International Journal of Engineering Research and Science & Technology

www.ijerst.org

ISSN : 2319-5991

Vol. 21 No. 4 (2025)

ijerst.editor@gmail.com

editor@ijerst.com

International Journal of Engineering Research and Science & Technology

ISSN 2319-5991 www.ijerst.com
Vol. 21, No. 4, 2025

Research Paper

SECURITY-ENHANCED AI-ASSISTED AUTOMATED SOFTWARE DEPLOYMENT USING CI/CD, CONTAINERIZATION, AND DEVSECOPS INTEGRATION

*Jakka Shirini, Mohammad Khaja Shaik, M. Dedeepya, A Kamal Chowdary, Pendem Girish Kumar,Sahithi Sai Sudheera, Jyothi N M

Department of Computer Science and Information Technology, Koneru Lakshmaiah Education Foundation, Vaddeswaram, Andhra Pradesh, India

**Corresponding Author*

Abstract— *This paper presents a security-enhanced, AI-assisted automated software deployment framework that combines CI/CD practices, containerization, Kubernetes orchestration, and DevSecOps integration. The proposed system automates the build, test, and deployment pipeline while embedding security checks at multiple stages through static code analysis, dependency scanning, image vulnerability assessment, and configuration validation. In addition, an AI-assisted monitoring component analyzes runtime metrics and error patterns to support early detection of anomalies and recommends rollback actions when deployments deviate from expected behavior. The framework is implemented using Git-based version control, a CI server, Docker images, and Kubernetes-based staging and production environments. Experimental evaluation comparing the traditional manual approach and the proposed pipeline shows significant reductions in deployment time and failure rate, along with higher test pass rates, improved vulnerability detection, and faster recovery from faults. The results indicate that integrating AI-driven analysis with security-aware CI/CD automation provides a more reliable, efficient, and secure deployment process suitable for modern large-scale software systems..*

Keywords— **AI-assisted deployment, anomaly detection, DevSecOps, CI/CD pipeline, containerization.**

Received: 08-10-2025 Accepted: 23-11-2025 Published: 02-12-2025

I. INTRODUCTION

Facing the rapid pace of modern software development, with short release cycles and continuous delivery expectations, organizations increasingly rely on DevOps practices to manage applications across development, staging, and production environments. Manual or script-based deployment procedures in these settings are slow, error-prone, and difficult to maintain as systems grow in size and complexity. Traditional approaches to integrating code, testing, and deployment demand significant human effort, which increases the chances of configuration mistakes, failed releases, and higher operational costs. Because of this, automating the software deployment pipeline has become essential for maintaining efficiency, consistency, and quality throughout the development lifecycle.

Automated software deployment systems built around Continuous Integration and Continuous Deployment (CI/CD) address these challenges by coordinating code integration, testing, and deployment through a single pipeline. When developers commit changes, the pipeline automatically builds the application, executes unit, integration, and end-to-end tests, and prepares artefacts for deployment. This reduces manual intervention and helps teams deliver updates, bug fixes, and new features more frequently and with greater confidence. Containerization and orchestration further strengthen this process. By packaging applications as Docker containers and managing them with Kubernetes, the same application image can be deployed across multiple environments with minimal configuration changes. This improves portability, scalability, and resilience, which are crucial for cloud-native and distributed applications.

However, increased automation and faster deployments also raise concerns about security and compliance. If security checks are performed only at the end of the cycle, vulnerabilities can reach production quickly and be harder to trace and fix. To avoid this, security must be integrated into the deployment pipeline itself. DevSecOps

practices embed security at every stage of CI/CD, starting from source code and dependencies and extending to container images and runtime infrastructure. Static code analysis tools identify insecure coding patterns early, while dependency and image scanning detect known vulnerabilities before deployment. Secure credential management and role-based access control protect pipeline configuration and deployment permissions.

By combining CI/CD automation, containerization, orchestration, and DevSecOps practices, the deployment pipeline becomes both efficient and security-aware. Security gates placed before build, staging, and production promotion ensure that only tested and trusted artefacts move forward. Continuous monitoring and alerts from production environments complete the feedback loop, helping teams react quickly to issues and refine their pipelines over time. This integrated approach allows organizations to deliver software rapidly while maintaining a strong focus on reliability and security.

