w_k = (1 - E_k) / \sum_{h=1}^K (1 - E_h), \quad (13)$$

where K is the total number of partial criteria. The proposed method uses six criteria: timing characteristics, failed requests, insufficient throughput, resource utilization, result variability, and scaling inefficiency; therefore, $K=6$.

If the value of a particular criterion is zero in all scenarios, as may occur in the absence of failed requests, its weighting coefficient is set to zero. The weights of the remaining criteria are then renormalized so that their sum equals one. The integral indicator of the risk of defect manifestation for an individual experimental scenario is determined as the weighted sum of the partial components:

$$R_a = \sum_{k=1}^K w_k r_{ak}, \quad (14)$$

In expanded form, the formula is as follows:

$$R = w_t r_t + w_e r_e + w_q r_q + w_r r_r + w_v r_v + w_s r_s. \quad (15)$$

The weighting coefficients must satisfy the conditions:

$$w_t + w_e + w_q + w_r + w_v + w_s = 1, \quad (16)$$

The value of the integral indicator belongs to the interval $R \in [0; 1]$. When $R \in [0; 0,33]$ the risk level is considered low; when $R \in [0,33; 0,67]$ it is considered medium, and when $R \in [0,67; 1]$, it is considered high. This assessment is comparative and reflects the risk of unstable operation of the software component under the investigated conditions rather than the probability of a defect being present in its source code.

3. 3 Method for Assessing the Risk of Defect Manifestation in .NET Microservices and Its Experimental Study

The proposed method for assessing the risk of defect manifestation in .NET microservices involves sequential acquisition, statistical processing, and comprehensive assessment of the operational metrics of a software component. The application of the method includes the following stages:

- determining the software component variant and the parameters of the load scenario;
- conducting a series of repeated experimental runs;
- collecting timing characteristics, throughput indicators, the percentage of failed requests, CPU and memory utilization, and the number of service replicas;
- calculating mean values, standard deviations, and coefficients of variation;
- normalizing the obtained operational indicators;
- forming partial risk components based on timing, resource, and scaling characteristics;
- determining the weighting coefficients of the partial criteria;
- calculating the integral risk indicator and determining the corresponding level of risk of defect manifestation.

The experimental study was conducted to verify the applicability of the proposed method for assessing the impact of a .NET microservice software implementation on the risk of defect manifestation during its operation in the Kubernetes container orchestration system. As part of the experiment, a testbed was created to reproduce a typical deployment environment for containerized microservice applications and compare the behavior of two functionally similar services under identical load conditions. The study is based on a comparison of the basic and optimized service implementations. This makes it possible to determine how changes in the internal request-

processing logic affect the average response time, the 95th percentile of response time, throughput, the percentage of failed requests, result variability, CPU and memory utilization, and the response of the horizontal autoscaling mechanism.

The experimental testbed was built on the .NET 10 platform. Two web services in the minimal API format, ServiceBasic and ServiceOptimized, were created for the study. Both services implement the same set of HTTP interfaces, including a root route, the /health health-check endpoint, and a group of test endpoints for simulating different types of loads. The /api/heavy endpoint was selected as the primary target of the experimental load because it simulates computationally intensive request processing and creates conditions under which the differences between the two software implementations are most evident.

The basic implementation performs computations with additional blocking delays and represents a typical request-processing arrangement without specialized optimization tools. The optimized implementation uses an asynchronous execution model, reduces the number of blocking operations during computations, applies simplified caching mechanisms, and improves the load-processing structure. Thus, both services are functionally similar but differ in their internal execution organization, making it possible to investigate the impact of the software implementation on the operational risk profile.

A separate Dockerfile was created for each service using a multi-stage build with the `mcr.microsoft.com/dotnet/sdk:10.0` image at the compilation stage and the `mcr.microsoft.com/dotnet/aspnet:10.0` image at the runtime stage. Both services were configured to listen on port 8080, ensuring their correct operation in the container environment. The build process produced the local Docker images `service-basic:1.0` and `service-optimized:1.0`, which were used for subsequent deployment in Kubernetes.

