COMPUTOLOGY: Journal of Applied Computer Science and Intelligent Technologies
Volume 1, Issue 1, Jan-Jun 2021, pp. 40-51
DOI: 10.17492/computology.v1i1.2105
https://www.journalpressindia.com/cjacsit
© 2021 Journal Press India

A Systematic Review of the Impact of Containerization on Software Development and Deployment Practices

Sarishma and Abhishek***

ABSTRACT

Containerization[1] has altered software development and deployment practices by allowing for more efficient program packaging, shipping, and execution. This systematic review provides an in-depth examination of containerization's benefits, problems, and new trends. It investigates the effects of containerization on software development workflows, to increase cooperation and enhancing the software delivery lifecycle. The study looks into the impact of containerization on deployment, including application distribution, scalability, and management, and emphasizes the benefits of [2]container orchestration systems like Kubernetes. It also solves adoption constraints including security concerns and resource allocation. The findings give useful insights for scholars, practitioners, and organizations, assisting in the optimization of workflows, increasing productivity, and achieving consistent software delivery. This evaluation is a significant resource for executives and scholars in the industry.

Keywords: Containers; Containerization; Continuous Integration; Continuous Deployment; CI/CD; DevOps; Software Development.

1.0 Introduction

Containerization has developed as a prominent approach in modern software development for developing and distributing applications. [3]This study will undertake a systematic review and meta-analysis of the literature to examine the influence of containerization on software development and deployment practices. The following research questions will be the focus of the study:

1. What is the impact of containerization on the software development process, including design, coding, testing, and deployment?
2. What are the advantages and disadvantages of using containers in software development and deployment?
3. What is the impact of containerization on DevOps culture and practices, such as continuous integration and delivery (CI/CD)?
4. What are the optimal containerization practices for software development and deployment?

Containerization has emerged as a disruptive technique in software development and deployment, disrupting traditional approaches and enabling more efficient and simplified procedures.

**Corresponding author; Department of Computer Science and Engineering, Graphic Era (Deemed to be University), Dehradun, India (E-mail: sarishma.cse@geu.ac.in)*

**Department of Computer Science and Engineering, Graphic Era (Deemed to be University), Dehradun, India (E-mail: abhishekguptageu@gmail.com)*

Its ability to encapsulate applications and their dependencies into portable and self-contained pieces has had a profound impact on how software is produced, distributed, and performed. [4] As a result, it has piqued the interest of researchers, industry professionals, and organizations looking to improve their software development and deployment practices. The goal of this study is to undertake a complete systematic review of the influence of containerization on software development and deployment practices. [5]

This study attempts to provide a full overview of the consequences, benefits, problems, and emerging trends related to containerization in the software industry by analyzing a wide range of scholarly articles, research papers, and case studies. Providing various benefits in software development, including increased program portability, uniformity in development and testing environments, and greater cooperation among development teams. Containerization of applications and their dependencies enables smooth deployment across several platforms and operating systems, providing uniform behavior regardless of the underlying infrastructure. This characteristic of portability has a significant impact on software development workflows, allowing developers to operate in a uniform environment and easing the process of moving applications from development to production.

Furthermore, containerization has transformed deployment practices by introducing scalable and efficient means for application delivery and management. Containers allow for speedier deployment, more efficient resource utilization via container orchestration systems, and improved fault tolerance and scalability via automated scaling capabilities. These developments have major consequences for both traditional and cloud-based deployment strategies, allowing organizations to deploy applications faster, manage them more effectively, and adapt to changing demands more swiftly. While containerization has many advantages, it also raises new difficulties that must be handled. Security problems, container sprawl, and interaction with existing development and deployment practices are just a few of the issues that must be addressed carefully. Understanding these issues is essential for successful containerization adoption and integration into existing software development and deployment processes. This paper intends to integrate existing knowledge, identify gaps, and provide insights into the impact of containerization on software development and deployment practices by undertaking a systematic review. The study's findings will be a great resource for researchers, practitioners, and organizations looking to efficiently embrace containerization and optimize their software development and deployment procedures.

