ILKOM Jurnal Ilmiah Vol. 17, No. 2, August 2025, pp. 107-119
Accredited 2nd by RISTEKBRIN No. 10/C/C3/DT.05.00/2025; E-ISSN 2548-7779 | P-ISSN 2087-1716

Research Article Open Access (CC–BY-SA)

Enhancing Kubernetes-Based Microservices Deployment Efficiency Through DevOps and GitOps

Irvan Maulana [a,1,*]; **Rusydi Umar** [a,2]; **Anton Yudhana**[a,3]
[a]*Magister Teknik Informatika, Universitas Ahmad Dahlan, Yogyakarta, Indonesia*
[a]*Magister Teknik Informatika, Universitas Ahmad Dahlan, Yogyakarta, Indonesia*
[a]*Magister Teknik Elektro, Universitas Ahmad Dahlan, Yogyakarta, Indonesia*
[1]*vanzoelmaulana@gmail.com,* [2] *rusydi@mati.uad.ac.id,* [3]*eyudhana@ee.uad.ac.id,* [4]*prodi@mti.uad.ac.id*

Article history: Received February 08, 2025; Revised April 22, 2025; Accepted July 08, 2025; Available online August 14, 2025.

Abstract

An effective and resilient means to deploy microservices to Kubernetes is an ongoing challenge. This challenge becomes more difficult with ever increasingly complex application architectures. This research explored a DevOps model based on GitOps that integrates ArgoCD and GitLab CI/CD, as a means to create a more effective, resilient, and scalable deployment. Twelve microservices that were deployed in a controlled experimentation format were used in a comparative approach to previous deployment practices that only considered manual deployments. The results show an overall deployment time improvement of 40%. For the deployments that were executed incorrectly, ArgoCD ensures service availability leveraging its self-healing capabilities. During the computation of each run we also experienced system performance in a sustained high-load environment. Upon high demand, we experienced the desired autoscaling behavior requested, which resulted in higher service responsiveness. In comparison to previous studies, this research considered statistical analysis, while also looking at an aspect of real-world orchestration and networking efficiency while adopting Kubernetes. Altogether, this research gives organizations practical advice on how they may optimize their deployment pipelines for efficient, scalable and resilient microservices.

Keywords: Kubernetes, DevOps, Microservices, GitOps, CI/CD

Introduction

In the ever-evolving digital era, organizations across sectors such as finance, manufacturing, and healthcare are under pressure to deliver responsive, scalable, and reliable applications to meet growing business demands [1]. This pressure has led to significant challenges in application deployment and management, particularly as systems become more complex and distributed.

Recent research in cloud computing and application deployment has highlighted several critical challenges [2]. However, large-scale deployment and management of microservices applications bring huge challenges, such as longer deployment time, service downtime instances, and losses in efficiency. Conventional deployment approaches commonly experience limitations in ensuring service reliability and scalability against fluctuating workloads [3], [4]. Pham et al. identified that traditional deployment methods struggle with elastic scalability and resource optimization in edge computing environments [5]. Similarly, Vayghan et al. found that managing the availability of stateful applications in Kubernetes environments presents significant operational challenges, with manual deployment methods often leading to increased downtime and reduced reliability [6].

Current studies have highlighted the significance of DevOps practices in resolving deployment inefficiencies. DevOps facilitates CI/CD pipelines to achieve faster and more consistent deployment. However, the majority of existing research has already significantly compared automation based on DevOps with traditional manual deployment processes [7], [8]. Therefore, there is an urgent necessity to investigate deeper innovations beyond core DevOps practices.

One such emerging approach is GitOps, whereby deployment activities are entirely managed by Git repositories as a single source of truth, enabling more stringent automation, versioning, and rollback [9]. Another direction in this vein is the integration of AI into DevOps pipelines (AI-Driven DevOps), which is also gaining traction, with possibilities of dynamic pipeline optimization, predictive failure identification, and automated scaling decisions [10], [11], [12].

Another key aspect that is usually neglected in deployment studies is DevSecOps, which integrates security controls into the DevOps pipeline to make sure automation will not undermine application security, especially in Kubernetes deployments [13], [14].

N. Vemuri et al. discuss in their study how AI can optimize DevOps workflows by automating repetitive tasks, improving deployment accuracy, and accelerating the software release cycle [15]. Their research highlights the potential of AI to reduce operational bottlenecks and enable faster, more reliable cloud-based deployments, positioning AI-optimized DevOps as a critical evolution in modern software development practices.Turin et al. demonstrated that predicting and optimizing resource consumption in Kubernetes container systems remains a significant challenge, particularly when dealing with multiple services and varying workloads [16]. This complexity is further emphasized by Zahoor et al., who identified security policy management as another critical concern in Kubernetes deployments, highlighting the need for automated and consistent deployment processes [17].

Besides, the effect of deployment automation on network performance, latency, and service discovery is not yet thoroughly examined. Service mesh technologies like Istio and Linkerd have been suggested as overlays for Kubernetes cluster observability, fault tolerance, and traffic management [18], [19]. It is needed to know how deployment patterns influence inter-service communication, network bottlenecks, and service resilience in order to optimize microservices orchestration.

DevOps practices have shown promise in addressing these deployment challenges. Wiedemann et al. found that integrating development and operations teams through DevOps practices can significantly improve deployment efficiency [20]. Langerman and Leung further explored the impact of organizational structures on DevOps implementation success [21].

To address deployment complexities, many organizations are adopting microservices architecture. This architectural approach breaks down monolithic applications into smaller, independent service units, enabling more flexible development, testing, and deployment [22], [23]. Waseem et al. conducted a systematic mapping study of microservices architecture in DevOps environments, identifying gaps in current implementation approaches [24]. Faustino et al. further demonstrated that while microservices offer improved scalability and maintenance, they introduce new challenges in deployment coordination and service orchestration [25].

This research addresses these gaps by exploring and analyzing optimal deployment strategies for microservices applications within a Kubernetes cluster through a comprehensive DevOps approach. Building upon fundamental Kubernetes concepts and recent practical implementations [26], [27], we aim to:

1. Evaluate the impact of DevOps implementation on deployment efficiency and reliability
2. Analyze error handling and recovery capabilities in automated versus manual deployment processes
3. Measure and compare deployment times and service availability across different deployment methods

Through this study, we aim to provide organizations with empirical evidence and practical guidance for optimizing their microservices deployment processes.