The services were deployed in a local Kubernetes cluster operating in the Docker Desktop environment. Separate Deployment and Service YAML configurations were created for each service. At the Deployment level, one initial replica, identical `imagePullPolicy` values, container port 8080, and identical `resources.requests` and `resources.limits` parameters for CPU and memory were specified. Defining the resource parameters is necessary because the horizontal autoscaling mechanism determines the relative CPU utilization based on `requests.cpu`. The `NodePort` type was used at the Service level, while the `kubectl port-forward` command was applied for local load testing, ensuring that test traffic was consistently directed to the corresponding service in the cluster.

A separate horizontal autoscaling object was configured for each deployment. The corresponding Deployment was specified as the scaling target. Identical scaling limits of one to five replicas and a target CPU utilization of 50% were set for the basic and optimized services. This configuration ensures identical infrastructure conditions and makes it possible to associate the identified differences in operational indicators primarily with the software implementation characteristics of the services.

To support the operation of the horizontal autoscaling mechanism, metrics-server was additionally installed in the cluster to collect and provide information on computing resource consumption. During its configuration, the `--kubelet-insecure-tls` parameter was added, which is required for the stable operation of the metrics service in the local Docker Desktop cluster.

After the infrastructure had been prepared, load testing was performed using k6. Separate `basic-test.js` and `optimized-test.js` scenarios were created to send requests to the `/api/heavy` endpoint through local port forwarding for the corresponding service. The load was simulated at three intensity levels: 10, 20, and 50 virtual users. Each test lasted 60 seconds. Each experimental scenario was repeated three times, which made it possible to reduce the influence of random fluctuations, assess the repeatability of the results, and calculate mean values, standard deviations, and coefficients of variation. The B10_1–B10_3, B20_1–B20_3, and B50_1–B50_3 run series were formed for the basic service, while the O10_1–O10_3, O20_1–O20_3, and O50_1–O50_3 series were formed for the optimized service.

During each run, k6 recorded the average response time, the 90th and 95th percentiles of response time, the maximum response time, throughput, and the percentage of failed requests. At the same time, Kubernetes tools were used to determine the number of service replicas after the test, CPU utilization, and memory utilization. The obtained values were used to calculate the partial components and the integral indicator of the risk of defect manifestation in accordance with the mathematical model presented in Subsections 3.1 and 3.2. Fig. 1 shows an example of the structure of a CSV file containing performance metrics, load parameters, and horizontal scaling characteristics of the services.

Service	Load Level	avg_ms	p90_ms	p95_ms	max_ms	rps	failed_percent
ServiceBasic	10	1050	1100	1150	1200	10	0
ServiceBasic	20	650	700	750	800	20	0
ServiceBasic	50	1450	1500	1550	1600	50	0
ServiceOptimized	10	160	170	180	190	10	0
ServiceOptimized	20	160	170	180	190	20	0
ServiceOptimized	50	160	170	180	190	50	0

Fig. 1. Fragment of the Experimental Dataset Generated from the Results of .NET Microservice Load Testing

During each test run, the system records a set of indicators characterizing both service performance and scaling behavior. From the k6 output data for each run, the average response time `avg_ms`, the p90 and p95 response-time percentiles `p90_ms` and `p95_ms`, the maximum response time `max_ms`, the number of requests per second `rps`, and the percentage of failed requests `failed_percent` are extracted. At the same time, horizontal scaling and resource consumption metrics are collected using Kubernetes tools. After each run, the `kubectl get hpa` and `kubectl top pods` commands are used to record the number of service replicas before and after the test, the current CPU load per pod, and memory utilization. Since the number of replicas may change when the horizontal autoscaling mechanism is triggered, CPU and memory values are averaged across the available pods of the corresponding service at the time of

measurement. All these data are entered into a single `experiment_results.csv` file, where each row corresponds to a separate experimental run.

After the series of runs has been completed, the results collected are processed in Google Colab using Python, the pandas library for tabular analysis, and matplotlib for plotting graphs. At this stage, the data are grouped according to two attributes: service type and load level. For each group, the mean values and standard deviations of the average response time, the 95th percentile of response time, throughput, the number of replicas after the test, CPU utilization, and memory utilization are calculated. Fig. 2 shows the dependence of the average response time of the two .NET microservice implementations on the load level specified by the number of virtual users. The results obtained demonstrate that the optimized implementation consistently provides lower average response times throughout the investigated load range. While the average response time of the basic service varies from approximately 659 to 1,445 ms and increases significantly under a load of 50 virtual users, this indicator remains almost constant for the optimized service at approximately 164–176 ms. This indicates the higher efficiency of the optimized service implementation under container deployment conditions.

