Research Article

Pavle Dakić*

Software compliance in various industries using CI/CD, dynamic microservices, and containers

https://doi.org/10.1515/comp-2024-0013
received December 28, 2023; accepted May 24, 2024

Abstract: The microservices architecture is widely used in modern businesses due to its ability to offer speed, efficiency, adaptability, autonomy, and usability. On the other hand, this architectural paradigm demands a well-designed infrastructure for optimal container and cluster utilization. Establishing version control and a solid continuous integration/continuous deployment (CI/CD) infrastructure becomes critical for accelerating software delivery to production and ensuring code alignment with best practices. This scientific investigation investigates the development of customized, specialized CI/CD procedures for software compliance, expanding its scope beyond traditional software delivery to include the complexities of design, testing, and server deployment. By integrating real-world examples from industry and reviewing crucial tools, the goal is to provide organizations with empirical information to navigate the difficulties of modern software development. The upcoming research seamlessly fits into the larger discussion, providing a deep understanding of sophisticated structures and their design. This interdisciplinary research combines the scientific principles governing microservices with the practical specifics of CI/CD methodologies, giving businesses a thorough understanding and practical insight into the tools needed to navigate the diverse landscape of modern software engineering. Among the main findings of this research is a suggestion for a new approach known as General repository compliance operations.

Keywords: software compliance, dynamic microservices, CI/CD and YAML, pipeline scripts, containers using IoC

*** Corresponding author: Pavle Dakić,** Faculty of Informatics and Computing, Singidunum University, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Bratislava, Slovakia, e-mail: pavle.dakic@stuba.sk

1 Introduction

The rise of microservice architecture in modern organizational frameworks demonstrates its profound impact on structural dynamics. This architectural paradigm offers significant benefits, such as increased speed, efficiency, adaptability, autonomy, and usability, making it essential for navigating the ever-changing corporate landscape. Achieving high performance requires skillful engineering that takes advantage of recent technology developments, particularly containers, and clusters in computer science [1,2].

Microservices essentially represent small, independently deployable services embedded within applications that improve modularity, scalability, and flexibility in software development. However, accessing these services and native cloud computing presents significant hurdles for everyone due to cost limits and complexity. This article studies the complex orchestration and control of microservices within infrastructure frameworks, with a focus on their possible integration into the automotive industry. Standardization is especially important in operations such as collecting and processing data for vehicle parking identification, as it ensures precision and compliance with specific rules [3].

Diverse technologies and container distribution approaches are thoroughly investigated to determine the best deployment tactics [4–6]. Throughout the development lifecycle, a variety of challenges combine to impede collaboration, including task distribution issues, organizational process ambiguities, specification uncertainties, communication breakdowns, cultural differences, and standard adherence complications. Overcoming these difficulties requires excellent management processes for software compliance and information exchange throughout the car production process. Rapid testing emerges as a vital component in ensuring compliance with industry standards and expediting software deployment into production systems.

This initiative aims to provide businesses with the empirical insights needed for informed decision-making in modern software development and deployment paradigms by a thorough evaluation of real-world case studies and critical explanations of key technologies. Our core goal remains to enable a standardized procedure to create,

deploy, and administer applications within Kubernetes clusters while adhering to common compliance criteria in the industry. Clusters promote immersive experiential learning by providing hands-on experience with key components such as CI/CD pipelines, Helm charts, and namespace management. The use of new ideas that the incorporation of CI/CD and their connections enables the integration of static and dynamic components within the CI/CD pipelines [7,8].

The specific issue is the difficulty in managing microservices within infrastructure frameworks. This specifically refers to communication hurdles, organizational uncertainty, and ambiguity in the requirements and planning to be managed. These constraints restrict effective coordination throughout the development lifecycle, highlighting the critical role of software compliance management and knowledge sharing in automotive software development. Our examination focuses on the effective application of new ways to describe requirements and navigate compliance issues. This analytical pursuit ends in a complete exposition, methodically analyzing the complexities inherent in the creation of advanced systems and providing essential information on the prudent deployment of the necessary instruments [9,10]. Applicability of our discourse includes key aspects such as implementation challenges, technology modalities, productivity impact assessments, and instructional learning from experiential projects. Furthermore, enhancing the scalability and adaptability of deployed systems is critical to ensure seamless alignment with changing business imperatives and demands.

This study is organized as follows: Section 1 is introduction, Section 2 is contributions and novelty, Section 3 is literature review, Section 4 is materials and methods, Section 5 is microservices and different software architecture designs, Section 6 is containerizations, Section 7 is analysis of continuous integration (CI) in agile deployment, Section 8 is results, Section 9 is limitations of the study, Section 10 is discussion, and Section 11 is conclusions. According to the stated research questions defined Section 4, an appropriate organization was performed within the sections that follow.

2 Contributions and novelty

The research's contributions and innovation stem from its complete examination and integration of numerous aspects of cloud computing, real-world applications, microservice architecture, CI/continuous deployment (CD), and so on. Noteworthy contributions and novelty include the following:

(1) Empirical knowledge for future decision making: The goal of this article is to provide companies with the empirical knowledge they need to make informed decisions in the dynamic terrain of current software development and deployment to understand the construction of microservices. This helps professionals and students manage the complexities of microservices and native cloud computing by breaking down real-world scenarios and assessing technology.
(2) Accessibility for students and professionals: Recognizing the challenges that students face while accessing microservices and native cloud computing environments, the text addresses this topic with insights and practical examples. Our primary focus was on providing hands-on learning while also developing and managing local clusters that can increase accessibility and skill development in a short amount of time.
(3) Unique CI/CD approaches to software compliance: This article advances the field by investigating and presenting novel CI/CD approaches built exclusively for software compliance. We experimented with this novel way to address the issues of design, testing, and server deployment processes, pushing beyond the normal distribution and development of software.
(4) Kubernetes cluster administration: By providing actual tasks linked to standard compliance, the investigation aids in gaining a better understanding of Kubernetes cluster administration. Hands-on experience with CI/CD pipelines, Helm charts, microservices, and namespace management in a cluster is required.
(5) Comprehensive case study and practical insights: A detailed case study that provides significant value by providing practical information and a review of key literature on this topic. This hands-on test demonstrates the complexity of designing advanced microservice architectures, bridging the gap between theoretical principles and real application, and improving comprehension and applicability.
(6) Application in the industry: The investigation adds to previous contributions by demonstrating a practical implementation of microservices in CI/CD with the creation of a new method and acronym General Repository Compliance Operations (GRCopOps). The fundamental aim behind this term is to achieve synergy and accelerate the process of software standardization. With the use of microservices to connect several technologies, we can tackle specific challenges, including parking and container distribution.

In summary, the research stands out for its comprehensive approach, which combines theoretical principles

with practical applications, making it a valuable resource for both academics and industry practitioners seeking a deeper understanding of CI/CD, microservices, and their real-world implications. Understanding the complexities of service-oriented architecture (SOA) is critical to maximizing its benefits. Similarly, the distinction between standard virtual machines (VMs) and containerization is critical. This insight guides strategic decisions about system architecture, scalability, resource optimization, and deployment efficiency. In today's rapidly changing technology scene, such understanding promotes adaptation and assures alignment with industry best practices, eventually increasing competitiveness and innovation. Our study seeks to address and resolve difficulties such as scalability, fault tolerance, modularity, container orchestration, application programming interface (API) administration, service communication, regulatory compliance, and efficient management of distributed transactions in computing infrastructure.

3 Materials and methods

The foundation for the research approach will be the application of the research questions. These research questions should address the educational component of microservices and cloud computing environments, with a focus on enhancing accessibility in the future. To explore applications of microservices approaches to enable hands-on learning while taking into account the problems that students and professionals confront, and emphasizing the practical aspects of cluster administration and standard compliance.

3.1 Research questions

The research within this study will be organized and implemented based on the following research questions:

(1) How can unique CI/CD methodologies be developed to ensure software compliance within the context of a microservices architecture?
(2) What insights and practical lessons can be derived from a comprehensive case study of designing, testing, and deploying intricate microservices architectures, considering real-world examples and industry applications, particularly in the automotive sector?
(3) How can accessibility and hands-on learning be enhanced for students and professionals in the domain of microservices, native cloud computing, and Kubernetes cluster administration, particularly regarding standard compliance and practical applications in diverse industries?

3.2 Keywords

The process of finding relevant literature was carried out using some of the following combinations:

(1) Software compliance case study and microservices architecture.
(2) Real-world examples CI/CD and microservices architecture in the automotive industry.
(3) CI/CD compliance in Kubernetes cluster and microservices architecture.
(4) Dynamic landscape of software development and Kubernetes cluster administration.

4 Literature review

The widespread adoption of microservices in cloud-based applications is evident, both internally within enterprises and externally in diverse environments. While microservice architecture improves scalability, it poses performance difficulties that must be carefully considered while designing performance models and scheduling jobs. Despite their prevalence, these difficulties remain largely unexplored. Microservices provide granular control over cloud applications, resulting in widespread use across sectors [11].

Using the collected literature and the findings of a thorough investigation, we can examine the factors that influence the performance of microservices provided by this type of serverless computing. In addition, we can see that the model driven engineering (MDE) paradigm is utilized to construct agnostic languages that automatically subsequently change into infrastructure as a service – infrastructure as code – in the majority of scenarios [12,13].

4.1 Microservice capacity (MSC)

Jindal et al. [14] addressed the issue of identifying the MSC for each microservice individually. Each microservice implements its functionality and communicates with the others via a language and a platform-independent API. Jindal et al. showed that their resource consumption varies according to the functionality implemented and the workload. Continuously increasing load or a sudden load spike may result in a service level objective violation. Podolskiy et al. [15] covered the concept of MSC as the maximum rate of requests. Their research examines the possibilities of successful functioning without violating the SLA. Based on this, in their study, defining their behavior in the application of a microservice that is appropriate for the user.

As a result, the construction and usage of automated computer flows can be simplified when the later connected interchange of different tools is implicitly derived from the need to properly harmonize with each other by having the needed file format, which we could see within these mentioned papers [16,17]. As in this case, after adopting a standardized YaML format, while looking for examples of direct application, we found articles and some of our research in this field that cover some of these challenges [18–20].

