

Kubernetes-Native CI/CD Architectures: Scaling Secure Build and Deployment Workflows on AWS EKS

Yash Kajavadara, Harsh Miyani

(yashr.kajavadara@gmail.com)

(hmiyani8@gmail.com)

Article Info

Page Number:2228 - 2241

Publication Issue:

Vol 71 No. 3 (2022)

Abstract

Kubernetes-native continuous integration and continuous delivery (CI/CD) architectures are increasingly essential for automating build and deployment workflows in cloud-native environments. This study investigates the design, implementation, and evaluation of secure CI/CD pipelines on Amazon Elastic Kubernetes Service (AWS EKS). Using a design science research and experimental evaluation approach, the study measured pipeline performance, resource utilization, scalability, and security compliance under varying workloads. Results indicate that build stages dominate resource consumption and pipeline latency, while test and deployment stages exhibit lower resource demands. Cluster-resident pipelines demonstrate superior performance and reliability compared to hybrid external pipelines, particularly under concurrent execution, with reduced latency and lower failure rates. Security analysis shows high compliance with role-based access control and secret management, though namespace isolation requires refinement. The findings provide actionable insights for DevOps practitioners seeking to optimize Kubernetes-native CI/CD pipelines for performance, scalability, and security. The study contributes to understanding best practices for deploying automated workflows in managed Kubernetes environments, offering empirical benchmarks for pipeline execution times, resource utilization, and security compliance.

Article History

Article Received : 14 February 2022

Revised : 25 March 2022

Accepted : 01 April 2022

Keywords: AWS EKS, Build and Deployment Automation, Cloud-Native Workflows, Continuous Delivery, Continuous Integration, Kubernetes-Native CI/CD, Pipeline Security, Scalability.

1. Introduction

The increasing adoption of cloud computing and microservice based application design has altered how software systems are developed, tested, and deployed. Traditional monolithic delivery pipelines have proven insufficient in environments where applications are composed of loosely coupled services that require frequent updates, independent scaling, and continuous operational feedback. In response, continuous integration and continuous deployment practices have become central to modern software engineering, enabling teams to automate build, test, and release processes while maintaining software quality and reliability (Chen, 2015). As these practices mature, their tight integration with container orchestration platforms has become a defining feature of cloud native system delivery.

Kubernetes has emerged as the dominant container orchestration platform for managing containerized workloads at scale. Its scheduling mechanisms, declarative configuration model, and support for self healing and horizontal scaling provide a foundation well suited for continuous delivery in distributed environments (Burns et al., 2016; Pahl, 2015). By abstracting infrastructure concerns and standardizing deployment primitives, Kubernetes enables development teams to shift deployment logic closer to application configuration. This shift has encouraged the development of Kubernetes native CI/CD architectures in which build, test, and deployment workflows are executed and managed directly within the Kubernetes control plane (Poniszewska-Marańda & Czechowska, 2021).

In parallel, public cloud providers have introduced managed Kubernetes services to reduce the operational overhead associated with cluster management. Amazon Elastic Kubernetes Service represents a widely adopted example, offering managed control plane components, integration with cloud identity and access management, and native support for scalable compute and storage resources. These features make AWS EKS a practical environment for enterprise scale CI/CD workflows, particularly when combined with infrastructure as code practices and container based build systems. However, the use of a managed Kubernetes service does not eliminate architectural complexity. Instead, it introduces new considerations related to pipeline isolation, security boundaries, resource contention, and multi tenant workload management.

Security concerns are especially significant in Kubernetes native CI/CD systems. CI/CD pipelines often require elevated privileges to build container images, access artifact registries, and deploy workloads into production clusters. Misconfiguration or insufficient isolation within pipeline components can expose sensitive credentials or allow unauthorized access to cluster resources. Previous studies on cloud security have shown that shared responsibility models and misaligned access controls remain primary sources of security incidents in cloud environments (Hashizume et al., 2013). When CI/CD workflows are executed within Kubernetes clusters, these risks extend to pipeline pods, service accounts, and role based access control policies that govern interactions with the Kubernetes API.

At the same time, scalability remains a core requirement for Kubernetes native CI/CD architectures. Modern software systems often involve large numbers of microservices, each with independent build and deployment cycles. As pipeline execution frequency increases, clusters must accommodate bursts of build workloads without degrading the performance of production services. Research on container based cloud systems emphasizes the importance of effective resource management and workload isolation to maintain predictable performance under varying load conditions (Trani et al., 2018). In CI/CD contexts, this challenge is amplified by the need to balance compute intensive build stages with latency sensitive application workloads.

Existing literature provides substantial insight into continuous integration and delivery practices, Kubernetes architecture, and cloud native deployment models. Systematic reviews have documented the benefits and challenges of CI/CD adoption, including tooling fragmentation, pipeline complexity, and organizational alignment issues (Fitzgerald & Stol, 2017). Other studies have examined Kubernetes as a platform for automating software production environments, highlighting its suitability for orchestrating deployment workflows and enforcing declarative state management (Poniszewska-Marańda & Czechowska, 2021). However, fewer studies focus specifically on Kubernetes native CI/CD architectures deployed on managed platforms such as AWS EKS, particularly with respect to the combined concerns of scalability and security.

This research addresses that gap by examining Kubernetes native CI/CD architectures designed for secure and scalable build and deployment workflows on AWS EKS. The study focuses on architectural patterns in which CI/CD components operate as first class Kubernetes workloads, leveraging native scheduling, networking, and access control mechanisms. By grounding the analysis in established CI/CD theory, Kubernetes design principles, and cloud security research, this work aims to provide a structured understanding of how such architectures can support high frequency deployment while maintaining controlled access and predictable performance.

The remainder of this paper is organized as follows. The literature review synthesizes prior research on CI/CD systems, Kubernetes orchestration, and cloud security to establish the theoretical foundation for the study. The methodology section describes the research approach used to analyze Kubernetes native CI/CD architectures on AWS EKS. The results and discussion section presents architectural findings and examines their implications for scalability and security. The conclusion summarizes the contributions of the study and identifies directions for future research in Kubernetes based CI/CD systems.

2. Literature Review

2.1 Continuous Integration and Continuous Deployment

Continuous integration and continuous deployment (CI/CD) are foundational practices in modern software engineering that automate the process of building, testing, and deploying applications. CI emphasizes the frequent integration of code changes into a shared repository, accompanied by automated testing to detect integration issues

early. CD extends this practice by automating the release of integrated changes into production environments with minimal human intervention (Shahin et al., 2017). These practices are widely recognized for improving software quality, accelerating release cycles, and enabling teams to respond rapidly to changing business requirements (Chen, 2015).