2.0 Literature Review

A. Containerization

Containerization is a method of developing and deploying software that allows applications and their dependencies to be packaged as self-contained entities known as containers. [6]These containers provide a lightweight, isolated runtime environment, ensuring consistency and mobility between systems. Containers encapsulate the application code, runtime environment, libraries, and dependencies, allowing them to run on any containerized infrastructure with confidence. This method has several advantages, including:

- Containers are capable of running on a wide range of operating systems and infrastructure platforms, including Linux, Windows, and cloud environments. They include all of the required components, allowing applications to be readily moved between development, testing, and production environments.

- Containers are lightweight and share the operating system kernel of the host machine, resulting in minimum resource overhead. They start up faster and have a smaller memory footprint than traditional virtual machines, enabling more efficient use of infrastructure resources.
- Containers separate programs and their dependencies, ensuring that changes in one container do not have an impact on modifications in others. This isolation increases security, stability, and fault tolerance by minimizing conflicts and limiting the impact of failures.
- Scalability: Containers may be readily scaled horizontally by running many instances of the same containerized application. Container orchestration technologies can automate the scaling process, allowing applications to successfully handle increased traffic and demand.
- Containerization promotes agile development by enabling developers to operate in consistent and repeatable contexts. Containers make it easier to set up development environments, ease the deployment process, and result in shorter development cycles.

Container engines have developed as effective tools to aid with containerization, while orchestration platforms aid in the management and scaling of containerized applications. Some noteworthy container engines and orchestration platforms are as follows:

1. Docker is a well-known container engine that helped popularise the concept of containers. It comes with an easy-to-use interface and a set of tools for creating, distributing, and executing containers. Docker Engine is the runtime environment that enables the execution of containers on host machines.
2. CRI-O: The Container Runtime Interface (CRI) is supported by CRI-O, a lightweight and efficient container engine. It focuses on Open Container Initiative (OCI)-compliant container operations. CRI-O provides a dependable and safe runtime environment for Kubernetes clusters.
3. Podman is a container engine that has a command-line interface comparable to Docker but does not require a daemon. The use of pods, containers, and images as primary structural elements is emphasized. Podman is a lightweight container runtime that is suitable for both development and production contexts.
4. Containerd is a runtime that controls the lifetime of containers with a focus on simplicity, dependability, and performance. It is the container runtime backend for a number of container platforms and orchestration systems, including Docker and Kubernetes.
5. rkt: In CoreOS, rkt (pronounced "rocket") is a container runtime. Its goals are to provide security, usability, and adaptability. rkt adheres to OCI standards and includes capabilities such as image signing and verification, isolation, and resource management efficiency.

[7]Among container orchestration platforms, the following stand out:

1. Kubernetes is a popular and effective container orchestration platform. It simplifies the deployment, scalability, and maintenance of containerized applications. Kubernetes facilitates complex and scalable deployments by including capabilities like service discovery, load balancing, self-healing, and horizontal scaling.
2. Docker Swarm is Docker's native clustering and orchestration solution. It simplifies the deployment and management of Docker containers in a swarm cluster. Docker Swarm prioritizes usability, scalability, and Docker ecosystem integration.
3. Apache Mesos is a distributed systems kernel that enables efficient resource sharing and cluster separation. It provides frameworks for container orchestration, such as Marathon, enabling large-scale application deployment and management.
4. Amazon Elastic Container Service (ECS): [8]Amazon Web Services (AWS) provides ECS, a fully managed container orchestration solution. It simplifies container deployment, management, and scalability by leveraging AWS services and seamlessly connecting to other AWS capabilities.