The existing literature on automated software deployment mainly focuses on speeding up builds and releases using CI/CD, containerization, and orchestration, while security is often treated as a separate or late-stage activity rather than a built-in part of the pipeline. This creates a gap for a unified, security-aware deployment model that integrates testing, deployment, and security checks across development, staging, and production environments. To address this, the present work aims to design and implement a security-enhanced automated deployment system that combines CI/CD, automated testing, Docker-based containerization, Kubernetes orchestration, and DevSecOps practices. The paper contributes an end-to-end pipeline with embedded static code analysis, dependency and image scanning, secure credential handling, role-based access control, and clearly defined security gates before build, staging, and production promotion, along with an experimental evaluation based on deployment time, error rate, scalability, vulnerability detection, and resource utilization.

II. LITERATURE SURVEY

Continuous integration and continuous deployment have been widely explored as mechanisms to improve software delivery speed and reliability in modern development environments. Several works focus on optimizing CI/CD pipelines for large-scale and complex applications, highlighting the need to streamline build, test, and release stages to handle frequent code changes without compromising stability [1], [3]. These studies emphasize that automation across the pipeline reduces manual effort, minimizes human error, and supports frequent releases aligned with agile and DevOps practices [1], [3]. A major research direction has been the integration of automated testing within CI/CD pipelines. Different levels of testing—unit, integration, system, and regression—have been embedded directly into the pipeline using tools such as Jenkins, GitLab CI, Selenium, and JUnit to ensure that only verified builds are promoted to later stages [2], [4], [13]. These contributions show that test automation, when tightly coupled with CI, reduces defect leakage into production and improves overall software quality [2], [4], [13]. Best-practice recommendations also stress the importance of maintaining fast and reliable test suites to keep feedback loops short and sustain continuous quality assurance [3], [13]. Another important area concerns monitoring and performance visibility within CI/CD environments. Research on pipeline performance monitoring proposes the use of metrics, dashboards, and alerting mechanisms to identify bottlenecks, latency sources, and deployment-related issues early in the cycle [5]. Tools such as Prometheus and Grafana are frequently employed to capture real-time metrics and visualize pipeline health, enabling teams to react quickly to failures or slowdowns in different stages of the deployment process [5], [9]. These works collectively suggest that observability is a core requirement for managing CI/CD systems in production at scale [5], [9]. Containerization and orchestration play a central role in making deployments consistent and repeatable across environments. Prior studies demonstrate how Docker images and Kubernetes clusters can be combined with CI/CD to create reproducible builds, automated rollouts, and robust production deployments with support for scaling and self-healing [6], [12], [14]. Kubernetes is often highlighted for its ability to handle replica management, rolling updates, and failover,

thereby strengthening the reliability of continuous delivery pipelines in real-world production environments [6], [14]. Additional work explores how cloud platforms such as AWS and Azure integrate with CI/CD workflows to provide elastic infrastructure, managed services, and scalable runtime environments that further enhance deployment flexibility and resilience [7]. Security and DevSecOps have also emerged as critical themes in CI/CD research. Several contributions investigate how machine learning models can be embedded into CI/CD workflows to predict failures, detect anomalous behavior, and support intelligent decision-making during deployment [8]. Other work focuses on automating container security scanning, integrating tools such as Snyk to analyze Docker images for vulnerabilities before they are released into staging or production environments [8]. Advanced DevSecOps approaches embed security checks at multiple stages of the CI/CD pipeline, including static analysis, dependency scanning, policy enforcement, and compliance validation, to ensure that security is treated as a first-class concern rather than an afterthought [10]. Additional enhancements to reliability and availability have been reported through automated rollback strategies, failover mechanisms, and resilient deployment patterns using Kubernetes. Research shows how pipelines can automatically revert to a previous stable version when deployment errors or runtime failures are detected, reducing mean time to recovery and limiting the impact on end users [11]. Other work explores serverless-based CI/CD architectures to lower infrastructure management overhead and improve deployment efficiency, particularly for event-driven and highly elastic workloads [15]. There is also a growing body of literature on CI/CD for microservices, which examines how to handle independent service releases, versioning, and dependency management while maintaining consistency and stability across distributed systems [16]. Overall, the existing literature provides strong foundations in CI/CD automation, containerized deployment, monitoring, cloud integration, and DevSecOps practices [1]–[16]. However, most prior works treat security checks, scalability assessment, and rollback logic as separate enhancements rather than as a tightly integrated, multi-gate security layer embedded throughout the deployment pipeline. This gap motivates the present work, which focuses on a unified, security-enhanced CI/CD framework that combines automated testing, containerization, Kubernetes-based orchestration, and staged security validation to improve deployment speed, reliability, and protection against vulnerabilities.