Method

This study employs a quantitative research design with an experimental approach. This design is chosen because it allows for direct testing of the effectiveness of implementing DevOps in optimizing the deployment of microservices applications within Kubernetes clusters. Quantitative research focuses on the collection and analysis of numerical data to measure variables and the relationships between them. In this study, numerical data related to deployment metrics such as deployment time, error count, and deployment success rate will be collected. This data will then be statistically analyzed to determine the effectiveness of DevOps implementation. The experiment involved deploying 12 different services using both DevOps and manual deployment methods. Each service was deployed ten times to capture a range of deployment times. The services included Adservice, Cartservice, Checkoutservice, Currencyservice, Emailservice, Frontend, Loadgenerator, Paymentservice, Productcatalogservice, Recommendationservice, Shippingservice, and Shoppingassistantservice. The average deployment times for each service were calculated and compared.

A. Infrastructure

The experiments were conducted using a private on-premises Kubernetes cluster deployed across six nodes. The full specifications of the system used are presented in **Table 1**. This configuration reflects a moderately distributed cluster suitable for real-world testing in development or pre-production environments. Each node was connected over a local area network and orchestrated by a kubeadm-initialized Kubernetes setup, with containerd as the container runtime.

Table 1. Kubernetes Cluster Nodes Spesifications

No	Hostname	Spesifications	Operating System	Role
1	master	4cpu, 6Gi Memory	Ubuntu 22.04	Master node (control plane)
2	worker1	4cpu, 8Gi Memory	Ubuntu 22.04	Worker node
3	worker2	4cpu, 6Gi Memory	Ubuntu 22.04	Worker node
4	worker3	6cpu, 12Gi Memory	Ubuntu 22.04	Worker node
5	worker4	6cpu, 16Gi Memory	Ubuntu 22.04	Worker node
6	worker5	8cpu, 24Gi Memory	Ubuntu 22.04	Worker node

The structure of the cluster is illustrated in Figure 1. The control plane manages scheduling, API access, and the state of the cluster, and all workloads run in the worker nodes [28]. To expose microservices outside, Istio was deployed. Specifically, the Istio Ingress Gateway was utilized to route traffic to particular VirtualService definitions for two primary categories of microservices. This made external HTTP access and DNS-based routing to test endpoints possible [29], [30]. It is noteworthy that even subsequent to the Istio installation, the advanced features of service mesh such as mutual TLS, traffic shifting, circuit breaking, and distributed tracing were not utilized. The research targets deployment and availability concerns more than internal network configuration or fine-grained observability [31].

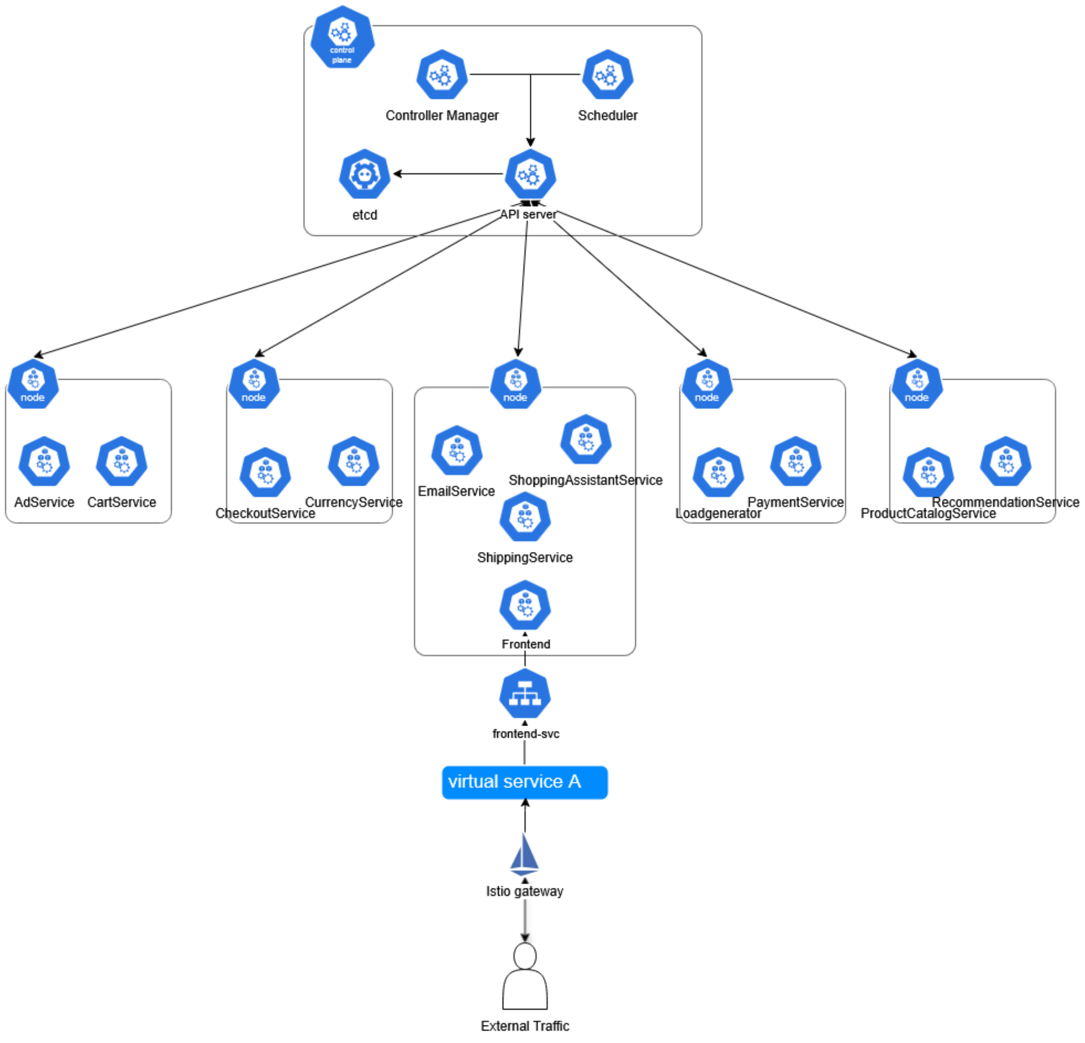


Figure 1. Kubernetes Architecthure

B. Deployment Models Comparission

The experimental approach involves manipulating the independent variable (DevOps implementation) and measuring its impact on the dependent variable (deployment metrics). In this research, two groups will be formed: a control group (without DevOps) and an experimental group (with DevOps). The differences in deployment outcomes between these two groups will be analyzed to assess the effectiveness of DevOps implementation.

