ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 9, No. 1, 2024

AN APPROACH TO IMPROVING AVAILABILITY OF MICROSERVICES FOR CYBER-PHYSICAL SYSTEMS

Oleh Chaplia[1], Halyna Klym[1], Anatoli I. Popov[2]

[1]*Lviv Polytechnic National University, 12, Bandera Str, Lviv, 79013, Ukraine,*
[2]*Institute of Solid State Physics, University of Latvia, 8, Kengaraga, Riga, LV-1063, Latvia.*
Authors' e-mails: *oleh.y.chaplia@lpnu.ua, halyna.i.klym@lpnu.ua, popov@latnet.lv*

https://doi.org/10.23939/acps2024.01.016

Submitted on 15.04.2024

***Abstract*: The design of modern Cyber-Physical Systems (CPS) connects physical and digital realms from cloud systems to edge devices. Microservice architecture has been widely used for IT solutions and emerges as a promising approach for supporting CPS that are more efficient, adaptable, and interconnected. However, there is an increasing need to improve the availability, reliability, and resilience of microservice systems according to the needs. This paper summarizes the challenges and drawbacks of microservice architecture used for CPS. Then, the simplified microservice model has been created, initial properties have been defined, and an improvement plan has been presented. The microservice model's availability has been improved using a novel approach with endpoint containerization. Then, the discussion and conclusions have been offered to explore the full potential of integrating the physical and digital realms.[1]**

***Index Terms*: cloud computing, cyber-physical systems, Industry 4.0, Internet of Things, microservices**

I. INTRODUCTION

In modern computing, combining digital technologies with physical processes has given rise to an intricate and highly dynamic domain known as cyber-physical systems (CPS) [1]. These systems integrate computation, networking, and physical processes, with embedded computers and networks monitoring and controlling the physical processes, often with feedback loops where physical processes affect computations and vice versa [1]. CPS's complexity and critical nature, encompassing sectors such as autonomous vehicle networks, smart grids, robotic systems, and industrial automation, demand highly resilient, flexible, and scalable architectural solutions [2].

Adopting Microservices Architecture (MSA) in cyber-physical systems is a strategic move toward addressing these demands effectively, especially for the needs of Industry 4.0 [3]. Microservices have been characterized by their small, modular, and independently deployable nature [3]. This architectural style has gained prominence for its ability to enhance availability, scalability, resilience, and reliability [4]. MSA's decentralized nature enhances CPS's resilience by isolating failures from individual services without impacting the system [4]. Through patterns such as circuit breakers, bulkheads, and retries, MSA ensures that CPS remains operational and responsive, even during service failures or external disruptions [4]. Containers like Docker and Kubernetes are often used too [5]. However, containerization, such as Docker, takes too long to build, deploy, or restart [5]. This time, it costs money and increases the service downtime. System design patterns are beneficial but sometimes not implemented when the project starts, especially for startups, but issues arise later [5]. Today, cloud computing technologies evolve very fast. Therefore, it leads to additional expenses on resource allocation. Thus, resource allocation optimization is welcomed [6]. Security incidents also significantly impact resources and degrade system availability [7].

This paper reviews different microservice architectures and common approaches for Cyber-Physical Systems and finds techniques that are very commonly used. The authors propose a novel approach based on "virtual containers" in the source code for each endpoint described in the paper [8]. These "containers" handle different errors on the endpoint level and try to restart when some error arises. However, this approach integrates into the source code as a framework. Therefore, previously described techniques may be omitted, and costs reduced. The main goal of this research paper is to find an approach that can improve microservice system availability without additional costs and overheads on cloud computing infrastructure tools.

II. LITERATURE REVIEW AND PROBLEM STATEMENT

This section defines and describes microservices' availability, reliability, and resilience properties. It also reviews the cloud provider's SLAs and necessary metrics related to these properties.

Availability, reliability, and resilience are critical attributes of a microservices architecture that ensure services are robust, perform as expected, and are accessible

[1] This article uses the materials and results obtained by the authors during the research work "Optimized nanocomposites and sensor structures for defense systems security control and threat detection," state registration number 0122U000807, which is carried out at the Department of Specialized Computer Systems of the Institute of Computer Technologies, Automation and Metrology of Lviv Polytechnic National University in 2022-2024.

when needed, contributing significantly to the overall system's quality of service [9]. They are tightly related to the DevOps Metrics and KPIs [10]. These properties are related to each other.