4.2 Web application microservices

The current investigation examined the impact of search engine optimization and manual ad set optimization (Google dynamic ads) vs the usage of microservices and YaML configuration files. Reading most of the papers obtained, one can discern the goal of building a knowledge model through a microservice measurement architecture with an explicit structure of mutual relations within the WINNER research project [21]. They were developing acceptable suggestions for continuous data transmission, dealing with dynamic container provisioning, and scaling appropriate microservices inside the cloud computing environment in data centers in this manner.

Subsequently, allocation profiles could be created for the demands of various web application microservices, which together with containers could be geographically dispersed to cloud data centers. Another example is the presentation of a software package for investigating the rate of movement/detection of the earth's surface in areas of extensive coal mining, as well as the usage of various computational demands for computer hardware resources. Saboor et al. [22] provided an overview of a green cloud computing environment and conducted a literature review focusing on the importance of different Docker files. They analyzed previous studies that included the largest collection of Docker files from 2013 to 2020, comprising 9.4 million files extracted from the World of Code infrastructure [22,23].

4.3 Error prediction and fault localization for microservice applications

Identifying failures and faults in microservice systems for prediction and localization in production remains difficult, due to the dispersed and dynamic nature of the design of microservices. Although highly desirable, the intricate interdependencies and different runtime environments that characterize microservice ecosystems make it difficult to handle these challenges efficiently. Zhou et al. [24] proposed MEPFL, a technique for latent error prediction and fault localization for microservice applications based on system trace logs. Thus, the prediction models may be utilized in the production environment to forecast latent faults, malfunctioning microservices, and fault kinds for runtime trace instances. This has allowed data center operators to approach hosting microservices in a very different way from traditional infrastructures.

Kannan et al. [25] presented GrandSLAm, a framework for efficiently running microservices in data centers. They show that implementing a microservice design dramatically reduces the work required for server adoption and maintenance. Furthermore, providing a catalog of functions as services might serve as a basis for developing applications in this ecosystem. This paradigm change enables data center operators to manage data centers and host microservices in ways that are not possible with traditional architectures. As a result, it requires a reconsideration of resource management strategies in various execution scenarios.

4.4 Monolithic systems

The rapid development of mobile edge computing (MEC) in recent years has provided an efficient execution platform at the edge for Internet of Things (IoT) applications. However, MEC delivers ideal resources to various microservices; yet the underlying network conditions and infrastructures intrinsically alter the execution process in MEC. As a result, in the presence of fluctuating network conditions, it is required to optimize end-user task execution while maximizing energy efficiency on the edge platform, as well as to ensure equitable quality-of-service.

Monolithic systems come into play, offering standard development environments that are simple, efficient, and straightforward to deploy. Their key strengths are scaling and operating expenses, which are manageable despite a lack of flexibility and maintainability as applications grow. The decision between monolithic and modular architectures is influenced by application size, complexity, and team expertise. Although monolithic systems have various advantages, they may have drawbacks such as independence and debugging time [26].

4.5 Biometric three-tier microservice architecture

The three-tier, biometric-based microservice architecture is designed to decrease fraudulent activity by employing

advanced authentication technologies and assuring secure user verification. Although monolithic systems offer several advantages, they also present limitations, such as reduced independence and prolonged debugging periods [27]. Consequently, there has been a paradigm shift towards adopting microservice architecture for application development. A promising solution involves the integration of Passport (passport.js), a Node.js (nodeAuth-passport) module renowned for its simplistic yet robust authentication procedures. Using Passport as a middleware within the Node.js application and integrating it with streamlined authentication processes allows simple integration. In particular, Passport boasts compatibility with various authentication schemes, including OAuth 1.0 and OAuth 2.0, and integrates seamlessly with popular providers such as Twitter, Facebook, and LinkedIn. Moreover, Passport's flexibility allows for customization, enabling adaptation to unique requirements, a crucial feature given the nuanced specifications inherent in the OAuth 2.0's intricate authentication framework [27,28].

In response to the challenge of duplication of passport (passport.js) packages, an innovative biometric-based monolithic authentication architecture is gaining traction. This approach circumvents the need to scrutinize all tokens and their functionalities within Node and the passport.js library. Alternatively, another strategy involves Service A and Service B interfacing with an authentication service to validate authorization tokens, albeit at the expense of increased interservice traffic due to the validation of each HTTP request containing a token. However, to mitigate network latency and cost, dynamic scheduling of microservices becomes imperative. Unlike standard techniques, a best-practice proposal is presented for a dynamic microservice scheduling strategy for MEC, aimed at optimizing resource allocation and improving overall system efficiency [29].

4.6 MicroCause framework

An increasing number of Internet applications are applying microservice architecture due to its flexibility and clear logic. Meng et al. [30] proposed a framework, MicroCause, to accurately locate root cause monitoring indicators in a microservice. MicroCause combines a simple yet effective path condition time series algorithm that accurately captures the sequential relationship of time series data and a novel temporal cause-oriented random walk (TCORW) method that integrates causal relationship, temporal order, and priority information of the monitoring data. We could see that Meng et al. [30] evaluated MicroCause based on 86 real-world failure tickets collected from a top-tier global online shopping service. Their experiments show that the top five accuracy (AC@5) of MicroCause for intra-microservice failure root cause localization is 98.7%, which is greatly higher (by 33.4%) than the best baseline method.

4.7 Complexity and dynamism of microservice systems

The complicated and unpredictable structure of these systems presents unique problems for a variety of software engineering activities, particularly failure diagnosis and debugging. Despite their extensive use and importance in industry, little research has been done on failure analysis and debugging processes. Given their frequency and importance in current software development, there is a significant need for research to address the analysis and debugging problems associated with microservice designs.

To fill this gap, Zhou et al. [31] performed an industrial survey to learn about common microservice system problems, existing debugging practices, and the challenges experienced by developers in practice. The study's findings demonstrate that existing industrial debugging processes can be improved by using proper tracing and visualization techniques and methodologies. The findings by Zhou et al. also indicate that there is a considerable need for more intelligent trace analysis and visualization, such as integrating trace visualization and enhanced fault localization, as well as using data-driven and learning-based recommendations for guided trace exploration and comparison.

4.8 MEC

With MEC, microservices can be dynamically deployed on edge clouds, quickly launched, and easily transferred between edge clouds, providing better services to users near them who use a wireless connection [32,33]. User mobility can result in a frequent switch to nearby edge clouds, increasing the service delay when users move away from their edge clouds. To overcome this issue, Wang et al. [34] investigated the coordination of microservices between edge clouds to provide real-time and seamless responses to requests for mobile user service. Their study introduces a novel service architecture that facilitates the fragmentation of a monolithic web service into a series of lightweight, independent services through microservices.

MEC allows microservices to be dynamically deployed in edge clouds, launched quickly, and easily transferred between edge clouds, resulting in improved services for nearby users. However, user mobility can result in frequent switching of neighboring edge clouds, increasing

service latency as users move away from their serving edge clouds. Loosely coupled, lightweight microservices running in containers are gradually replacing monolithic applications.

Luo et al. [35] provided a solid analysis of large-scale microservice deployments in Alibaba clusters [36]. They do a thorough study to distinguish the differences between microservice call graphs and traditional data-parallel directed acyclic graphs. Their analysis uncovers distinguishing features of microservice call graphs, such as heavy-tailed distribution, tree-like topology, and frequent hotspots between microservices. In their research, they identified three important call dependencies that are critical for optimizing microservice designs. Furthermore, their research shows that microservices are significantly more vulnerable to CPU interference than memory interference, based on an assessment of microservice runtime performance.

5 Microservices and different software architecture designs

Microservices, a modern paradigm of software architecture, decomposes programs into independently deployable services, revolutionizing the previous monolithic method. Each microservice is self-contained, allowing for greater flexibility, scalability, and autonomy in development and deployment. Microservices, as opposed to monolithic structures, allow teams to focus on individual functionality, facilitating parallel development and faster iterations. Service-oriented architectures contain functional, autonomous, and reusable components that can be accessed remotely via a local or external Internet WAN network. They facilitate polyglot programming (microservices do not need to share the same technologies, frameworks, and libraries) and can be created in multiple programming languages. They allow developers to create independent services without disrupting the flow of other services and make development organization much easier by allowing microservices to develop autonomously. They can be deployed independently and updated separately, eliminating the need to reinstall the entire program [37].

Various software architecture designs coexist, each catering to a different set of requirements. In contrast to microservices, monolithic designs combine components into a single unit, simplifying development, but potentially restricting scalability. In contrast, serverless architectures abstract infrastructure administration, allowing developers to focus exclusively on code. Asynchronous communication is used in event-driven systems to improve responsiveness and scalability [38].

The decision between these designs is influenced by project needs, scalability objectives, and the dynamics of the development team. They are renowned for their ability to adapt to complex, dynamic contexts, promoting a paradigm shift in software development that stresses agility, resilience, and resource efficiency. The interaction with other architectural designs continues to influence the landscape as technology progresses, giving developers a varied toolkit to meet diverse difficulties in the ever-changing world of software engineering.

5.1 Monolithic design shortcomings

Monolithic architecture has various flaws that become apparent as the applications become more complicated. Scalability is a significant challenge because scaling requires replicating the entire monolithic application, which may result in resource inefficiencies. A monolith's lack of modularity and independence limits development flexibility and agility, making it difficult to update or modify individual components without redeploying the entire system. Technology stack restrictions stifle innovation even further, because monoliths frequently rely on a single technology stack, limiting the use of varied tools and frameworks. Monolithic architectures' high coupling and complexity make them difficult to understand, maintain, and troubleshoot. The necessity to relaunch the complete application raises the risks of deployment, which can generate errors and disruptions [39,40].

Inefficiencies in resource usage, resistance to adopting new technology, and development bottlenecks in large teams are all disadvantages of monolithic architecture. Although such limitations are suitable for simpler applications, they highlight the need for more scalable, modular, and adaptive architectural paradigms, such as microservices, to meet the changing needs of modern and sophisticated software systems [38].