- Manual Deployment (Baseline Group)

  In the traditional workflow shown in **Figure 2**, deployment process involves several manual steps: image building, pushing to a container registry, updating manifests, and deploying Kubernetes manifests with kubectl. This process is very operator-dependent, which can be a source of delays as well as potential human error.

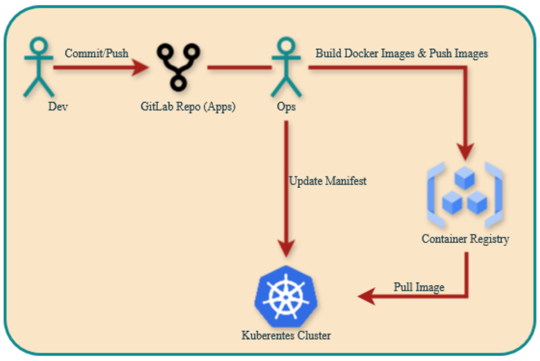


Figure 2. Manual deployment

- GitOps-based DevOps Deployment (Experimental Cohort)

  Depicted in **Figure 3**, this end-to-end automated process is GitOps-driven. Developers commit application code and deployment manifests to GitLab. This triggers a GitLab CI/CD pipeline that:

  1. Creates Docker images.
  2. Pushes them to the GitLab Container Registry.
  3. Updates the Kubernetes manifests located in the Git repository.

  ArgoCD monitors the Git repository in real time, identifies differences, and automatically synchronizes the desired state to the cluster. ArgoCD self-healing ensures configuration drift is fixed and failed deployments don't affect service right away. This deployment is based on GitOps principles: declarative configuration, automated reconciliation, version control, and observability [32], [33].

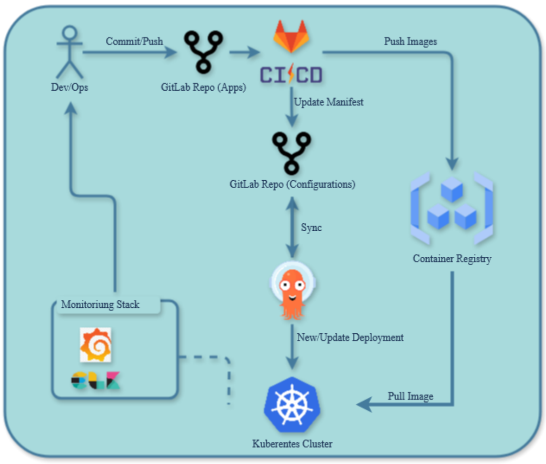


Figure 3. DevOps deployment

In modern IT environments, optimizing the deployment of microservices applications in Kubernetes clusters by adopting DevOps practices is crucial for ensuring the continuous availability of critical services [34]. High-availability (HA) infrastructure solutions and the application of DevOps principles are designed to achieve this by employing redundant components and failover mechanisms to prevent service downtime [35]. Adopting DevOps for the deployment of microservices applications with a focus on continuous principles as we can see in Figure 3, begins with developers committing and pushing code to a GitLab repository. GitLab then triggers a webhook that automatically initiates the CI/CD pipeline whenever there are changes in the repository. Within the CI/CD pipeline, the application is packaged into Docker images and these images are pushed to a Gitlab docker image registry [36]. Subsequently, ArgoCD identifies changes in the Kubernetes manifest within the GitLab repository and updates the deployment in the Kubernetes cluster using the new Docker images that were pushed by GitLab CI, all that process running automatically [37].

After all these processes, monitoring the running microservices applications in the Kubernetes cluster is essential to gather feedback and implement necessary updates. This cycle continues perpetually.

In contrast, traditional methods handle these processes manually. In **Figure 2**, the developer's role typically ends at committing and pushing code to the GitLab repository or pushing Docker images to the gitlab registry, with subsequent steps managed by the operations team. This entire process must be executed manually, which is inefficient, time-consuming, and prone to human error (potentially impacting the stability of the overall system).

Results and Discussion

This section analyzes and comparing the results of two primary deployment techniques: manual deployment and DevOps deployment as orchestrated with GitOps practices. The comparison evaluates parameters such as deployment time, error handling, and service uptime. While the scope of this research does not include large-scale dynamic load testing or detailed network traffic analysis, the findings offer relevant information regarding deployment trends and system stability in controlled environments.

A. Deployment Time

This research was conducted by 12 microservices with DevOps and manual deployment 10 times to view the different deployment time between DevOps and manual deployment. The detailed experiment available at **Table 2** .

Table 2. Avarange Deployment Time

Service	DevOps Avg Time (s)	Manual Avg Time (s)	Overall average		Standard Deviation (seconds)	
			DevOps	Manual	DevOps	Manual
Adservice	777.8	932.8	522.525	872.2	150.35	61.39
Cartservice	784.8	830				
Checkoutservice	498.1	774.4				
Currencyservice	484.3	864				
Emailservice	556.3	817.5				
Frontend	520.7	919.4				
Loadgenerator	498.6	933.8				
Paymentservice	243.9	839.1				
Productcatalogservice	602.5	976.5				
Recommendationservice	443.5	828.6				
Shippingservice	506.5	841.3				
Shoppingassistantservice	353.3	909				