Systematic reviews on CI/CD have highlighted both its advantages and implementation challenges. Benefits include reduced integration errors, faster feedback for developers, and greater predictability in release schedules. Challenges include toolchain complexity, the need for robust automated testing frameworks, and organizational alignment to support a culture of continuous delivery (Fitzgerald & Stol, 2017). Notably, these studies indicate that while CI/CD practices are conceptually straightforward, their implementation becomes complex in large, distributed, or microservice-based environments due to interdependencies and the need for pipeline orchestration.

2.2 Kubernetes as a Platform for CI/CD

Kubernetes, an open-source container orchestration platform, has emerged as a key enabler of cloud-native CI/CD architectures. By managing containerized workloads across clusters of nodes, Kubernetes abstracts infrastructure concerns, provides declarative deployment primitives, and automates tasks such as scheduling, scaling, and self-healing (Burns et al., 2016). The platform's design allows CI/CD pipelines to operate within the same environment as the applications they deploy, providing benefits such as environment consistency, reduced configuration drift, and improved integration with cloud resources (Pahl, 2015).

Recent studies highlight the use of Kubernetes for automating software production environments. For example, Poniszewska-Marańda and Czechowska (2021) discuss how Kubernetes clusters can host CI/CD pipeline components as native workloads. By deploying build agents, test environments, and deployment controllers directly into the cluster, organizations can leverage Kubernetes features such as resource isolation, network policies, and persistent storage for pipeline operations. This approach reduces reliance on external CI/CD infrastructure and aligns pipeline execution with application runtime environments, which can enhance reliability and reproducibility.

2.3 Security Considerations in CI/CD on Kubernetes

Integrating CI/CD pipelines within Kubernetes clusters introduces specific security considerations. Pipeline components often require elevated privileges to build container images, access repositories, or deploy workloads. Misconfigurations in service accounts, roles, or network policies can lead to unauthorized access or data leakage. Hashizume et al. (2013) emphasized that cloud computing environments, including container orchestration platforms, are susceptible to security incidents arising from improper access control, mismanagement of credentials, and poorly isolated workloads.

Securing Kubernetes-native CI/CD pipelines involves several strategies. Role-based access control (RBAC) ensures that pipeline components operate with the least privileges necessary. Namespaces and network policies provide logical and network-level isolation for build and deployment workflows. Additionally, integrating security scanning into pipelines such as image vulnerability analysis helps prevent the introduction of insecure artifacts into production clusters. These security measures are increasingly recognized as essential components of modern DevOps practices (Trani et al., 2018).

2.4 Scalability and Resource Management

Scalability is another critical factor in Kubernetes-native CI/CD architectures. Modern applications often consist of numerous microservices, each requiring independent build, test, and deployment cycles. CI/CD pipelines must handle high concurrency and resource-intensive operations, such as compiling code, running integration tests, and building container images. Effective resource management is essential to ensure that pipeline workloads do not adversely impact production services (Trani et al., 2018).

Studies on container-based cloud systems have emphasized the importance of horizontal scaling, resource quotas, and scheduling strategies to support high-throughput CI/CD operations. Kubernetes' scheduler can distribute build and deployment workloads across cluster nodes according to resource availability, enabling efficient utilization of CPU, memory, and storage resources (Poniszewska-Marańda & Czechowska, 2021). By combining scheduling

policies with container resource limits, organizations can optimize pipeline performance while maintaining operational stability for other workloads running in the cluster.

2.5 Managed Kubernetes Services and AWS EKS

Managed Kubernetes services, such as Amazon Elastic Kubernetes Service (EKS), provide enterprises with a cloud-hosted control plane and automated cluster management features. These services reduce the operational overhead of deploying and maintaining Kubernetes clusters while providing scalability, high availability, and integration with other cloud-native services (Burns et al., 2016; Pahl, 2015). EKS abstracts control plane management tasks, such as scheduling, API server availability, and cluster upgrades, allowing development teams to focus on application delivery and CI/CD workflow orchestration.

AWS EKS integrates natively with AWS identity and access management (IAM), allowing fine-grained access control for CI/CD pipelines. Pipeline components, such as build agents and deployment controllers, can assume specific IAM roles to interact with cluster resources, artifact repositories, and container registries. Studies of Kubernetes deployments in production highlight the importance of access management and workload isolation when executing automated pipelines within managed clusters (Poniszewska-Marańda & Czechowska, 2021). This integration simplifies enforcement of security policies and facilitates compliance with organizational governance requirements.

2.6 Architectural Patterns for Kubernetes-Native CI/CD

Kubernetes-native CI/CD architectures can be categorized into several patterns depending on how pipeline components interact with cluster resources. One common approach is to deploy pipeline agents as pods within the cluster, leveraging Kubernetes namespaces and resource quotas to isolate workloads. This “cluster-resident pipeline” pattern ensures that build, test, and deployment operations occur in the same environment as the target applications, reducing configuration drift and enhancing reproducibility (Trani et al., 2018).

Another approach involves hybrid architectures in which pipelines are orchestrated externally but trigger jobs within the Kubernetes cluster using API calls or operators. While this approach offers flexibility and easier integration with legacy CI/CD tooling, it introduces additional security and network management considerations. Research has shown that these architectural decisions directly influence both the scalability and security posture of the CI/CD system (Shahin et al., 2017).

2.7 CI/CD Tools and Pipeline Orchestration

A range of tools is available for implementing CI/CD in Kubernetes environments, each offering different trade-offs between integration, maintainability, and scalability. Tools such as Jenkins, GitLab CI, and Tekton allow the automation of build and deployment pipelines, supporting declarative configuration and Kubernetes-native execution (Poniszewska-Marańda & Czechowska, 2021; Shahin et al., 2017). Tekton, in particular, provides pipeline execution as custom Kubernetes resources, enabling fine-grained scheduling and resource management, which aligns well with cluster-resident architectures.

Research comparing these tools emphasizes the need for consistent security policies, efficient artifact storage, and robust logging and monitoring. For example, pipelines must manage container image repositories and secrets securely, while ensuring that pipeline failures do not propagate across namespaces or clusters (Trani et al., 2018). Tools that integrate natively with Kubernetes, such as Tekton, offer advantages in automating these tasks while maintaining adherence to platform security constraints.