Availability measures the system's operational time when it can execute all necessary operations without failures [9]. It is often expressed as an uptime percentage against overall execution time [10]. High availability in microservices is attained through practices like auto-scaling, load balancing, and deploying services across multiple zones or regions to withstand failures [11].

Reliability concerns the system's capability to perform its required functions under stated conditions for a specified period [9]. It is about the consistency and accuracy of the service outputs. Techniques such as retries with exponential backoff and deploying redundant instances of services can enhance reliability [9].

Resilience refers to a system's ability to handle and recover from failures, ensuring minimal impact on performance and user experience [9]. In microservices, resilience can be achieved through patterns like Circuit Breaker, which prevents a network or service failure from cascading to other services, and Bulkhead, which isolates failures within one from affecting others [9].

Reliability and resilience critically impact availability time [9]. By enhancing reliability, a system reduces the probability of failures. This improvement directly contributes to increased operational time, boosting the system's availability [11]. This approach could mean implementing robust error handling, effective load balancing, and thorough testing for microservices to ensure each service performs reliably under various conditions [9].

Even with high reliability, failures can occur due to unforeseen issues [9]. Resilience ensures that when failures happen, the system's recovery mechanisms activate swiftly to minimize downtime. In a microservices architecture, resilience might involve practices like automatic failover, replication, circuit breakers, and quick rollback capabilities for deployments that do not go as planned [11]. The faster a system can recover from failure, the less its overall downtime, thus enhancing availability [10].

A Service Level Agreement (SLA) for a cloud provider is a contractual document that outlines the expected level of service and specifies performance metrics such as uptime, response times, and data integrity [12]. It details scheduled and unscheduled downtime procedures, data management policies, security, backup, recovery processes, and compliance standards to ensure data protection [12].

The SLA also defines customer support parameters and additional agreement notes, providing a comprehensive framework for service delivery between the cloud provider and the customer [12]. Service Level Agreements (SLAs) often quantify these attributes in metrics.

Standard SLA metrics include availability (uptime), performance (response time), error rates, and latency. Availability is when services are expected to be available and operational within a given period, such as a month or year, typically expressed as a percentage (e.g., 99.9 % uptime). Performance (response time) is the expected performance level regarding response times. Error Rate is the acceptable rate of errors or faults in the service. Latency specifies the maximum delay that can be expected when processing requests.

It is essential to recognize that SLAs ensure reliability, resilience, and availability from the cloud provider's side concerning their infrastructure. Software engineers and DevOps teams can control only some of the provided components [12]. However, these properties improve system reliability. Even when the cloud provider handles different cases of failures of his hardware and software, the system may still fail on network connections, application errors, third-party issues, service integration, and deployment issues, among others [12]. Microservice architecture is complex, often connecting many services and third-party dependencies. Each of the microservice system's components may fail. In the context of CPS and IoT, the chances of additional complexity and error rate increase. Testing and validating complex systems are necessary to cover most error cases [13]. MSA architecture also fits well for testing CPS, including reliability properties [13].

Many research papers are oriented toward the precise design of Cyber-Physical Systems and IoT, which is essential in their scope of work. For example, a cloud and IoT-based green healthcare system provides a solution to facilitate remote monitoring and support for patients [14]. The paper about emergent CPS systems mentions the wide usage of containerized microservices [15]. Docker is a popular solution, but additional infrastructure costs and deployment time are needed. Another interesting approach is chaos engineering, which improves system reliability by adapting and self-healing [16]. Chaos Engineering is a methodology that integrates unexpected failures into a system. Self-healing means that microservices can restore themselves automatically.

However, they use widespread techniques and tools to support microservices architecture. The provider's SLA, containers, and tools like load balancers, proxies, and service meshes cover some reliability. This approach works well for ideal cases or testing environments when the system state is controlled and observed. Reviewed papers also show a tendency to use containers (Docker; Kubernetes).

III. SCOPE OF WORK AND OBJECTIVES

This section highlights the central research questions guiding our investigation of reliable and resilient Microservice Architecture (MSA) within Cyber-Physical Systems (CPS). First, the paper provides the context and drawbacks for the MSA, challenges, and applications in CPS, specifically for microservices' resilience, reliability, and availability properties and patterns. The main research question answers how the availability of microservices used for CPS systems may be improved by at least 99,99 % for 24 hours of uptime. The default availability of microservice provided by SLA from cloud providers is near 99,95 %, which can be increased. All additional

features provided by the cloud provider cost money, impacting the overall cost of the CPS system. Cost optimization may also be helpful.