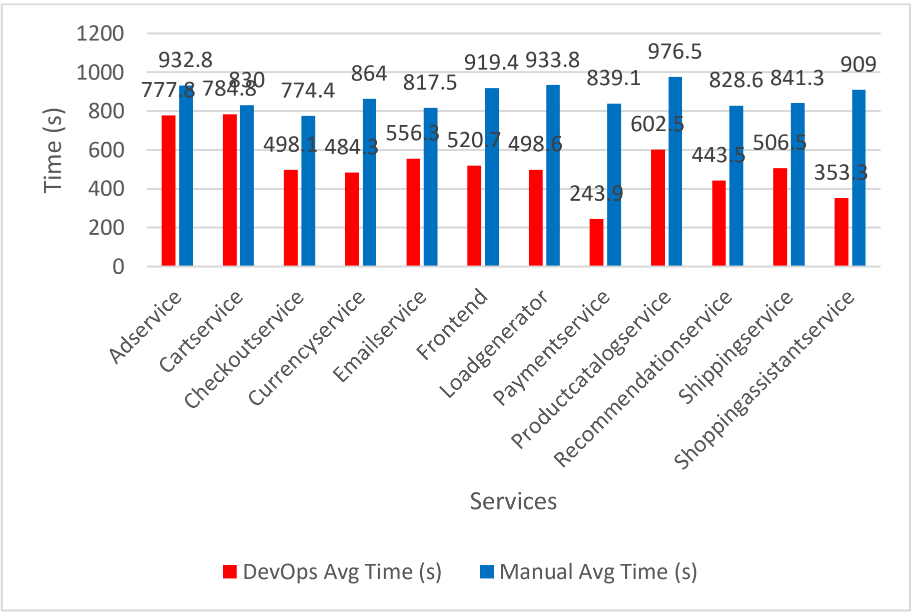


Figure 4. Deployment time comparission

B. Statistical Analysis

To evaluate whether the difference in deployment times between manual and DevOps methods is statistically significant, a two-sample t-test was conducted assuming equal variances. Each method was tested across 12 services (n = 12), and the average deployment times were previously summarized in **Table 2**.

Two-Sample t-Test with :

Hypotheses

Null Hypotheses (H_0)

$\mu_1 = \mu_2$

(No significant difference in mean deployment time)

Alternative Hypothesis (H_1)

$\mu_1 \neq \mu_2$

(There is a significant difference)

Pooled Standard Deviation (s_p) = 36.29

Since we are assuming that both groups have equal population variances, we need to compute the pooled standard deviation, which combines the variability of both groups into a single estimate. This is especially useful when the sample sizes are equal, as in this case.

$$s_p = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_1 - 1)s_2^2}{n_1 + n_2 - 2}}$$

T-Statistic Calculation (t) = −23.62

Where

- $\bar{x}_1$ and $\bar{x}_2$ are the sample means
- s_p is the pooled standard deviation
- n is the number of samples per group.

The t-statistic measures the size of the difference relative to the variation in the sample data. It is calculated using the formula below:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{s_p \cdot \sqrt{\frac{2}{n}}}$$

Degrees of Freedom (df) = 22, with this formula

$$df = n_1 + n_2 - 2$$

p-value < 0.00001, using a statistical software (SPSS), a t-statistic of -23.62 with 22 degrees of freedom results in a p-value < 0.00001.

Since this p-value is far below the conventional alpha level of 0.05, we reject the null hypothesis. This indicates a statistically significant difference between the two deployment methods.

The measured deployment durations indicate a clear advantage for the automated DevOps pipeline. Across twelve trials each, the CI/CD-driven DevOps process averaged 522.5 s per deployment versus 872.2 s for manual deployment roughly a 40% reduction in time. A two-sample t-test (equal variances, n = 12 per group) confirms this difference is highly significant. The pooled standard deviation (36.29),combined with a t-statistic of $-23.62$and p-value smaller than 0.001 confirm that the difference in avarage deployment time between the DevOps and manual methods is statistically significant. This substantial result confirms that the automated DevOps pipeline offers a consistent and measurable improvement over manual processes. The low variability and high significance level strengthen the conclusion that DevOps practices not only reduce deployment time but also contribute to a more reliable and repeatable deployment workflow, which is crucial for maintaining service availability in microservices-based architectures. Indeed, prior work has similarly found that introducing a CI/CD pipeline significantly reduced the deployment time compared to manual deployment.

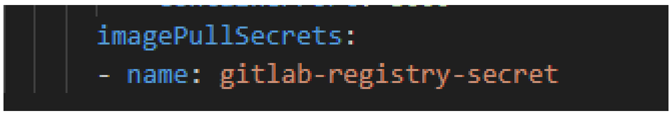
This magnitude of improvement is consistent with the general understanding that DevOps automation accelerates delivery. In manual deployments each step (building, testing, container creation, etc.) must be performed by hand, making the process slow and error-prone. In contrast, a CI/CD system like gitlab-ci or Jenkins can execute many steps automatically and in parallel maximizing time efficiency. Our result echoes studies reporting dramatic time savings (on the order of tens of percent) from automated pipelines. We note that AI-driven CI/CD tools are emerging as well, for example, pipelines that use machine learning to predict failures or optimize test order can further speed deployments but evaluation of such approaches is beyond this paper's scope.

The analysis reveals that DevOps deployment methods significantly reduce the average deployment time for all services tested. This reduction is particularly notable in services such as Paymentservice and Productcatalogservice, where the time savings exceed several hundred seconds. The automated processes inherent in DevOps likely contribute to these efficiencies by minimizing human error and increasing consistency across deployments.

These findings should be interpreted in light of the study's limitations. First, the tests did not include high-load or stress conditions; as experts emphasize, testing the scalability of a micro-service is very critical to ensure that deployments remain fast under increased workload. Second, we measured only end-to-end deployment time and did not collect lower-level metrics such as container startup latency or data transfer overhead. In a real microservices environment, each service-to-service "hop" adds latency, and cumulative inter-service communication can introduce nontrivial network overhead and bottlenecks. We also did not isolate the impact of inter-service messaging patterns on performance. Accounting for these factors (heavy load, latency, and communication overhead) in future experiments would provide a more complete view of deployment performance.

C. Error handling and service availability

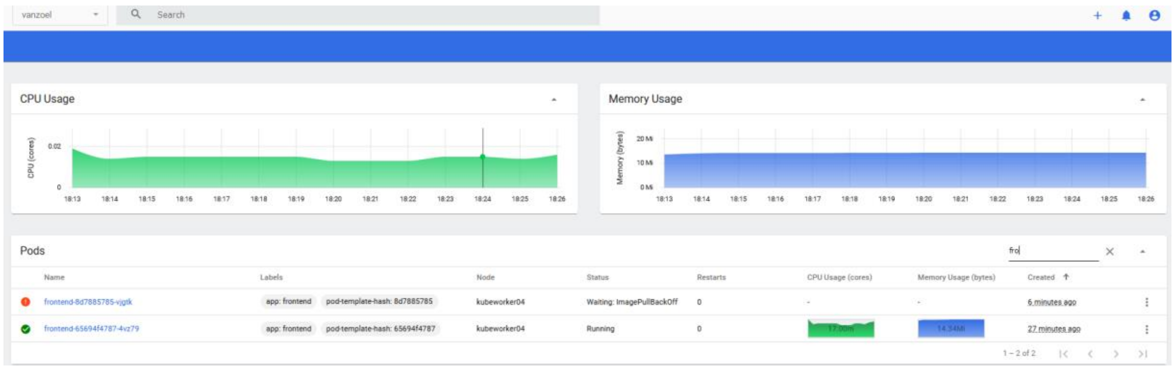
In this study, we collected the error handling and service availability from DevOps deployment and manual deployment within wrong Kubernetes manifest with removing the imagePullSecret in kuberentes manifest. The wrong Kubernetes manifest can be seen in Figure 5.



```
imagePullSecrets:
- name: gitlab-registry-secret
```