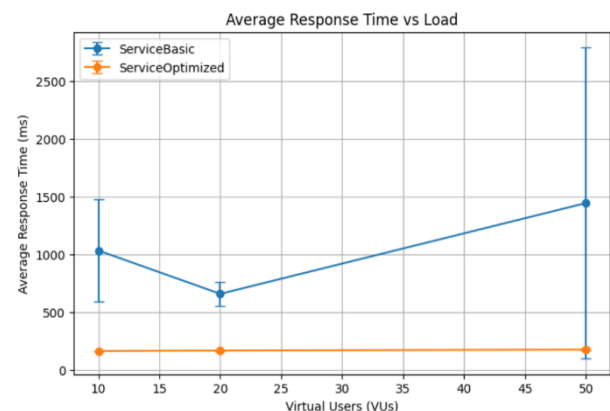


Fig. 2. Dependence of the Average Response Time of the Services on the Load Level

An important feature of the graph presented is also the difference in the variability of the results. The basic implementation exhibits significantly greater deviations, especially under high load, indicating lower predictability of its behavior. In contrast, the optimized service demonstrates minimal dispersion of values, confirming its stability across repeated experimental runs.

Fig. 3 shows the dependence of the p95 response-time percentile on the load level for the basic and optimized .NET microservice implementations. Unlike the average response time, this indicator characterizes service behavior at high latency values and makes it possible to assess the stability of request processing under less favorable operating conditions.

The obtained results indicate that, for the basic implementation, the 95th-percentile response-time values remain substantially higher across all scenarios studied and increase particularly sharply under a load of 50 virtual users. At the same time, the corresponding indicator for the optimized service remains at a significantly lower level and changes only slightly as the load increases.

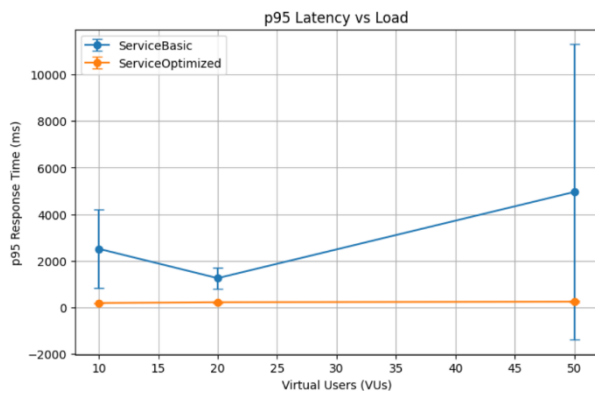


Fig. 3. Dependence of the p95 Response-Time Percentile of the Services on the Load Level

This relationship between the results confirms that the optimized implementation not only reduces the average request processing time but also ensures greater predictability of system operation under peak loads.

The considerable dispersion of values for the basic service indicates its increased sensitivity to rising request intensity and less stable operation under scaling conditions. In contrast, the low p95 response-time percentile values and minor variability between repeated experiments for the optimized service indicate greater resilience to load. Fig. 4 shows the dependence of service throughput on the load level specified by the number of virtual users. The obtained results indicate that, as the load increases, both implementations increase the number of processed requests per second; however, the optimized service demonstrates higher throughput in all scenarios. This difference becomes particularly noticeable at 50 virtual users, where the advantage of the optimized implementation is most pronounced.

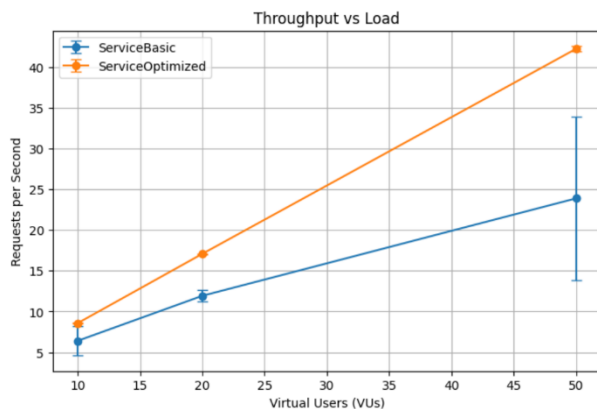


Fig. 4. Dependence of the Throughput of the Services on the Load Level

The presented graph confirms that the applied .NET platform tools and software improvements ensure more efficient resource utilization during request processing. The basic implementation also demonstrates greater variability in the results, whereas the optimized service operates more stably. In the context of this research, this confirms the feasibility of the proposed approach to implementing microservices in container orchestration systems. Fig. 5 shows the change in the number of service replicas after load testing at different request intensity levels.