IV. THE MODEL DEFINITION OF THE MICROSERVICE SYSTEM

This section defines a simplified microservice model to evaluate further the microservice architecture commonly used in CPS systems.

As it has been expected, in real-life cloud systems, networks and designed microservice architectures may face increased errors because Cyber-Physical Systems include hardware devices and edge computing tools within an unstable environment. These unstable environmental properties increase the need for reliable microservice architectures in these systems.

Fig. 1 presents the system architecture layers from application to edge layers. This figure depicts the place of microservice architecture within the overall CPS system.

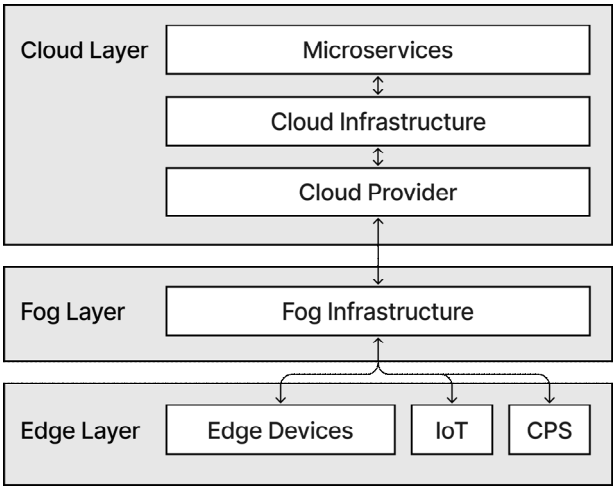


Fig. 1. Overall system architecture

Before proposing improvements, a simplified microservice system model has been defined. The model contains common properties described in most reviewed papers but does not contain specific or unique components. Cloud provider availability and reliability provided by SLAs and DevOps deployment practices before the system starts are omitted. Chaos engineering, a non-deterministic way of testing microservices, is excluded from the model. Microservice system design and patterns are the focus of the improvement. The model's basic blocks will be microservices within Docker containers. Load balancer, replication, and health checks are commonly used patterns for reliability and are often implemented on the cloud provider's side.

Fig. 2 presents a simplified model consisting of microservices.

This model will be used for further evaluation, validation, and calculations. MS stands for microservice, and the number stands for its ID. For example, MS1 means microservice number 1.

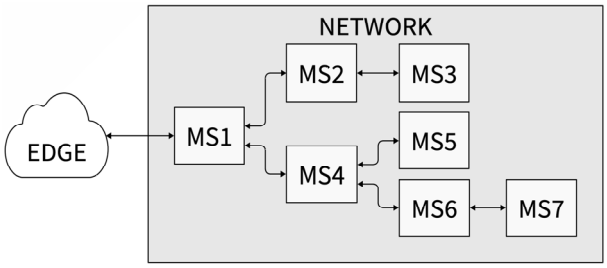


Fig. 2. A simplified model of microservice architecture

V. IMPROVING THE RELIABILITY OF MICROSERVICES FOR CYBER-PHYSICAL SYSTEMS

This section provides methods and approaches for improving the reliability of microservices in cyber-physical systems based on the simplified microservice model. The improvements provided in this section are expected to answer the main research question.

Main terms, formulas, and definitions are provided to calculate a microservice system's availability, reliability, and resilience. It is essential to state that these properties may be calculated differently for different cases, but common intuition is similar overall.

Availability is the leading property of these systems because it states how much time the system works against the total system working time. Reliability, resilience, and fault tolerance for microservices are components of total availability. Total availability (Availability) is often calculated as follows

$$Availability = \left(\frac{TotalOperatingTime}{TotalTime} \right) \times 100\ \%. \quad (1)$$

Based on this formula, availability is calculated in percentages by dividing operating time (TotalOperating-Time) by total measurement time (TotalTime). Very often, when system models, prototypes, and new approaches are designed, they are assumed always to work, almost like in an ideal world. However, in real life, errors and failures are widespread, especially for IoT and CPS systems, because these systems depend on unstable conditions of the real world. Each failure has a different level at which it appears. Achieving 100 % is almost impossible. That is why SLAs provide percentages like 99,9 %, 99,99 %, and 99,999 %.