Figure 5. Kubernetes manifest without pullSecret

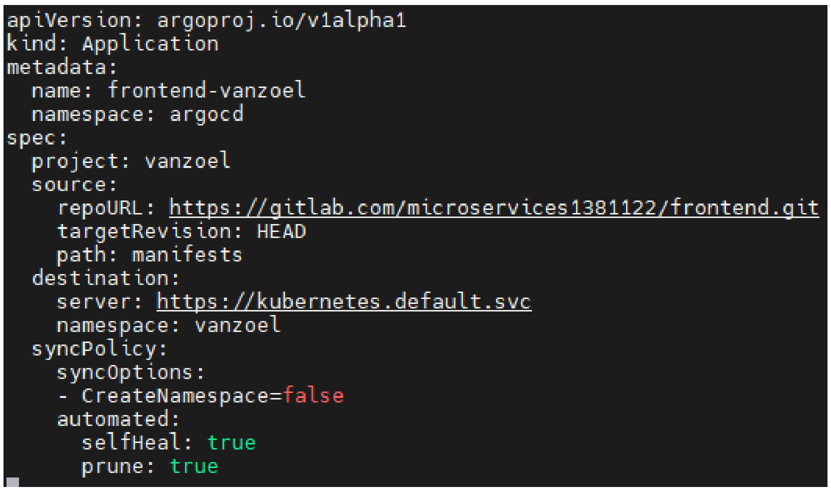
This experiment made some service with wrong Kubernetes manifest wich not including the imagePullSecret wich is the credential for authenticating with the gitlab images registry as we can see in fig 4 to see how DevOps deployment and manual deployment handle the error. The DevOps deployment using ArgoCD with application configuration can be seen in Fig 6 and the service availability in DevOps deployment can be seen in Figure 6.



	Name	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
●	frontend-8d7885785-vjgtk	app: frontend; pod-template-hash: 8d7885785	kubeworker04	Waiting: ImagePullBackOff	0	-	-	6 minutes ago
✓	frontend-65694f4787-4vz79	app: frontend; pod-template-hash: 65694f4787	kubeworker04	Running	0	17.00m	14.34Mi	27 minutes ago

1 – 2 of 2

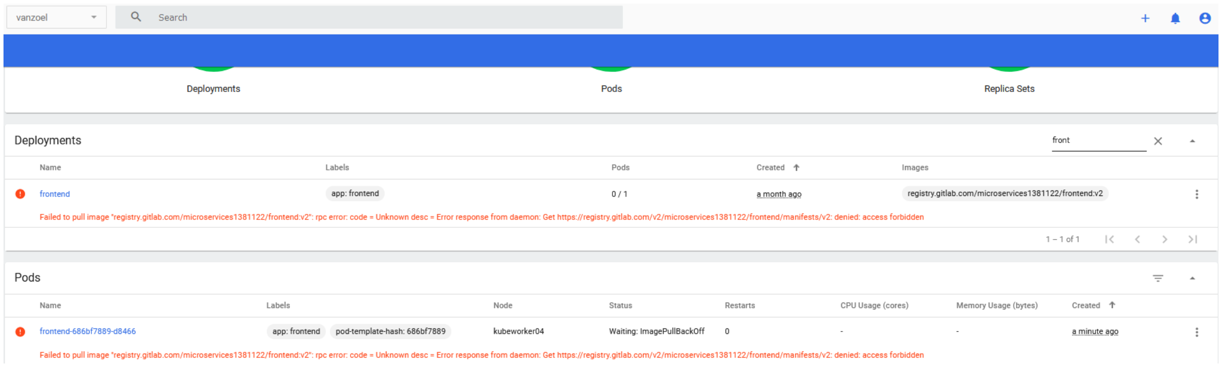
Figure 6. Service availability with DevOps deployment



```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: frontend-vanzoel
  namespace: argocd
spec:
  project: vanzoel
  source:
    repoURL: https://gitlab.com/microservices1381122/frontend.git
    targetRevision: HEAD
    path: manifests
  destination:
    server: https://kubernetes.default.svc
    namespace: vanzoel
  syncPolicy:
    syncOptions:
    - CreateNamespace=false
    automated:
      selfHeal: true
      prune: true
```

Figure 7. Argocd application configuration

The DevOps deployment method using ArgoCD as Continous Deployment and uses the Git repository as the source of truth to determine the desired state of the application in Kubernetes cluster event he configuration was wrong, ArgoCD will not instantly delete the old pod with selfHeal feature as we described in Figure 6. In summary, argocd can minimalize the downtime for application, because argocd will make sure the new deployment running properly before taking out or delete old deployment. The pod will not instantly replace with new pod , summary, the service was still available as we describe in **Figure 7**.



	Name	Labels	Pods	Created ↑	Images
●	frontend	app: frontend	0 / 1	a month ago	registry.gitlab.com/microservices1381122/frontend:v2

Failed to pull image "registry.gitlab.com/microservices1381122/frontend:v2": rpc error: code = Unknown desc = Error response from daemon: Get https://registry.gitlab.com/v2/microservices1381122/frontend/manifests/v2: denied: access forbidden

1 – 1 of 1

Pods

	Name	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
●	frontend-686bf7889-d8466	app: frontend; pod-template-hash: 686bf7889	kubeworker04	Waiting: ImagePullBackOff	0	-	-	a minute ago

Failed to pull image "registry.gitlab.com/microservices1381122/frontend:v2": rpc error: code = Unknown desc = Error response from daemon: Get https://registry.gitlab.com/v2/microservices1381122/frontend/manifests/v2: denied: access forbidden

Figure 8. Service availability without DevOps deployment

On the other hand, manual deployment not verifying the Kubernetes manifest, with this experiment, the pod will not avaibale because the pod can't pull the new images because don't have authentication credential. Summary this can leading to service downtime with new deployment if the pod did not running properly as we describe in **Figure 8**.

