BULLETIN of Cherkasy State Technological University

Journal homepage: https://bulletin-chstu.com.ua/en

Vol. 30 No. 4. 2025

UDC 004.42: 004.7
DOI: 10.62660/bcstu/4.2025.155

# High availability in a microservice architecture

**Bohdan Fedoryshyn***
Postgraduate Student
Lviv Polytechnic National University
79000, 12 Stepana Bandery Str., Lviv, Ukraine
https://orcid.org/0009-0005-3779-0186

**Abstract.** The purpose of this study was to investigate approaches to ensuring high availability of microservice systems with a focus on fault tolerance, scalability, and continuous operation of services. The study applied a comparative and analytical method to analyse technical solutions for ensuring high availability, systematise the characteristics of container orchestration platforms, and evaluate load balancing tools according to the criteria of performance, flexibility, reliability, and integration convenience. Fault-tolerance patterns – retry, circuit breaker, and fallback – that provide flexible error management, reduce the risk of cascading failures, and maintain system continuity are investigated. The study found that the behaviour of fault-tolerance patterns depends on the configuration of execution parameters, such as timeouts, retry limits, and conditions for activating fallback mechanisms. The effectiveness of such tools as NGINX, HAProxy, Envoy, and Amazon Web Services Elastic Load Balancing is evaluated in terms of their impact on the scalability and resilience of the architecture, as well as the possibility of automatic scaling on the example of Amazon Web Services and Google Cloud platforms. It was found that built-in autoscaling services ensure stable operation of services under variable load and enable a rapid response to peak loads. An overview of container orchestrators (Kubernetes, OpenShift, Amazon ES) was provided, among which Kubernetes is recognised as the most effective due to the support of self-healing mechanisms, distributed deployment, health checks, and integration with Continuous Integration/ Continuous Delivery. The findings of this study can serve as an analytical basis for designing sustainable microservice architectures in cloud and enterprise environments to improve the reliability, scalability, and continuity of business processes

**Keywords:** container orchestration platform; distributed deployment; scaling; monitoring; load balancing

## INTRODUCTION

In the context of digital business transformation, microservice architecture has become one of the key approaches to building scalable and flexible software systems. Thanks to their distributed structure, where each service performs a separate function, microservices are easy to develop, deploy, and maintain. However, with distribution comes increased demands on system resilience and uptime. High availability becomes a major factor, as even a short-term downtime can lead to extensive financial losses or a degraded user experience. However, ensuring high availability of such distributed systems is a complex engineering task due to the considerable number of interdependent components, the possibility of partial failures, and the requirement for continuous operation around the clock. It also involves choosing the right container orchestration tools, setting up monitoring systems, and implementing distributed deployment strategies. Despite the availability of powerful platforms such as Kubernetes or OpenShift, in practice their capabilities are not always fully utilised due to a lack of experience or a lack of a systematic approach.

In the field of microservice architecture, a significant challenge is to ensure high availability of the system while maintaining scalability and management efficiency. M.K. Foka (2024) analysed the key advantages of microservice architecture, including development

**Article's History:** Received: 05.07.2025; Revised: 25.10.2025; Accepted: 15.12.2025.

**Suggested Citation:**
Fedoryshyn, B. (2025). High availability in a microservice architecture. *Bulletin of Cherkasy State Technological University*, 30(4), 155-165. doi: 10.62660/bcstu/4.2025.155.

*Corresponding author

flexibility, simplified updates, and the ability to work in parallel by teams. The study showed that automatic scaling and health checks markedly increase the resilience of Web applications, but it was noted that the implementation of these mechanisms is often complicated due to the complexity of the interaction between microservices. T. Seliviorstrova & N. Krasnoshapka (2023) described the role of automatic monitoring and health checks in improving system reliability. The study confirmed the effectiveness of using Kubernetes and analogous platforms to maintain high availability. The specific features of integrating automatic scaling, monitoring, load balancing, and disaster recovery mechanisms should be studied.

In a microservice architecture, ensuring high availability is challenging due to the need to effectively coordinate numerous services, ensure load balancing, maintain data consistency, and resilience to possible failures in the distributed infrastructure. J. Li (2025) proposed an optimisation model to increase the availability and flexibility of blockchain systems built on a microservice architecture. The researcher developed an approach to load balancing and service redundancy, which allows increasing the overall system resilience to failures. D. Rossi (2020) studied the Consistency Availability Partition (CAP-theorem) dilemma in microservice systems, analysing the trade-offs between consistency and availability. The results showed that achieving high availability often requires a reduction in the level of consistency, which affects the consistency of data in distributed systems.

One of the challenges is the complexity of choosing the most suitable tools, frameworks, and platforms to ensure high availability, service consistency, and scalability of a microservice architecture under variable loads. G. Márquez *et al.* (2020) analysed the most popular frameworks and technological solutions used in industrial microservice systems to achieve high availability. The researchers found that most companies use off-the-shelf orchestrators (e.g., Kubernetes) and monitoring tools, but there are challenges with their proper configuration and integration into concrete business cases. G. Liu *et al.* (2020) reviewed the general challenges of microservice architecture, including high availability, containerisation, and service management. The researchers highlighted the complexity of debugging services in a distributed environment, the need to closely monitor the status of services, and manage their interaction as key issues.

