VOL. 12. NO. 1 AUGUST 2026
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v12i1.8208

JITK (JURNAL ILMU PENGETAHUAN DAN TEKNOLOGI KOMPUTER)

IMPLEMENTATION OF DEVOPS APPROACH USING CI/CD WITH A SHARED-PIPELINE TO ACCELERATE WEB SERVICE DEVELOPMENT

Ahmad Lukman Hakim[1*]; Diah Aryani[1]; Daniati Uki Eka Saputri[2]

Informatics Engineering[1]
Universitas Esa Unggul, Jakarta, Indonesia[1]
www.esaunggul.ac.id[1]
lukmanlabz@student.esaunggul.ac.id*, diah.aryani@esaunggul.ac.id

Information Systems[2]
Universitas Nusa Mandiri, Jakarta, Indonesia[2]
www.nusamandiri.ac.id[2]
daniati.due@nusamandiri.ac.id

(*) Corresponding Author
(Responsible for the Quality of Paper Content)

Abstract— *The development of microservices-based web service applications requires fast, consistent, and automated integration and deployment processes. However, in practice, many organizations still rely on manual procedures and non-standardized pipeline configurations, which lead to release delays and increase the risk of operational errors. The DevOps approach through Continuous Integration and Continuous Delivery (CI/CD) has emerged as a relevant solution. Nevertheless, its practical implementation still requires clear and reusable guidance. This study aims to implement DevOps approach using CI/CD practices for the backend microservices at JEL Tech Inc by leveraging GitLab and a shared-pipeline scheme. The research adopts an applied research methodology with a case study approach, following the stages of the DevOps lifecycle, including plan, code, build, test, release, deploy, operate, and monitor. Pipeline automation is constructed using modular shell scripts executed by GitLab Runner and integrated with Docker, a Container Registry, and Kubernetes. The results demonstrate that the implementation of a shared-pipeline successfully standardizes build, package, and deployment processes across multiple microservices repositories. The implementation results show measurable improvements, including up to 90% reduction in deployment time, 400% increase in release frequency, and significant reduction in failure rates. This study contributes a reusable shared-pipeline architecture that extends existing DevOps implementation practices by providing a standardized, modular, and testable CI/CD model for microservices environments.*

Keywords: *CI/CD, DevOps, GitLab, Shared Pipeline, Shell Script*

Intisari— *Pengembangan aplikasi web service berbasis layanan mikro (microservices) menuntut proses integrasi dan penyebaran (deployment) yang cepat, konsisten, dan terotomatisasi. Namun, pada praktiknya masih banyak organisasi yang mengandalkan proses manual dan konfigurasi alur kerja (pipeline) yang tidak terstandarisasi, sehingga menimbulkan keterlambatan perilisan serta meningkatkan risiko kesalahan operasional. Pendekatan DevOps melalui Continuous Integration dan Continuous Delivery (CI/CD) menjadi solusi yang relevan, namun implementasi praktisnya masih memerlukan panduan yang jelas dan dapat digunakan kembali. Penelitian ini bertujuan untuk mengimplementasikan pendekatan DevOps dan praktik CI/CD pada backend microservices di perusahaan JEL Tech dengan memanfaatkan GitLab dan skema alur kerja terpusat (shared-pipeline). Metode yang digunakan adalah applied research dengan pendekatan studi kasus, mengikuti tahapan siklus hidup DevOps (DevOps lifecycle) mulai dari perencanaan (plan), pengkodean (code), pembangunan (build), pengujian (test), perilisian (release), penyebaran (deploy), pengoperasian (operate), dan pemantauan (monitoring). Otomatisasi pipeline dibangun menggunakan skrip Shell modular*

VOL. 12. NO. 1 AUGUST 2026
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v12i1.8208

yang dijalankan oleh GitLab Runner dan diintegrasikan dengan Docker, Container Registry, serta Kubernetes. Hasil penelitian menunjukkan bahwa penerapan shared-pipeline mampu menstandarisasi proses build hingga deployment lintas repositori microservices. Hasil implementasi menunjukkan peningkatan yang terukur, termasuk pengurangan waktu deployment hingga 90%, peningkatan frekuensi rilis hingga 400%, dan pengurangan tingkat kegagalan yang signifikan. Studi ini memberikan kontribusi berupa arsitektur shared-pipeline yang dapat digunakan kembali yang memperluas praktik implementasi DevOps yang ada dengan menyediakan model CI/CD yang terstandarisasi, modular, dan dapat diuji untuk lingkungan microservices.