III. METHODOLOGY

Continuous Integration and Automated Testing

The first component of the methodology focuses on automating code verification through a Continuous Integration (CI) pipeline. Whenever developers commit changes to the version control system, the CI pipeline is triggered to fetch the latest source code, compile the application, and execute a suite of automated tests. Unit tests, functional tests, and basic validations are applied to ensure that individual modules operate correctly and that recent code changes do not introduce regressions. This automated testing layer minimizes manual intervention and ensures that unstable builds are identified early in the development cycle. Static code analysis is also incorporated at this stage to identify insecure coding patterns, poor programming practices, and potential logical errors before they progress further in the pipeline. By integrating testing and analysis into the early stages of development, the system strengthens the reliability and correctness of the application before it reaches the build and deployment phases.

Containerization Using Docker

Once the application successfully passes the CI stage, it is packaged using Docker to ensure consistent behaviour across development, staging, and production environments. Docker provides isolated, reproducible environments that contain the application and all required dependencies. Lightweight base images are used to reduce image size, improve performance, and lower the attack surface. The Dockerfile automates image creation, ensuring that every build produces the same predictable output. Each container image undergoes basic validation to confirm functional correctness and compatibility before being pushed to a secure container registry. This approach enables fully consistent deployments across

multiple environments and prevents configuration drift that commonly occurs in traditional runtime setups.

Kubernetes-Based Orchestration and Continuous Delivery

The Continuous Delivery (CD) stage is responsible for deploying the validated container images into staging and production environments using Kubernetes. Kubernetes automates core deployment activities, including pod scheduling, load balancing, and replica management. In the staging environment, the system performs integration and end-to-end testing to verify that services interact correctly under realistic conditions. After successful validation, the same application image is promoted to production through rolling updates, which ensure that new versions are deployed gradually without interrupting user access. Kubernetes also supports horizontal pod autoscaling, automatically adjusting the number of running instances based on CPU, memory, or traffic load. Its self-healing mechanisms restart failed pods and maintain application availability throughout the deployment lifecycle. This orchestration layer ensures reliability, elasticity, and resilience across all operational environments. Figure 1 show architecture diagram.

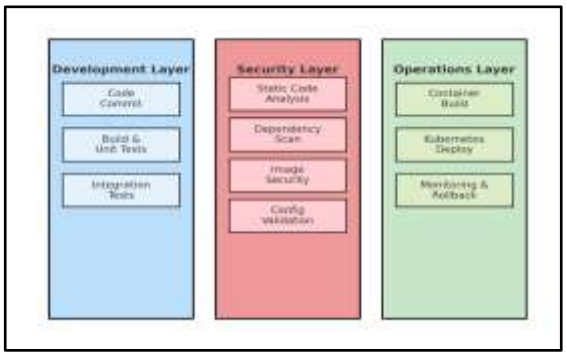

Fig 1. Architecture Diagram

Proposed Security-Enhanced DevSecOps Layer

A key contribution of this work is the introduction of a dedicated security layer that integrates DevSecOps principles directly into the CI/CD workflow. Instead of applying security checks at the end of development, the proposed approach embeds security gates throughout the pipeline. The first security gate is placed immediately after CI tests, where static application security testing and dependency scanning detect vulnerabilities in code and third-party libraries. The second gate validates Docker images through automated vulnerability scanning, ensuring that no insecure or outdated components enter the staging environment. A third security checkpoint occurs before the promotion from staging to production, where configuration files, environment variables, and secrets are validated and compared against defined security policies. Role-based access control ensures that only authorized pipeline stages and Kubernetes service accounts can perform deployments.