5.2 SOA

SOA introduces distinct security considerations that are critical for securing distributed systems. In SOA, where services communicate across networks, data integrity, confidentiality, and authentication become critical. One of the inherent security challenges is the potential exposure of sensitive information during service interactions. Securing SOA involves implementing strong authentication systems

to authenticate the identities of service providers and clients. Credential management, secure token exchange, and the adoption of standards such as Security Assertion Markup Language all contribute to a safe authentication architecture [41].

For protecting communication channels between services, encryption techniques such as secure sockets layer or transport layer security are used to maintain data integrity. Additionally, digital signatures can be used to validate the integrity and validity of sent messages. Authorization techniques are critical to managing access to services, since they ensure that only authenticated and authorized entities can use certain features. To secure SOA, fine-grained access control and role-based authorization frameworks are often used [42].

Additionally, the inclusion of security gateways and firewalls helps to monitor and filter incoming and outgoing traffic, fortifying the overall security posture of a service-oriented ecosystem. Regular audits, threat assessments, and adherence to best-in-security practices contribute to a comprehensive security strategy, fostering trust and reliability in SOA implementations. SOA is a style consisting of discrete services rather than a monolithic design. Breaking down a monolithic design into a system of services interacting via messages via REST APIs allows businesses to have as minimal production downtime as possible (improve availability), isolate issues, and expand applications [43]. This implies the persistence of a different structure that divides an application into smaller services that communicate with each other. The goal is to bring together multiple services without relying on each other.

5.3 Stateless/stateful microservices

Our current research on this subject aims to improve the scalability of both models by striking a balance between efficiency and reliability, ultimately increasing overall performance and user experience in a wide range of organizations. This endeavor requires weighing the advantages and disadvantages of each strategy, as well as recognizing the trade-offs between scalability and data persistence for optimal system design in certain applications [43,44].

5.3.1 Stateless microservices

As the name stateless indicates, this type of microservice does not preserve any actual state. Instead, they receive a request, process it, and return a response to the caller without storing any information, relying on databases to provide the necessary information if needed.

5.3.2 Stateful microservices

Stateful microservices, on the other hand, keep information in some form. Many users want to be able to resume where they left off, and stateful microservices make that possible [44].

5.4 Scalability

To obtain optimal system performance, we need to rigorously evaluate stateless and stateful microservices for scalability. The importance of stateless microservices stems from their ability to handle requests independently without keeping session data, which promotes scalability and fault tolerance. However, stateful microservices are used to retain session data and ensure consistency. However, data management complexities and interconnections can complicate scalability and result in a loss of failure tolerance. Here we must be very discriminating in picking among them based on application requirements and trade-offs while using them.

As mentioned, stateless microservices have a high degree of scalability, allowing seamless horizontal expansion to meet increasing demand while maintaining excellent responsiveness in dynamic resource settings. Stateful microservices, on the other hand, although providing persistent data benefits, require careful scaling considerations due to the complexity of maintaining a synchronized state across instances where these attempts were addressed in our earlier research [2,8,45–47].

5.5 Serverless architecture

Serverless computing transforms computing by abstracting server management from developers. In this architecture, code execution is event-driven and scales automatically based on demand. This paradigm is facilitated by services such as AWS Lambda and Azure Functions, which charge only for real code execution time. Developers are freed from infrastructure issues by focusing only on writing code.

We improve agility, decrease operational overhead, and ensure cost effectiveness by utilizing serverless architecture. Event triggers, such as HTTP requests or file uploads, start functions invisibly. Serverless computing, despite its name, is based on the underlying servers, but

abstracts their management, easing the creation and deployment of scalable and cost-effective applications [48].

5.5.1 Serverless microservices

Serverless microservices are usually cloud-based services that perform a highly specific task within the application. Tasks vary depending on the application. Tasks are executed in response to HTTP requests, database updates, at a certain period (such as batch jobs), or at any other event that could trigger a service.

5.5.2 Serverless architecture

Serverless architecture is a technology that allows developers to build and run applications without having to worry about the upkeep of the infrastructure or the administration of the cluster server. Cloud providers manage the servers that run their applications and databases in serverless architectures. Teams that lack the time, experience, or resources to administer servers can outsource the task to cloud providers such as AWS, Google, and Azure [48].

5.6 Communication

As one of the most important aspects of microservice design, communication between microservices must be quick and efficient. Effective communication across microservices is crucial for complex systems to run smoothly. A well-planned communication strategy promotes coherence and responsiveness, allowing microservices to communicate data effortlessly and coordinate activities.

The communication protocols used, such as RESTful APIs or message queues, have a significant impact on system performance. Real-time updates and efficient data flow among microservices promote agility, scalability, and dependability. However, issues such as latency and potential service outages require sophisticated error handling and monitoring techniques. Adroit microservices communication is critical to orchestrating coherent and high-performance distributed systems at the intersection of reliability and efficiency.

5.6.1 Synchronous architecture (Service-to-service)

In the service-to-service paradigm, the interaction between microservices is largely composed of HTTP requests such as GET, POST, and PUT. If one microservice requires something from another, it sends a GET request to that microservice, which returns with the appropriate response, as shown in Figure 1.

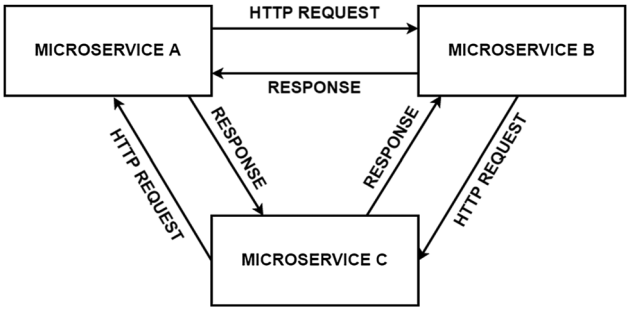

Figure 1: Example or synchronous service-to-service communication. Source: the study of Gordesli and Varol [49].

5.6.2 Asynchronous architecture

Microservices with an asynchronous design do not communicate with each other but instead with the Composer API [49]. In Figure 2, this can be seen in greater detail how the communication itself is realized.

5.6.3 Service mesh

Service mesh is a method to integrate the communication of a large number of autonomous services into a working application. Requests in the service mesh are typically routed through proxies, which capture all aspects of service-to-service interactions within the infrastructure (Figure 3). The figure depicts the procedure before and after the installation of service mesh, with the left side showing the cluster and the environment itself, where traffic balancing and exposing only

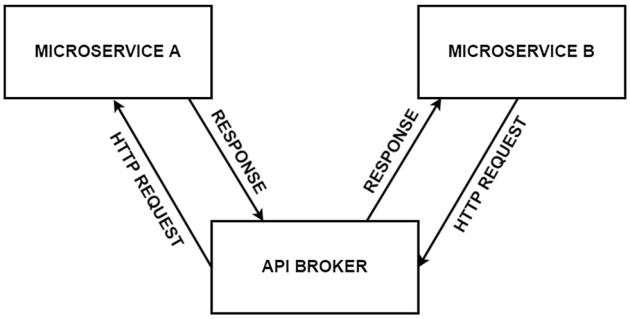

Figure 2: Example of Asynchronous architecture. Source: the study of Gordesli and Varol [49].

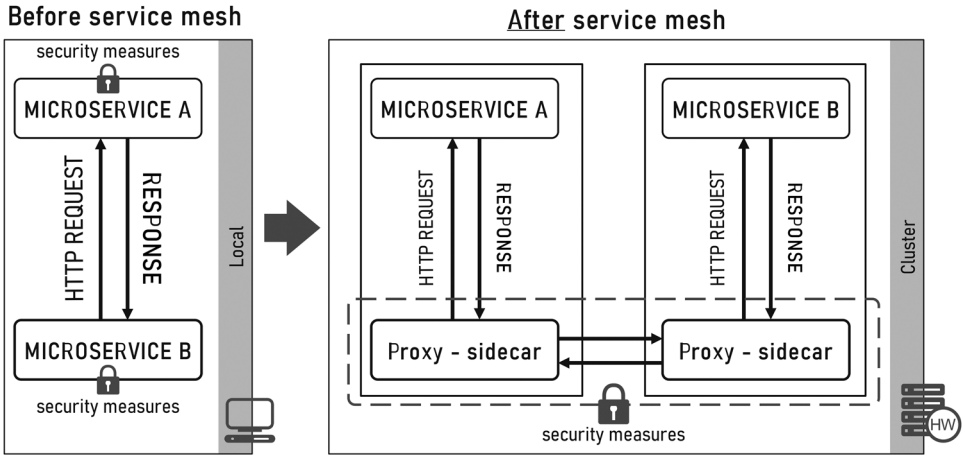

Figure 3: Example of service mesh – before and after deployment within the cluster using proxy sidecar. Source: author's contribution.

public ports are enabled. While the internal ports within the cluster are protected, the user does not have direct access to all services.

A "sidecar" is a proxy that operates alongside rather than within each microservice. The sidecar is in charge of all communication and security measures. Without a service mesh, it is impossible to isolate faults or bottlenecks in a large and intricate infrastructure [50].