5. Google Kubernetes Engine (GKE): GKE is a managed Kubernetes service provided by Google Cloud Platform (GCP). It provides a reliable and high-availability Kubernetes environment while also managing infrastructure and communicating with other GCP services.
6. Microsoft Azure Kubernetes Service (AKS): [9]A managed Kubernetes service, Microsoft Azure Kubernetes Service (AKS). It streamlines the deployment, maintenance, and scaling of containerized applications on Azure infrastructure and integrates with Azure services.
7. Nomad is an open-source orchestration platform developed by HashiCorp. It supports application deployment and administration across a variety of infrastructure platforms, including containers, virtual machines, and bare metal.
8. Red Hat developed OpenShift[10], a Kubernetes-based container orchestration platform. It provides a complete platform for designing, deploying, and supporting containerized applications with additional capabilities and tools.
9. Rancher is an open-source container management platform that works with several orchestration frameworks, including Kubernetes, Docker Swarm, and Apache Mesos. It simplifies container deployment and management in a variety of infrastructure setups.
10. IBM Cloud Kubernetes Service (IKS): IKS is a Kubernetes service managed by IBM Cloud. It offers a secure and scalable Kubernetes environment with integrated application deployment, administration, and management tools and services.

These container orchestration solutions offer some features and capabilities that aid in the administration of containerized applications, automate scaling, provide high availability, and expedite deployment procedures. Organizations can choose the best platform for their needs based on factors like ease of use, scalability, integration with existing infrastructure, and cloud provider preferences.

B. Traditional Methods

In the industry, numerous [11]Software Development Life Cycle (SDLC) approaches are utilized, each with its own set of advantages and disadvantages. In this section, we will look at some of the most common SDLC methodologies and their drawbacks:

1. Waterfall: The Waterfall method is a linear approach that requires each phase to be completed before going on to the next. While it provides a planned and sequential procedure, it does not allow for adjustments or input throughout development.
2. Agile methodologies, such as Scrum, place a premium on iterative development, cooperation, and adaptation. They may, however, struggle with documentation and may suffer scaling issues for larger projects or scattered teams.
3. Spiral: The Spiral approach combines waterfall and iterative development components. It enables risk minimization and early prototyping, but it can be resource-intensive and difficult to maintain.
4. The V-Model links testing activities with each development phase to ensure thorough test coverage. It can, however, be stiff and less responsive to changing circumstances.
5. Rapid Application Development (RAD) focuses on rapid prototyping and user feedback to enable rapid software delivery. However, this can result in lost documentation and requires highly skilled engineers to maintain quality.
6. Lean Development: The goal of lean approaches is to decrease waste while increasing value delivery. However, they may lack formal processes and documentation, making consistency and scalability difficult to maintain.

7. DevOps: This term refers to the integration of development and operations teams to achieve continuous integration, delivery, and deployment. However, its execution might be difficult, necessitating cultural shifts as well as automated tooling.

It is vital to remember that the SDLC method chosen is determined by the project scope, team size, organizational culture, and other considerations. Teams frequently customize or blend several approaches to meet their objectives and overcome the shortcomings of a single methodology[12].

C. Drawbacks

Traditional [13]SDLC approaches can present developers with several obstacles and downsides that might stymie the software development process. Among these difficulties are[14]:

- Rigidity: Traditional SDLC approaches, such as the Waterfall model, are sequential and rigid. This might make accommodating changes or incorporating feedback during the development process challenging, resulting in delays and a lack of response to changing requirements.
- Limited Customer interaction: Until the later stages of development, traditional approaches sometimes include little or no customer or end-user interaction. This can lead to a mismatch between client expectations and the final product, perhaps leading to rework and unhappiness.
- lengthier Time-to-Market: Traditional approaches often have lengthier development cycles due to their sequential structure. This can cause a delay in time-to-market, particularly if changes or alterations are required.
- Lack of Flexibility: Traditional SDLC methodologies are frequently incapable of adapting to changing corporate needs or market circumstances. Incorporating new features or addressing emergent difficulties without upsetting the established development schedule might be difficult.
- Traditional approaches may provide limited visibility into the development process, making tracking progress and identifying bottlenecks difficult. Transparency can stymie successful project management and decision-making.
- Increased danger: Because traditional processes are sequential, the danger of producing a product that does not match customer expectations or market needs is increased. It may also increase the risks associated with late defect detection and resolution.
- Limited Collaboration: Traditional techniques frequently include siloed development phases in which teams work in isolation and hand over deliverables to the next phase. This can stifle collaboration and communication among team members, resulting in inefficiencies and potential misalignment.
- To solve these problems, several organizations have shifted to more agile and iterative methodologies, such as Agile or DevOps, which provide better flexibility, customer interaction, and adaptability throughout the development lifecycle.