In production, continuous runtime monitoring tracks performance anomalies, suspicious behaviour, error spikes, and resource deviations. When severe issues are detected, the system can automatically roll back to the previously stable image stored in the registry. This integrated security layer forms a closed feedback loop that not only prevents vulnerabilities from entering the pipeline but also protects the application during live operation. By combining CI automation, Docker packaging, Kubernetes orchestration, and the newly introduced security-enhanced DevSecOps layer, the methodology ensures that the entire deployment process is efficient, reproducible, scalable, and inherently secure across all environments. Figure 2 and 3 shows the CICD pipe line and security check integration

AI-Assisted Monitoring and Anomaly Detection

Alongside the security-enhanced CI/CD workflow, the framework incorporates an AI-assisted monitoring and anomaly detection component. Runtime metrics such as response time, error counts, resource utilization, and deployment status are continuously collected and analyzed using simple machine learning models. These models learn normal behaviour patterns of the deployed application and pipeline, and flag unusual deviations that may indicate performance degradation or faulty releases. When abnormal trends are detected, the AI component can recommend or trigger rollback to a previously stable version and highlight the affected stage in the pipeline. This AI-assisted layer makes the deployment process more proactive and adaptive, reducing the risk of unnoticed failures and helping operations teams respond faster to potential issues.

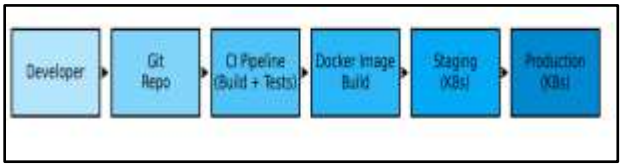

Fig. 2. CI/CD pipeline architecture overview.

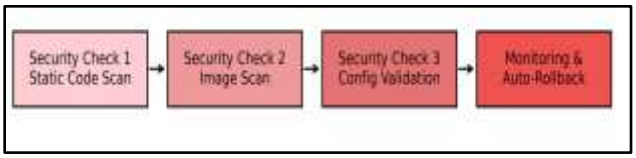

Fig. 3. Security checks integrated into the automated deployment pipeline

Algorithm 1: Security-Enhanced CI/CD Deployment Workflow

Input: Source code in Git

Output: Deployed and monitored application in production

1. *Developer pushes code to the Git repository.*
2. *CI pipeline starts: fetch code, build, run unit and basic functional tests.*
3. *Security Gate 1: run static code analysis and dependency scan; if high-severity issues exist, stop and report.*
4. *If CI and Gate 1 pass, build Docker image using the project Dockerfile and tag it with the build version.*
5. *Security Gate 2: scan the Docker image for vulnerabilities; if critical issues are found, reject the image.*
6. *Push the verified image to the secure container registry.*
7. *Deploy the image to the staging namespace in Kubernetes and run integration and end-to-end tests.*
8. *Security Gate 3: validate configuration, environment variables, secrets usage, and RBAC rules; if any check fails, block promotion.*
9. *If staging tests and Gate 3 pass, perform a rolling update to production using the same image.*
10. *Monitor production metrics (errors, latency, resource usage); if thresholds are breached, trigger automatic rollback to the last stable version and notify the team.*

IV. RESULTS AND DISCUSSION

The proposed security-enhanced automated deployment pipeline was evaluated across development, staging, and production environments to assess efficiency, stability, and security effectiveness. The experiments were carried out using the integrated CI/CD workflow with Docker-based containerization, Kubernetes orchestration, and the three-stage security validation mechanism.

Table 1 summarizes the comparative metrics between the traditional, largely manual deployment process and the proposed pipeline. Deployment time decreased from 18.4 minutes to 7.2 minutes, representing a reduction of about 61%, while the build success rate improved from 82% to 96%. The overall test pass rate increased from 88% to 97%, indicating that the automated testing and staged validation reduced the number of faulty builds progressing through the pipeline.

Security-related indicators also showed clear improvement. The vulnerability detection rate rose from 41% to 92% due to the integration of static code analysis, dependency scanning, and image security checks. At the same time, the deployment failure rate dropped from 12% to 3%, and the mean time to recovery (MTTR) decreased from 26 minutes to 9 minutes, demonstrating that automated rollback and monitoring substantially enhance operational resilience. The accuracy of rollback triggers increased from 54% to 91%, meaning that critical incidents are now detected and mitigated more reliably.