6 Containerizations

A container is a code contained exclusively by the operating system, libraries, and dependencies required for the program to run. The containerization code results in an executable file that we can run on any machine. Containers are similar to VMs. Specific differences can be seen in Figure 4 where we see a comparative presentation of their

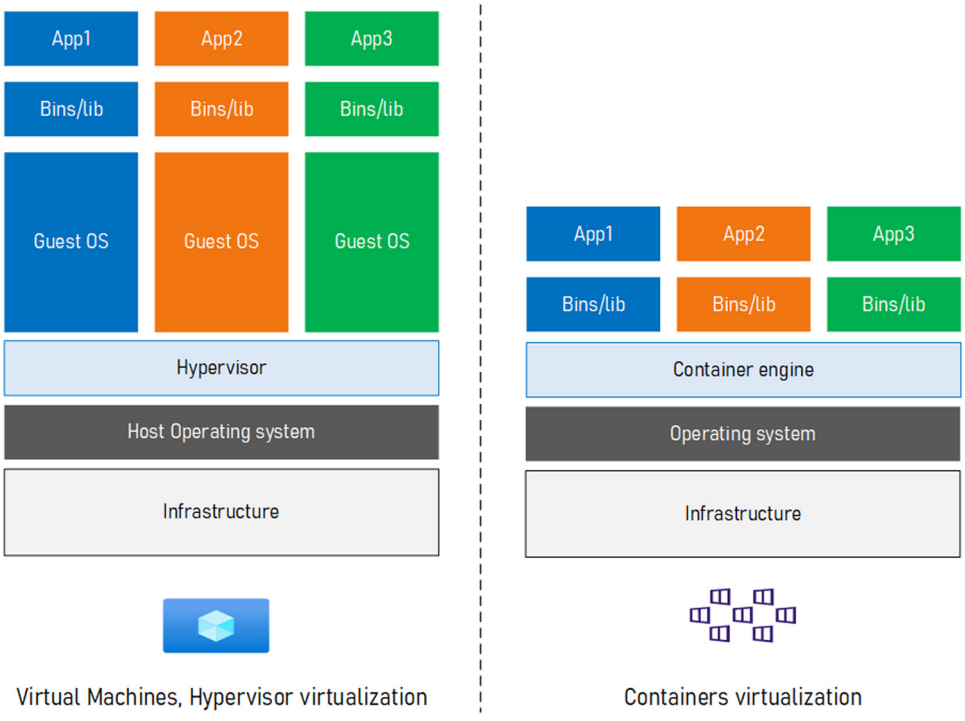

Figure 4: Side by side view of VMs vs containers. Source: Adopted to improve the quality [53].

differences in the architecture they use during implementation. However, they use fewer resources and are faster. Previously, moving code to other computers was significantly more difficult. Containers have solved this difficulty ("written once, run anywhere") [51]. Docker is the leading containerization platform, widely recognized and used across sectors. Its value lies in accelerating application deployment, providing consistent environments across several systems, and facilitating scaling. Understanding Docker's popularity is critical for businesses looking for efficient software development, deployment, and management solutions in today's fast-paced digital environment [52].

6.1 Docker image

A Docker image is a package that contains all the code, libraries, dependencies, and other components used to run an application. A file called DockerFile is used to construct a base image, since it provides all of the relevant information about the image that will be built.

Because images are immutable and easy to share, once you produce one, you can share it with others, who will receive the identical program when they execute it. Docker images are, in some ways, templates for running apps or adding extra image layers. Docker images can be pulled and submitted to the Docker hub, which serves as a public image repository [54].

7 Analysis of CI in agile deployment

CI/CD offers value to Agile development approaches by automating building, testing, and deployment. Although all of these phases can be performed manually, the primary benefit of this strategy is automation, which reduces delivery time [47,55,56]. Now, we can obtain error isolation much more easily by using this solution. The basic purpose is to increase productivity and discover problems before they are deployed to production.

7.1 Pipeline

The pipeline displays a succession of specified actions before releasing the code. Depending on the repository platform utilized, the format in which the pipeline is written may differ. The most well-known platforms that provide CI/CD pipelines are GitLab, Github, and Jenkins. The pipelines are built in YAML, a human-readable data serialization language [57].

7.1.1 Jobs

Jobs are an essential component of the pipeline because they describe tasks that consist of a series of commands that the pipeline must execute, such as compiling or testing code. If all tasks are completed in a stage, the pipeline moves on to the next stage.

7.1.2 Stages

The stage consists of a set of tasks. Stages are pipeline steps that can be defined by the developer, such as build, lint, test, and deploy. To streamline development and deployment, the pipeline steps follow a defined sequence throughout various stages. The first step is to define the project's goals and requirements. Task allocation and resource coordination are included in planning. Coding entails the creation of software, which is then tested to find and correct faults. Individual code components are consolidated into a unified system through integration.

The program is deployed for user access, while monitoring incorporates real-time performance and issue detection. Feedback loops help shape future advancements. Because each stage is interconnected, a methodical evolution from conception to execution is ensured, supporting agile development and optimizing the efficiency and reliability of pipeline steps in the software development lifecycle. An example of a development of CI/CD pipeline is shown in Figure 5, where the sequence of steps that is realized depending on the calling of a certain scene (build, test, deploy, and production) can be seen.

7.2 Container orchestration

Managing hundreds of containers and communication between them is rather difficult without a Container orchestration tool. "Container orchestration automates the deployment, management, scaling, and networking of containers." Container orchestration gives the user greater control over individual containers, resources, configurations, and life cycles. The most well-known containers orchestration tools are Kubernetes, Openshift, and Docker Compose [58].

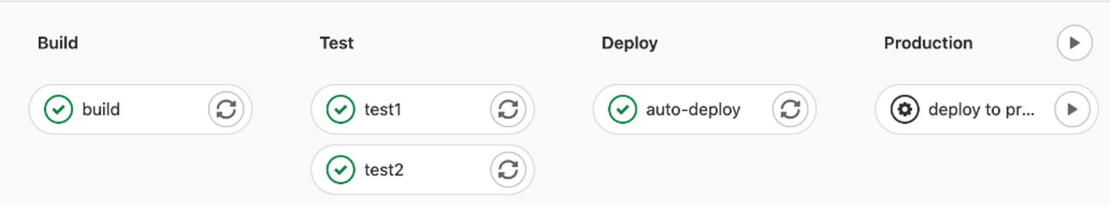

Figure 5: Example of simple GitLab CI/CD pipeline. Source: the study of Malviya and Dwivedi [58].

7.3 Kubernetes

Kubernetes, an open-source container orchestration mechanism, includes the ability to deploy, manage, and scale multiple containers. The key advantages of Kubernetes automation include dependability, autonomous deployment, load balancing, and storage orchestration. Pods emerge as critical components of Kubernetes orchestration, encapsulating containers, and enabling efficient deployment and scalability. This declarative approach offers seamless configuration management, allowing large systems to be orchestrated with services and labels. This also allows for dynamic load balancing and simple identification of connected components.

Data durability is ensured by persistent storage, which is accomplished through volumes. The self-healing capabilities of Kubernetes immediately replace broken containers, ensuring high availability. CRDs improve Kubernetes capability by allowing the introduction of bespoke resources where this broad ecosystem places Kubernetes as the cornerstone in modern containerized application deployment, facilitating scalability, resilience, and maintenance [59].

7.4 Kubernetes cluster

A Kubernetes cluster combines computing resources by coordinating numerous nodes to run containerized applications efficiently. In a distributed architecture, the master node directs the orchestration, while the worker nodes execute tasks, ensuring scalability, fault tolerance, and optimal performance, providing the backbone of resilient environments built for cloud infrastructures. The cluster is made up of Worker Nodes, Master Nodes, and a Control plane, which has six major components: Kubelet, Kube-proxy, control-manager, etcd, API Server, and scheduler.

7.5 Namespace

Kubernetes provides a namespace, which is a place to separate a group of resources within a cluster. The Namespaces are intended for use with a large number of clients via a cluster. They also serve as a mechanism to split cluster resources among various users, with four predefined namespaces in a cluster: default, Kube-system, Kube-public, and Kube-node-lease.

For items with no defined namespace, the default namespace is used. The Kube-system is used for Kubernetes-created objects. The Kube-public class represents things that are accessible to all users. Kube-node-lease is used to lease things.

7.6 Worker node

The worker node itself can be defined as a node that has containers that have been built and are currently operating. Every worker node has its own Kube proxy, container runtime engine, and Kubelet. Kubelet serves as a communication agent for the Kubernetes API Server. Kube-proxy facilitates communication between containers created on separate working nodes, as well as the provision of an IP address for the container and load balancing. The container runtime engine is responsible for the initialization of deployed containers.

7.7 Master node

A Kubernetes cluster is managed by a master node, which is made up of several components: etcd, Kube-controller-manager, Kube-API Server, and Kube-scheduler. Kubernetes supports several master nodes, ensuring load balancing, and when one dies, another takes its place.

Containers are scheduled using Kube-scheduler, while nodes and replicas are managed via Kube-controller-manager. Etcd maintains and monitors information about other nodes. The Kube-API Server authenticates the clients and allows them to communicate with the Kubernetes cluster.

7.8 Helm charts

Helm is a Kubernetes package management that comprises preconfigured Kubernetes resources such as Secrets,

Services, Configmaps, and so on, as well as a package description (Chart.yaml). It enables repeatable build of Kubernetes applications and simplifies installation and administration of Kubernetes applications via Helm charts. Helm charts, a key component of Kubernetes package management, enable smooth deployment rollbacks. Helm's versioned releases enable easy reversal to previous states, ensuring robustness in the event of problems.

When problems develop after deployment, Helm's rollback command can be used to quickly return the application to the last known stable version, lowering risks and minimizing downtime. This versioning system, together with Helm's easy management, enables users to navigate and correct deployment difficulties with agility, offering a critical safety net in the changing Kubernetes application deployment landscape [60].

Users have more confidence in managing applications, since the rollback feature provides a robust mechanism for quickly resolving errors, boosting resilience and efficiency in the volatile landscape of containerized environments. The final result is a more secure, adaptive, and user-friendly experience to coordinate Kubernetes installations with Helm charts.

8 Results

Due to the end product and the capability of direct application, it is necessary to use appropriate hardware resources, which are at the heart of computer systems. To deploy Helm charts, perform CI/CD, and manage Kubernetes successfully, we must prioritize appropriate hardware resources. Helm charts, CI/CD artifacts, and Kubernetes deployments all require adequate storage, whether HDD or SSD, as well as a robust network and a consistent power source to ensure continuous communication and operation. In the case of graphics processing unit (GPU) resources, we may boost our performance for GPU-accelerated tasks in AI training and standard compliance when they are available. Multicore systems and effective cooling should further enhance processing efficiency. To build a highly performant and resilient development and deployment environment, we must align these resources with the deployment scale.