3.0 DevOps

[15]DevOps' origins date back to the early 2000s when there was a clear need for close communication between software development and operations teams. The traditional disconnected approach creates inefficiencies, slows time-to-market, and creates inconsistencies between development and operational goals. DevOps has emerged as a way to bridge this gap and foster a culture of collaboration and agility.

Patrick Debois and Andrew Shafer coined the term DevOps in 2009 after a discussion about Agile infrastructure. Meanwhile, the core concepts and practices of DevOps have evolved over the years. DevOps can be traced back to several key movements and practices:

- Agile Development: Beginning in the early 2000s, the Agile movement emphasized iterative development, cross-functional teams, and customer collaboration. It established the collaborative ethos that DevOps epitomizes.
- DevOps was inspired by lean manufacturing techniques, specifically the Toyota Production System, which focused on minimizing waste, optimizing workflows, and encouraging continuous improvement.
- Continuous Integration (CI): Martin Fowler popularised the practice of continuous integration in the early 2000s, which emphasized integrating code changes often and automating the build and test procedures. CI foreshadowed the automation and integration components of DevOps.
- Infrastructure as Code (IaC): The introduction of infrastructure as code in the mid-2000s enabled infrastructure management through code, allowing for version control, repeatability, and automation. IaC set the groundwork for DevOps infrastructure automation.

DevOps evolved as a cultural and technological trend over time, driven by the need for faster software delivery, higher quality, and increased collaboration. It promotes shared responsibility and ownership of the full software delivery lifecycle by breaking down silos between development, operations, and other stakeholders. DevOps practices and tooling are rapidly evolving. Automation, continuous integration and delivery (CI/CD), infrastructure automation, and monitoring have all become crucial DevOps toolkit components. Industry standards such as the DevOps Handbook and the CALMS framework (Culture, Automation, Lean, Measurement, and Sharing) have aided organizations in implementing DevOps principles. DevOps has now become a mainstream technique, adopted by organizations of all sizes and industries trying to improve time-to-market, quality, and customer happiness. It has changed the landscape of software development and operations, allowing organizations to provide value to end users more efficiently and effectively.

A. DevOps Rise

Because of the changing dynamics of software development and deployment, DevOps must be used instead of traditional Software Development Life Cycle (SDLC) approaches. [12]Traditional SDLC processes are usually limited, preventing firms from reaching optimum efficiency and output. Here are some of the primary reasons why DevOps has become the preferred method:

- Acceleration of Time-to-Market: DevOps focuses on streamlining development and deployment processes, allowing firms to provide software more quickly. By automating the development, test, and deployment processes, DevOps eliminates bottlenecks and time-consuming manual activities, resulting in a speedier time-to-market.
- By removing barriers between development and operations teams, DevOps promotes collaboration and communication. This collaborative culture promotes better goal alignment, more visibility, and stronger collaboration, all of which result in higher software quality and fewer errors.
- DevOps focuses on an agile and iterative method, which enables businesses to respond fast to changing business requirements. By automating infrastructure provisioning and leveraging infrastructure as code, DevOps enables flexible growth and adaptation to changing needs.
- DevOps practices such as continuous integration, continuous delivery, and automated testing ensure greater software quality. Frequent and automated testing helps to spot faults earlier, resulting in more reliable deployments.

- DevOps optimize resource utilization by leveraging cloud infrastructure, containerization, and efficient deployment procedures. As a result, cost savings, enhanced scalability, and the flexibility to handle changing workloads are realized.
- Continuous Feedback Loop: Feedback from stakeholders such as customers and end users is valued highly in DevOps. By including feedback loops throughout the development process, DevOps enables firms to iterate and improve software based on real-time information, ensuring customer satisfaction.
- By encouraging the use of metrics, monitoring, and feedback tools, DevOps fosters a culture of continuous improvement. This can be used by businesses to identify areas for improvement, make data-driven decisions, and drive ongoing optimization.