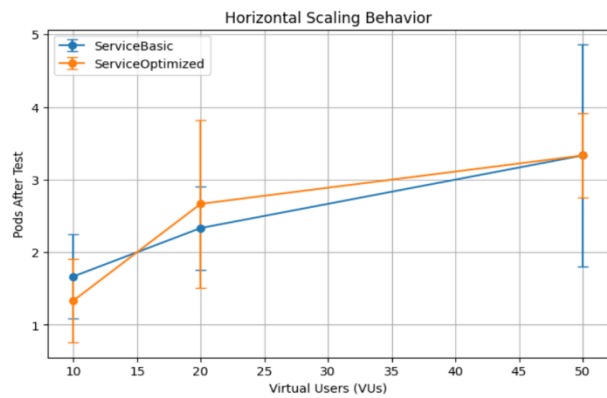


Fig. 5. Change in the Number of Service Replicas During Horizontal Scaling

The obtained results demonstrate that both implementations respond to increased load by increasing the number of replicas, which means that the horizontal autoscaling mechanism operates correctly for both the basic and optimized services. At the same time, the optimized implementation demonstrates a slightly higher average scaling level even under medium load, whereas at 50 virtual users, the values for both services become nearly identical.

The presented graph confirms that the proposed approach makes it possible to study not only the timing characteristics and throughput of .NET microservices but also the specific features of their behavior during horizontal autoscaling in the Kubernetes environment.

The weights were determined using the entropy method based on six aggregated scenarios and are presented in Fig. 6. The weight w_e is equal to zero because the percentage of failed requests was 0% in all 18 runs. This does not mean that the indicator was excluded from the mathematical model – in this experiment, it simply had no discriminative capability. The obtained operational metrics were used to calculate the partial and integral indicators of the risk of defect manifestation. They are presented on Fig. 7. Before performing the calculations, the results of the three repeated runs were averaged separately for each service implementation and load level. For the timing characteristics and throughput, the mean values, standard deviations, and coefficients of variation were determined.

Component	Designation	Weights
Timing characteristics	w_t	0,233
Failed requests	w_e	0
Insufficient throughput	w_q	0,182
Resource utilization	w_r	0,129
Result variability	w_v	0,206
Scaling inefficiency	w_s	0,251

Fig. 6. Weights

Realization	Users	r_t	r_e	r_q	r_r	r_v	r_s	R	Level
ServiceBasic	10	0,556	0	0,67	0,68	0,48	0,205	0,489	medium
ServiceBasic	20	0,285	0	0,56	0	0,2	0,198	0,26	low
ServiceBasic	50	1	0	1	0,74	1	0,8	0,917	high
ServiceOptimized	10	0	0	0,02	0,36	0	0,003	0,052	low
ServiceOptimized	20	0,006	0	0	0,15	0,06	0,001	0,033	low
ServiceOptimized	50	0,011	0	0,05	0,08	0,04	0,022	0,036	low

Fig. 7. Partial and Integral Indicators of the Risk of Defect Manifestation

Since the number of requests processed per second depends on the load intensity, relative throughput, defined as the ratio of the average number of requests per second to the number of virtual users, was used to ensure comparability between the scenarios. The operational indicators were then normalized, and the timing risk component r_t , failed-request component r_e , insufficient-throughput component r_q , resource component r_r , variability component r_v , and scaling-inefficiency component r_s were calculated. The weighting coefficients of the partial indicators were determined using the entropy method. The highest weights were assigned to the scaling-inefficiency component, $w_s = 0.251$, the timing component, $w_t = 0.233$, and the result-variability component, $w_v = 0.206$. The weights of insufficient throughput and resource utilization were $w_q = 0.182$ and $w_r = 0.129$, respectively. Since the percentage of failed requests was zero in all experimental runs, the r_e indicator had no discriminative capability, and its weight within the conducted experiment was set to zero.