8.1 Prerequisites

To ensure a smooth sailing experience, certain prerequisites must be met before launching Helm chart-based deployment and rollback in Kubernetes. This requires the use of a fully operational Kubernetes cluster with Kubectl installed for command-line interactions. Additionally, Helm, the package management, must be properly installed and initialized within the cluster. Understanding the application's Helm chart structure and keeping version control on chart releases are critical. Access to the required application chart repository and knowledge of the relevant chart values help ensure successful deployments. Furthermore, familiarity with Helm commands, particularly helm installation and helm rollback, is necessary for effective chart management in Kubernetes deployments.

For our prerequisites, we investigated the use of the Windows OS environment as part of the criteria for the execution and testing of our laboratory environment. Where Windows container orchestration requires specific conditions for a seamless deployment, including appropriate modern CPU resources for speedy container operation and adequate RAM for properly managing concurrent workloads. Storage capacity is required to store container images and associated data, whether on an HDD or on an SSD. Strong network design facilitates communication between containers and orchestrators.

Compatibility with Windows Server versions that support containerization is required, as well as Docker Engine for Windows. Depending on desire and workload, Kubernetes or Docker Swarm can be used as orchestrators. These requirements must be addressed to ensure a smooth and effective orchestration process for Windows containers in a variety of computing environments.

According to our knowledge and the latest available information, there are the following requirements for Windows container orchestration that can be divided into the following categories:

(1) Operating system requirements
(2) Virtualized container hosts
(3) Memory requirements

As a result, we must meet the following basic requirements:

(1) To have a CPU processor that supports AMD or intel virtualization (x86/x64).
(2) A host computer with at least one CPU and two processor cores is required for the use of VMs.
(3) 4 GB of RAM on the host machine for container virtualization (Windows Server 2022 or Windows 10/11).

8.2 Implementation

We have decided to use a Minikube to deploy and test Kubernetes locally to test microservices. The goal is to

have a Kubernetes cluster running locally and to allow users to automatically deploy and manage their applications within our local cluster.

8.3 Web App

The web application will run as a pod inside the Minikube cluster, where other users will add the SSH key, as well as all the necessary parameters, configurations, and secrets, to the Git repository. The repository must include the application's source code and a DockerFile.

8.4 Microservice for handling login

Because namespaces can include sensitive information that must be kept hidden. Users who want to deploy their application or control specific namespaces must be logged in. The log-in microservice interfaces with a database to verify that no unauthorized user can install, inspect, or manage unassigned namespaces and pods within. Every user will be able to claim three namespaces.

8.5 Microservice for building image from Git repository

Microservice obtains the source code and DockerFile from a Git repository, builds it, performs any tests, and gives the Docker image to the Microservice, which talks to minikube. An example of a DockerFile that can be downloaded from the link: https://bit.ly/3eZoEBl

```
FROM node:14-alpine
WORKDIR /usr/src/app
COPY ["package.json",
"package-lock.json", "./"]
RUN npm install
COPY . .EXPOSE 3001
RUN chown -R node /usr/src/app
USER node
CMD ["npm", "start"]
```

8.6 Microservice for communicating with Minikube

Following the acquisition of a Docker image, the microservice interacts with kubectl to create a new namespace within a minikube cluster and constructs a node that runs the container generated by the Docker image in the newly created namespace.

8.7 Microservice for generating pipeline and helm charts

Once the deployed pod is operational, the microservice generates a pipeline on the Git repository (customers can select between GitLab and Github) and helm charts for them to download. The downloaded files will be saved to the user's Git repository for automatic deployment the next time a branch is merged into "main." The user can continue to edit the files provided for application reasons.

8.8 Databases

Databases in containerized environments work within orchestrated systems such as Kubernetes. Containerized databases contain application components, making them more portable. Data durability is ensured by persistent storage solutions such as Kubernetes persistent volumes. Orchestration solutions easily manage database setup, scaling, and connection.

Container orchestration solutions, such as Kubernetes, can be used by stateful applications, such as databases, to provide consistent storage and network configurations. Operators and Helm charts facilitate database deployment, while containerization concepts increase scalability and resource utilization. Database containers interact within clusters to promote dynamic, robust, and scalable data management in modern cloud-based applications. In Figure 6, the database contains each user's login credentials and namespace names. The database may be local or distributed across numerous containers, depending on the requirements.

8.9 Compliance operations and new GRCopOps method

Implementing a conventional CI/CD pipeline for code compliance involves a series of procedures to ensure that the code adheres to defined standards, passes tests, and can be delivered reliably. Investors should bear in mind that developing a code compliance pipeline is a complicated process that must include safety, dependability, and compliance criteria. It is vital to work closely with domain experts, follow industry best practices, and tailor the pipeline to the needs of your specific project.

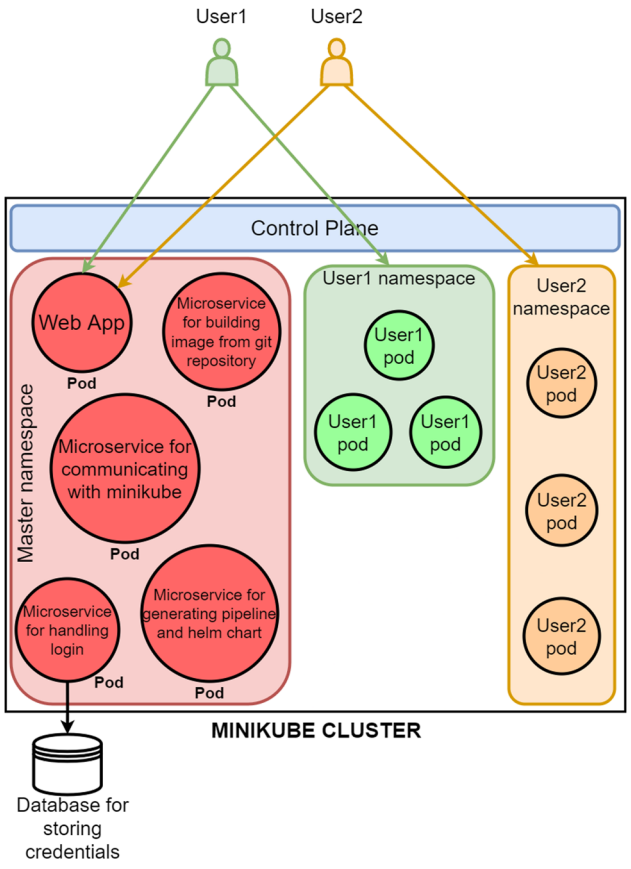

Figure 6: Visual representation of the implementation. Source: the study of Larrucea et al. [37].

Every business needs to conduct a thorough review of where it is now in its IT modernization journey, as well as where it intends to go. Companies in the automotive industry can then build a roadmap with a timeline. Those in charge of implementation must keep strong and flexible thinking as the final aim as well as the continuing mentality. To stay current with world changes, every modernization strategy should evolve as time goes on. In the future, we believe that repository compliance operations that incorporate collaboration are the best attitude for this.

One of the results of this research was the development of a way to achieve compliance with standardized software with certain standards through the design of a new approach and acronym. The latest proposal (Figure 7) includes new proposals that are part of the new GRCopOps method. In the figure, we can see the division into static and dynamic parts of the implementation. Depending on the action that was launched from the repository, an alternative option would be for the user to create a workflow-specific file within the user repository. Then, the client repository would contain yml (YAML) code as well as limited Python code execution capabilities.

Figures 7 and 8 contain a proposal for processing dynamic and static requests within the process itself, which could be completed within different cluster environments. Therefore, it relies on the previously created infrastructure. GRCopOps is a new approach and acronym. In GRCopOps, a set of steps represents the direction of

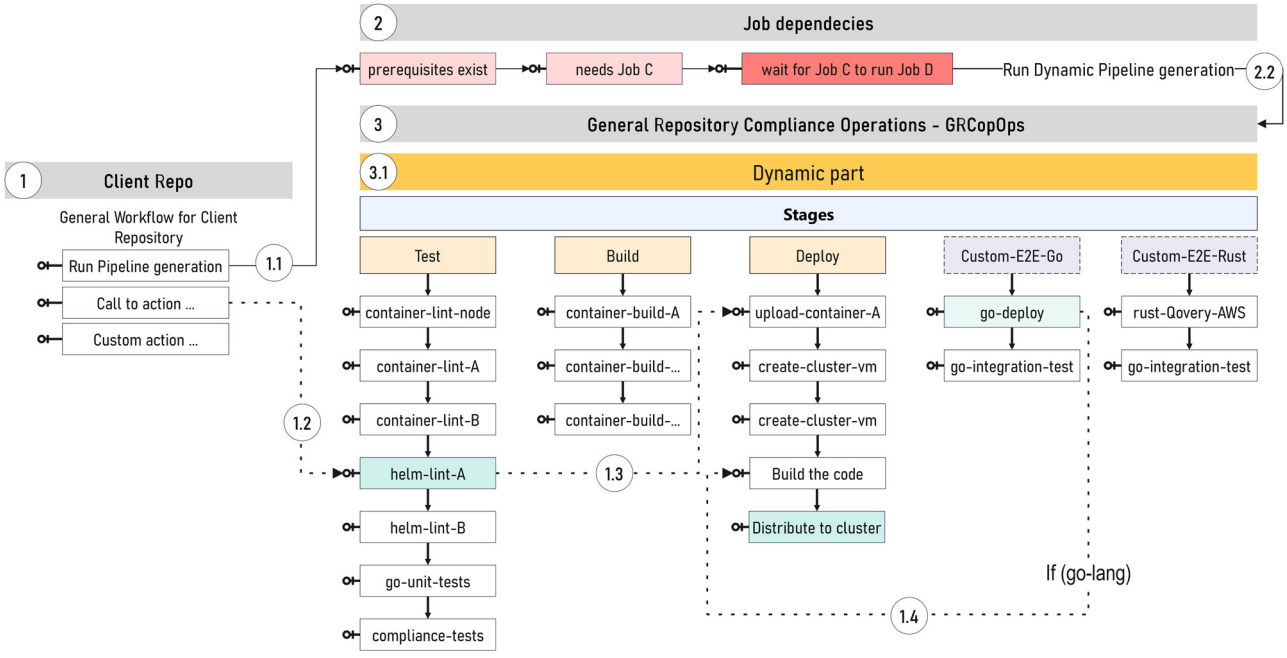

Figure 7: Dynamic part – GRCopOps. Source: author's contribution.