In today's fast-paced and highly competitive business, organizations must deliver high-quality software quickly. DevOps overcomes the limitations of traditional SDLC techniques by prioritizing collaboration, automation, agility, and continuous improvement. By deploying DevOps, organizations can benefit from faster time-to-market, higher software quality, improved customer satisfaction, and a competitive advantage in the software business.

B. Stages

DevOps is comprised of numerous stages, each of which contributes to the successful implementation of its concepts and practices. [16]These stages encompass the entire software development and deployment life cycle. The stages of DevOps are as follows:

- Continuous Planning entails working with stakeholders to define project requirements, set goals, and develop a shared knowledge of the software development process. Backlog grooming, sprint planning, and prioritization are all part of it.
- Continuous Integration (CI): During this stage, developers often merge their code changes into a shared repository. CI guarantees that modifications from several developers are combined smoothly, avoiding integration difficulties. To analyze the modifications and detect any vulnerabilities early on, automated builds and tests are launched.
- The goal of Continuous Delivery (CD) is to automate deployment and prepare software releases for production. Packaging the software, creating the requisite environments, and delivering the artifacts to various stages such as development, testing, and production are all part of the process.
- Continuous Testing: Throughout the development phase, continuous testing ensures that the program meets quality criteria. It includes automated testing at multiple levels, such as unit tests, integration tests, and end-to-end tests. Testing frameworks and technologies aid in the detection of faults, vulnerabilities, and performance issues.
- Continuous Deployment: Continuous Deployment extends CD automation by automatically deploying software to production environments once all relevant tests have been passed. Stringent monitoring, rollback procedures, and thorough validation are essential at this step to ensure that only dependable and stable releases reach the production environment.
- The collection and analysis of data from production systems to gain insights into their performance, stability, and user experience is known as continuous monitoring. Monitoring tools aid in the identification of possible issues, the tracking of vital metrics, and the assurance of the application's reliability and availability.
- Continuous Feedback requires soliciting feedback from a wide range of stakeholders, such as consumers, end-users, and operations teams. Feedback assists in identifying areas for improvement,

validating assumptions, and prioritizing future development actions. This stage enables companies to iterate and enhance their software based on real-time feedback.

- Continuous Learning places a premium on the culture of continuous improvement. It requires creating a mindset of learning from experiences, accepting new technology and practices, and sharing information within the team. Retrospectives, post-implementation evaluations, and knowledge-sharing sessions are crucial at this stage.

These steps are interconnected and iterative, rather than linear. They encourage collaboration, automation, and feedback throughout the software development and deployment process. By embracing these steps, organizations can achieve faster delivery, higher quality, increased agility, and improved customer satisfaction.

4.0 Containerization Impact on the Software Development Process

Containerization has had a profound impact on the software development process, transforming stages ranging from design to deployment. Consider how containerization has impacted each stage:

- Containerization promotes a modular, loosely coupled design style. Large apps can be divided into smaller, self-contained components known as microservices by developers. This increases the scalability, maintainability, and flexibility of the design process.
- Containers provide consistent and predictable coding environments for developers. Containers enable developers to package their applications, including all dependencies, libraries, and customizations. This eliminates the "it works on my machine" problem and ensures consistent behavior across contexts.
- Testing: Containerization facilitates testing. Test environments may be easily reproduced by spinning up containers with similar setups. Automated testing may be easily integrated into the containerization workflow, allowing developers to notice and resolve issues more quickly.
- Containers facilitate collaboration among development teams. Using containers, developers may distribute their software as self-contained units. This promotes collaboration, shortens feedback loops, and allows developers to work on multiple application components concurrently.
- Containerization enables smooth and uniform deployment across several environments. Containers can be deployed to any infrastructure that supports containers, such as local PCs, on-premises servers, or cloud platforms. Containers encapsulate the application and its dependencies, eliminating compatibility issues and shortening deployment times.
- Containers facilitate horizontal scalability. By launching many instances of containerized services, developers can swiftly scale their applications using container orchestration technologies such as Kubernetes. This allows programs to handle increasing traffic and demand more effectively.
- Pipelines for Continuous Integration and Continuous Deployment (CI/CD) are inextricably linked to containerization. Containers enable the creation of standardized, version-controlled artifacts that can be consistently delivered from development to production. This makes it easier to offer updates and new features to end consumers.