Ensuring high availability in a microservice architecture is complicated by the need to scale, maintain performance, system resilience, and integrate numerous services in large distributed and cloud environments. N. Suleiman & Y. Murtaza (2024) reviewed comprehensive strategies for scaling microservices in enterprise applications, with a focus on increasing availability, optimising performance, and system resilience. The researchers highlighted the significance of horizontal scaling to adapt to load changes and emphasised the role of dynamic resource balancing in preventing overload. The researchers also analysed approaches to automatic scaling in cloud environments, considering the specifics of microservice architecture. L. Roda-Sanchez *et al.* (2023) proposed a comprehensive cloud-edge microservice architecture using service orchestration that improves the real-world conditions for deploying and managing services. The researchers emphasised the significance of hybrid interaction between cloud infrastructure and edge devices, which enabled increased performance and reduced delays. Despite the contribution to the development of approaches to ensuring high availability of microservice systems, the analysed studies are mostly applied or narrowly focused and do not cover the complex interaction between scaling, monitoring, load balancing, and automatic recovery mechanisms. Additionally, most studies do not account for the challenges associated with integrating these solutions in large-scale distributed cloud environments with high load dynamics.

The purpose of this study was to substantiate solutions to ensure the reliability, fault tolerance, and scalability of microservice systems, considering the requirements for their continuous operation. To fulfil the purpose of this study, the following tasks were set: studying existing approaches to ensuring the resilience of microservice systems; analysing methods of automatic scaling and load balancing; researching tools and platforms for microservice orchestration.

## MATERIALS AND METHODS

The study of microservice architectures and mechanisms for increasing their fault tolerance and performance was performed through the consistent application of four principal methodological approaches, which included analysis, comparison, and systematisation of technical characteristics and functional capabilities of key technologies. The first stage of the study was based on a functional and comparative analysis of the key fault-tolerance patterns in microservice architectures. Specifically, the study considered the retry, circuit breaker, and fallback patterns, which are fundamental to ensuring the continuity and stability of the system in the face of temporary and long-term failures. The method involved a detailed study of the functional principles of each pattern, their areas of application, and their impact on the stability and response time of the system.

The second stage focused on comparing load balancing systems used in microservice environments. The choice was focused on three key tools – NGINX, HAProxy, and Envoy – due to their ability to provide interactivity, gamification, and adaptive learning. They allow creating dynamic tasks, track student progress, analyse results, and personalise the learning process. This helps to increase motivation, engagement, and efficiency of learning, which directly corresponds to the purpose of

this study. Additionally, the Amazon Web Services (AWS) Elastic Load Balancer cloud solution is considered, the analysis of which covered such balancing models as client, server, Domain Name System (DNS) balancing, Application Programming Interface (API) Gateway, and service mash, based on official guides and typical use cases. A comparative review of Istio, Linkerd, and Consul Connect technologies was used to analyse the possibilities of implementing load balancing using a service mash. The review covered typical functionalities, including centralised traffic management, support for observability, authentication, and security in a dynamic microservices environment. A documented analysis of the implementation of automatic scaling mechanisms in leading cloud platforms was performed: AWS Auto Scaling, Google Cloud Autoscaler. The method of comparative analysis helped to assess their ability to dynamically respond to load changes, ensure fault tolerance, and rational use of resources.

At the final stage, the study reviewed the functionality of modern platforms that support microservice application lifecycle management, with a focus on deployment, scaling, monitoring, and automatic recovery tools. To investigate the life cycle management of microservice applications, the method of comparative analysis of the technical characteristics of tools and platforms was employed, which allowed structuring their functionality according to the key criteria of modern container orchestration platforms Kubernetes, OpenShift, and Amazon Electronic Stability Control (ECS). The analysis included a review of the functionality of these platforms in terms of automatic scaling, health checks, monitoring, and distributed service deployment.

Separately, the study reviewed the documentation on health checks mechanisms – liveness and readiness probes – used in orchestration platforms for prompt detection of faults and automatic restart of problematic components, with examples of application in Kubernetes. Integration of real-time monitoring with microservice lifecycle management to maintain continuous system availability is applied. The distributed deployment of microservices on physical and virtual nodes, including geographical distribution in different data centres or regions, avoids single points of failure, and increases scalability. Overall, the application of these methods helped to obtain a comprehensive and systematic overview of technologies, their advantages and limitations that affect the resilience, scalability, and performance of microservice architectures. The findings obtained became the analytical basis for developing recommendations for implementing the most suitable solutions in corporate and cloud environments.

## RESULTS

The key factors in the resilience of microservice systems are the use of fault tolerance patterns (retry, circuit breaker, fallback), isolation of services in containers to avoid cascading failures, automatic recovery through self-healing mechanisms, monitoring and health checks, and load balancing (Paz & Bernardino, 2018). In microservice architectures, the key patterns of fault tolerance are retry, circuit breaker, and fallback, each of which is used depending on the type and duration of failures that occur in the system (Raj & David, 2021). The retry pattern is used in situations of temporary service unavailability or short-term network errors when the probability of successful re-execution of the operation is high. In case of a failure, the operation is automatically repeated several times at specified intervals. This approach reduces the probability of failure due to temporary problems, but it can also increase system response times, especially if the number of retries and delays are not optimally configured.

In practice, a properly configured retry helps to avoid unnecessary failures without significantly degrading performance, increasing overall system stability. The circuit breaker pattern is used to protect the system from prolonged and stable service failures. It monitors the number of errors over a certain period and, if it exceeds a threshold, temporarily blocks further calls to the problematic service, putting it in an "open" state. In this state, no requests are sent, which gives the service time to recover. After a certain pause, the circuit breaker switches to the "half-open" state, checking whether the service has recovered, and if so, returns to normal operation. This pattern reduces the load on the faulty service and prevents failures from spreading to other system components. The impact on response time in the normal state is minimal, but during activation, there may be a delay due to temporary call blocking. Overall, circuit breakers increase system reliability, especially in case of long-term failures. The fallback mechanism is used as a backup option in cases where the main service is unavailable or malfunctions. In such situations, the system executes alternative logic – e.g., it returns cached data, a standard response, or a message about the temporary unavailability of the service. This allows preserving basic functionality and maintain the user experience even during partial failures. In terms of response time, fallback can be faster than waiting for the primary service to be restored, but with certain limitations in terms of data relevance or completeness. The use of fallback is a significant component of a fault tolerance strategy, as it ensures the continuity of the system. The generalised characteristics of the key fault tolerance patterns that contribute to the stability of microservice systems are presented in Table 1. It reflects their functions, purpose, and implementation features.