The calculation results showed that, for the basic implementation with 10 virtual users, the integral indicator was $R = 0.489$, corresponding to a medium risk level. This value is explained by the considerable variability in the average response time and its 95th percentile between repeated runs. With 20 virtual users, the indicator decreased to $R = 0.260$, corresponding to a low risk level. This decrease is associated with the more stable results of runs B20_1–B20_3 and lower average resource utilization. Under a load of 50 virtual users, the integral indicator of the basic implementation increased to $R = 0.917$, corresponding to a high-risk level. The obtained value was primarily influenced by increases in the average response time and its 95th percentile, a decrease in relative throughput, considerable dispersion of the results, and an increase in the number of replicas without sufficient stabilization of the timing characteristics. For the optimized implementation, the integral indicator values were $R = 0.052$, $R = 0.033$, and $R = 0.036$ for 10, 20, and 50 virtual users, respectively. All the obtained values correspond to a low risk level. The minor change in the indicator as the load increases confirms the stability of the optimized software implementation and its ability to maintain predictable timing characteristics and throughput throughout the investigated load range. Fig. 8 shows the dependence of the integral indicator of the risk of defect manifestation for the basic and optimized .NET microservice implementations on the number of virtual users.

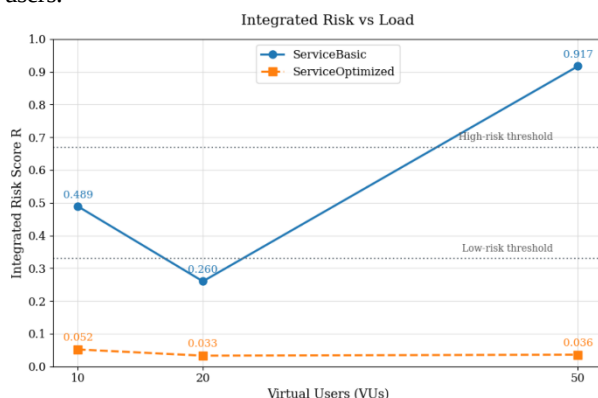


Fig. 8. Dependence of the Integral Indicator of the Risk of Defect Manifestation in .NET Microservices on the Load Level

A comparison of the obtained values shows that the optimized implementation reduced the integral risk indicator by 89.5% with 10 virtual users, by 87.3% with 20 virtual users, and by 96.1% with 50 virtual users compared with the basic implementation. The results obtained confirm the operability of the proposed method and the possibility of its application for the comparative assessment of the risk of defect manifestation in functionally similar software components based on operational metrics.

4. Discussion of the Results of the Study of the Method for Assessing the Risk of Defect Manifestation in .NET Microservices

The proposed mathematical model for assessing the risk of defect manifestation in .NET microservices is based on the comprehensive consideration of timing characteristics, throughput, the percentage of failed requests, computational resource utilization, result variability, and service behavior during horizontal scaling. A distinctive feature of the model is the integration of heterogeneous operational metrics into a single integral indicator R . The experimental study revealed substantial differences between the risk profiles of the basic and optimized implementations. For Service Basic with 10 virtual users, the value $R = 0.489$ was obtained, corresponding to a medium risk level. With 20 users, the indicator decreased to $R = 0.260$ and corresponded to a low risk level. This decrease is explained by the lower variability of the B20 series results compared with the B10 series, which exhibited considerable dispersion in the average response time and its 95th percentile. With 50 virtual users, the integral indicator of the basic implementation increased to $R = 0.917$, corresponding to a high-risk level. The main reasons were a sharp increase in the timing characteristics, a decrease in relative throughput, increased result variability, and an increase in the number of replicas without sufficient performance stabilization. For Service Optimized, the integral indicator values were $R = 0.052$, $R = 0.033$, and $R = 0.036$ with 10, 20, and 50 virtual users, respectively. All values correspond to a low risk level and change only slightly as the load increases. Compared with the basic implementation, optimization of the software logic reduced the integral risk indicator by 89.5% with 10 users, by 87.3% with 20 users, and by 96.1% with 50 users. The results obtained are explained using asynchronous request processing, reduced blocking, caching, and more efficient organization of computations. The absence of failed requests in all experimental scenarios resulted in zero values for the partial component r_e and its weighting coefficient. This does not indicate the absence of defects in the software implementation, since their manifestation may be associated not only with failed request completion but also with excessive delays, instability of timing characteristics, reduced throughput, and inefficient resource utilization. Therefore, the proposed method provides an assessment of a broader class of adverse states of a software component.