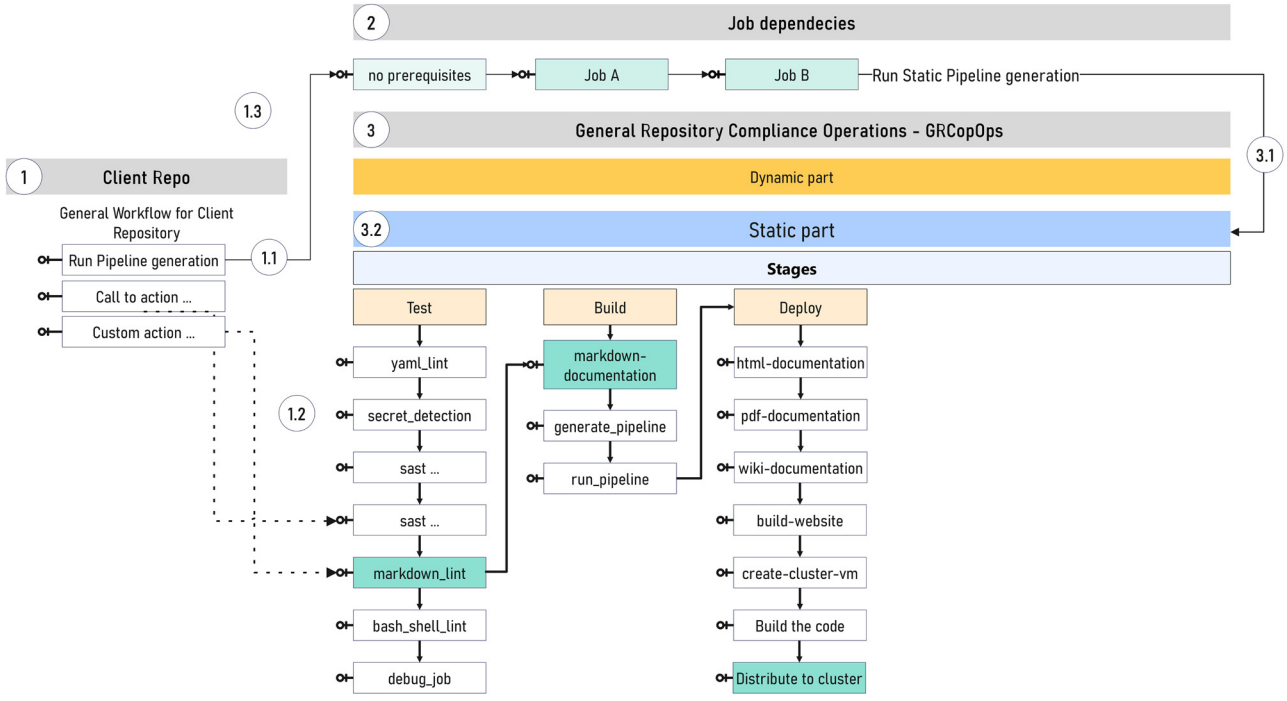

Figure 8: Static part – GRCopOps. Source: author's contribution.

movement based on the source and execution stage. The number of dots indicated in the figures represents the realization processes that begin with a certain beginning point.

The initial conceptual characteristics of GRCopOps are as follows:

- Design of the general repository's conceptual architecture and method of software team communication related to the knowledge management life cycle.
- Connecting the client to the general repository previously defined by the automotive company.
- Conceptual scaling of resources depending on the need for testing different standards, with the possibility of dynamic creation of microservices and other artifacts for these needs.

In both cases, the steps for the static and dynamic part (Figures 7 and 8) can generally be described and shown as follows:

- The reading process occurs from left to right, with the first step on the left being a repository that launches a specific activity (1 Client Repo). The launch process is described in the basic workflow for the client repository and the YAML file, which can launch a user-defined scene or activity. The file's content is determined by its content and the method of execution (Helm or YAML packages, packages.yaml). The document definition should adhere to the official specifications stated in the official public package settings documentation, spack.readthedocs.io. This approach enables certain possibilities of scaling and starting the process of early code compliance.
- Step 1 executes and invokes the prerequisites required by certain activities (1.1 Run pipeline creation). Through this approach, the car brand can repeat the defined testing procedure. In addition, this compliance process can be defined by the regulatory body or by the company's software engineer.
- Step 2 involves implementing the tasks and determining what may be waiting for the next task, after which it is implemented. This phase allows for either Dynamic or Static execution mode, depending on the action initiated within the CI/CD pipeline.
- Step 3 involves implementing traditional tests and tasks in three phases: test, build, and deployed or special stages established by the developer and manufacturer of a specific brand of automobile. The results can be extended and linked to certain dashboards used in the automotive sector.

The fundamental aim behind this term is to achieve synergy and accelerate the process of software standardization. However, employing its maximum capacity should be the best manner for the organization's strength and profitability, not because many standards organizations need it. Despite the

lack of confidence, this seems puzzling today. The core notion, on the other hand, represents the future of software compliance and can transform how technology evolves to adapt to new missions, goals, and end-user needs. As with other broad organizational enhancements, future software factory adoption should begin with an assessment.

To accelerate software code standardization, additional research and detailed information from various companies is required. Depending on the software project or component, integration testing, and deployment to autonomous vehicle platforms could require additional steps in the workflow. The artifact generation workflow in this example is not covered. The reason is that each project has different requirements and needs certain testing methods during dynamic implementation. The starting idea is that the developer has the first basic idea of how to start with understanding the ecosystem in which he should work for the automotive industry.

We may need to package the code into deployable artifacts for a specific platform with performance monitoring, again depending on the project. In this case, we will need to integrate with the relevant tools and platforms for autonomous vehicle deployment and monitoring in a hybrid cloud/edge environment. This may require the definition of additional workflow stages. The idea is that the developer's code cannot integrate new features unless it passes the workflow criteria in our GRCopOps. This will require adaptation of our example to the requirements of the multiple autonomous vehicle project over time.

For our client or developer to implement GRCopOps (Figures 7 and 8), he would need to have this example in his Git repository with the following steps:

(1) Create the **.git/workflows/** directory if it does not exist.
(2) Create in **.git/workflows/** a new file named <**Write_type_of_workflow**>**_workflow.yml** in that directory.
(3) Copy and paste our example YAML configuration into **.git/workflows/general_workflow.yml.**
(4) Commit and push the changes to the local or enterprise Git repository.
(5) The defined workflow will automatically trigger (start) on every push to the main branch, where the developer can adjust the trigger conditions as needed.
(6) The user should always refer to the latest GRCopOps documentation and adapt the pipeline to his project's requirements and tools.

In the context of autonomous vehicles, creating a conventional CI/CD pipeline for code compliance involves a series of steps to ensure that the code meets defined standards, passes tests, and can be supplied reliably. For this reason, the solution architect and the regulatory body specified the process as a template to be used in the future inside of GRCopOps.

9 Limitations of the study

Each research has limitations, in the case of our research they can be presented as scope limitations, resource constraints, educational access, etc. To obtain broadly relevant findings, we keep the following constraints in mind:

(1) Scope limitation: Research may be constrained by the specified scope, which focuses on microservices architectures, CI/CD approaches, and specialized applications such as the automotive industry. This constraint may limit the generalizability of the findings to other industries or upcoming technologies that have not been formally examined.
(2) Temporal constraints: Given the rapidly evolving nature of technology, there could be temporal constraints in which research findings may become outdated or less relevant over time. The pace of technological advancements may outstrip the research's ability to encompass the latest developments in microservices, CI/CD, and related fields covering mainframe architecture.
(3) Resource constraints: Constraints may arise due to a lack of resources, both financially and in terms of access to specific technologies or platforms. A shortage of resources may impede comprehensive real-world implementations and large-scale investigations, compromising the depth and breadth of the research. One of our key constraints is the lack of a funding grant, which would allow us to do more thorough testing and application development within the edge cloud environment.
(4) Generalizability challenges: It can be challenging to apply the findings to a broader context or to different cultural and organizational situations. Because organizational culture, geographical regulations, and technical infrastructures differ greatly, the generalizability of certain solutions is limited because our research considered perspectives from the perspective of students, as well as the environment/country in which the research was conducted.
(5) Educational access: Depending on the area, the practical applicability of the research for students may be limited by the availability of appropriate educational resources, tools, and technologies. It is possible that not all students or educational institutions have the same access to the settings required for hands-on learning in microservices, Kubernetes, and comparable technologies.

As a result, our research may be difficult to use in some circumstances, but it can be solved if appropriate research grants are obtained.

(6) External environmental factors: External factors such as economic, legal, political, or global events can have an impact on the industry under study, potentially affecting the contextual validity of the research. These factors are beyond our control as researchers, but may have an impact on the practical usefulness of the findings. One of these issues could be the establishment of new regulations about the deployment of Western intelligence or the limitation of particular technology or computer chips.

Understanding and embracing these limits is crucial for maintaining the research's integrity and providing a realistic framework for understanding and applying its findings in an ever-changing technical and business world. We anticipate that by gradually removing these constraints, we will be able to conduct even more robust research and get better results according to particular measures in the future.

10 Discussion

Within the dynamic arena of software development, the planned sequence of pipeline phases orchestrates a debate and an interesting dance of innovation and correctness. This is often referred to as choreography because it begins with the definition of the goal and proceeds through planning, coding, and testing, weaving intricate patterns of teamwork. Integration, like a symphony integrating instruments, harmonizes dissimilar components.

Monitoring functions as a diligent conductor, assuring precise execution, while deployment reveals great performance. This choreographed ballet captures the essence of agile development, in which each step is critical, promising a fascinating show of efficiency and innovation on the vast stage of software evolution.