The integrated implementation of Retry, Circuit Breaker, and Fallback patterns allows increasing the resilience and availability of microservice architecture, minimising the consequences of failures and ensuring stable system operation in dynamic environments. Service isolation in a microservice architecture ensures component independence by placing each service in its individual container or runtime environment. This

allows localising faults, avoiding cascading failures, and provides flexibility in updating and scaling individual services without affecting the entire system. Self-healing is implemented through mechanisms for automatic restart, replacement of faulty instances, and traffic redirection, which minimises downtime and maintains system stability without operator intervention. Orchestration platforms, such as Kubernetes, provide built-in health checks that automatically monitor the health of services and initiate their recovery in case of a failure.

**Table 1.** Key patterns of fault tolerance in microservices

| Pattern | Description | Purpose | Setup features |
|---|---|---|---|
| Retry | Repeat operation or request in case of temporary failure or unavailability | Reduction of the impact of temporary errors | Configuration of the number of attempts and intervals to avoid overload |
| Circuit Breaker | Mechanism that blocks calls to the service when a stable fault is detected | Prevention of cascading failures and overloading | Transitions between states: closed, open, half-open; error thresholds |
| Fallback | Alternative logic or fallback in case of unavailability of the main service | Maintenance of functionality during failures | Use of cached data or messages for the user |

**Source:** compiled by the author of this study based on F. De Souza Miranda *et al.* (2024)

Monitoring and health checks are key mechanisms that ensure the continuity and stability of microservice systems (Waseem *et al.*, 2021). Monitoring allows tracking the status of each service in real time, analysing performance, detecting anomalies, and potential problems before they lead to failures. In practice, monitoring systems such as Prometheus or Grafana provide visualisation of key metrics such as Central Processing Unit (CPU), memory, response time, which allows quickly detecting anomalies related to overload or "hanging" requests. Collecting metrics, logs, and call tracing ensures transparency of the system's internal processes, which enables prompt response to malfunctions and optimisation of performance. Health checks automate the monitoring of the health of individual components. They periodically check the availability and correctness of services, detecting failures or degradation. For example, the readiness-probe in Kubernetes allows excluding a component from balancing if it is not yet ready to process requests, while the liveness-probe restarts a container in case of a freeze. These checks reduce the average downtime of one instance to a few seconds without the need for an engineer. If a problem is detected, health checks can initiate a service restart or notify the orchestrator to redirect traffic to healthy instances, which prevents the failure from spreading to the entire system. Such mechanisms allow quickly isolating faults and minimising downtime. Implementation of monitoring and health checks helps to increase the reliability and availability of microservice systems, promptly detect problems, reduce the risk of cascading failures, and ensure stable operation even in case of partial failures.

The distributed deployment of microservices is a crucial factor in ensuring high system availability, as it avoids a single point of failure (Kansal & Balasubramaniam, 2024). Due to the fact that services run on different physical or virtual nodes, the failure of a single component does not lead to a complete shutdown of the system. This approach ensures resource redundancy and load balancing, which contributes to greater resilience to failures. Additionally, distributed deployment enables the geographical location of services in different data centres or regions, which increases resilience to localised disruptions, such as power outages or network problems. In practice, this means that delays or failures in one region do not affect the overall performance of the system if the routing of requests between regions is properly configured. This reduces downtime and improves overall system reliability. Implementing a distributed deployment also makes it easier to scale the system, as one can independently add resources where needed without affecting other components. Thus, distributed deployment is an effective means of increasing the availability and resilience of microservice architectures.

One of the key conditions for ensuring high availability and resilience of a microservice architecture is effective load balancing between service instances (Barua & Kaiser, 2024). This allows the system to evenly distribute requests, avoid overloading individual nodes, and ensure real-time scalability. There are several key approaches to implementing load balancing, each of which has its specific features, advantages, and applications. Client-based load balancing implies that the logic of selecting a service instance is delegated to the client. The client has a list of available instances (e.g., obtained through a service discovery service) and decides which one to send a request to. This approach reduces the load on the central components but requires a more complex implementation on the client side and up-to-date information about the instances. Examples of client-side balancing include the use of Netflix Ribbon or Spring Cloud LoadBalancer in microservice applications, when the client receives a list of instances from Eureka, Consul, or ZooKeeper and selects them by an algorithm (e.g., round-robin or random). Google Remote Procedure Call also has built-in client balancing mechanisms, where the client library distributes requests between known instances. En-

voy employs an analogous approach in sidecar mode, where balancing is performed directly at the proxy level, which is integrated into the client loop. Server balancing is implemented through a separate component, the load balancer, which independently routes requests to the relevant instance. For this, special tools such as NGINX, HAProxy, Envoy, or cloud solutions such as AWS Elastic Load Balancing (ELB) are used. The advantage of this approach is centralised request management, but there is a risk of a bottleneck if the balancer is not scalable or fails.

Another relevant approach to organising load balancing is the use of a service mash, which is implemented by introducing proxy servers (sidecars) to each microservice. Istio, Linkerd, and Consul Connect provide automatic traffic redirection based on performance metrics, centralised control, tracing, monitoring, authentication, and other service functions without changing business logic. Istio provides the broadest configuration and integration capabilities with Kubernetes, Linkerd is distinguished by its simplicity and minimal overheads, while Consul Connect focuses on integration with HashiCorp infrastructure and support for hybrid environments. All these solutions increase the reliability and manageability of systems, but require extra computing resources and complex configuration, which should be considered when designing the architecture. To better compare the key load balancing tools, the study analysed their characteristics according to the key criteria: reliability, routing flexibility, configuration complexity, and compatibility with other services. This enables a clearer understanding of the strengths and weaknesses of each solution and helps to choose the best tool depending on the specific needs and features of the microservice architecture. The results of this comparison are presented in Table 2.