Unlike the approaches presented in [1, 6], which focus on the formal verification of autoscaling policies, the proposed method assesses the risk associated with the behavior of the software implementation itself. In [2–5, 7], operational metrics are used primarily for predicting resource requirements, managing scaling, and ensuring

quality of service, whereas the proposed method considers them as interrelated indicators of the risk of defect manifestation. Unlike [8], the proposed approach involves forming an integral assessment that enables the quantitative comparison of the operational profiles of different software implementations. The research results should be applied with consideration of the fact that the risk assessment is comparative and does not represent the probability that a defect is present in the program code. The experiments were conducted in a local Kubernetes cluster for two implementations of one microservice, three load levels, and a fixed test duration. Normalization was performed within the generated dataset; therefore, when applying the method to other systems, the threshold values of the metrics and the weighting coefficients must be determined again.

Further research should focus on validating the method for a larger number of software components, extending the set of load scenarios, and conducting experiments in production clusters. The use of data on confirmed defects, the simulation of individual component failures, and the consideration of network delays. This will make it possible to refine the boundaries of the risk levels and move from a comparative integral assessment to predicting the probability of defect manifestation.

5. Conclusions

1. A mathematical model for assessing the risk of defect manifestation in .NET microservices within distributed computer systems was proposed. A distinctive feature of the model is the comprehensive consideration of the average response time, its 95th percentile, throughput, the percentage of failed requests, CPU and memory utilization, the number of replicas, and the variability of the experimental results. The statistical processing of repeated runs made it possible to consider not only the mean metric values but also the stability of the software component's behavior. To ensure the correct comparison of scenarios with different load intensities, relative throughput, defined as the number of requests per second per virtual user, was used.

2. Partial indicators and an integral criterion for assessing the risk of defects in a software component were defined. The timing risk component, failed-request component, insufficient-throughput component, resource component, result-variability component, and horizontal-scaling-inefficiency component were formed. To reduce the subjectivity of the assessment, the weighting

coefficients were determined using the entropy method. The integral indicator (R) takes values from 0 to 1 and enables the operational state of the component to be classified as having a low, medium, or high-risk level. In the experiments conducted, the percentage of failed requests was zero; therefore, the corresponding component did not affect the integral assessment but was retained in the general mathematical model.

3. A method for assessing the risk of defect manifestation in .NET microservices based on the collection, statistical processing, normalization, and comprehensive analysis of operational metrics was developed. The experimental study was conducted for basic and optimized implementations of a functionally similar service under identical Kubernetes container orchestration conditions. For ServiceBasic, the integral indicator was $R = 0.489$ with 10 virtual users, $R = 0.260$ with 20 users, and $R = 0.917$ with 50 users, corresponding to medium, low, and high-risk levels, respectively. For ServiceOptimized, the value obtained were $R = 0.052$, $R = 0.033$, and $R = 0.036$, all of which corresponded to a low risk level. The optimized implementation reduced the integral risk indicator by 89.5–96.1% compared with the basic implementation. The obtained results confirmed the operability of the proposed method and the possibility of using it for the comparative assessment of the risk of defect manifestation in software components based on operational data without directly analyzing their program code.

Conflict of Interest

The authors declare that they have no conflicts of interest related to this research, including those of a financial, personal, authorship-related, or other nature that could have influenced the research and its results presented in this article.

Funding

The research was conducted without financial support.

Data Availability

The manuscript has no associated data.

Use of Artificial Intelligence Tools

The authors confirm that they did not use artificial intelligence technologies in the preparation of the presented work.