D. Load Test and Cluster Behavior under Dynamic Workload

To complement the deployment time analysis, we executed a load testing scenario to simulate production-level high traffic and exercise dynamic scaling behavior in the Kubernetes environment. The test targeted the productcataloge service endpoint /product/0PUK6V6EV0, simulating product detail fetch under maximum concurrent access. The test provided insight into runtime behavior of the Kubernetes cluster and exercised the deployment infrastructure's tolerance to sustained pressure.

The load was simulated using Apache JMeter, generating 7,712 samples of HTTP requests over 10 minutes. The mean response time of 81,860 milliseconds was at a minimum of 68 milliseconds and a maximum of 675,480 milliseconds, as can be seen from **Table 3**. The standard deviation of 56583,29 milliseconds was extremely high, indicating extremely high variation in response times. The system achieved a throughput of 11.3 requests per second and an error rate of 27.17%, which is most likely caused by resource saturation occurring at periods of peak latency.

Table 3. Summary Report

Label	**Samples**	**Average**	**Min**	**Max**	**Std. Dev**	**Error%**	**Throughput**	**Received KB/sec**	**Sent KB/sec**	**Avg. Bytes**
HTTP Request	7712	81860	68	675.480	56583.29	27.17	11.3/sec	85.96	1.93	7778.3
TOTAL	7712	81860	68	675.480	56583.29	27.17	11.3/sec	85.96	1.93	7778.3

This performance distribution is visualized in **Figure 9**, which illustrates a progressive rise in response time during the early phase of the test, peaking at ~140 seconds, followed by a mix of sustained latency and temporary drops. These drops correspond with temporary recovery points, possibly due to internal load distribution or cached responses.

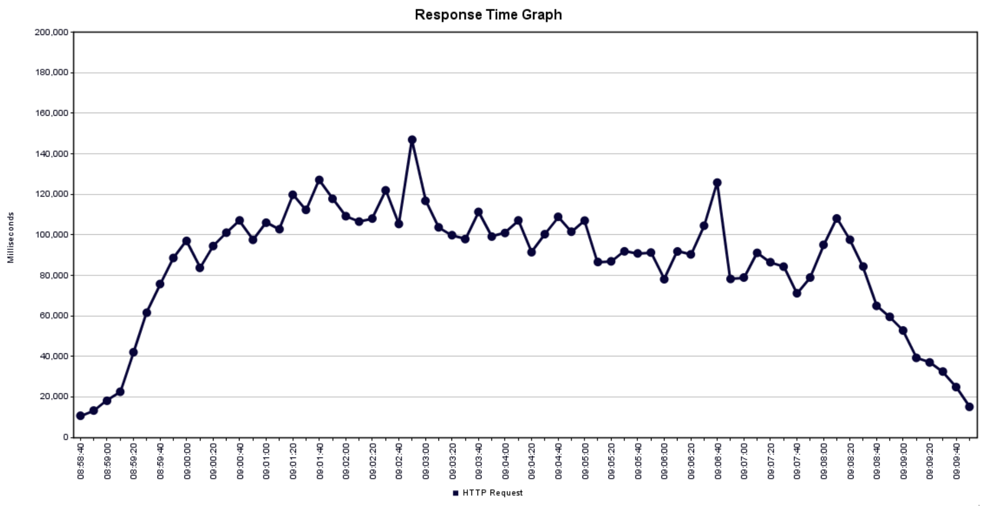


Figure 9. Response Time

During the load testing phase of the product catalog service, we monitored traffic and CPU and memory utilization using Grafana in real-time at the pod level. **Figure 10** illustrates the immediate change in memory usage that correlates to sustained traffic. In the first two minutes, memory usage was consistently above 80%, so the original pod utilization exceeded memory limits configured for a Horizontal Pod Autoscaler (HPA), which triggered a configured Kubernetes cluster scaling policy, resulting in the automatic creation of a new pod running the product catalog service.

This activity supports our original hypothesis that autoscaling is providing elasticity in that it was accommodating service levels of availability for the dynamic workloads it was serving. When two product catalog service pods were instantiated, upon arrival rate, the traffic was being satisfied by two pods and the load was distributed more efficiently without service degradation under added traffic.

As this example demonstrates, resource monitoring at runtime and autoscaling policy offers an important mechanism for elasticity and fault tolerance in microservices-enabled architectures. In addition, this example highlights the importance of selecting resource limits thresholds in order to make sure they are aligned to your workloads, such that autoscaling occurs with maximum influence and least overlap, respectively.

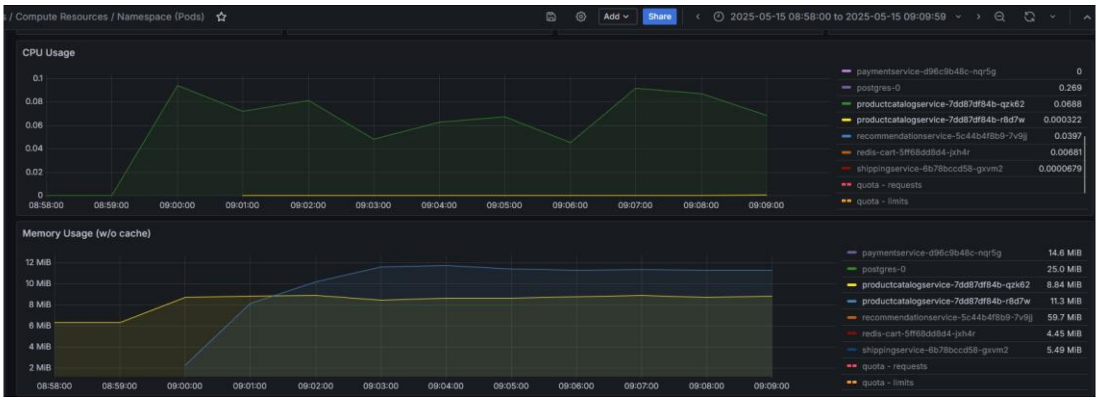


Figure 10. Grafana pod utilizations

Together, these results reflect the service's performance degradation under sustained load without autoscaling support. In contrast to earlier tests involving the frontend service, where Horizontal Pod Autoscaler (HPA) was triggered based on memory thresholds, the productcatalogservice instance did not trigger scaling events. This suggests either more conservative HPA configuration, insufficient resource limits, or a need for revised threshold tuning.