**Table 2.** Comparison of load balancing tools by key characteristics

| Tool | Reliability | Routing flexibility | Customisation complexity | Compatibility with other services |
|---|---|---|---|---|
| NGINX | High, proven over years | Support for basic and advanced routing HyperText Transfer Protocol (HTTP), Transmission Control Protocol (TCP) | Moderate, requires configuration knowledge | Extensive module support, integration with CI/CD |
| HAProxy | Very high, widely used | High, support for complex balancing rules | Medium, configuration via configuration files | Solid integration with various protocols and monitoring |
| Envoy | Highly cloud-focused | Very high, support for L3-L7 routing, service mesh | High, requires time for configuration and training | Perfectly integrates with service mash-up platforms |
| AWS ELB | High, automatic scaling and high availability | Limited compared to NGINX and Envoy, standard methods | Low, manageable configuration via AWS console | Tight integration with AWS ecosystem and services |
| Istio | High, centralised traffic management, fault-tolerance | Very high, support for intelligent routing based on performance metrics | High, requires knowledge of Kubernetes and configurations | Support for Kubernetes, Prometheus, Jaeger, and other monitoring services |
| Linkerd | High, simple and lightweight solution | High-level, basic routing for services | Low, easy installation and basic configuration | Solid integration with Kubernetes, Prometheus |
| Consul Connect | Highly centralised service management | High-performance routing with security policy support | Moderate, requires customisation of the HashiCorp infrastructure | Integration with HashiCorp Vault, Terraform, and other HashiCorp services |

**Notes:** CI/CD – Continuous Integration/Continuous Delivery
**Source:** compiled by the author of this study

The comparison shows that all of the tools under consideration provide a high level of reliability but differ in terms of routing flexibility and configuration complexity. Envoy is ideal for complex cloud and service-mix environments that require high adaptability, although it is more resource intensive to implement. NGINX and HAProxy offer an optimum balance between functionality and simplicity and are well suited for traditional systems with a medium workload. AWS ELB is best used in the AWS ecosystem, valuing ease of configuration and deep integration, although with some routing limitations. The choice of a particular tool should be based on infrastructure requirements and project specifics.

Automatic scaling methods are key to ensuring high availability of microservice systems, as they enable the number of service instances to be dynamically adapted to the current load. The key approaches to automatic scaling are horizontal and vertical scaling. Horizontal scaling involves adding or removing instances of

the same service, which allows quickly responding to changes in load and reduces the risk of overloading individual components. Vertical scaling involves changing the CPU, Random Access Memory (RAM) resources of individual instances, which is limited by the physical characteristics of the infrastructure but can be effective for short-term peaks. In cloud environments, automatic scaling is implemented through specialised services that provide an adaptive response to load changes. Specifically, AWS Auto Scaling and Google Cloud Autoscaler are the leading solutions that are widely used to maintain stable operation of microservices.

AWS Auto Scaling works on the principle of continuous monitoring of key metrics such as CPU, number of requests, memory usage, and others. Based on the set thresholds and scaling policies, the service automatically adds or removes instances, maintaining the optimum level of resources. Another major advantage is the tight integration with other AWS services, such as Elastic Load Balancer and CloudWatch, which provides detailed monitoring, alerts, and flexible scaling management. This enables quick response to peak loads and ensures traffic balancing between active instances. The key limitations are a strong dependence on the AWS ecosystem and the complexity of settings when dealing with complex scaling scenarios. Google Cloud Autoscaler supports scaling for both virtual machines and containerised environments in Kubernetes. It monitors CPU, memory, network load, and even custom metrics, enabling precise resource allocation to meet workload demands. Thanks to integration with the Google Kubernetes Engine (GKE), the service provides flexible horizontal container scaling with high response speed. The advantage is support for various types of metrics and fine-tuning of thresholds, which avoids over- or under-scaling. The disadvantage may be a slightly more complex configuration compared to other providers, as well as the need for additional Kubernetes knowledge to fully utilise the features. Both services support integration with health checks and load balancing mechanisms, which allows not only scaling resources but also ensuring business continuity by quickly replacing faulty instances. The choice between these platforms is usually based on the chosen cloud provider, the specifics of the infrastructure, and the scaling needs of the project.

A variety of container orchestration platforms are used to ensure high availability, scalability, and efficient lifecycle management of microservices. Kubernetes is a leading container orchestration system that provides extensive capabilities for automating the deployment, scaling, and management of microservice applications (Zhou *et al*., 2021). Its architecture provides high availability due to mechanisms for automatically checking the state of containers through liveness and readiness probes, which enable the rapid detection of inoperable service instances. Scaling is performed using Horizontal Pod Autoscaler, which responds to the load by measuring resource consumption (CPU, memory) or custom metrics. Additionally, support for geo-distributed deployment across multiple availability zones avoids a single point of failure and thus increases system resilience. Kubernetes integrates closely with CI/CD processes (e.g., through GitLab CI or Jenkins), supports Infrastructure as a Code, and can work in conjunction with service mashups such as Istio, which provides advanced traffic, security, and observability management (Mustyala, 2022). Due to its modularity and distributed nature, Kubernetes is considered the most complete and flexible solution for building a scalable, isolated, and self-healing microservice architecture.