REFERENCES

1. Jawaddi S.N.A., Johari M.H., and Ismail, A. (2022), "A review of microservices autoscaling with formal verification perspective", *Softw Pract Exper.*, vol. 52, is.11, pp. 2476–2495, doi: <https://doi.org/10.1002/spe.3135>
2. Taheri, J., Gördén, A., and Al-Dulaimy, A. (2024), "Using Machine Learning to Predict the Exact Resource Usage of Microservice Chains", *UCC '23*, article 25, pp. 1–9, doi: <https://doi.org/10.1145/3603166.3632166>
3. Nunes, J. P. K. S., Nejati, S., Sabetzadeh, M., and Nakagawa E. Y. (2024), "Self-adaptive, Requirements-driven Autoscaling of Microservices", *SEAMS '24*, pp. 168–174, doi: <https://doi.org/10.1145/3643915.3644094>
4. Pintye, I., Kovács, J., and Lovas, R. (2024), "Enhancing Machine Learning-Based Autoscaling for Cloud Resource Orchestration", *J Grid Computing*, vol. 22(68), doi: <https://doi.org/10.1007/s10723-024-09783-1>
5. Alharthi, S., Alshamsi, A., Alseiyari, A., and Alwarafy, A. (2024), "Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions", *Sensors*, vol. 24(17), art. 5551, doi: <https://doi.org/10.3390/s24175551>
6. Jawaddi, S.N.A., Ismail, A. et al. (2025), "Analyzing Energy-Efficient and Kubernetes-Based Autoscaling of Microservices Using Probabilistic Model Checking", *J Grid Computing*, vol. 23(3), doi: <https://doi.org/10.1007/s10723-024-09789-9>
7. Gurung, N., Shrestha, S., and Chulyadyo, R. (2025), "Scalability in Microservices: A systematic literature review", *Journal of Computer Science and Technology*, vol. 25(2), e.11, doi: <https://doi.org/10.24215/16666038.25.e11>
8. Medavarapu, S.V., and Medavarapu, S.S. (2023), "Enhancing Scalability and Performance of .NET Microservices on Azure Kubernetes Service (AKS)", *J. Sc. and Eng. Res.*, vol. 10(6), pp. 133–139, doi: <https://doi.org/10.5281/zenodo.13597405>

9. Siswanto, M., and Yuhana, U.L. (2023), "Software Defect Prediction Based on Optimized Machine Learning Models: A Comparative Study", *Teknika*, vol. 12(2), 1pp. 66–172, doi: <https://doi.org/10.34148/teknika.v12i2.634>
10. Kabir, S.M.H., Rahman, Md. T., and Mridul, A.H. (2025), "Software Defect Prediction Using Traditional Machine Learning and Ensemble Learning Algorithms", *Smart Wearable Tech.*, vol. 1, A9, doi: <https://doi.org/10.47852/bonviewSWT52025645>
11. Zain, Z.M., Sakri, S., and Ismail, N.H.A. (2023), "Application of Deep Learning in Software Defect Prediction: Systematic Literature Review and Meta-analysis", *Inf. Softw. Technol.*, vol. 158, doi: <https://doi.org/10.1016/j.infsof.2023.107175>
12. Marijan, D. (2023), "Comparative study of machine learning test case prioritization for continuous integration testing", *Software Qual J.*, vol. 31, pp. 1415–1438, doi: <https://doi.org/10.1007/s11219-023-09646-0>
13. Cabral, G.G., Minku, L.L. et. al. (2023), "An investigation of online and offline learning models for online Just-in-Time Software Defect Prediction", *Journal of Systems and Software*, vol. 28, doi: <https://doi.org/10.1007/s10664-023-10335-6>
14. Bayramova, T., (2023), "Software defect prediction using the machine learning methods", *Problems of Information Technology*, vol. 14(2), doi: <http://doi.org/10.25045/jpit.v14.i2.03>
15. Chen, Z., Wang, K., Yang, C., Cao, B., and Fan, J., (2025), "Learning Top-k Classification with Label Ranking", *Communications in Computer and Information Science*, vol. 2755, doi: https://doi.org/10.1007/978-981-95-4094-5_9
16. Tang, P., Wang, J., and Liu, M. (2025), "Variational learning to rank for Test Case Prioritization via prioritizing metric inspired differentiable loss", *Engineering Appl. of Artificial Intelligence*, vol. 141, doi: <https://doi.org/10.1016/j.engappai.2024.109776>
17. Stradowski, S., and Madeyski L. (2023), "Machine learning in software defect prediction: A business-driven systematic mapping study", *Information and Software Technology*, vol. 155, doi: <https://doi.org/10.1016/j.infsof.2022.107128>
18. Robredo, M., Esposito, M., Taibi, D., Peñaloza, R., and Lenarduzzi, V., (2025), *SQuAD: The Software Quality Dataset - Dataset* [Data set], doi: Zenodo. <https://doi.org/10.5281/zenodo.17566691>
19. Mahbub, P., Shuvo, O., and Rahman, M.M. (2023), "Defectors: A Large, Diverse Python Dataset for Defect Prediction", 2023 IEEE/ACM 20th Int. Conf. on MSR, pp. 393-397, doi: <https://doi.org/10.1109/MSR59073.2023.00085>
20. Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., and Abraham A. (2022), "A systematic literature re-view on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools", *Engineering Applications of Artificial Intelligence*, vol. 111, pp. 1–33, doi: <https://doi.org/10.1016/j.engappai.2022.104773>
21. Cabral, G.G., Minku, L.L., Oliveira, A.L., Pessoa, D.A., and Tabassum, S. (2023), "An investigation of online and offline learning models for online Just-in-Time Software Defect Prediction", *Journal of Systems and Software*, vol. 28, 121, doi: <https://doi.org/10.1007/s10664-023-10335-6>
22. Guo, Z., Sun, Y., Fu, B., and Cheng, X., (2022), "Research on software defect prediction method based on machine learning oriented to large-scale complex information system", *ICIEIS2021*, doi: <https://doi.org/10.1117/12.2627233>
23. Giray, G., Bennin, K. E., Köksal, Ö., Babur, Ö., and Tekinerdogan, B. (2023), "On the use of deep learning in software defect prediction", *Journal of Systems and Software*, vol. 195, pp. 1–26, doi: <https://doi.org/10.1016/j.jss.2022.111537>