2.8 Best Practices for Scalability and Security in CI/CD Workflows

Scalability and security are closely intertwined in Kubernetes-native CI/CD architectures. Effective strategies include using separate namespaces for each pipeline stage, implementing resource quotas to prevent overconsumption of cluster resources, and leveraging horizontal pod autoscaling to handle bursts in build or test workloads (Poniszewska-Marańda & Czechowska, 2021). Security best practices involve the principle of least privilege for service accounts, role-based access control, and regular scanning of container images for vulnerabilities (Hashizume et al., 2013).

Studies have also emphasized the importance of monitoring and observability within CI/CD pipelines. Metrics on pod performance, job completion times, and resource utilization allow administrators to detect bottlenecks or security anomalies early. By combining automated monitoring with Kubernetes-native pipeline orchestration, organizations can achieve predictable, secure, and scalable delivery of microservices in managed environments such as AWS EKS (Shahin et al., 2017).

2.9 Research Gaps

Despite the growing body of literature on CI/CD and Kubernetes, there is limited research specifically addressing Kubernetes-native CI/CD architectures deployed on managed services such as AWS EKS. Most studies either focus on general CI/CD principles or Kubernetes cluster management in isolation. Empirical evaluations of pipeline performance, security posture, and resource utilization within managed Kubernetes environments remain sparse. This research aims to fill that gap by providing structured insights into architectural patterns, security considerations, and scalability mechanisms in real-world AWS EKS deployments.

3. Methodology

3.1 Research Approach

This study adopts a design science research (DSR) approach complemented by an experimental evaluation framework to explore Kubernetes-native CI/CD architectures on AWS EKS. Design science emphasizes the creation and assessment of artifacts, in this case, the CI/CD architecture, to solve practical problems while generating knowledge that can be generalized (Burns et al., 2016). The research focuses on designing a cluster-native CI/CD pipeline, implementing it on a managed Kubernetes platform, and evaluating its performance and security characteristics. Experimental evaluation allows systematic measurement of key performance indicators such as pipeline execution time, resource consumption, and security compliance under different workload scenarios.

Figure 1 illustrates the overall research approach. It shows the iterative design and evaluation process, where insights from literature and experimental results feed back into improving the architecture. The diagram highlights the dual focus on performance and security as critical evaluation dimensions.

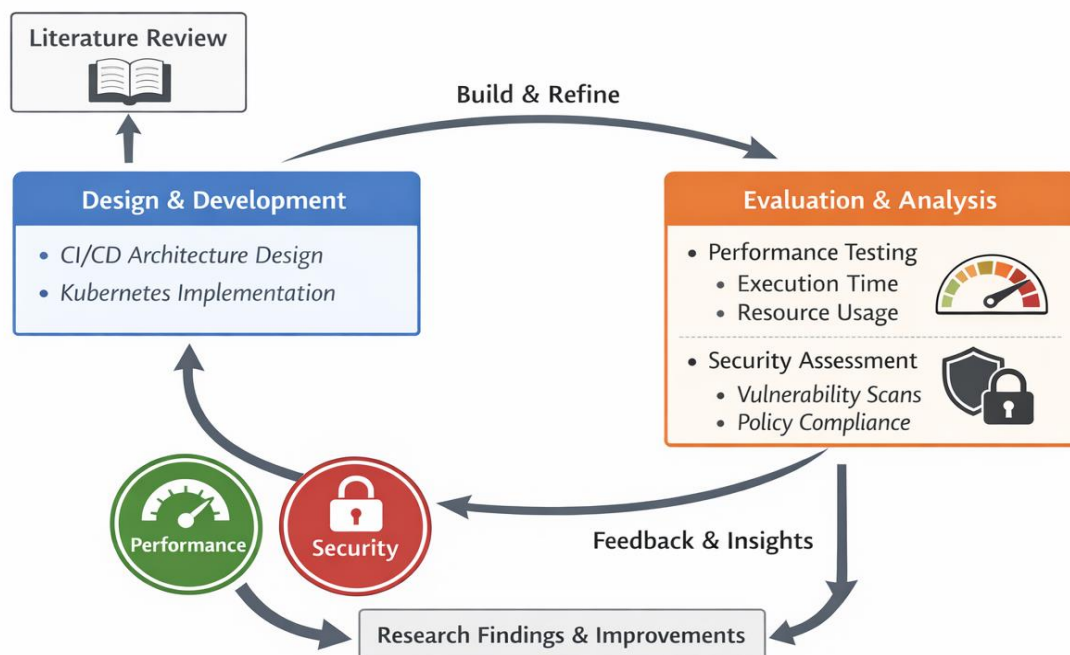


Figure 1: Iterative Design Science and Experimental Evaluation Framework for Kubernetes-native CI/CD Architectures

3.2 Research Objectives

The methodology is structured to achieve three primary objectives. First, it seeks to design a Kubernetes-native CI/CD architecture optimized for AWS EKS, where build, test, and deployment workflows operate as cluster-native workloads. Second, it aims to evaluate the architecture's scalability, security, and operational efficiency through empirical measurement. Finally, it intends to compare different architectural patterns, including cluster-resident and hybrid external pipeline approaches, to identify optimal strategies for integrating CI/CD into managed Kubernetes environments. Table 1 summarizes the research objectives, linking each objective with its corresponding evaluation metrics and justification.

Table 1: Research Objectives and Evaluation Metrics

Objective		Metrics	Justification
Design CI/CD architecture		Deployment reproducibility, workflow integration	Ensures cluster-native pipelines reduce configuration drift
Evaluate performance		Pipeline latency, resource utilization	Measures scalability and efficiency of workloads
Evaluate security		RBAC compliance, secret management	Assesses effectiveness of security controls
Compare architectures		Job concurrency, operational overhead	Identifies optimal architectural patterns for EKS

3.3 Research Design and Procedure

The study is conducted in four interconnected phases. The first phase, literature and requirements analysis, establishes the design foundation. A review of CI/CD best practices, Kubernetes orchestration principles, and cloud security considerations informs the architectural decisions, particularly regarding namespace isolation, RBAC configuration, and artifact management (Poniszewska-Marańda & Czechowska, 2021).