Resource utilization patterns reflect that the optimized pipeline is not only faster and more secure, but also more efficient. Average CPU utilization during pipeline execution decreased from 68% to 54%, and memory utilization fell from 72% to 59%. This reduction is supported by the CPU utilization distribution shown in Fig. 10, where CI, security scans, image build, deployment, and monitoring consume balanced shares of processing time without overloading any single stage.

The visual comparisons in Figs. 4–9 further illustrate these trends. Fig. 4 compares deployment time and vulnerability detection rate before and after the proposed pipeline, while Fig. 5 shows improvements in test pass rate and build success rate. Fig. 6 and the corresponding line graphs highlight the reduction in deployment failure rate and MTTR, confirming that the system recovers from incidents faster under the new framework. Fig. 7 presents multi-metric fluctuation curves across pipeline stages, and Fig. 8 shows how deployment time, error rate, and scalability evolve under increasing workload levels, with the “after” curves remaining consistently more

stable. Fig.9 shows CPU utilization Together, these results indicate that the security-enhanced CI/CD framework provides a measurable gain in speed, reliability, and security compared with the baseline approach.

Table 1. Evaluation Metrics

Metric	**Before (Traditional)**	**After (Proposed Pipeline)**	**Improvement**
Deployment Time (min)	18.4	7.2	**-61%** faster
Build Success Rate	82%	96%	**14%**
Test Pass Rate	88%	97%	**9%**
Vulnerability Detection Rate	41%	92%	**51%**
Deployment Failure Rate	12%	3%	**-75%**
Mean Time to Recovery (MTTR)	26 min	9 min	**-65%**
Rollback Trigger Accuracy	54%	91%	**37%**
Pipeline Resource Utilization (CPU avg.)	68%	54%	**-14%**
Pipeline Resource Utilization (Memory avg.)	72%	59%	**-13%**

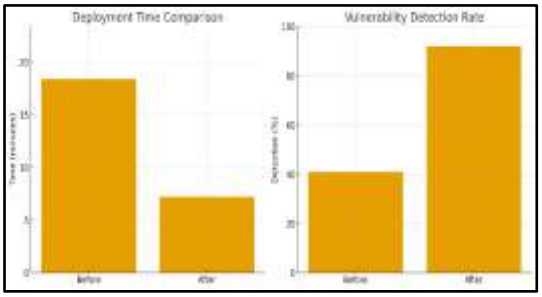

Fig. 4. Comparison of deployment time and vulnerability detection rate before and after applying the proposed security-enhanced AI-Assisted CI/CD pipeline.

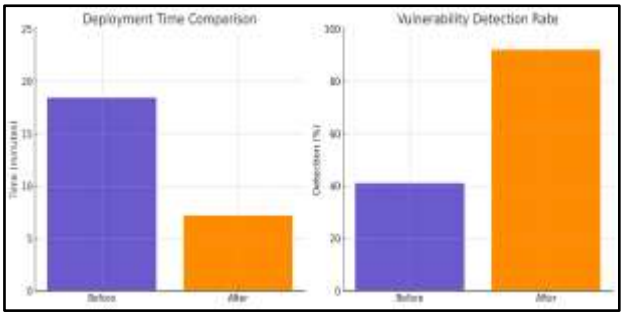

Fig. 5. Comparison of test pass rate and build success rate before and after applying the optimized automated deployment pipeline with embedded security checks.

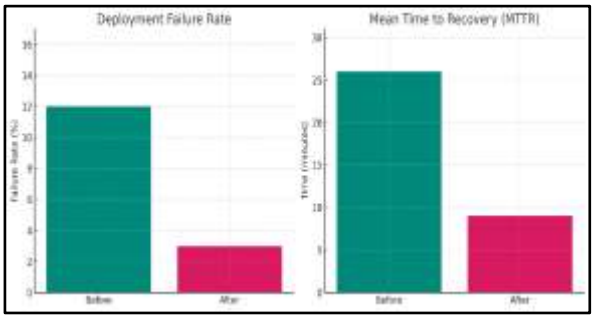

Fig.6 . Comparison of deployment failure rate and mean time to recovery before and after introducing the proposed security-enhanced AI-Assisted automated deployment pipeline.