Fig. 2 represents a system of seven microservices. A system is a set of connected microservices in a specific way. Each microservice may be dependent on another service. A set of microservices M where each microservice is represented as m_i, and i ranges from 1 to n, where n is the total number of microservices. Therefore, the set M can be defined as

$$M = \{m_1, m_2, m_3, ..., m_n\}. \quad (2)$$

With a set of microservices (M), the total operating time (*TotalOperatingTime*), is the sum of the operating

times (*OperatingTime*) for each microservice (*i*) divided by the number of all microservices (*n*), which is calculated as follows

$$TotalOperatingTime = \frac{\sum_{i=1}^{n} OperatingTime_i}{n}. \quad (3)$$

Total time is the timeframe for a system to operate from start to end. Each microservice runs in parallel. Therefore, a microservice's maximum operating time equals the total time. The sum of operating times for each microservice will exceed the total operating time by the number of microservices. Therefore, a correction for this is included – a sum of operating times for each microservice is divided by the count of microservices (*n*).

The operating time (*OperatingTime*) for microservice is the time it successfully executes during the total timeframe of measurement or system execution. Operating time is represented as a difference between total time (*TotalTime*) and total downtime (*TotalDowntime*), as follows

$$OperatingTime = TotalTime - TotalDowntime. \quad (4)$$

Downtime is the time when a microservice is unavailable due to planned or unexpected reasons.

Total downtime (*TotalDowntime*) for the microservice due to unplanned reasons, incorporating various unexpected events and failures, is a sum of all downtime events (*m*), as follows

$$TotalDowntime = \sum_{j=1}^{m} DowntimeEvent_j. \quad (5)$$

An unplanned failure causes downtime. *M* is the total number of unplanned downtime events (*DowntimeEvent*) considered over a defined period (*TotalTime*). $DowntimeEvent_j$ is the downtime of one event (*j*).

Reasons for downtime may be planned or unplanned. Planned reasons for downtime are testing, deployment, or maintenance. During planned reasons of downtime, developers and DevOps engineers optimize the cloud infrastructure and service architecture in a way that will not impact the system or impact only when it is not critical to the end users. Unplanned reasons include unexpected events, failures, dependency, network, infrastructure, system, or even microservice runtime errors. Downtime is also called inactivity time. Downtime also consists of an automatic or manual restoration microservice process because the service is still unavailable during the restoration. Service operation time during the overall execution time includes deployment, execution, failure event, downtime, restore process event and execution. Execution is a prosperous state of microservice execution. Deployment, failure event, downtime, and restore event are states when the microservice is inactive, e.g., states the microservice downtime.

Each microservice has a chance of failure, and its operability is independent. The operability of each microservice affects the system's overall availability. The system is considered fully operational only when all required microservices are functioning.

Failure events include several levels: cloud provider failures, cloud infrastructure failures, virtualization failures, OS system-level failures, microservice failures, unexpected events, failures, dependency issues, network problems, infrastructure issues, third-party dependency issues, failures due to security vulnerabilities, configuration errors, failures due to high-load, deployment issues, system errors, or microservice runtime errors. The microservice developer, DevOps, and IT expert can control and manage the cloud infrastructure resources, custom virtualization configurations, and microservices. However, they cannot maintain cloud infrastructure and all components related to the cloud provider software, hardware, and data center.

Table 1 shows the availability results for seven cases. Each row presents one case with an increased amount of failed microservices. The total measurement time is 24 hours, which is 1440 minutes. Each failed microservice has one downtime event. The downtime event is taken as 5 minutes for these calculations for each failed microservice. In real life, downtime depends on many factors, such as the event's cause, whether the server was redeployed, restarted, or even the underlying VM failed. Operating time per microservice is the difference between this microservice's total time and total downtime. For example, a total operating time for a set of seven microservices, where one failed with one downtime event, which takes 5 minutes, will consist of 1435 minutes for one failed microservice and 1440 minutes for the other six microservices. The same calculation approach applies to other cases.