***Kata Kunci**: CI/CD, DevOps, GitLab, Shared Pipeline, Shell Script.*

INTRODUCTION

The rapid advancement of digital technology requires organizations to develop and release software quickly, reliably, and sustainably [1], [2], [3]. At JEL Tech Inc, the development team faced challenges related to manual deployment processes and inconsistent environment configurations across web services, resulting in delayed releases and frequent configuration errors. As the number of services, development teams, and integration requirements increased, system complexity also grew significantly [4], [5]. Consequently, traditional software development approaches became insufficient, leading to slow time-to-market, repeated manual errors, and limited collaboration between development and operations teams.

Therefore, an integrated and continuous approach is required to bridge the gap between development and operations teams. Many organizations have adopted the DevOps approach and integrated it with cloud computing technologies and Agile methodologies [6], [7], [8], [9]. Numerous studies indicate that DevOps can accelerate technology adoption and support continuous improvement in software development processes [10], [11], [12], [13]. However, prior studies on DevOps adoption predominantly focus on conceptual frameworks, organizational aspects, or general CI/CD practices, with limited emphasis on reusable pipeline architectures and implementation-level automation. While these benefits appear promising, the implementation of DevOps also introduces significant complexity and requires substantial technical expertise, encompassing infrastructure, organizational structures, and human factors.

Nevertheless, the practical adoption of DevOps is not without challenges and complexities, whether from infrastructural, organizational, or human perspectives. Several studies report that DevOps adoption has successfully increased release frequency, shortened deployment time, and improved software quality through the implementation of Continuous Integration and Continuous Deployment (CI/CD) practices [6], [13], [14], [15], [16], [17], [18], [19]. However, most existing studies still focus primarily on conceptual discussions, general DevOps frameworks, or organizational and human aspects [1], [20]. In contrast, in-depth technical discussions on reusable CI/CD implementations—such as shared-pipelines, script-based automation, and their integration with cloud ecosystems and container orchestration platforms—remain limited. This limitation often leads to suboptimal outcomes in DevOps adoption within industry practices.

Based on these issues, this research is motivated by the need to present a practical, standardized, and reproducible case study of DevOps implementation. The primary objective of this study is to design and implement a CI/CD pipeline in GitLab using a shared-pipeline scheme and job automation based on shell scripting to accelerate the development and deployment of web service applications (backend) at JEL Tech Inc. This approach is expected to reduce manual tasks, minimize human errors, and enhance the consistency of release processes in a microservices environment.

Unlike traditional CI/CD setups where pipeline logic is duplicated across individual repositories, leading to 'configuration drift' and maintenance complexity, this shared-pipeline scheme centralizes all orchestration logic within a single parent repository. Furthermore, while many implementations rely on complex third-party plugins or high-level programming languages, this approach utilizes modular shell scripts to provide greater transparency, flexibility, and easier maintenance for operations teams. Critically, this model also introduces automation unit testing for the pipeline scripts themselves using Bats-core library, ensuring the reliability of automation logic before it is deployed to production.

This study extends existing DevOps research by introducing a reusable shared-pipeline architecture that centralizes CI/CD orchestration across multiple repositories. Unlike conventional CI/CD implementations, this approach integrates modular shell script–based automation with automated testing, providing a practical,

standardized, and maintainable solution for microservices-based development environments.

MATERIALS AND METHODS

1. Research Methodology

This study uses an applied research method with a case study approach to examine DevOps implementation in developing microservices-based web service applications at JEL Tech Inc. This method supports direct validation of the shared-pipeline CI/CD architecture in a real industrial environment rather than relying only on theoretical analysis.

The approach is chosen because the research focuses on practical and technical DevOps implementation to overcome integration and deployment challenges in complex software development environments [21], [22]. The research methodology follows the DevOps lifecycle, which is operationalized through a shared-pipeline scheme in GitLab. In this setting, the parent repository centrally manages the orchestration logic via modular shell scripts, while individual microservices (child repositories) only define unique variables such as application name and version.