From a deployment reliability perspective, these results highlight the importance of not only automated deployment but also runtime observability and scaling policies. While the system remained operational, the latency spikes and error rates suggest that autoscaling policies must be carefully tailored to each microservice's expected load pattern and criticality.

Future iterations of this experiment should include:

1. Application-specific HPA tuning based on both memory and CPU utilization.
2. Load redistribution policies or circuit-breaking rules.
3. Integration with service mesh observability tools (e.g., Istio telemetry, Kiali, or Jaeger) to trace failure origins and latencies at service hop-level resolution.

These findings underscore that although GitOps-based deployment pipelines provide operational efficiency, runtime performance assurance still depends heavily on post-deployment observability and scaling configuration.

Conclusion

The results of this research show that the implementation of DevOps practices to deploy microservices applications to uniformly orchestrated kubernetes clusters increases the deployment efficiency, reliability, and operational resiliency of those applications. The quantitative comparison between DevOps based automation and traditional manual practices showed statistically significant deployment efficiencies yielded average deployment efficiencies of 522.5 seconds (DevOps automation) and 872.2 seconds (manual methods). These efficiencies will allow organizations to reduce release cycles and human error while maintaining service continuity. When considering availability and fault tolerance, through the use of tools like ArgoCD, DevOps including self-healing functionality made it possible to maintain service uptime even with misconfigured deployments, whereas manual methods led to pod errors and service disruption. The automated rollback and verification capabilities in DevOps pipelines created a more flexible and reliable application development life cycle.

Considering the scalability and networking perspective involves load-testing under dynamic workload features, DevOps deployments combined well with the auto-scaling features of kubernetes and allowed for better load balancing of requests and service responsiveness, while only utilizing Istios Ingress Gateway level (not full-service mesh) features, the system demonstrated better orchestration behavior in under increasing traffic. The autoscaling events were triggered and occurred as expected and allowed the pods to autorecover when the requests exceeded memory usage, allowing the pod forecasted performance (reduced request latency spikes). In conclusion, as implied by the self-healing aspects of our system, DevOps contributes to an optimum orchestrated experience.

For industrial adoption, this research affirms that organizations can benefit from adopting GitOps-driven DevOps workflows to achieve faster deployment cycles, minimize operational risks, and scale services predictably. By aligning infrastructure as code, continuous deployment, and observability, companies can streamline their development-to-deployment processes while improving infrastructure responsiveness to traffic surges and failures. Organizations seeking rapid innovation and operational agility, particularly in industries reliant on high availability such as fintech, healthtech, and e-commerce are encouraged to adopt these practices.

This research is limited in scope to the evaluation of a single DevOps implementation using GitOps and ArgoCD. Alternative paradigms such as AI-driven DevOps, GitOps with full service mesh capabilities, or hybrid cloud multi-cluster deployment strategies were not explored. Furthermore, security aspects (DevSecOps) and cost-efficiency evaluations were excluded from the experiments, despite their increasing relevance in production-grade environments.

References

[1] S. Pallewatta, V. Kostakos, and R. Buyya, "MicroFog: A framework for scalable placement of microservices-based IoT applications in federated Fog environments," *Journal of Systems and Software*, vol. 209, Mar. 2024, doi: 10.1016/j.jss.2023.111910.

[2] S. Alzide, "Cloud Computing: Evolution, Challenges, and Future Prospects," *Journal of Information Technology, Cybersecurity, and Artificial Intelligence*, vol. 1, no. 1, pp. 52–63, Dec. 2024, doi: 10.70715/jitcai.2024.v1.i1.007.

[3] K. Vishnivetskii, "Dynamic Scaling and Performance Optimization for Microservices using Kubernetes," *Asian Journal of Research in Computer Science*, vol. 18, no. 3, pp. 213–220, Feb. 2025, doi: 10.9734/ajrcos/2025/v18i3587.

[4] S. Hassan, R. Bahsoon, and R. Buyya, "Systematic scalability analysis for microservices granularity adaptation design decisions," *Softw Pract Exp*, vol. 52, Jan. 2022, doi: 10.1002/spe.3069.

[5] K. Q. Pham and T. Kim, "Elastic Federated Learning with Kubernetes Vertical Pod Autoscaler for edge computing," *Future Generation Computer Systems*, vol. 158, pp. 501–515, Sep. 2024, doi: 10.1016/j.future.2024.04.047.

[6] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes controller for managing the availability of elastic microservice based stateful applications," *Journal of Systems and Software*, vol. 175, May 2021, doi: 10.1016/j.jss.2021.110924.

[7] S. Rama Krishna, J. Srinivasa Rao, Y. Venkata Durga, L. Prem Venkatesh, and P. Sridhar, "Enhancing Software Deployment Efficiency: A Comparative Analysis of Agile Application Deployment Using CI/CD Pipelines," 2024.

[8] G. Hyun, J. Oak, D. Kim, and K. Kim, "The Impact of an Automation System Built with Jenkins on the Efficiency of Container-Based System Deployment," *Sensors*, vol. 24, no. 18, Sep. 2024, doi: 10.3390/s24186002.

[9] K. Sakinala, "Advancements in Devops: The Role Of Gitops In Modern Infrastructure Management," *International Journal Of Information Technology And Management Information Systems*, vol. 16, no. 1, pp. 632–646, Feb. 2025, doi: 10.34218/IJITMIS_16_01_045.

[10] A. Kumar, M. Nadeem, and M. Shameem, "Machine learning based predictive modeling to effectively implement DevOps practices in software organizations," *Automated Software Engineering*, vol. 30, Jul. 2023, doi: 10.1007/s10515-023-00388-8.

[11] V. M. Tamanampudi, "Distributed Learning and Broad Applications in Scientific Research Automating CI/CD Pipelines with Machine Learning Algorithms: Optimizing Build and Deployment Processes in DevOps Ecosystems," 2019.

[12] N. Vemuri, V. Manoj Tatikonda, and N. Thaneeru, "Integrating Deep Learning with DevOps for Enhanced Predictive Maintenance in the Manufacturing Industry," 2022.