Given the same *DowntimeEventTime* for each failed microservice, a simplified formula was used to calculate *TotalOperatingTime* the Table 1, as follows

$$TotalOperatingTime = \\ = TotalTime - \frac{DowntimeEventNum \cdot DowntimeEventTime}{n}. \quad (6)$$

Table 1

Availability for the simplified model with n=7 microservices, TotalTime=1440 minutes and DowntimeEventTime=5 minutes

DowntimeEventNum	TotalOperatingTime (minutes)	Availability
0	1440	100 %
1	1439,28	99,95 %
2	1438,57	99.90 %
3	1437,85	99.85 %
4	1427,14	99.80 %
5	1436,42	99.75 %
6	1435,71	99.70 %
7	1435	99,65 %

DowntimeEventNum is the number of all downtime events that occurred for each microservice.

Improvements to reliability and resilience have been added to increase the availability of microservice systems. The model can only cover some existing cases. However, essential failure cases can be mitigated. Outage and recovery times should be minimized as much as possible.

The first approach is to containerize each microservice execution code within its execution framework. This approach is described in "An approach for automatic self-recovery for a Node.js microservice" [8]. These containers are special "wrappers" around the API endpoints inside the execution framework. They exist within the microservice application itself without the additional infrastructure overhead of managing VMs, machines, and networking traffic.

This approach covers the cases when dependency microservice fails or unexpected errors for specific endpoints appear. When one container fails, the system tries to execute the code in the next available container by switching the containers in the runtime. The service remains operational, ensuring no downtime. This method significantly reduces outage time without affecting the time required for redeployment or recovery. According to the "container" design, the switch time is less than one second [8]. Therefore, an assumption is made for calculation purposes that the switch time is 1 second. Applying this approach also reduces costs to the cloud infrastructure. However, each framework needs additional development if the system is developed using different programming languages and frameworks. Recovery time will be zero because everything works. The pessimistic case may impact the microservice when it fails to execute all three or more "containers," and the error bubbles through the service to the bottom layer of the system, causing the microservice process to crash. According to the model, this approach is applied to each microservice (from MS1 to MS7). The time measure is in minutes, so conversion to seconds is practical. Twenty-four hours is 86400 seconds. One second is 0.016 minutes.

Table 2 presents calculation results using the formula for availability for this approach. For a system of seven microservices, where one of them has a slight delay of one second because of container switching, the availability time is 99,99 %.

So, the availability of the microservices system with one failed microservice increased from 99,95 % to 99,99 %. Similar calculations will show that the overall downtime is reduced when more microservices fail.

The second microservice pattern that helps improve the system reliability is the Retry pattern. Retry will try to execute some endpoint several times (in this case, three times are selected). When the first execution fails, some time is given to the dependency microservice to switch the execution container and try to execute the endpoint one more time. The subsequent execution will likely be successful. At least three tries are expected. There is no outage time in this case because the system tries to execute the dependency code. This pattern works well with the previous one. According to the model, this pattern is applied to each microservice with a dependency - MS1, MS2, MS4, and MS6.

Table 2

Availability for the improved model with n=7 microservices, TotalTime=1440 minutes and DowntimeEventTime=0.016 minutes

DowntimeEventNum	TotalOperatingTime (minutes)	Availability
0	1440	100 %
1	1439,99	99,99 %
2	1439,99	99.99 %
3	1439,99	99.99 %
4	1439,99	99.99 %
5	1439,99	99.99 %
6	1439,99	99.99 %
7	1439,99	99,99 %

The third helpful pattern is the Response Caching with predefined responses when failure appears. The data may be prefetched and cached for specific cases for each microservice. When some API endpoints do not need to provide real-time updates or actual data at a particular moment, this API may degrade functionality in a tradeoff of reducing outage time. When some failure happens, or even after the switch of containers, the system responds to the client microservice with a cached response. The service does not have an outage and recovery time in this case. Only a network delay will happen to prefetch cached response. As an assumption, it may take up to several seconds or even less. For example, Redis provides high throughput and access rates, so only network latency may impact the overall cache request turnaround time. According to the model, this pattern is applied to each microservice with a dependency - MS1, MS2, MS4, and MS6. Therefore, availability is not impacted for microservices, which have caching instances. On the other hand, this approach impacts data consistency because the data in the cache may not be up to date compared to the database storage.