Each part of the process is executed as follows: (1) Plan: requirements are modeled into pseudocode; (2) Code: modular scripts are developed for automation; (3) Build: scripts and Java binaries are packaged; (4) Test: scripts undergo automated unit testing via Bats-core to prevent pipeline failure; (5) Release: successful code is merged and tagged; (6) Deploy: applications are pushed to Amazon ECR and deployed to Kubernetes (EKS) using Helm; (7) Operate: standardized configurations are applied across all services, and (8) Monitor: continuous feedback is gathered from the Kubernetes environment or user [23], [24]. Each stage is designed to support an automated, standardized, and continuous Continuous Integration and Continuous Delivery (CI/CD) process [25].

In this study, the researchers design a shared-pipeline scheme in GitLab and develop process automation using shell scripts as the primary mechanism to fulfill continuous integration and deployment requirements. This approach is intended to support operations teams in managing CI/CD pipelines in a standardized manner across multiple microservices repositories, thereby enabling software release processes to be executed consistently and efficiently.

2. Research Environment and Tools

This research is conducted within a cloud-based software development environment that utilizes a centralized repository management system and a microservices architecture. As the number of services grows, each microservice is maintained in a separate repository, thereby requiring a CI/CD mechanism that is consistent, scalable, and easy to maintain.

The materials used in this study are summarized in Table 1.

Table 1. DevOps Tools Used in the Research

No	Name of tools	Function
1	GitLab	Source code repository management and CI/CD pipeline orchestration
2	GitLab Runner	Execution of CI/CD pipeline jobs
3	Docker	Packaging applications into Docker images
4	Amazon Elastic Container Registry (ECR)	Storage and management of Docker images
5	Helm	A tool to deploy applications to Kubernetes
6	Shell Script	Automation language of build until deployment
7	Bats-core	Unit testing of shell scripts
8	Elastic Kubernetes Service (EKS)	An orchestrator of container-based applications

Source: (Research Results, 2026)

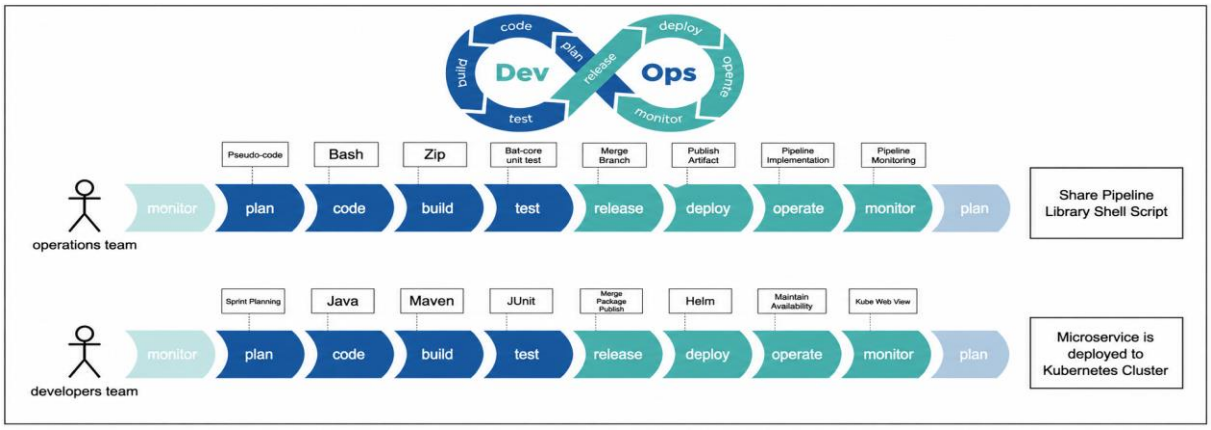
The selected tools were chosen based on their integration capabilities, scalability, and compatibility with microservices-based CI/CD environments. Compared to alternative platforms such as Jenkins or GitHub Actions, GitLab provides integrated repository management and pipeline orchestration within a single platform, simplifying CI/CD management. In addition, Amazon EKS and Helm were selected to support scalable container orchestration and automated deployment in cloud-native environments. The combination of these tools allows the process from development to application implementation to be carried out in an integrated and automated manner.