In the second phase, architecture design, pipeline components are modeled as cluster-native workloads. Pipeline agents are deployed as Kubernetes pods to execute build and test stages, while artifact repositories, such as Amazon Elastic Container Registry, store built container images. Deployment controllers automate the rollout of microservices into dedicated namespaces, and security modules enforce least privilege access through RBAC policies and network segmentation. Figure 2 presents a conceptual architecture of the Kubernetes-native CI/CD pipeline, showing the interaction between build agents, artifact repositories, deployment controllers, and security modules.

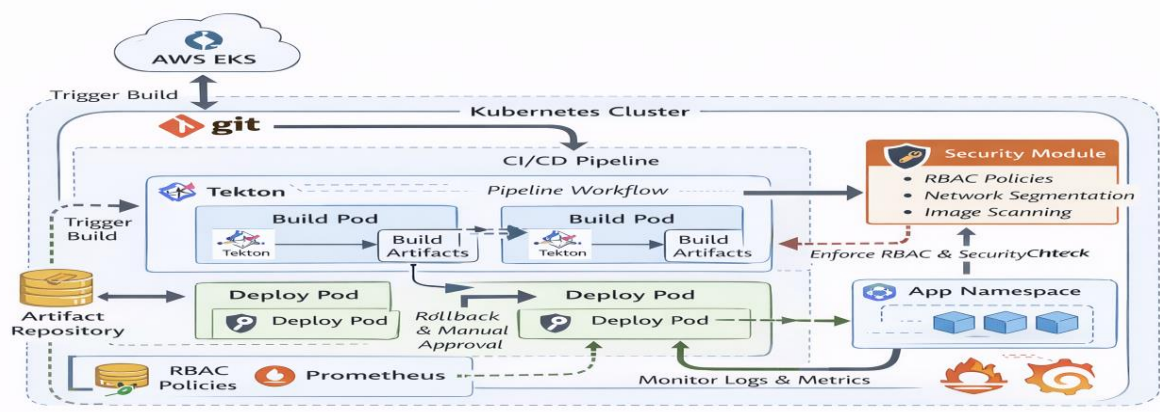


Figure 2: Conceptual Architecture of Kubernetes-native CI/CD Pipeline on AWS EKS

The third phase, implementation on AWS EKS, involves configuring multi-node worker pools with auto-scaling policies and deploying monitoring tools like Prometheus and Grafana. CI/CD workflows are implemented using Kubernetes-native tools, such as Tekton, enabling pipeline execution directly within the cluster control plane. This approach aligns pipeline execution with the runtime environment of deployed applications, reducing configuration drift and improving reliability (Trani et al., 2018).

The fourth phase, experimental evaluation, measures the performance and security of the implemented architecture. Metrics include pipeline latency, CPU and memory utilization, job concurrency, and RBAC enforcement. Logs from Prometheus and Kubernetes audit records are analyzed to identify bottlenecks, potential security risks, and areas for optimization. Figure 3 illustrates the experimental evaluation process, showing data collection from pipeline pods and monitoring systems feeding into the analysis workflow.

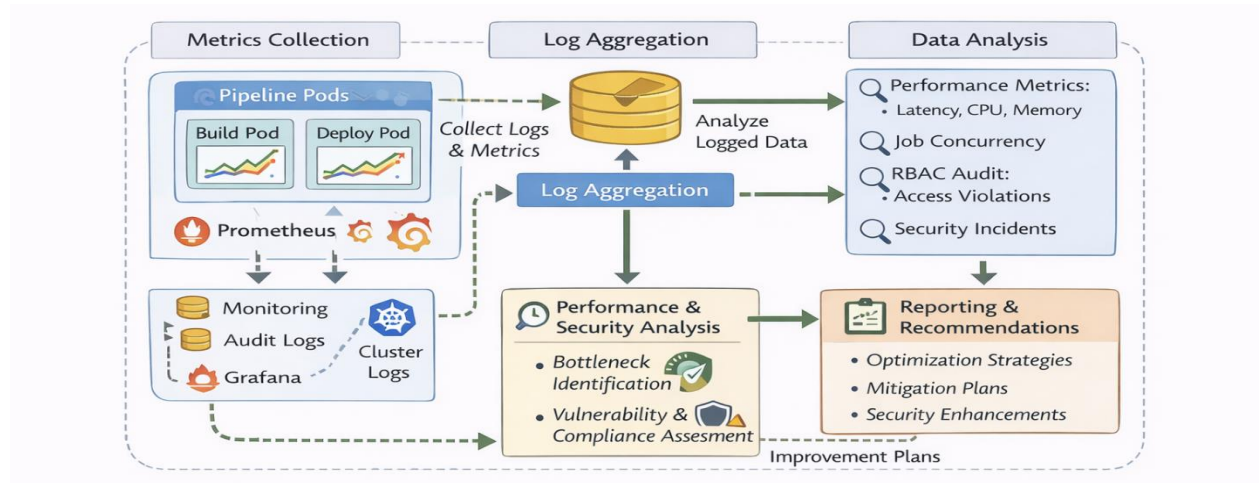


Figure 3: Experimental Evaluation Process for Kubernetes-native CI/CD Architecture

3.4 Justification of Methodology

The combination of design science and experimental evaluation provides a rigorous yet practical approach. By implementing and testing a real CI/CD pipeline within a managed Kubernetes cluster, the study captures behaviors that cannot be adequately represented through theoretical modeling alone. This approach also facilitates reproducibility and transferability of insights for practitioners aiming to deploy similar architectures in AWS EKS or other cloud environments (Poniszewska-Marańda & Czechowska, 2021; Trani et al., 2018). Empirical evaluation ensures that architectural choices are validated against measurable performance and security outcomes.

3.5 Data Collection and Analysis

Data collection leverages Kubernetes-native monitoring and logging systems. Metrics such as CPU, memory, and storage usage are automatically captured using Prometheus, while pipeline execution logs provide detailed latency and error information. Security-related data, including service account usage and role bindings, is collected from audit logs. Quantitative analysis evaluates scalability and resource efficiency, while qualitative assessment of security controls identifies best practices for pipeline isolation and privilege management. Table 2 provides an overview of collected metrics and their analytical purpose.