Load balancing will help with the availability of microservices. This approach needs additional deployments, network configuration, and costs to balance. It forwards the traffic to a set of replicated microservices instead of one microservice, adding further complexity to the system and maintaining the load balancer service and replicas of the microservice. It distributes network traffic and balances resources across each microservice, enhancing performance and availability. Load balancers are often placed before a system's critical services or entry points. As an assumption, load balancing may be used for MS4 because it has dependencies and may need more processing power. A load balancer is a well-defined microservice with its own configuration time.

A Rate limiter is a mechanism or software component designed to control the rate of operations or requests sent to or processed by a system, microservice, or API. Its primary purpose is to ensure that the throughput of requests does not exceed the system's capacity to handle them, which can prevent system overload, ensure fair usage, and maintain service availability. Rate limiting is essential in scenarios where resources are limited and

costly or when a sudden surge in traffic could lead to service degradation or failure. The Rate limiter pattern should be implemented within the microservice application. As it is difficult to calculate the rate limiter's impact on the system's availability without monitoring the specific details of the model in runtime, it is excluded from the current calculation.

Fig. 3 presents the improved microservice architecture for the model. Fig. 3 depicts that MS1, MS2, MS4-1, MS4-2, MS4-3, and MS6 microservices contain a cache instance for response caching, and a load balancer is added before MS4-1, MS4-2, and MS4-3. Each microservice contains logical containers within the application and framework layer. Rate limiting and retry patterns should be implemented within the code of the microservice application.

In addressing the challenge of enhancing microservice availability, reliability, and resilience, this work emphasizes the critical role of integrating resilience patterns and strategies at the stage of new prototyping approaches for CPS. Focusing on microservice and dependency failures proposes a robust approach to effectively minimize outage and recovery times. The critical strategy involves using specific containers within the framework, significantly reducing outage times to nearly zero, and limiting recovery times to about 1 second per instance. This approach is complemented by implementing patterns such as Retry, Response Caching, and Load Balancing, each contributing to the system's ability to maintain high availability and withstand failures.

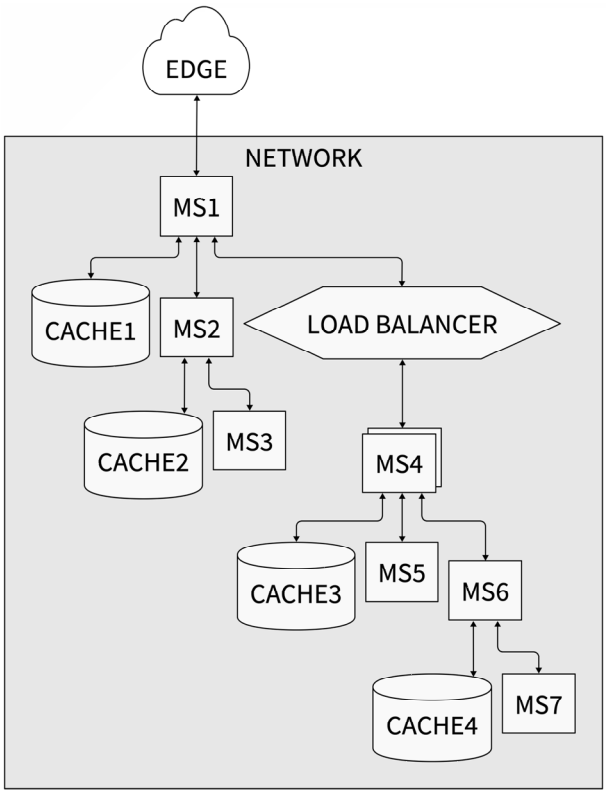


Fig. 3. Improved microservice architecture

Retry patterns eliminate outage times by facilitating multiple execution attempts, ensuring service continuity. Response Caching is an immediate fallback mechanism, and load balancing distributes workload and prevents resource saturation. Together, these strategies form a comprehensive model for developing resilient microservice architectures that are both highly available and cost-effective.

VI. DISCUSSION

The inherently distributed nature of MSA introduces significant complexities in service coordination, requiring advanced orchestration tools and expertise to manage inter-service dependencies and communications effectively. Such complexities are compounded by CPS's real-time and near-real-time operational demands, where network latency and the need for instantaneous data consistency and synchronization pose substantial obstacles to maintaining system responsiveness and reliability.