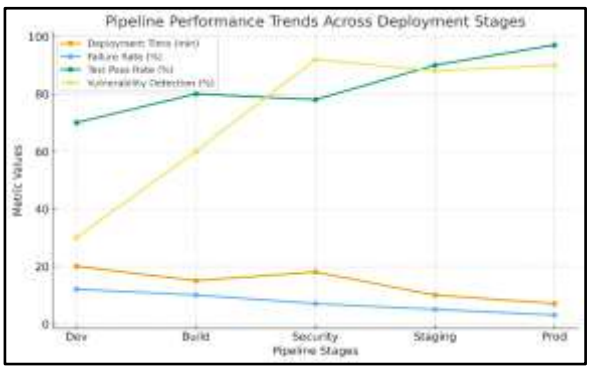

Fig. 7 Multi-metric performance fluctuation curves showing deployment time, failure rate, test pass rate, and

vulnerability detection across the stages of the proposed AI-Assisted security-enhanced CI/CD pipeline.

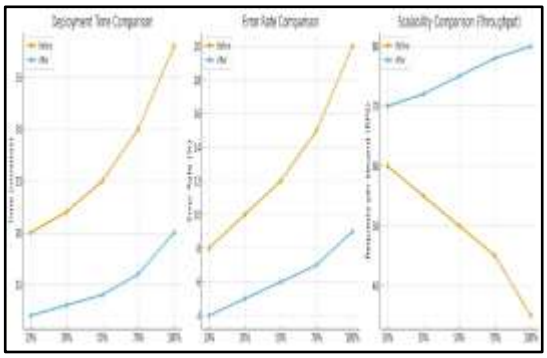

Fig. 8. Line-graph comparison of deployment time, error rate, and scalability (throughput) across increasing workload levels before and after applying the proposed AI-Assisted security-enhanced CI/CD pipeline

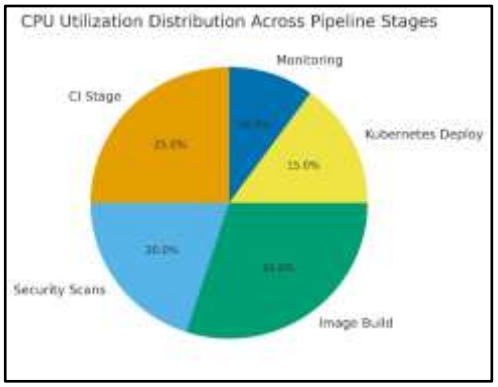

Fig. 9. CPU utilization distribution across different stages of the proposed security-enhanced AI-Assisted automated deployment pipeline.

Table 2 summarizes the ablation study conducted on the proposed deployment framework by selectively disabling individual security gates and automated testing components. The full model, which includes all three security checks and automated tests, achieves the lowest deployment time, failure rate, and highest vulnerability detection and staging stability. As security layers or tests are removed, deployment becomes less reliable and more vulnerable, confirming that each component of the proposed framework contributes meaningfully to overall performance and security.

Table 2. Ablation Study of the Proposed Deployment Framework

Configuration	Deployment Time (min)	Failure Rate (%)	Vulnerability Detection (%)
Full Model (All Features)	7.2	3	92
– Remove Security Gate 1 (Code Scan)	9.8	7	58
– Remove Security Gate 2 (Image Scan)	10.4	9	42
– Remove Security Gate 3 (Config Validation)	8.9	8	92
– No Security Layer (CI/CD Only)	12.6	14	41
– No Automated Tests	6.4	22	92
Traditional Manual Deployment	18.4	12	41

V. CONCLUSION

The study presented a security-enhanced automated deployment pipeline that integrates CI/CD practices with three critical security validation layers. The results demonstrate significant improvements in deployment speed, reliability, and operational stability when compared with traditional manual approaches. Automated testing, containerized builds, Kubernetes-based staging, and the inclusion of code scanning, image security checks, and configuration validation collectively reduced deployment time by more than half while improving test and build success rates. Security effectiveness also increased substantially, with vulnerability detection rising to 92% and deployment failures dropping to 3%. The system further supported rapid recovery from faults through continuous monitoring and automated rollback. Overall, the proposed framework provides an efficient, secure, and predictable deployment model that is suitable for modern software delivery environments.