Overall, containerization has improved cooperation, scalability, and development cycle speed in the software development process. It promotes a more modular and flexible design approach, simplifies testing and deployment, and enables enterprises to implement agile practices such as continuous integration and continuous deployment. Containerization enables software development teams to more efficiently and confidently create high-quality products.

5.0 Containerization Impact on the Software Development Process

Containerization has had a profound impact on the software development process, transforming stages ranging from design to deployment[17]. Consider how containerization has impacted each stage:

- Containerization promotes a modular, loosely coupled design style. Large apps can be divided into smaller, self-contained components known as microservices by developers. This increases the scalability, maintainability, and flexibility of the design process.
- Containers provide consistent and predictable coding environments for developers. Containers enable developers to package their applications, including all dependencies, libraries, and customizations. This eliminates the "it works on my machine" problem and ensures consistent behavior across contexts.
- Testing: Containerization facilitates testing. Test environments may be easily reproduced by spinning up containers with similar setups. Automated testing may be easily integrated into the containerization workflow, allowing developers to notice and resolve issues more quickly.
- Containers facilitate collaboration among development teams. Using containers, developers may distribute their software as self-contained units. This promotes collaboration, shortens feedback loops, and allows developers to work on multiple application components concurrently.
- Containerization enables smooth and uniform deployment across several environments. Containers can be deployed to any infrastructure that supports containers, such as local PCs, on-premises servers, or cloud platforms. Containers encapsulate the application and its dependencies, eliminating compatibility issues and shortening deployment times.
- Containers facilitate horizontal scalability. By launching many instances of containerized services, developers can swiftly scale their applications using container orchestration technologies such as Kubernetes. This allows programs to handle increasing traffic and demand more effectively.
- Pipelines for Continuous Integration and Continuous Deployment (CI/CD) are inextricably linked to containerization. Containers enable the creation of standardized, version-controlled artifacts that can be consistently delivered from development to production. This makes it easier to offer updates and new features to end consumers.

Overall, containerization has improved cooperation, scalability, and development cycle speed in the software development process. It promotes a more modular and flexible design approach, simplifies testing and deployment, and enables enterprises to implement agile practices such as continuous integration and continuous deployment. Containerization enables software development teams to more efficiently and confidently create high-quality products.

6.0 Impact of Containerization on DevOps Culture and Practices

[3]Containerization has transformed DevOps culture and practices, revolutionizing how software is built, tested, and delivered. The improved collaboration and communication between development and operations teams are one of the most important consequences of containerization on DevOps.Developers can construct consistent environments that closely mirror production circumstances by encapsulating applications and their dependencies into containers. This enables more effective collaboration because developers and operations teams can use the same containerized environments throughout the software development lifecycle.

Containerization encourages the usage of automation in DevOps practices as well. It is now possible to automate the construction, testing, and deployment processes using containers. This automation reduces manual errors, boosts efficiency, and allows for continuous integration and continuous deployment (CI/CD) pipelines.

7.0 Optimal Containerization Practices

[18]To ensure productivity, [19]scalability, and reliability, optimal containerization practices for software development and deployment include several critical elements. Here are some suggested practices:

- Microservices Architecture: Using a microservices architecture allows for the application to be divided into smaller, loosely linked services. Each service can be containerized independently, enhancing scalability, flexibility, and maintenance ease.
- Containerization Best Practises: Use containerization best practices such as lightweight and single-purpose containers, optimizing container image size, and minimizing dependencies. These practices improve productivity, security, and resource utilization.
- Infrastructure as Code: To specify and automate container environment setup, use infrastructure as code tools such as Docker Compose or Kubernetes YAML manifests. This assures uniformity and reproducibility across the lifespan of software development and deployment.
- Continuous Integration and Continuous Deployment (CI/CD) pipelines that automate the build, testing, and deployment procedures should be implemented. This enables rapid and dependable software releases, guaranteeing that containerized applications are thoroughly tested before deployment.
- Container Orchestration: To manage and scale containerized systems, use container orchestration platforms such as Kubernetes. To optimize resource utilization and ensure high availability, use capabilities like load balancing, auto-scaling, and service discovery.
- Monitoring and logging: For containerized applications, implement robust monitoring and logging systems. Monitor resource utilization, performance metrics, and container health to detect and rectify problems as soon as possible.
- Security considerations include employing secure base images, analyzing container images for vulnerabilities, and establishing access controls. To reduce potential security threats, update and patch containers regularly.
- Documentation and Versioning: Keep detailed records of container images, dependencies, and configurations. Use versioning to keep track of changes and assure reproducibility.
- Encourage good collaboration and communication among development, operations, and other relevant teams. Encourage cross-functional teams to share information, solve problems, and collaborate throughout the containerization process.
- Maintenance and updates should be performed regularly to incorporate bug fixes, security patches, and feature additions. Review and optimize container setups regularly to increase performance.

These practices lay the groundwork for efficiently exploiting containerization in software development and deployment, resulting in streamlined processes, scalability, and resilience in application delivery.

8.0 Future work

The project "Scalable and Resilient DevOps Microservices Architecture for Web Applications" can be improved further by combining numerous new technologies to improve its [20]scalability, dependability, and security. To begin, the project can be expanded to include OpenShift, an enterprise-grade container platform, to simplify the deployment and management of microservices in a hybrid cloud environment. This will increase the architecture's flexibility and scalability while also providing sophisticated capabilities like service mesh and autonomous scaling. Second, the project can be augmented with [21]AI and ML capabilities such as anomaly detection and predictive analysis to increase the performance and reliability of the microservices architecture. This would allow the architecture to recognize and respond to any faults before they caused severe interruptions. Third,[22] to maintain the security and integrity of the microservices and the entire architecture, the project can be augmented with increased security methods such as container image scanning, runtime protection, and vulnerability management. Fourth,[23] the project can be expanded to include Ceph, a distributed storage system, to provide dependable and scalable storage for microservices and data. This would improve the architecture's resilience and availability while lowering the chance of data loss. Fifth, Ansible, an automation tool, can be used to automate the deployment, configuration, and maintenance of the microservices architecture. This would save time and increase the consistency and dependability of the deployment process.

Finally, [24]DevSecOps practices can be used to strengthen the overall security posture of the architecture and ensure that security is a basic component of the development and deployment process. [25]These practices incorporate security into the entire software development lifecycle.

Overall, future development on the project "Scalable and Resilient DevOps Microservices Architecture for Web Applications" may concentrate on combining modern technologies and practices to improve the architecture's scalability, reliability, and security.

References

1. E. Casalicchio, “Container Orchestration: A Survey,” *EAI/Springer Innov. Commun. Comput.*, no. Vm, pp. 221–235, 2019, doi: 10.1007/978-3-319-92378-9_14.
2. I. M. Al Jawarneh *et al.*, “Container Orchestration Engines: A Thorough Functional and Performance Comparison,” *IEEE Int. Conf. Commun.*, vol. 2019-May, pp. 1–6, 2019, doi: 10.1109/ICC.2019.8762053.
3. I. M. Al Jawarneh *et al.*, “Container Orchestration Engines: A Thorough Functional and Performance Comparison,” *IEEE Int. Conf. Commun.*, vol. 2019-May, 2019, doi: 10.1109/ICC.2019.8762053.
4. “Traditional Deployment VS Virtualization VS Container | by Bikram | Medium.” https://bikramat.medium.com/traditional-deployment-vs-virtualization-vs-container-f9b82ce98a50
5. “What is Container Orchestration? | VMware Glossary.” https://www.vmware.com/topics/glossary/content/container-orchestration.html
6. U. Zdun, E. Wittern, and P. Leitner, “Emerging Trends, Challenges, and Experiences in DevOps and Microservice APIs,” *IEEE Softw.*, vol. 37, no. 1, pp. 87–91, 2020, doi: 10.1109/MS.2019.2947982.