OpenShift is an extension of Kubernetes aimed at enterprise use and includes advanced tools for improved management, security, and application lifecycle automation. Apart from the basic features of Kubernetes, OpenShift integrates built-in CI/CD support through OpenShift Pipelines, providing controlled and repeatable application deployment. The platform features enhanced security policies implemented through the Security Context Constraints (SCC) and Role-Based Access Control (RBAC) mechanisms, which enable granular access control to resources. The monitoring and logging system is based on the Prometheus, Grafana, and EFK (Elasticsearch, Fluentd, Kibana) stack, which provides deep observability. Service scaling is supported by built-in controllers, analogous to Kubernetes. Due to its extensive management capabilities, advanced security, and centralised approach to administration, OpenShift is a reasonable choice for large organisations that require not only high availability but also strict control over infrastructure, access policies, and auditing (Al-Harbi & Al-Qahtani, 2024).

Amazon ECS is a managed container orchestration service on the AWS cloud infrastructure focused on simplifying the deployment and management of microservices. ECS provides basic availability mechanisms through health checks, automatic scaling through AWS Auto Scaling, and load balancing through AWS Elastic Load Balancer. All these features are tightly integrated with other AWS services, such as Identity and Access Management for access control, Virtual Private Cloud for network isolation, and Secrets Manager for secure storage of confidential information. Monitoring is implemented through Amazon CloudWatch, which allows creating triggers and alerts based on load metrics, response time, or number of requests. One of the key advantages of ECS is the minimal complexity of setup – the platform hides most of the technical details, which allows deploying a stable and scalable environment with minimal effort. However, this convenience is accompanied by a deep tie to the AWS ecosystem, which can limit flexibility in multi-cloud or hybrid scenarios (Bagai, 2024). Table 3 presents the key features of each platform, such as automatic scaling, load balancing, monitoring, and implementation features that directly affect the resilience and reliability of services.

**Table 3.** Comparison of container orchestration platforms

| Platform | Scaling | Load balancing | Health Checks | Features | Application |
|---|---|---|---|---|---|
| Kubernetes | Automatic horizontal scaling | Built-in, supports L4 and L7 levels | Liveness, readiness, startup probes | The most popular, with a developed ecosystem, supports Helm, and has a large community | Suitable for all environments, especially in large distributed systems |
| OpenShift | Automatic, improved | Built-in, extended | Advanced features | Security, CI/CD, access control | Enterprises, corporate solutions |
| Amazon ECS | Automatic scaling with AWS Auto Scaling | Built-in, integrated | Supported | Tight integration with AWS infrastructure | Cloud applications, scalable services |

**Source:** compiled by the author of this study based on E. Truyen *et al.* (2019), A.M. Kovalenko (2021)

The integrated use of these platforms allows optimising the deployment of microservices, ensuring their continuity, and responding quickly to failures, which ultimately increases the overall availability and efficiency of the system. Based on this analysis, the following practical recommendations for implementing high availability in microservice architectures can be formulated. It is worth implementing well-known fault tolerance patterns – retry, circuit breaker, and fallback – that allow the system to automatically respond to temporary failures, avoid excessive loads on problematic components, and provide alternative ways to process requests. Correctly configuring the parameters of these patterns is critical to achieving the optimal balance between reliability and performance. One should also use a distributed deployment of microservices across different physical or virtual nodes, as well as in different geographical regions. This approach eliminates a single point of failure, increases resilience to localised failures, reduces the risk of large-scale disruptions, and facilitates faster system recovery. Furthermore, it is vital to implement automatic scaling mechanisms that allow adapting resources to the current load without disrupting services. This ensures system flexibility and saves resources while maintaining stable operation even at peak times. Along with automatic scaling, it is necessary to implement effective load balancing between service instances, which prevents overloading of individual nodes and increases overall system performance.

Monitoring and health checks should be integrated into all levels of the architecture to promptly detect failures and automatically launch recovery processes. This allows promptly responding to problems, minimising downtime, and improving user experience. The use of modern container orchestration platforms such as Kubernetes, Docker Swarm, or OpenShift is a key element of high availability. These platforms support automated deployment, scaling, load balancing, and recovery of microservices, which greatly simplifies the management of complex systems. Finally, the implementation of continuous integration and delivery (CI/CD) practices ensures fast and secure service updates without interruptions, which increases the overall reliability and efficiency of software support.

Comprehensive adherence to these recommendations helps to increase the resilience, reliability, and availability of microservice architectures, minimises the risk of failure and ensures business continuity even in case of a malfunction. To summarise the results, high availability of microservice systems is achieved through a combination of service isolation, fault-tolerance patterns, autoscaling, load balancing, and self-healing mechanisms. The most effective solution in practice is to use Kubernetes with autoscaling, service mash, and distributed deployment. In contrast, more simplistic approaches such as DNS balancing have limitations in scalability and adaptability.

## DISCUSSION

The findings confirmed the significance of using fault tolerance patterns (retry, circuit breaker, fallback), isolation of services in containers, self-healing mechanisms, monitoring and health checks, and load balancing to ensure the resilience of microservice architectures. The implementation of these approaches has helped minimise the impact of failures, localise malfunctions, and ensure system continuity by automating recovery processes. These solutions help to increase system reliability and efficiency, reducing the need for human intervention and improving productivity. The results obtained are consistent with the provisions highlighted in M. Kuppam (2024), where the researcher emphasises the need to develop a resilient architecture through the introduction of self-healing mechanisms that minimise human intervention and automate the response to failures. These research findings have confirmed the practical effectiveness of such approaches when automatic restart or replacement of faulty instances helps to ensure continuous operation of the system. B. Arugula (2024) proposed a model for building resilient cloud-native APIs in event-driven microservice ecosystems that involves autonomous disaster recovery and adaptive request routing. These approaches echo the conclusions of this paper regarding the significance of load balancing between healthy instances, as well as the use of fallback mechanisms that allow the system to continue functioning even in case of individual service failures.