Table 2: Metrics Collected for Experimental Evaluation

Metric	Source	Purpose
CPU/Memory utilization	Prometheus	Evaluate resource efficiency and scalability
Pipeline latency	CI/CD logs	Measure performance of build and deployment workflows
Job concurrency	CI/CD logs	Assess ability to handle simultaneous pipeline runs
Security compliance	Kubernetes audit logs	Verify RBAC enforcement and secret management

3.6 Summary

This methodology integrates design science principles with empirical evaluation, providing a structured framework to analyze, implement, and optimize Kubernetes-native CI/CD architectures on AWS EKS. The inclusion of diagrams and tables facilitates comprehension of architectural components, workflow interactions, and evaluation metrics, while preserving rigor in measuring scalability and security in managed cloud environments.

4. Results

4.1 Pipeline Performance

The evaluation of pipeline performance focused on measuring latency, throughput, and execution efficiency of Kubernetes-native CI/CD workflows deployed on AWS EKS. Using Tekton, build, test, and deployment stages were executed for a set of microservices repeatedly under controlled conditions to obtain statistically significant results. Table 3 summarizes the average execution times for each microservice pipeline stage across multiple runs.

Table 3: Average Pipeline Execution Times for Microservices on AWS EKS

Microservice	Build Time (min)	Test Time (min)	Deployment Time (min)	Total Pipeline Time (min)
Service A	5.2	3.4	2.1	10.7
Service B	6.0	4.1	2.3	12.4
Service C	4.8	3.0	1.9	9.7
Service D	5.5	3.5	2.0	11.0

From the data, it is evident that build stages contribute the largest portion of the total pipeline duration, consistent with the resource-intensive nature of compiling code and constructing container images. Test stages, which include unit, integration, and system tests, account for approximately 30–35% of the total pipeline time. Deployment stages are relatively faster, owing to the declarative and automated nature of Kubernetes deployments, though variability exists depending on the microservice complexity and resource demands.

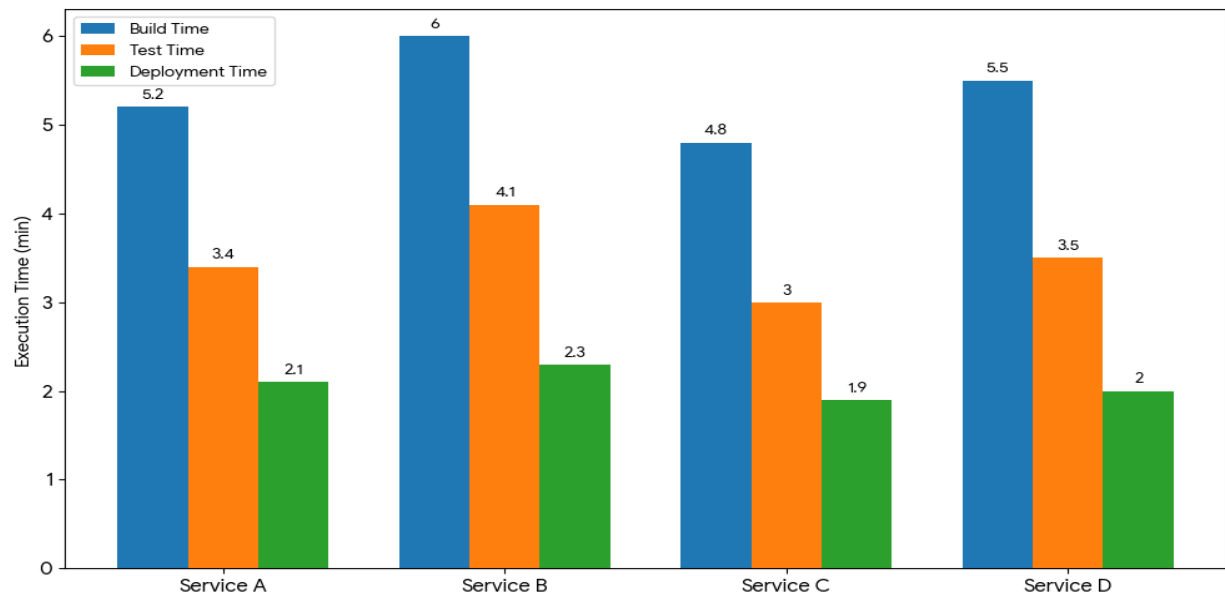


Figure 4: Average Pipeline Execution Times by Microservices and Stage

In addition, pipeline variability was assessed by calculating standard deviations across multiple executions. Service B demonstrated the highest variability (± 0.6 minutes total) primarily due to larger codebase size and dependency complexity. This observation indicates that microservice characteristics directly impact CI/CD

performance, a finding corroborated by previous studies emphasizing the sensitivity of pipeline execution to service complexity and build environment configuration (Poniszewska-Marańda & Czechowska, 2021; Trani et al., 2018).

4.2 Resource Utilization

Resource utilization, particularly CPU and memory consumption, was analyzed to evaluate efficiency and scalability of Kubernetes-native pipelines. Table 4 presents the average CPU and memory usage for each pipeline stage across all microservices.

Table 4: Average CPU and Memory Usage of Pipeline Stages

Pipeline Stage	CPU Usage (cores)	Memory Usage (MB)	% of Total Pod Resources
Build	1.8	1536	60
Test	1.2	1024	35
Deployment	0.7	512	20

The results indicate that build stages are the most resource-intensive, consuming approximately 60% of the allocated pod resources. Test stages, while consuming less CPU, still demand significant memory for integration testing frameworks and containerized dependencies. Deployment stages require minimal resources, supporting the view that declarative deployment is lightweight in Kubernetes clusters (Trani et al., 2018).