[13] S. Ugale and A. Potgantwar, "Container Security in Cloud Environments: A Comprehensive Analysis and Future Directions for DevSecOps †," *Engineering Proceedings*, vol. 59, no. 1, 2023, doi: 10.3390/engproc2023059057.

[14] L. Prates and R. Pereira, "DevSecOps practices and tools," *Int J Inf Secur*, vol. 24, no. 1, Feb. 2025, doi: 10.1007/s10207-024-00914-z.

[15] N. Vemuri, N. Thaneeru, and V. M. Tatikonda, "AI-Optimized DevOps for Streamlined Cloud CI/CD," 2024. [Online]. Available: www.ijisrt.com504

[16] G. Turin, A. Borgarelli, S. Donetti, F. Damiani, E. B. Johnsen, and S. L. Tapia Tarifa, "Predicting resource consumption of Kubernetes container systems using resource models," *Journal of Systems and Software*, vol. 203, Sep. 2023, doi: 10.1016/j.jss.2023.111750.

[17] E. Zahoor, M. Chaudhary, S. Akhtar, and O. Perrin, "A formal approach for the identification of redundant authorization policies in Kubernetes," *Comput Secur*, vol. 135, Dec. 2023, doi: 10.1016/j.cose.2023.103473.

[18] M. R. Saleh Sedghpour, C. Klein, and J. Tordsson, "An Empirical Study of Service Mesh Traffic Management Policies for Microservices," in *ICPE 2022 - Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering*, Association for Computing Machinery, Inc, Apr. 2022, pp. 17–27. doi: 10.1145/3489525.3511686.

[19] K. V. Palavesam, M. V. Krishnamoorthy, and A. S M, "A Comparative Study of Service Mesh Implementations in Kubernetes for Multi-cluster Management," *Journal of Advances in Mathematics and Computer Science*, vol. 40, no. 1, pp. 1–16, Jan. 2025, doi: 10.9734/jamcs/2025/v40i11958.

[20] A. Wiedemann, M. Wiesche, H. Gewald, and H. Krcmar, "Integrating development and operations teams: A control approach for DevOps," *Information and Organization*, vol. 33, no. 3, Sep. 2023, doi: 10.1016/j.infoandorg.2023.100474.

[21] J. Langerman and W. S. Leung, "The effect of outsourcing and insourcing on Agile and DevOps," *Journal of Information Technology Teaching Cases*, 2023, doi: 10.1177/20438869231176841.

[22] A. Raharjo, P. Andyartha, W. Wijaya, Y. Purwananto, D. Purwitasari, and N. Juniarta, *Reliability Evaluation of Microservices and Monolithic Architectures*. 2022. doi: 10.1109/CENIM56801.2022.10037281.

[23] A. Nicolas-Plata, J. L. Gonzalez-Compean, and V. J. Sosa-Sosa, "A service mesh approach to integrate processing patterns into microservices applications," *Cluster Comput*, vol. 27, no. 6, pp. 7417–7438, Sep. 2024, doi: 10.1007/s10586-024-04342-5.

[24] M. Waseem, P. Liang, and M. Shahin, "A Systematic Mapping Study on Microservices Architecture in DevOps," *Journal of Systems and Software*, vol. 170, Dec. 2020, doi: 10.1016/j.jss.2020.110798.

[25] D. Faustino, N. Gonçalves, M. Portela, and A. Rito Silva, "Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation," *Performance Evaluation*, vol. 164, May 2024, doi: 10.1016/j.peva.2024.102411.

[26] S. C and M. S, *Security Aware Resource Management Framework (SARMF) for Edge-Cloud Computing*. 2023. doi: 10.1109/ICOEI56765.2023.10125845.

[27] J. B. Adelusi, "Kubernetes for Microservices Deployment Across Cloud Platforms." .

[28] D. D. Vu, M. N. Tran, and Y. Kim, "Predictive Hybrid Autoscaling for Containerized Applications," *IEEE Access*, vol. 10, pp. 109768–109778, 2022, doi: 10.1109/ACCESS.2022.3214985.

[29] P. Priya Patharlagadda, "Kubernetes Traffic Management using Istio," *Journal of Media & Management*, pp. 1–4, Feb. 2022, doi: 10.47363/JMM/2022(4)E101.

[30] M. Chigurupati and A. Jagtap, "Enhancing Microservice Resiliency and Reliability on Kubernetes with Istio: A Site Reliability Engineering Perspective," *International Journal of Computer Trends and Technology*, vol. 72, no. 11, pp. 17–22, Nov. 2024, doi: 10.14445/22312803/IJCTT-V72I11P103.

[31] A. Malhotra, A. Elsayed, R. Torres, S. Venkatraman, and A. S. Kaul, "Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000. Evaluate Canary Deployment Techniques using Kubernetes, Istio and Liquibase for Cloud Native Enterprise Applications to Achieve Zero Downtime for Continuous Deployments," no. 1, 2017, doi: 10.1109/ACCESS.2024.07512.

[32] K. Sakinala, "Advancements in Devops: The Role of Gitops in Modern Infrastructure Management," *International Journal Of Information Technology And Management Information Systems*, vol. 16, pp. 632–646, Feb. 2025, doi: 10.34218/IJITMIS_16_01_045.

[33] Ramadoni, E. Utami, and H. al Fatta, *Analysis on the Use of Declarative and Pull-based Deployment Models on GitOps Using Argo CD*. 2021. doi: 10.1109/ICOIACT53268.2021.9563984.

[34] B. Chandra Vadde and V. B. Munagandla, "Cloud-Native DevOps: Leveraging Microservices and Kubernetes for Scalable Infrastructure," 2024.

[35] P. Somasekaram, R. Calinescu, and R. Buyya, “High-Availability Clusters: A Taxonomy, Survey, and Future Directions”, doi: 10.48550/arXiv.2109.15139.

[36] A. Singh, V. Singh, and A. Aggarwal, “Improving Business Deliveries for Micro-services-based Systems using CI/CD and Jenkins,” *Journal of Mines Metals and Fuels*, Dec. 2023, doi: 10.18311/jmmf/2023/33936.

[37] T. Kormanik and J. Poruban, “Exploring GitOps: An Approach to Cloud Cluster System Deployment,” in *ICETA 2023 - 21st Year of International Conference on Emerging eLearning Technologies and Applications, Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2023, pp. 318–323. doi: 10.1109/ICETA61311.2023.10344182.