Received (Надійшла) 06.06.2026

Accepted for publication (Прийнята до друку) 19.08.2026

Publication date (Дата публікації) 25.09.2026

ВІДОМОСТІ ПРО АВТОРІВ / ABOUT THE AUTHORS

Дяченко Дмитро Олександрович – аспірант кафедри електронних обчислювальних машин, Харківський національний університет радіоелектроніки, Харків, Україна;

Dmytro Diachenko – PhD student, Department of Electronic Computers, Kharkiv National University of Radio Electronics Kharkiv, Ukraine;

e-mail: dmytro.diachenko2@nure.ua; ORCID Author ID: <http://orcid.org/0009-0006-5751-3511>.

Шостак Максим Віталійович – аспірант кафедри електронних обчислювальних машин, Харківський національний університет радіоелектроніки, Харків, Україна;

Maksym Shostak – PhD student, Department of Electronic Computers, Kharkiv National University of Radio Electronics Kharkiv, Ukraine;

e-mail: maksym.shostak@nure.ua; ORCID Author ID: <https://orcid.org/0009-0003-1510-7425>.

Метод оцінки ризику прояву дефектів .NET-мікросервісів у розподілених комп'ютерних системах на основі експлуатаційних метрик

Д. О. Дяченко, М. В. Шостак

Анотація. Актуальність. Дефекти програмного забезпечення, не виявлені під час функціонального тестування, можуть проявлятися під навантаженням, що зумовлює актуальність оцінювання ризику на основі експлуатаційних метрик.

Предмет дослідження: методи та засоби оцінки ризику прояву дефектів .NET-мікросервісів у розподілених комп'ютерних системах з урахуванням експлуатаційних метрик. **Мета дослідження:** розробка методу оцінки ризику прояву дефектів .NET-мікросервісів у розподілених комп'ютерних системах на основі комплексного аналізу експлуатаційних метрик. **Завдання дослідження.** Для досягнення мети визначено сукупність експлуатаційних метрик і розроблено модель формування інтегрального показника ризику на основі їх нормованих значень. **Методи дослідження.** Використано методи системного аналізу, нормування та інтегрального оцінювання, навантажувального тестування, а також статистичної обробки. **Результати дослідження.** Запропоновано метод інтегральної оцінки ризику прояву дефектів .NET-мікросервісів за нормованими експлуатаційними метриками. **Висновки.** Запропонований метод дає змогу оцінювати ризик прояву дефектів програмних компонентів розподілених комп'ютерних систем без втручання в їхній програмний код на основі експлуатаційних даних, отриманих під час навантажувального тестування.

Ключові слова: ризик прояву дефектів; .NET-мікросервіси; розподілені комп'ютерні системи; експлуатаційні метрики, інтегральний індикатор ризику; навантажувальне тестування; пропускну здатність; Kubernetes; Docker; надійність програмного забезпечення.