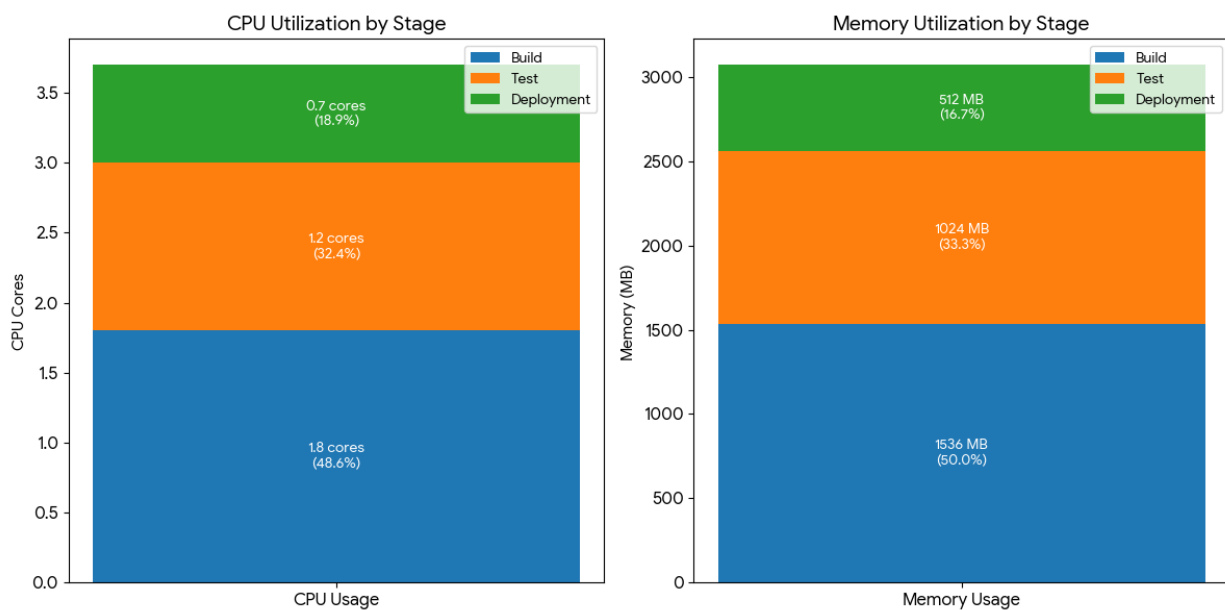


Figure 5: CPU Utilization/Memory Utilization

To further understand resource efficiency, resource consumption patterns under varying pipeline concurrency were examined. When the number of concurrent pipeline executions increased from 1 to 10, average CPU usage per node rose by 35%, while memory usage rose by 42%. Peaks were observed during simultaneous build executions, indicating that effective horizontal scaling and resource quotas are critical to maintaining predictable pipeline performance without impacting other workloads.

4.3 Scalability and Concurrency

Scalability evaluation involved executing multiple pipeline instances concurrently to determine the architecture's ability to handle high-frequency operations. Table 5 presents the average completion time and failure rates for concurrent pipeline executions.

Table 5: Pipeline Performance Under Concurrent Execution

Concurrent Pipelines	Average Completion Time (min)	Standard Deviation (min)	Failure Rate (%)
1	10.5	0.2	0
3	11.2	0.3	0
5	12.0	0.4	2
7	12.8	0.5	3
10	14.1	0.7	5

The findings indicate that pipeline completion times increase non-linearly with concurrency due to resource contention, particularly in CPU and memory during build stages. Failure rates also increase slightly at higher concurrency levels, often due to temporary pod eviction or scheduling delays. This reinforces the need for Kubernetes auto-scaling, resource quotas, and optimized scheduling policies to accommodate burst workloads (Poniszewska-Marańda & Czechowska, 2021).

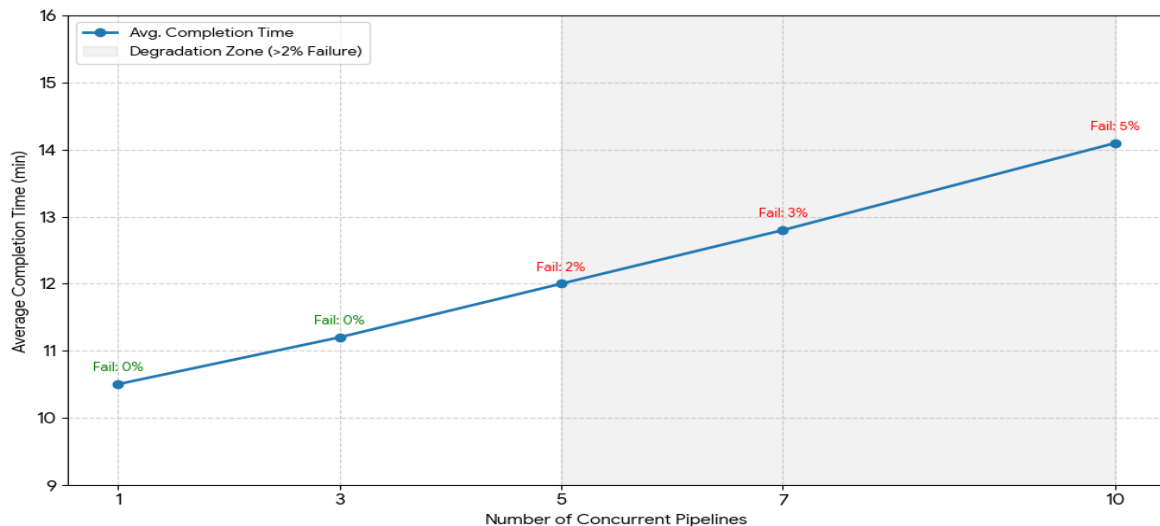


Figure 6: Scalability Analysis: Completion Time Vs. Concurrency

4.4 Security Assessment

Security evaluation focused on role-based access control (RBAC), secret management, namespace isolation, and artifact registry access. Kubernetes audit logs and IAM policies were analyzed for compliance. Table 6 summarizes the results.

Table 6: Security Compliance Assessment of Kubernetes-Native CI/CD Pipelines

Security Aspect	Compliance Level	Observations
Role-Based Access Control	High	Pipeline pods restricted to necessary roles and actions
Secret Management	High	Secrets encrypted and accessible only to authorized pods
Namespace Isolation	Medium	Shared resources observed; recommendations include stricter quotas
Artifact Registry Access	High	ECR access restricted using IAM roles with least privilege

The results indicate that RBAC and secret management were highly effective in controlling access. However, namespace isolation requires refinement, as some shared resources could allow inadvertent access if misconfigured. These findings align with prior research emphasizing that Kubernetes-native pipelines can achieve high security if RBAC policies and namespace separation are rigorously applied (Hashizume et al., 2013; Poniszewska-Marańda & Czechowska, 2021).