As a result, depending on the study questions previously stated, we were able to arrive at specific answers and conclusions. So, the findings based on the above are as follows:

(1) The research aimed to develop groundbreaking methodologies that guarantee software compliance. We found that building such procedures within a microservices architecture and GRCopOps requires a thorough and rigorous approach. The first step is to know how to examine regulatory requirements and industry standards. Subsequently, modular compliance tests and automated testing must be introduced in microservices. Containerization with Docker and Kubernetes must be embraced for a streamlined deployment process. Continuous compliance monitoring and making changes based on feedback are fundamental. By integrating these elements, software compliance can be achieved effectively and efficiently.

(2) We learned from this question that it dives into the practical ramifications of the research, attempting to extract significant insights from a detailed case study. Its goal is to bridge the gap between academic principles and real-world applications, with a particular emphasis on industry.

(3) This study topic was successful in addressing the educational side, with a focus on improving access to microservices and cloud computing environments. We discovered how we can investigate techniques for facilitating hands-on learning, taking into account the problems faced by students and experts, and focusing on the practical aspects of cluster administration and standard compliance incorporated into our approach (GRCopOps).

10.1 Open questions

Open conversations and questions in software development should cover a wide range of topics. Improving teamwork during the planning phase leads to more innovative results. Optimization solutions for bug resolution during testing are key factors in the continuation and expansion of existing and future research. The integration dance seeks to achieve a seamless blending of diverse components in a code symphony. The deployment strategies for a user-friendly reveal remain a priority.

During the observation stage, we tried to develop informative approaches for future real-time performance evaluation. This introspection within feedback loops should lead to the evolution of methodology and collaboration, and these open questions should drive the industry forward. This future topic should explore the synergies between microservices and cutting-edge technologies, highlighting the importance of ethical integration research and the potential benefits of technologies such as edge computing and artificial intelligence. These open questions will lead future research efforts that will focus on the adaptability of CI/CD methodologies, the sector-agnostic difficulties of microservice adoption, and the ethical integration of new technologies.

Due to their distributed nature, microservices systems require the highest level of security. Each microservice is a potential entry point for security vulnerabilities that require effective safeguards across several layers. Authentication and authorization mechanisms are crucial to managing access to

sensitive resources. Implementing sophisticated protocols such as OAuth and JWT, along with fine-grained access control techniques, enhances security. However, they can also be a source of failure, necessitating regular monitoring and supervision.

Future container security measures will include image scanning and runtime protection to ensure the integrity and security of containerized microservices. Overall, a comprehensive security solution will be required to protect microservice architectures from attacks and vulnerabilities.

Accordingly, we can pose the following open questions that can be addressed in further research:

(1) How can CI/CD strategies respond to changing software compliance standards while staying in step with changing industry rules and technological advancements?
(2) What are the lasting consequences and industry-agnostic problems of widespread adoption of microservices and how can these challenges be efficiently addressed for long-term success?
(3) How can emerging technologies such as edge computing and artificial intelligence be easily integrated with microservices designs to improve system performance, scalability, and ethical issues in a variety of applications?

Addressing these concerns will help ensure that microservice architectures continue to evolve, ensuring their relevance, compliance, and ethical considerations in an ever-changing technological context. These challenges should help future efforts to solve the challenge of developing adaptive CI/CD strategies that are compatible with changing regulatory landscapes and technological advances while also supporting resilience and agility in software distribution. The purpose of this research was to uncover universal difficulties and successful strategies for long-term deployment and success in several sectors.

11 Conclusion

To our knowledge, we investigated and discussed the most crucial features of cloud computing in our study. We decided to use this study as a playground to help new aspiring DevOps engineers or Software Developers gain a better understanding of how the mentioned technologies work and how to implement them in their projects, assuming that we were not the only ones who struggled to understand microservices and their management. We were able to demonstrate that research of architectures, CI/CD methodologies, and their integration with evolving technologies provide useful insights for a wide range of audiences, including young professionals and students entering the dynamic world of technology.

As this industry develops, the highlighted challenges and adaptive strategies serve as a guide for experts in navigating challenging terrain. Furthermore, the emphasis on hands-on learning, particularly in Kubernetes cluster administration and compliance-related practices, matches the educational goals of students who want to work in the industry. One of the unique elements is the need for young employees to keep up with evolving software compliance standards and embrace CI/CD methodologies to remain flexible in their development operations.

Real-world applications and case studies, particularly in the industry, provide realistic examples that are appealing to both students and professionals, bridging the gap between academic principles and actual realities. One of the conclusions and verifications is the successful adoption of microservice architectures with CI/CD techniques, which is particularly obvious in real-world automotive applications. This underscores the relevance of our findings and the practical benefits of using these strategies in a variety of corporate settings. Because one of the major features must be a focus on hands-on learning, particularly in Kubernetes cluster administration and compliance-related practices, students will enter the dynamic workforce. Young professionals, as torchbearers of innovation, are well-positioned to effect positive change by embracing and developing the suggested strategies.

In a nutshell, this research lays a foundation for future research, showing the practical application of microservices and CI/CD approaches, and expects exciting participation from the younger generation in influencing the future trajectory of technology and software development. The journey continues as we develop our collective understanding and application of these important components in the ever-changing technology context.

As we look ahead, our research promotes a collaborative approach in which students and professionals can engage in continuous learning, adapt to technological advances, and contribute to the continued refinement of best practices. We enable the next generation of technology enthusiasts not only to understand the complexity of platforms and CI/CD, but also to actively shape the future of software development and deployment by building a learning environment that combines theoretical knowledge with hands-on experience.

In essence, this research serves as a guide for everyone interested in the IT industry as they navigate the ever-changing terrain of technology with awareness, adaptability, and innovation. Future research should focus on solutions that enable continued compliance with changing industry legislation and technology improvements, building on the foundations laid out in this study. We may be able to

inspire new experiments on adaptive CI/CD solutions that address the growing diversity of software compliance standards in future research.

Acknowledgement: The present investigation is an expansion and continuation of a previously published conference article [45] that was published as part of the 2022 IEEE 16th International Scientific Conference on Informatics (Informatics) https://informatics.kpi.fei.tuke.sk/.

Funding information: The work reported here was supported by the Slovak national project Increasing Slovakia's Resilience Against Hybrid Threats by Strengthening Public Administration Capacities (Zvýšenie odolnosti Slovenska voči hybridným hrozbám pomocou posilnenia kapacít verejnej správy) (ITMS code: 314011CDW7), and the Operational Programme Integrated Infrastructure for the project: Support of Research Activities of Excellence Laboratories STU in Bratislava (ITMS code: 313021BXZ1), co-funded by the European Regional Development Fund (ERDF), the Operational Programme Integrated Infrastructure for the project: Research in the SANET network, and possibilities of its further use and development (ITMS code: 313011W988), Advancing University Capacity and Competence in Research, Development, and Innovation (ACCORD) (ITMS code 313021X329), co-funded by the ERDF, rurALLURE project – European Union's Horizon 2020 Research and Innovation program under grant agreement number: 101004887 H2020-SC6-TRANSFORMATIONS-2018-2019-2020/H2020-SC6-TRANSFORMATIONS-2020, the Slovak Research and Development Agency under the contract No. APVV-15-0508, the Erasmus+ ICM 2023 program under the grant agreement 2023 No. 2023-1-SK01-KA171-HED-000148295, Model-based explication support for personalized education (Podpora personalizovanÃĺho vzdelávania explikovaná modelom) – KEGA (014STU-4/2024), and the Operational Program Integrated Infrastructure for the project: National infrastructure for supporting technology transfer in Slovakia II – NITT SK II, co-financed by the European Regional Development Fund.

Author contributions: Conceptualization, investigation, methodology, proofreading, project administration, visualization, and writing – original draft and editing: P.D.; The author has read and agreed to the published version of the manuscript.

Conflict of interest: The author declares that there is no conflict(s) of interest.

Data availability statement: Not applicable.

References

[1] S. Siddique, M. Naveed, A. Ali, I. Keshta, M. I. Satti, A. Irshad, et al., "An effective framework to improve the managerial activities in global software development," *Nonlinear Engineering*, vol. 12, no. 1, p. 4. Jan. 2023.

[2] T. Golis, P. Dakić, and V. Vranić, "Automatic deployment to kubernetes cluster by applying a new learning tool and learning processes," in: *SQAMIA 2023 Software Quality Analysis, Monitoring, Improvement, and Applications*, vol. 1613, 2023, p. 0073. https://ceur-ws.org/Vol-3588/p16.pdf.

[3] P. Dakić and V. Todorovć, "Isplativost i energetska efikasnost autonomnih vozila u eu," *FBIM Transactions*, vol. 9 no. 2, p. 10, 2021. https://www.meste.org/ojs/index.php/fbim/article/view/1198.

[4] M. Kročka, P. Dakić, and V. Vranić, "Extending parking occupancy detection model for night lighting and snowy weather conditions," in: *2022 IEEE Zooming Innovation in Consumer Technologies Conference (ZINC)*, 2022, pp. 203–208.

[5] R. Szarka, P. Dakić, and V. Vranić, "Cost-effective real-time parking space occupancy detection system," in: *2022 12th International Conference on Advanced Computer Information Technologies (ACIT)*, IEEE, Sep 2022.

[6] M. Kročka, P. Dakić, and V. Vranić, "Automatic license plate recognition using OpenCV," in: *2022 12th International Conference on Advanced Computer Information Technologies (ACIT)*, IEEE, Sep 2022.

[7] P. Dakić, V. Todorović, and V. Vranić, "Financial justification for using CI/CD and code analysis for software quality improvement in the automotive industry," in: *2022 IEEE Zooming Innovation in Consumer Technologies Conference (ZINC)*, 2022, pp. 149–154.

[8] A. Petričko, P. Dakić, and V. Vranić, "Comparison of visual occupancy detection approaches for parking lots and dedicated containerized REST-API server application," in: *Proceedings of the Ninth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, September 11–14, 2022, ser. CEUR Workshop Proceedings*, Z. Budimac, Ed., vol. 3237, CEUR-WS.org, 2022, http://ceur-ws.org/Vol-3237/paper-pet.pdf.