The findings confirmed that automatic scaling is a key mechanism for maintaining the resilience and adaptability of microservice architecture in the cloud environment. Modern services, such as AWS Auto Scaling and Google Cloud Autoscaler, demonstrate high efficiency by monitoring key metrics (CPU, memory, network load) and dynamically managing the number of instances based on predefined policies. Integration with load balancing and health checks ensures fast response to failures and minimises downtime. AWS Auto Scaling is distinguished by its close interaction with other components of the AWS ecosystem (CloudWatch, ELB), which contributes to flexible and centralised scale management. Google Cloud Autoscaler is deeply integrated with GKE and supports custom metrics, providing precise adaptation to load changes in Kubernetes clusters. Both approaches confirm the effectiveness of combining scaling with availability and load control, increasing the overall resilience of microservice infrastructure.

N. Singh *et al.* (2023) reached analogous conclusions, emphasising the role of load balancing mechanisms and service discovery in the Docker Swarm environment for distributed big data systems. The researchers noted the effectiveness of horizontal scaling in reducing the risk of overloading individual containers, which coincides with the conclusions drawn from the study on the benefits of horizontal scaling for flexible adaptation to load changes. S. Rabiu *et al.* (2022) highlighted the problems and challenges of load balancing and auto-scaling in cloud microservices environments, especially the limitations of vertical scaling and the need for dynamic resource management to ensure system resilience. These findings are also consistent with the findings of the present study, which showed that horizontal scaling in combination with service-mashups provides the best level of resilience and adaptability for microservice architectures.

These results confirm that Kubernetes is a leading container orchestration system that provides a prominent level of availability, scalability, and resilience for microservice architectures through deployment automation, horizontal scaling (via Horizontal Pod Autoscaler (HPA)), self-healing mechanisms, and geo-distribution of instances. Built-in health checks (liveness and readiness probes) enable rapid detection of failures, while integration with service mashups such as Istio enables advanced traffic, security, and availability management. The platform tightly integrates with CI/CD processes, supports infrastructure-as-a-service approaches, and is a flexible solution for building an isolated, scalable, and self-healing microservice architecture. In turn, OpenShift, as an enterprise-oriented extension of Kubernetes, offers additional security features (SCC, RBAC), centralised monitoring (EFK, Prometheus, Grafana), and native CI/CD integration through OpenShift Pipelines, making it a viable choice for large organisations with increased access control and audit requirements. Amazon ECS, as a managed solution within the AWS ecosystem, provides a simplified container management model, automatic scaling, CloudWatch monitoring, and deep integration with other AWS services. ECS enables rapid deployment of resilient services with minimal configuration but is less flexible in multi-cloud environments due to its strong tie to AWS.

The results of this study are in line with the findings of A. Saboor *et al.* (2022), who emphasised the significance of efficient orchestration of containerised microservices to ensure high availability, scalability, and reliability of cloud systems. Specifically, the researchers focused on conceptual approaches to automated deployment and life cycle management of microservices, which is in line with the recommendations identified in the study on the use of orchestration platforms (Kubernetes, etc.) to improve service resilience. R.R. Vangala (2018) confirmed the significance of dynamic orchestration and automatic scaling as key factors in the adaptive resilience of microservice systems. The adaptive framework proposed by the researcher emphasised the need for rapid response to load changes and automatic recovery from failures, which is consistent with the results of the present study on the effectiveness of horizontal scaling and automatic load balancing to maintain system continuity.

Monitoring and health checks have played a key role in ensuring the continuity and stability of microservice systems. By being capable of detecting faults at an early stage and responding to them quickly, these mechanisms considerably reduce the risk of cascading failures and minimise downtime. Most significantly, automated health checks help to isolate problematic components and ensure that traffic is redirected to healthy instances, which increases overall system reliability. It also confirms the significance of distributed deployment as a key factor in the high availability of microservice architecture. Running services on multiple physical or virtual nodes eliminates a single point of failure, providing resource redundancy and load balancing. The geographical location of components in different data centres or regions further increases the system's resilience to local failures, which is especially relevant for critical applications with continuous operation requirements.

The findings are consistent with the approaches described in O.V. Talaver & T.A. Vakaliuk (2023), which emphasises the significance of comprehensive system health management to improve system reliability. The researchers emphasised the need to use automated tools to monitor service performance, which enabled prompt detection of failures and minimised the impact of individual component failures on the functioning of the entire system – a conclusion that is consistent with the emphasis on the significance of health checks in the study. Additionally, the findings of the present study are consistent with the findings of Y. Wang *et al.* (2021), who examined both the benefits and challenges of

microservice architectures. Specifically, authors emphasised that the scalability and resilience of systems directly depend on the proper design of distributed deployment and well-established mechanisms for self-healing services. This coincides with the conclusions about the need to avoid single points of failure by distributing the load across different nodes and geographical areas, which allows the system to stay operational even in case of partial failures.

Overall, the findings of the study confirmed that microservice architecture is a modern and effective approach to creating scalable, flexible, and adaptive software systems that can meet the requirements of a dynamic business environment. The comprehensive implementation of microservice architecture, which includes not only technical tools but also suitable management approaches, enables organisations to create highly reliable, flexible, and scalable software systems that can effectively respond to market changes, reduce support costs, and provide high quality customer service.

## CONCLUSIONS

The study found that the stability, reliability, and continuity of microservice systems are achieved through the integrated implementation of a series of architectural approaches, mechanisms, and patterns. Service isolation, automatic recovery of instances, load balancing, and continuous monitoring form the foundation for ensuring stable operation even in case of partial failures or anomalies. Practical cases revealed that the use of fault tolerance patterns – retry, circuit breaker, and fallback – reduces the risk of cascading failures while optimising system response times through adaptive parameter settings. Specifically, retry reduces the probability of failure due to temporary problems. Circuit breaker effectively isolates faulty components, preventing excessive load, while fallback provides alternative request processing scenarios in case of prolonged failures.