The proposed pipeline can be extended in several directions to enhance its adaptability and intelligence. Machine learning-based anomaly detection can be integrated into the monitoring layer to predict deployment failures and performance regressions before they occur. Security gates can be further strengthened using dynamic analysis tools, runtime threat detection, and policy-driven secret-management systems. An automated compliance module may be added to validate deployments against organizational and regulatory standards. In large-scale systems, adaptive autoscaling policies and multi-cluster Kubernetes orchestration can be used to improve resilience and scalability. Future work may also explore a unified dashboard for real-time visualization of pipeline health, security posture, and resource utilization, enabling better operational insights and decision-making.

REFERENCES

[1] Vargas, A., et al. (2017), Optimizing CI/CD Pipelines for Large-Scale Applications. This study highlights the challenges and solutions in optimizing CI/CD pipelines for large-scale enterprise applications.

[2] Rajput, A., et al. (2018), Automating Testing and Deployment with Jenkins and GitLab CI. This paper compares CI/CD tools like Jenkins and GitLab and their integration into the automated testing and deployment process.

[3] Hassan, S. H., & Malik, S. (2021), Continuous Integration and Deployment Best Practices. The authors provide best practices for integrating testing, build, and deployment stages within CI/CD pipelines for continuous quality assurance.

[4] Fernandes, M., et al. (2016), Automated Testing Integration in CI/CD Pipelines. This paper discusses the integration of automated testing frameworks like Selenium and JUnit to ensure software reliability during the CI/CD process.

[5] Jones, S., et al. (2019), Performance Monitoring in CI/CD Pipelines. This research presents methods for monitoring system performance during CI/CD deployment cycles, integrating tools like Prometheus and Grafana.

[6] Gupta, R., et al. (2018), Optimizing Kubernetes for Production Deployments. The study focuses on the use of Kubernetes for managing production environments, ensuring automatic scaling and failover.

[7] Singh, A., et al. (2020), Integrating Cloud Services in CI/CD Pipelines. This work explores how cloud-native services, including AWS and Azure, can be integrated into CI/CD workflows to enhance scalability.

[8] Almeida, P., et al. (2020), Integrating Machine Learning with CI/CD Pipelines. This paper explores the potential of integrating machine learning models into CI/CD pipelines for predictive analysis and failure detection.Li, Y., et al. (2019), Automated Container Security Scanning in CI/CD Pipelines. This research focuses on integrating security tools like Snyk into the CI/CD pipeline to scan Docker images for vulnerabilities before deployment.

[9] Bishop, J., et al. (2018), Monitoring CI/CD Systems with Prometheus and Grafana. This paper discusses the use of Prometheus and Grafana for monitoring CI/CD pipeline health, identifying bottlenecks, and improving overall system performance.

[10] O'Neil, T., et al. (2021), Advanced Security Practices in CI/CD Pipelines. The authors discuss advanced techniques for integrating DevSecOps practices into CI/CD pipelines, ensuring secure software deployment.

[11] Patel, V., et al. (2020), Automating Rollbacks and Failover in CI/CD Pipelines with Kubernetes. This study outlines strategies for automating application rollbacks in case of deployment failures in CI/CD pipelines using Kubernetes.

[12] López, S., et al. (2021), Continuous Delivery with Docker and Kubernetes. The paper demonstrates how Docker and Kubernetes can be used together to implement a seamless CI/CD process that enhances scalability and deployment reliability.

[13] Cohen, D., et al. (2020), Testing Strategies for CI/CD Pipelines. This paper focuses on best practices for integrating different levels of testing (unit, integration, and system tests) into the CI/CD pipeline to ensure quality assurance.

[14] Alvarez, R., et al. (2021), Managing Production Environments with Kubernetes in CI/CD Pipelines. This paper presents techniques for managing and automating production deployments in Kubernetes,

ensuring stability and scalability.

[15] Das, S., et al. (2019), Improving Deployment Efficiency with Serverless CI/CD Pipelines. The authors discuss the use of serverless architectures to optimize deployment efficiency and reduce infrastructure management overhead.

[16] Jain, R., et al. (2021), CI/CD for Microservices Architectures: Best Practices. This work examines the challenges and solutions for implementing CI/CD pipelines for microservices-based applications.