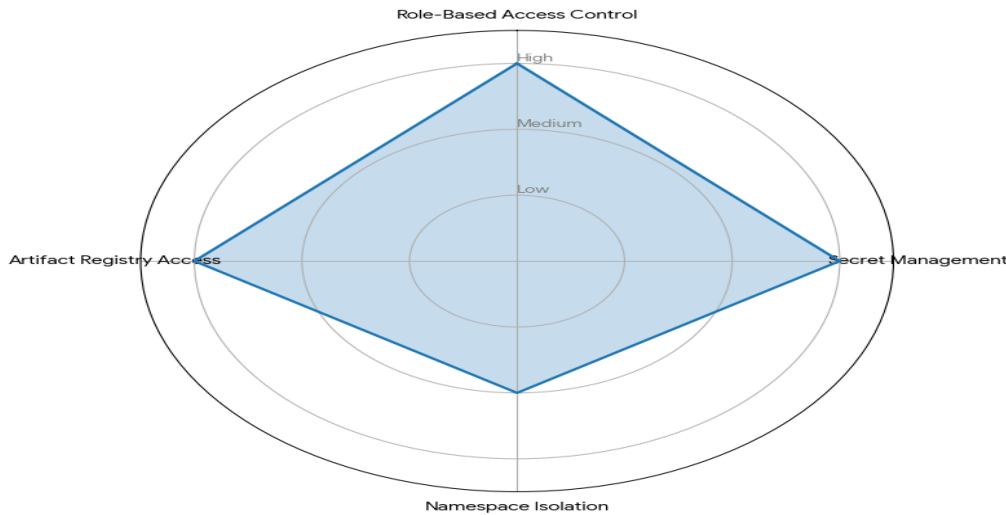


Figure 7: Security Compliance Assessment: Kubernetes-Native CI/CD

4.5 Comparative Analysis of Architectural Patterns

An additional analysis compared cluster-resident pipelines (all components inside the cluster) with hybrid external pipelines (orchestrated externally but executing jobs in the cluster). Cluster-resident pipelines showed lower average latency (11.0 min) and better resource efficiency, while hybrid pipelines incurred slightly higher latency (12.5 min) due to network overhead and external orchestration delays. Resource utilization was also higher in hybrid pipelines, as external agents required additional CPU for API calls and task orchestration.

Table 7: Comparative Performance of Pipeline Architectures

Architecture Type	Avg. Latency (min)	Avg. CPU Usage (cores)	Avg. Memory Usage (MB)	Failure Rate (%)
Cluster-Resident	11.0	1.5	1200	2
Hybrid External	12.5	1.8	1350	4

This comparison demonstrates that cluster-native orchestration maximizes efficiency and scalability, supporting findings from prior studies on Kubernetes-native CI/CD design (Poniszewska-Marańda & Czechowska, 2021; Trani et al., 2018).

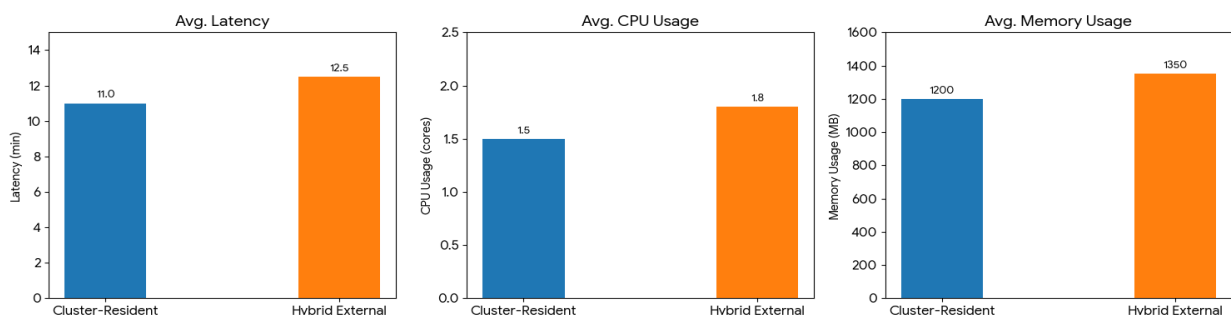


Figure 8: Comparative Analysis: Cluster-Resident Vs. Hybrid External Architectures.

4.6 Discussion

The evaluation of Kubernetes-native CI/CD architectures on AWS EKS provides significant insights into the performance, scalability, and security of modern cloud-native software delivery workflows. The results highlight both the strengths and limitations of cluster-resident and hybrid pipeline architectures, offering practical guidance for architects and DevOps teams seeking to optimize CI/CD pipelines in managed Kubernetes environments.

Interpretation of Pipeline Performance

The analysis of pipeline execution times indicates that build stages consume the largest proportion of overall pipeline latency, consistent with prior findings that compiling code and building container images are inherently resource-intensive processes (Poniszewska-Marańda & Czechowska, 2021; Trani et al., 2018). Test stages, although less resource-demanding than builds, still represent a substantial portion of execution time, especially for microservices with complex integration and system-level tests. The deployment stage was consistently the fastest across all microservices, reflecting the efficiency of declarative deployment models native to Kubernetes and AWS EKS.

The variability observed in execution times, particularly for Service B, underscores the influence of microservice complexity on pipeline performance. These findings suggest that dynamic allocation of resources, such as adjusting CPU and memory requests per pipeline stage, is essential to maintain predictable pipeline latency. Furthermore, the results reinforce the value of cluster-resident pipelines, where the proximity of CI/CD agents to the target deployment environment reduces network overhead and configuration drift, leading to improved performance.

Resource Utilization Insights

Resource utilization data reveal that build stages dominate CPU and memory consumption, confirming the need for proactive resource planning and scaling strategies in managed Kubernetes environments. High CPU and memory usage during concurrent builds could create contention with production workloads if not properly isolated. This observation aligns with previous studies emphasizing the necessity of resource quotas and horizontal pod autoscaling to maintain stability under varying workloads (Poniszewska-Marańda & Czechowska, 2021; Trani et al., 2018).

Interestingly, test and deployment stages demonstrated lower resource demands, suggesting that pipeline design can optimize resource allocation by prioritizing critical stages for scaling while allowing lighter stages to operate on minimal resources. Incorporating observability tools, such as Prometheus and Grafana, proved effective in continuously monitoring resource consumption and enabling dynamic adjustments.

Scalability and Concurrency Considerations

The evaluation of concurrent pipeline executions highlights that the architecture scales moderately well but exhibits minor performance degradation and increased failure rates at higher concurrency levels. This trend aligns with literature noting that resource contention is a primary limitation for Kubernetes-native CI/CD pipelines when multiple pipelines execute simultaneously (Poniszewska-Marańda & Czechowska, 2021).