[9] A. M. G. Esperón, F. M. Pérez, J. V. B. Martínez, M. D. D. Dapena, and I. L. Fonseca, "Specifying requirements for modern software development: A test-oriented methodology," *International Journal of Software Engineering and Knowledge Engineering*, vol. 34, no. 01, pp. 27–48, Sep. 2023.

[10] P. Dakić, V. Todorović, and P. Biljana, "Investment reasons for using standards compliance in autonomous vehicles," *ESD Conference, Belgrade 75th International Scientific Conference on Economic and Social Development, ESD Conference Belgrade, 02–03 December, 2021 MB University, Teodora Drajzera 27, 11000 Belgrade, Serbia*, 2021, https://www.shorturl.at/diMRS.

[11] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 2114–2129, Sep. 2019.

[12] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in: *2018 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, Apr 2018.

[13] M. Artac, T. Borovsak, E. D. Nitto, M. Guerriero, D. Perez-Palacin, and D. A. Tamburri, "Infrastructure-as-code for data-intensive architectures: A model-driven development approach," in: *2018*

IEEE International Conference on Software Architecture (ICSA), IEEE, Apr 2018.

[14] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19, ACM, Apr. 2019.

[15] V. Podolskiy, M. Mayo, A. Koay, M. Gerndt, and P. Patros, "Maintaining SLOs of cloud-native applications via self-adaptive resource sharing," in: *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, IEEE, Jun. 2019.

[16] J. Liu, E. Braun, C. Düpmeier, P. Kuckertz, D. Ryberg, M. Robinius, et al., "Architectural concept and evaluation of a framework for the efficient automation of computational scientific workflows: An energy systems analysis example," *Applied Sciences*, vol. 9, no. 4, p. 728, Feb 2019.

[17] S. Apel, F. Hertrampf, and S. Späthe, "Toward a knowledge model focusing on microservices and cloud computing," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 13, Jun 2019.

[18] P. Dakić, A. Todosijević, and M. Pavlović, "The importance of business intelligence for business in marketing agency," *International Scientific Conference ERAZ 2016 Knowledge Based Sustainable*, 2016, značaj poslovne inteligencije za poslovanje marketinške agencije.

[19] M. Popović, M. Milosavljević, and P. Dakić, "Twitter data analytics in education using IBM infosphere biginsights," in: *Sinteza 2016 - International Scientific Conference on ICT and E-Business Related Research*, Singidunum University, 2016, pp. 74–80.

[20] T. Semerádová and P. Weinlich, "Reaching your customers using Facebook and google dynamic ads," in: *Research Anthology on Strategies for Using Social Media as a Service and Tool in Business*, IGI Global, 2021, pp. 582–599.

[21] A. Lovska, O. Fomin, V. Píštěk, and P. Kučera, "Dynamic load modelling within combined transport trains during transportation on a railway ferry," *Applied Sciences*, vol. 10, no. 16, p. 5710, Aug 2020.

[22] A. Saboor, A. K. Mahmood, A. H. Omar, M. F. Hassan, S. N. M. Shah, and A. Ahmadian, "Enabling rank-based distribution of microservices among containers for green cloud computing environment," *Peer-to-Peer Networking and Applications*, vol. 15, no. 1, pp. 77–91, Aug 2021.

[23] S. E. Popov, R. Y. Zamaraev, N. I. Yukina, O. L. Giniyatullina, L. S. Mikov, I. E. Kharlampenkov, et al., "Software for calculating deformations of the earth's surface using satellite radar data," *Programmnaya Ingeneria*, vol. 12, no. 5, pp. 246–259, Aug 2021.

[24] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, et al., "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '19, ACM, Aug. 2019.

[25] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAam: Guaranteeing SLAs for jobs in microservices execution frameworks," in: *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19, ACM, Mar. 2019.

[26] Z. Shah, U. Javed, M. Naeem, S. Zeadally, and W. Ejaz, "Mobile edge computing (MEC)-enabled UAV placement and computation efficiency maximization in disaster scenario," *IEEE Transactions on Vehicular Technology*, vol. 72, No. 10, pp. 13406–13416, 2023.

[27] S. Prayla Shyry, *Biometric-based three-tier microservice architecture for mitigating the Fraudulent behaviour*, Springer Nature, Singapore, Dec. 2019, pp. 399–404.

[28] N. Sänger and S. Abeck, "User authorization in microservice-based applications," *Software*, vol. 2, no. 3, pp. 400–426, Sep. 2023.

[29] A. Samanta and J. Tang, "Dyme: Dynamic microservice scheduling in edge computing enabled IoT," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6164–6174, Jul. 2020.

[30] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, et al., "Localizing failure root causes in a microservice through causality inference," in: *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, Jun. 2020.

[31] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, et al., "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, Feb. 2021.

[32] X. Wang, J. Li, Z. Ning, Q. Song, L. Guo, S. Guo, et al., "Wireless powered mobile edge computing networks: A survey," *ACM Computing Surveys*, vol. 55, no. 13, Art. No. 263, 2023.

[33] M. Reiss-Mirzaei, M. Ghobaei-Arani, and L. Esmaeili, "A review on the edge caching mechanisms in the mobile edge computing: A social-aware perspective," *Internet of Things*, vol. 22, p. 100690, 2023.

[34] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 939–951, Mar. 2021.

[35] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, et al., "Characterizing microservice dependency and performance: Alibaba trace analysis," in: *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21, ACM, Nov. 2021.

[36] M. Xu, L. Yang, Y. Wang, C. Gao, L. Wen, G. Xu, et al., "Practice of Alibaba cloud on elastic resource provisioning for large-scale microservices cluster," *Software: Practice and Experience*, vol. 54, no. 1, 39–57, 2023.

[37] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Software*, vol. 35, no. 3, pp. 96–100, 2018.

[38] T. Pikkumäki, *Comparison of monolithic, micro-service, and cloud development*, JAMK University of Applied Sciences, Jyväskylä, Finland, 2023.

[39] S. H. A. Hamed, "Reusability of legacy software using microservices: An online exam system example," *Journal of Al-Qadisiyah for Computer Science and Mathematics*, vol. 15, no. 3, pp. 35, 2023.

[40] H. Bai and X. Liu, "Design and implementation of intelligent medical system based on microservices," in: *Proceedings of the 4th Management Science Informatization and Economic Innovation Development Conference, MSIEID 2022, December 9–11, 2022*, Chongqing, China, 2023.

[41] S. Primer, "Service-oriented architecture and legacy systems *SOA Primer*, 2023.

[42] C. Maniveena and R. Kalaiselvi, "A survey on IoT security and privacy," in: *AIP Conference Proceedings*, vol. 2904, no. 1, AIP Publishing, 2023.

[43] V. A. Vasil'ev, P. S. Chernov, N. V. Gromkov, and M. A. Shcherbakov, "Service-oriented architecture and its application to smart capabilities of sensors," in: *2017 International Siberian Conference on Control and Communications (SIBCON)*, 2017, pp. 1–4.

[44] Sparkequation, Stateless vs stateful, 2022, https://sparkequation.com/2020/11/12/stateless-vs-stateful-microservices-addressing-the-benefits-and-quandaries/.

[45] T. Golis, P. Dakić, and V. Vranić, "Creating microservices and using infrastructure as code within the CI/CD for dynamic container

creation," in: *2022 IEEE 16th International Scientific Conference on Informatics (Informatics)*, IEEE, Nov. 2022.

[46] P. Dakić and M. Živković, "An overview of the challenges for developing software within the field of autonomous vehicles," in: *7th Conference on the Engineering of Computer Based Systems*, ser. ECBS 2021, New York, NY, USA: Association for Computing Machinery, 2021, doi: https://doi.org/10.1145/3459960.3459972.

[47] N. Hroncová and P. Dakić, "Research study on the use of CI/CD among Slovak students," in: *2022 12th International Conference on Advanced Computer Information Technologies (ACIT)*, IEEE, Sep 2022.

[48] K. J. P. G. Perera and I. Perera, "TheArchitect: A serverless-microservices based high-level architecture generation tool," in: *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, 2018, pp. 204–210.

[49] M. Gördesli and A. Varol, "Comparing interservice communications of microservices for e-commerce industry," in: *2022 10th International Symposium on Digital Forensics and Security (ISDFS)*, 2022, pp. 1–4.

[50] RedHat, What is server mesh, 2022. https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh.

[51] IBM, Containerization, 2022. https://www.ibm.com/cloud/learn/containerization.

[52] R. Muddinagiri, S. Ambavane, and S. Bayas, "Self-hosted kubernetes: Deploying docker containers locally with minikube," in: *2019 International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET)*, 2019, pp. 239–243.

[53] NetApp, Container vs VMS, 2022. https://www.netapp.com/blog/containers-vs-vms/.

[54] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, et al., "Large-scale analysis of docker images and performance implications for container storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 918–930, 2021.

[55] M. R. Pratama and D. Sulistiyo Kusumo, "Implementation of continuous integration and continuous delivery (CI/CD) on automatic performance testing," in: *2021 9th International Conference on Information and Communication Technology (ICoICT)*, IEEE, Yogyakarta, Indonesia, 2021, pp. 230–235.

[56] Atlassian, Continuous integration, 2022. https://www.atlassian.com/continuous-delivery/continuous-integration/how-to-get-to-continuous-integration.

[57] IBM, A practical guide to the continuous integration/continuous delivery (CI/CD) pipeline, 2022, https://www.ibm.com/cloud/blog/ci-cd-pipeline.

[58] A. Malviya and R. K. Dwivedi, "A comparative analysis of container orchestration tools in cloud computing," in: *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, 2022, pp. 698–703.

[59] S. Telenyk, O. Sopov, E. Zharikov, and G. Nowakowski, "A comparison of Kubernetes and Kubernetes-compatible platforms," in: *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, 2021, pp. 313–317.

[60] Helm, Helm, 2022, https://github.com/helm/helm.