Tools such as Consul provided efficient service management, detection, traffic routing, and implementation of fault tolerance patterns (retry, circuit breaker, fallback). In turn, the use of Istio as a service mashup enabled centralised management of inter-service interaction, load balancing, security, and observability. The introduction of distributed deployment of services in different geographical locations considerably reduced the risk of large-scale failures, ensuring high availability and fast recovery of the system even in case of local failures. AWS and Google Cloud provided built-in services for automatic scaling, load balancing, redundancy, and monitoring, which greatly simplifies the construction of fault-tolerant systems with a prominent level of availability. Automatic scaling based on monitoring of key metrics (CPU, latency, number of requests) allowed dynamically adjusting resources to the current load, maintaining system stability at peak times and saving resources during downturns.

The prominent level of reliability of microservice systems was ensured by container orchestration platforms (Kubernetes, OpenShift, Amazon ECS) that automate application lifecycle management: deployment, scaling, monitoring, and recovery. They support load balancing, container isolation, self-healing, and horizontal scaling, which reduces the risk of failures. Integration with CI/CD and infrastructure as code increases stability and control of changes in the production environment. Comparative analysis revealed that Kubernetes is the most flexible and scalable solution, while OpenShift and Amazon ECS offer complementary enterprise and cloud integrations. The proposed recommendations will help to improve the efficiency, reliability, and resilience of microservice systems. Further research could focus on developing adaptive mechanisms for automatically responding to complex failures in microservice architectures using artificial intelligence and machine learning to predict possible failures and optimise resources. Another major area will be exploring the impact of various security and data protection strategies on the overall availability and resilience of systems, which will ensure both reliability and compliance with modern information security requirements.

## ACKNOWLEDGEMENTS

None.

## FUNDING

None.

## CONFLICT OF INTEREST

None.

## REFERENCES

[1] Al-Harbi, F., & Al-Qahtani, A. (2024). Software-defined storage (SDS): Architecture, benefits, and leading platforms. *International Journal of Informatics and Data Science Research*, 1(8), 36-49.

[2] Arugula, B. (2024). Architecting for resilience: Designing fault-tolerant systems in multi-cloud environments. *International Journal of Emerging Trends in Computer Science and Information Technology*, 5(2), 113-121. doi: 10.63282/3050-9246.IJETCSIT-V5I2P112.

[3] Bagai, R. (2024). Comparative analysis of AWS model deployment services. *International Journal of Computer Trends and Technology*, 72(5), 102-110. doi: 10.14445/22312803/ijctt-v72i5p113.

[4] Barua, B., & Kaiser, M.S. (2024). Enhancing resilience and scalability in travel booking systems: A microservices approach to fault tolerance, load balancing, and service discovery. *ArXiv*. doi: 10.48550/arXiv.2410.19701.

[5] De Souza Miranda, F., dos Santos, D.S., Vilela, R.F., Guez Assunção, W.K., dos Santos, R.C., & Costa Pinto, V.H.S. (2024). A proposed catalog of development patterns for fault-tolerant microservices. In *SBQS '24: Proceedings of the XXIII Brazilian symposium on software quality* (pp. 406-416). New York: Association for Computing Machinery. doi: 10.1145/3701625.3701678.
[6] Foka, M.K. (2024). *Research on the effectiveness and advantages of microservice architecture in Web applications*. (Master's thesis, Zaporizhzhia National University, Zaporizhzhia, Ukraine).
[7] Kansal, S., & Balasubramaniam, V.S. (2024). Microservices architecture in large-scale distributed systems: Performance and efficiency gains. *Journal of Quantum Science and Technology (JQST)*, 1(4), 633-663. doi: 10.63345/jqst.v1i4.139.
[8] Kovalenko, A.M. (2021). *Methods and tools for auditing the security of the Kubernetes automatic container orchestration system*. (Masters's dissertation, Igor Sikorsky Kyiv Polytechnic Institute, Kyiv, Ukraine).
[9] Kuppam, M. (2024). The resilient design techniques. In *Enterprise digital reliability* (pp. 87-115). Berkley: Apress. doi: 10.1007/979-8-8688-1032-9_4.
[10] Li, J. (2025). Research on optimization model of high availability and flexibility of blockchain system based on microservice architecture. *Procedia Computer Science*, 261, 207-216. doi: 10.1016/j.procs.2025.04.191.
[11] Liu, G., Huang, B., Liang, Z., Qin, M., Zhou, H., & Li, Z. (2020). Microservices: Architecture, container, and challenges. In *2020 IEEE 20th international conference on software quality, reliability and security companion (QRS-C)* (pp. 629-635). Macau: IEEE. doi: 10.1109/QRS-C51114.2020.00107.
[12] Márquez, G., Soldani, J., Ponce, F., & Astudillo, H. (2020). Frameworks and high-availability in microservices: An industrial survey. In *Proceedings of the XXIII Ibero-American conference on software engineering (CIbSE)* (pp. 57-70). Montevideo: Curran Associates.
[13] Mustyala, A. (2022). CI/CD pipelines in Kubernetes: Accelerating software development and deployment. *International Journal of Science and Engineering*, 8(3), 1-11. doi: 10.53555/ephijse.v8i3.238.
[14] Paz, S., & Bernardino, J. (2018). Web platform assessment tools: An experimental evaluation. In T.A. Majchrzak, P. Traverso, K.-H. Krempels & V. Monfort (Eds.), *Web information systems and technologies* (pp. 45-63). Cham: Springer. doi: 10.1007/978-3-319-93527-0_3.
[15] Rabiu, S., Yong, C.H., & Mohamad, S.M.S. (2022). A cloud-based container microservices: A review on load-balancing and auto-scaling issues. *International Journal on Data Science*, 3(2), 80-92. doi: 10.18517/ijods.3.2.80-92.2022.
[16] Raj, P., & David, G.S.S. (2021). Engineering resilient microservices toward system reliability: The technologies and tools. In R. Achary & P. Raj (Eds.), *Cloud reliability engineering: Technologies and tools* (pp. 77-116). Bora Raton: CRC Press. doi: 10.1201/9781003030973-3.
[17] Roda-Sanchez, L., Garrido-Hidalgo, C., Royo, F., Maté-Gómez, J.L., Olivares, T., & Fernández-Caballero, A. (2023). Cloud-edge microservices architecture and service orchestration: An integral solution for a real-world deployment experience. *Internet of Things*, 22, article number 100777. doi: 10.1016/j.iot.2023.100777.
[18] Rossi, D. (2020). Consistency and availability in microservice architectures. In M.J. Escalona, F.D. Mayo, T.A. Majchrzak & V. Monfort (Eds.), *Web information systems and technologies* (pp. 39-55). Cham: Springer. doi: 10.1007/978-3-030-35330-8_3.
[19] Saboor, A., Hassan, M.F., Akbar, R., Shah, S.N.M., Hassan, F., Magsi, S.A., & Siddiqui, M.A. (2022). Containerized microservices orchestration and provisioning in cloud computing: A conceptual framework and future perspectives. *Applied Sciences*, 12(12), article number 5793. doi: 10.3390/app12125793.
[20] Seliviorstrova, T., & Krasnoshapka, N. (2023). Aspects of designing scalable microservices architecture for web services. *Information Technology Computer Science Software Engineering and Cyber Security*, 4, 58-66. doi: 10.32782/it/2023-4-7.
[21] Singh, N., Hamid, Y., Juneja, S., Srivastava, G., Dhiman, G., Gadekallu, T.R., & Shah, M.A. (2023). Load balancing and service discovery using Docker Swarm for microservice based big data applications. *Journal of Cloud Computing Advances Systems and Applications*, 12, article number 4. doi: 10.1186/s13677-022-00358-7.
[22] Suleiman, N., & Murtaza, Y. (2024). Scaling microservices for enterprise applications: Comprehensive strategies for achieving high availability, performance optimization, resilience, and seamless integration in large-scale distributed systems and complex cloud environments. *Applied Research in Artificial Intelligence and Cloud Computing*, 7(6), 46-82.
[23] Talaver, O.V., & Vakaliuk, T.A. (2023). Reliable distributed systems: Review of modern approaches. *Journal of Edge Computing*, 2(1), 84-101. doi: 10.55056/jec.586.
[24] Truyen, E., van Landuyt, D., Preuveneers, D., Lagaisse, B., & Joosen, W. (2019). A comprehensive feature comparison study of open-source container orchestration frameworks. *Applied Sciences*, 9(5), article number 931. doi: 10.3390/app9050931.