7. E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A comprehensive feature comparison study of open-source container orchestration frameworks," *Appl. Sci.*, vol. 9, no. 5, 2019, doi: 10.3390/app9050931.
8. "Welcome - kOps - Kubernetes Operations." https://kops.sigs.k8s.io/
9. "Managed Kubernetes Service (AKS) | Microsoft Azure." https://azure.microsoft.com/en-in/products/kubernetes-service
10. "Openshift Vs. Kubernetes : What is the Difference?." https://www.simplilearn.com/kubernetes-vs-openshift-article
11. R. Arora and N. Arora, "Analysis of SDLC Models," *Int. J. Curr. Eng. Technol.*, vol. 6, no. 1, pp. 2277–4106, 2016, [Online]. Available: http://inpressco.com/category/ijcet.
12. S. Balaji, "Waterfall vs v-model vs agile : A comparative study on SDLC," *WATEERFALL Vs V-MODEL Vs Agil. A Comp. STUDY SDLC*, vol. 2, no. 1, pp. 26–30, 2012.
13. S. Radack, "Security Considerations in the System Development Life Cycle," *Natl. Inst. Stand. Technol.*, pp. 1–7, 2002.
14. V. K. Sarkania and V. K. Bhalla, "International Journal of Advanced Research in," *Android Internals*, vol. 3, no. 6, pp. 143–147, 2013.
15. C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Software Engineering for DevOps," *Ieee Comput. Soc.*, no. June, pp. 94–100, 2016.
16. L. Surya, "An Exploratory Study of DevOps and It's Future in the United States," *Int. J. Creat. Res. Thoughts*, vol. 3, no. 2, pp. 2320–2882, 2016.
17. L. Chen, "Microservices: Architecting for Continuous Delivery and DevOps," *Proc. - 2018 IEEE 15th Int. Conf. Softw. Archit. ICSA 2018*, no. March, pp. 39–46, 2018, doi: 10.1109/ICSA.2018.00013.
18. T. Yarygina, "Exploring Microservice Security," 2018, [Online]. Available: http://bora.uib.no/handle/1956/18696.
19. B. El Khalyly, A. Belangour, A. Erraissi, and M. Banane, "Devops and Microservices Based Internet of Things Meta-Model," *Int. J. Emerg. Trends Eng. Res.*, vol. 8, no. 9, pp. 6254–6266, 2020, doi: 10.30534/ijeter/2020/217892020.
20. L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A survey of DevOps concepts and challenges," *ACM Comput. Surv.*, vol. 52, no. 6, 2019, doi: 10.1145/3359981.
21. L. Surya, "AI and DevOps in information technology and its future in the United States," *Int. J. Creat. Res. Thoughts*, vol. 9, no. 2, pp. 2032–2036, 2021, [Online]. Available: https://ssrn.com/abstract=3786535.
22. C. Hochreiner, O. Kopp, and O. Kopp, "ZEUS 2018 10 th ZEUS Workshop , ZEUS 2018 ," no. 10, pp. 8–9, 2018.
23. S. Hoque, M. S. De Brito, A. Willner, O. Keil, and T. Magedanz, "Towards Container Orchestration in Fog Computing Infrastructures," *Proc. - Int. Comput. Softw. Appl. Conf.*, vol. 2, pp. 294–299, 2017, doi: 10.1109/COMPSAC.2017.248.
24. L. Prates, J. Faustino, M. Silva, and R. Pereira, "DevSecOps metrics," *Lect. Notes Bus. Inf. Process.*, vol. 359, no. 351, pp. 77–90, 2019, doi: 10.1007/978-3-030-29608-7_7.
25. H. Myrbakken and R. Colomo-Palacios, "DevSecOps: A multivocal literature review," *Commun. Comput. Inf. Sci.*, vol. 770, no. October, pp. 17–29, 2017, doi: 10.1007/978-3-319-67383-7_2.