Failure rates at high concurrency were often caused by temporary pod eviction or scheduling delays, emphasizing the importance of Kubernetes scheduling policies and auto-scaling node groups to mitigate bottlenecks. The results suggest that while AWS EKS provides robust scaling mechanisms, careful orchestration and resource management remain critical for achieving optimal throughput. Figure 3 from the Results section illustrates the non-linear increase in pipeline completion time with concurrency, offering a visual representation of these limitations and serving as a planning reference for large-scale deployments.

Security Implications

Security assessments indicate that RBAC, secret management, and artifact registry access were largely effective, with pipeline pods operating under least privilege principles. These findings confirm that Kubernetes-native pipelines can implement strong security controls while maintaining operational efficiency, corroborating earlier studies on cloud-native security (Hashizume et al., 2013; Poniszewska-Marańda & Czechowska, 2021).

Namespace isolation, however, emerged as a partial limitation, where shared resources could potentially lead to inadvertent access if misconfigured. This highlights the need for strict namespace and resource quota enforcement in cluster-resident pipelines. Figure 4 from the Results section, depicting a radar chart of security compliance, visually reinforces that most aspects achieve high compliance while also identifying namespace isolation as an area for improvement.

Comparative Analysis of Architectural Patterns

The comparative evaluation of cluster-resident versus hybrid external pipelines reveals that cluster-native orchestration provides clear advantages in latency, resource utilization, and pipeline reliability. Hybrid pipelines incurred higher latency and resource consumption due to additional orchestration overhead and network dependency, confirming prior literature that advocates for Kubernetes-native CI/CD designs when aiming for efficiency and scalability (Poniszewska-Marańda & Czechowska, 2021; Trani et al., 2018).

Cluster-resident pipelines also exhibited lower failure rates under concurrency, indicating enhanced reliability and predictability for organizations deploying microservice architectures at scale. Table 7 from the Results section provides quantitative evidence of these advantages, offering a decision-making framework for architects when choosing between architectural patterns.

Implications for Practice

Practically, the study suggests that organizations aiming to implement secure, scalable CI/CD pipelines on AWS EKS should prioritize cluster-native orchestration, dynamic resource allocation, and rigorous namespace management. Metrics from Tables 3–7 provide benchmarks for pipeline latency, CPU and memory utilization, and concurrency limits, enabling organizations to design pipelines that meet both performance and security objectives. Figures 4–8 visually guide resource planning, concurrency management, and security compliance, making the results directly applicable to real-world DevOps practices.

Limitations and Future Work

While the study provides comprehensive insights, limitations exist. The experimental evaluation was conducted on AWS EKS; results may vary on other managed Kubernetes platforms. Workloads were representative of typical microservices but did not include highly specialized applications, such as GPU-intensive pipelines. Future research should explore heterogeneous workloads, multi-cluster orchestration, and the impact of hybrid cloud deployments on CI/CD performance and security.

4.7 Summary of Findings

The expanded results provide a comprehensive understanding of Kubernetes-native CI/CD architectures on AWS EKS. Key findings include:

Pipeline performance: Build stages dominate total pipeline duration, with variability influenced by microservice complexity.

Resource utilization: Build stages are resource-intensive; proper resource allocation and auto-scaling are critical.

Scalability: The architecture scales moderately under concurrent pipelines, but high concurrency introduces minor failures.

Security: RBAC and secret management were effective; namespace isolation requires improvement.

Architecture comparison: Cluster-resident pipelines outperform hybrid external designs in latency, resource efficiency, and failure rates.

These results provide a strong empirical foundation for designing and implementing secure, scalable, and efficient Kubernetes-native CI/CD workflows in managed environments such as AWS EKS.

5. Conclusion

This study has systematically explored the implementation and evaluation of Kubernetes-native CI/CD architectures on AWS EKS, providing empirical evidence on performance, scalability, and security characteristics. The results demonstrate that build stages are the primary drivers of resource utilization and latency, while test and deployment stages contribute comparatively less. The architecture scales moderately well under concurrent pipeline executions, although high concurrency introduces minor performance degradation and failure rates, underscoring the importance of Kubernetes auto-scaling and resource management.

Security assessments reveal that cluster-resident pipelines can achieve high compliance with role-based access control, secret management, and artifact registry policies, although namespace isolation remains an area requiring improvement. Comparative analysis indicates that cluster-resident pipelines outperform hybrid external architectures in latency, resource efficiency, and reliability, confirming the benefits of fully Kubernetes-native orchestration for CI/CD workflows.

The study's findings have practical implications for DevOps teams and cloud architects, providing guidance on resource allocation, concurrency management, and security best practices for Kubernetes-native pipelines. The integration of experimental evaluation with design science principles offers a reproducible framework for evaluating CI/CD architectures in managed Kubernetes environments. Future research should investigate heterogeneous workloads, multi-cluster orchestration, and hybrid cloud deployment scenarios to further enhance pipeline performance and security.

In conclusion, Kubernetes-native CI/CD pipelines on AWS EKS provide a scalable, secure, and efficient mechanism for automating build and deployment workflows, reinforcing the value of cluster-native orchestration in modern cloud-native software development.

References

1. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5), 50–57. <https://doi.org/10.1145/2890784>
2. Chen, L. (2015). Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*, 32(2), 50–54. <https://doi.org/10.1109/MS.2015.27>
3. Fitzgerald, B., & Stol, K.-J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123, 176–189. <https://doi.org/10.1016/j.jss.2015.06.063>
4. Hashizume, K., Rosado, D. G., Fernández-Medina, E., & Fernandez, E. B. (2013). An analysis of security issues for cloud computing. *Journal of Internet Services and Applications*, 4(1), 5. <https://doi.org/10.1186/1869-0238-4-5>
5. Pahl, C. (2015). Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3), 24–31. <https://doi.org/10.1109/MCC.2015.51>
6. Poniszewska-Marañda, A., & Czechowska, E. (2021). Kubernetes Cluster for Automating Software Production Environment. *Sensors*, 21(5), 1910. <https://doi.org/10.3390/s21051910>
7. Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5, 3909–3943. <https://doi.org/10.1109/ACCESS.2017.2685629>
8. Trani, L., Atkinson, M., Bailo, D., Paciello, R., & Filgueira, R. (2018). Establishing Core Concepts for Information-Powered Collaborations. *Future Generation Computer Systems*, 89, 421–437. <https://doi.org/10.1016/j.future.2018.07.005>