[25] Vangala, R.R. (2018). Adaptive resilience framework: A comprehensive study on dynamic orchestration and auto-scaling of microservices in cloud-native systems. *International Journal of Computer Engineering and Technology*, 9(6), 278-288.
[26] Wang, Y., Kadiyala, H., & Rubin, J. (2021). Promises and challenges of microservices: An exploratory study. *Empirical Software Engineering*, 26, article number 63. doi: 10.1007/s10664-020-09910-y.
[27] 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, article number 111061. doi: 10.1016/j.jss.2021.111061.
[28] Zhou, N., Georgiou, Y., Pospieszny, M., Zhong, L., Zhou, H., Niethammer, C., Pejak, B., Marko, O., & Hoppe, D. (2021). Container orchestration on HPC systems through Kubernetes. *Journal of Cloud Computing Advances Systems and Applications*, 10, article number 16. doi: 10.1186/s13677-021-00231-z.

## Висока доступність в мікросервісній архітектурі

**Богдан Федоришин**

Аспірант
Національний університет «Львівська політехніка»
79000, вул. Степана Бандери, 12, м. Львів, Україна
https://orcid.org/0009-0005-3779-0186

**Анотація.** Метою роботи було дослідження підходів до забезпечення високої доступності мікросервісних систем з акцентом на стійкість до відмов, масштабування і безперервну роботу сервісів. У дослідженні застосовано порівняльно-аналітичний метод, аналізу технічних рішень забезпечення високої доступності, систематизації характеристик платформ оркестрації контейнерів і оцінки інструментів балансування навантаження за критеріями продуктивності, гнучкості, надійності та інтеграційної зручності. Досліджено fault-tolerance патерни – retry, circuit breaker і fallback – які забезпечують гнучке управління помилками, знижують ризик каскадних відмов і підтримують безперервність роботи систем. Встановлено, що поведінка fault-tolerance патернів залежить від конфігурації параметрів виконання, таких як таймаути, ліміти повторних спроб і умови активації fallback-механізмів. Оцінено ефективність таких інструментів, як NGINX, HAProxy, Envoy та Amazon Web Services Elastic Load Balancing, з огляду на їх вплив на масштабованість і стійкість архітектури, а також можливості автоматичного масштабування на прикладі хмарних платформ Amazon Web Services і Google Cloud. Виявлено, що вбудовані сервіси autoscaling забезпечують стабільну роботу сервісів при змінному навантаженні та дозволяють оперативно реагувати на пікові навантаження. Надано огляд оркестраторів контейнерів (Kubernetes, OpenShift, Amazon ECS), серед яких Kubernetes визнано найбільш ефективним завдяки підтримці механізмів самовідновлення, розподіленого розгортання, health checks та інтеграції з CI/CD. Результати дослідження можуть слугувати аналітичною основою для проєктування стійких мікросервісних архітектур у хмарному та корпоративному середовищах з метою підвищення надійності, масштабованості та безперервності бізнес-процесів

**Ключові слова:** платформа оркестрації контейнерів; розподілене розгортання; масштабування; моніторинг; балансування навантаження