Each of the proposed approaches and patterns provides some improvements to the system. However, they also need some time for integration and experience in system design to adapt and integrate them into the microservice system. Development time will be increased in containerized code execution within the framework described in the paper “An approach for automatic self-recovery for a Node.js microservice” [8]. Caching provides faster responses even when a third-party service fails but needs additional infrastructure management, including machines and deployed instances of cache databases like Redis. These additional overheads lead to increased costs. Load balancing is similar in terms of infrastructure overhead. Retry and rate-limiting patterns are commonly implemented within the microservice code. Therefore, additional development and testing time is needed. These challenges underscore the need for a deliberate, well-considered approach to implementing MSA in CPS, balancing the architecture's inherent benefits with the complexities it introduces.

VII. CONCLUSION

Maintaining availability and resilience against various failures is paramount in the evolving the landscape of microservices. This work has dissected the layers of potential failure events within a microservice architecture, ranging from cloud provider and infrastructure failures to more granular levels, such as OS system-level failures, microservice failures, and dependency issues. It shows that while developers may have limited control over external cloud failures, they significantly influence microservice configurations, cloud resource management, and implementing resilience patterns that can dramatically enhance system reliability and availability.

An approach introduced is the wrapping (like a virtual container) of the executed code within the microservice framework, allowing for an automatic failover mechanism that ensures minimal outage times by swiftly switching execution to backup containers upon failure.

According to the availability of the microservices system with one failed microservice, it increased from the default value of 99,95 % to at least 99,99 %. This strategy significantly reduces outage time and offers a cost-effective solution to infrastructure redundancy without requiring extensive redevelopment across different programming frameworks.

Integrating resilience patterns such as Retry, Response Caching, and Load Balancing fortifies the system against failures. The Retry pattern eliminates outage time by attempting operation executions multiple times, allowing dependent services to recover seamlessly. Similarly, Response Caching provides immediate fallback responses to ensure uninterrupted service, even during backend failures. Load Balancing distributes traffic and computational load evenly across service instances, thereby preventing outages due to resource overutilization and enhancing overall system performance and availability.

This comprehensive model of incorporating patterns into microservice architectures offers a robust framework for developing highly available and resilient systems. It underscores the necessity of adopting a multi-faceted approach to system design that anticipates and mitigates potential failures at every level of the service stack. By doing so, developers can ensure that microservices survive in the face of disruptions and thrive, maintaining operational integrity and providing uninterrupted service to users. This model, represented by an improved architecture diagram, serves as a blueprint for future developments in microservice resilience, promising a more stable, efficient, and reliable ecosystem for software applications.

References

[1] Tyagi A. K., N. Sreenath., (2021). Cyber physical systems: analyses, challenges and possible solutions, *Internet of Things and Cyber-Physical Systems*, vol. 1, pp. 22–33, DOI: 10.1016/j.iotcps.2021.12.002.

[2] Serôdio C., Mestre P., Cabral J., Gomes M., Branco F., (2024). Software and architecture orchestration for process control in Industry 4.0 enabled by cyber-physical systems technologies, *Applied Sciences*, vol. 14, p. 2160, DOI: 10.3390/app14052160.

[3] Pontarolli R. P., Bigheti J. A., De Sá L. B. R., Godoy E. P., (2023). Microservice-oriented architecture for Industry 4.0, *Eng*, vol. 4, no. 2, pp. 1179–1197, DOI: 10.3390/eng4020069.

[4] Mena M., Criado J., Iribarne L., Corral A., Chbeir R., Manolopoulos Y., (2023). Towards high-availability cyber-physical systems using a microservice architecture, *Computing*, vol. 105, no. 8, pp. 1745–1768, DOI: 10.1007/s00607-023-01165-x.

[5] Fritzsch J., et al., (2023). Adopting microservices and DevOps in the cyber□physical systems domain: A rapid review and case study, *Software: Practice and Experience*, vol. 53, no. 3, pp. 790–810, DOI: 10.1002/spe.3169.

[6] Kniazhyk T., Muliarevych O., (2023). Cloud computing with resource allocation based on ant colony optimization, *Advances in Cyber-Physical Systems*, vol. 8, no. 2, pp. 104–110, DOI: 10.23939/acps2023.02.104.

[7] Malik M. I., Ibrahim A., Hannay P., Sikos L. F., (2023). Developing resilient cyber-physical systems: a review of state-of-the-art malware detection approaches, gaps, and future directions, *Computers*, vol. 12, no. 4, p. 79. DOI: 10.3390/computers12040079.

[8] Chaplia O., Klym H., (2023). An approach for automatic self-recovery for a Node.js microservice in *2023 13th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, Athens, Greece, pp. 1–4. DOI: 10.1109/DESSERT61349.2023.10416461.

[9] Yin K., Du Q., (2020). On representing resilience requirements of Microservice Architecture Systems, *arXiv*. DOI: 10.48550/arXiv.1909.13096

[10] Amaro R., Pereira R., Da Silva M. M., (2024). DevOps metrics and KPIs: a multivocal literature review, *ACM Computing Surveys*, vol. 56, no. 9, pp. 1–41. DOI: 10.1145/3652508.

[11] Boor M. V., Borst S. C., Van Leeuwaarden J. S. H., Mukherjee D., (2022). Scalable load balancing in networked systems: a survey of recent advances, *SIAM Review*, vol. 64, no. 3, pp. 554–622. DOI: 10.1137/20M1323746.

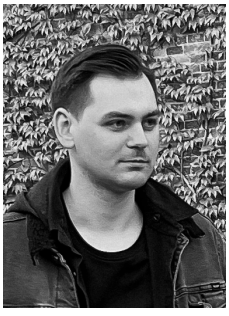
[12] Bernal A., Cambronero M. E., Núñez A., Cañizares P. C., Valero V., (2022). Evaluating cloud interactions with costs and SLAs," *The Journal of Supercomputing*, vol. 78, no. 6, pp. 7529–7555. DOI: 10.1007/s11227-021-04197-2.

[13] Aldalur I., Arrieta A., Agirre A., Sagardui G., Arratibel M., (2024). A microservice-based framework for multi-level testing of cyber-physical systems, *Software Quality Journal*, vol. 32, no. 1, pp. 193–223. DOI: 10.1007/s11219-023-09639-z.

[14] Islam Md. M., Bhuiyan Z. A., (2023). An Integrated scalable framework for cloud and IoT based green healthcare system, *IEEE Access*, vol. 11, pp. 22266–22282, DOI: 10.1109/ACCESS.2023.3250849.

[15] Ward G., Janczewski L., (2022). Investigating data risk considerations in emergent cyber physical production systems, *Journal of Systemics, Cybernetics and Informatics*, vol. 20, no. 2, pp. 51–62, DOI: 10.54808/JSCI.20.02.51.

[16] Naqvi M. A., Malik S., Astekin M., Moonen L., (2022). On evaluating self-adaptive and self-healing systems using chaos engineering, in *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 1–10. DOI: 10.1109/ACSOS55765.2022.00018.



Oleh Chaplia was born in Lviv, Ukraine. He is a PhD student in the Specialized Computer Systems Department at Lviv Polytechnic National University, where he received his B.S. and M.S. degrees in Computer Engineering.

After completing his master's degree in 2015, he has been working in the software engineering field. He has extensive commercial experience designing and developing enterprise-grade cloud solutions incorporating innovative technologies, state-of-the-art approaches, and high-quality system architecture.

His research interests include emerging cloud computing technologies, distributed systems, microservices, artificial intelligence, and software architecture.

Halyna Klym - doctor of technical sciences, professor, professor of the department of Specialized Computer Systems of the Institute of Computer Technologies, Automation and Metrology of Lviv Polytechnic National University.

In 2008, she received a degree of Doctor of Philosophy in the specialty: Physical and Mathematical Sciences at Ivan Franko Lviv National University.

In 2016, she received a Doctor of Science degree in Technical Sciences at Lviv Polytechnic National University. She conducts lecture courses on the design of ultra-large integrated circuits and methods and means of automated design of computer systems. She is an author of more than 170 scientific articles in international publications.

Anatoli I Popov is a Doctor of Physics, Senior scientist at the Institute of Solid State Physics, University of Latvia, one of the world's leading experts in the field of solid-state radiation physics, sensor materials for cyber-physical systems, a board member of Crystal Clear Collaboration at CERN, board member of Enabling Research Projects on Materials, EUROfusion, is an author and co-author of more than 250 articles (Scopus), including articles from the first quartile Q1 (Scientific Reports, Ceramics International, Surfaces and Interfaces, Journal of Materials Chemistry C, Nanomaterials, Symmetry, Journal of Materials Research and Technology, Journal of Nuclear Materials). In total, over the past 5 years, according to SCOPUS, 104 articles have been published.