3. DevOps-Based Research Workflow

The research methodology workflow is designed based on the DevOps lifecycle to support CI/CD automation in a microservices environment [23], [24]. This workflow is illustrated in Figure 1, which depicts the collaboration between development and operations teams through a shared-pipeline implemented in GitLab, as well as the integration of containerization technologies and Kubernetes for the deployment of web service applications.

Accredited Rank 2 (Sinta 2) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No.225/E/KPT/2022, December 07, 2022. Published by LPPM Universitas Nusa Mandiri

JITK (JURNAL ILMU PENGETAHUAN DAN TEKNOLOGI KOMPUTER)
VOL. 12. NO. 1 AUGUST 2026
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v12i1.8208

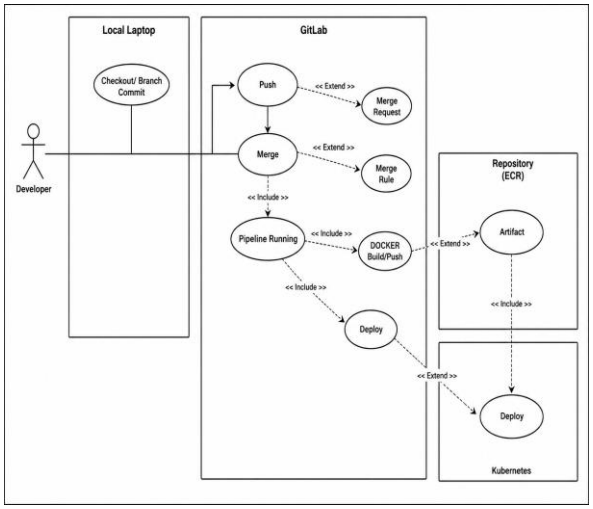


Source: (Research Results, 2026)

Figure 1. Research Methodology Workflow Based on DevOps Lifecycle

A. Plan

In the implementation phase, the researchers develop and configure the shared-pipeline CI/CD architecture using DevOps tools and container-based technologies to support automated build, testing, and deployment processes. The developed system is then evaluated through functional testing, deployment monitoring, and performance analysis to assess the effectiveness, consistency, and scalability of the proposed pipeline architecture. This phase is critical as it ensures alignment between development and operations requirements, reducing miscommunication and enabling a structured foundation for CI/CD automation.



Source: (Research Results, 2026)

Figure 2. Use Case Diagram of CI/CD Workflow in DevOps Planning Phase

Figure 2 illustrates the interaction workflow among developers, the GitLab repository, the container registry, and the Kubernetes cluster within the CI/CD process, starting from source code management to application deployment.

B. Code

Furthermore, containerization technologies and orchestration tools are integrated to ensure reliable deployment and environment consistency across development and production stages. The implemented shared-pipeline also supports centralized maintenance and easier scalability, allowing the operations team to efficiently manage updates, monitor deployment activities, and accelerate the software delivery process.

C. Build

The final shell scripts are packaged into archive file format (.zip). This approach is adopted because shell scripts do not undergo a compilation process like high-level programming languages. Therefore, packaging is used to maintain version consistency and facilitate the distribution of pipeline artifacts. This phase ensures artifact consistency and reproducibility, which are fundamental for reliable CI/CD processes and version control.

D. Test

Shell script testing is conducted using the Bats-core library to ensure that each function operates as expected. Unit testing includes validation of helper functions, application build processes, authentication to the container registry, Docker image creation, and deployment procedures. If any test fails, the pipeline is automatically terminated. This phase plays a crucial role in ensuring pipeline reliability by detecting errors early, thereby preventing faulty deployments and improving system stability.

E. Release

The release phase is performed by merging code changes that have successfully passed the

testing stage from a feature branch into the main branch of the GitLab repository. This process is accompanied by version increments (patch or minor) using a tagging mechanism to ensure consistency and traceability of pipeline artifacts. This phase ensures traceability and version control, enabling controlled and consistent delivery of validated pipeline configurations.

F. Deploy

During the deployment phase, the released shell script artifacts are stored as GitLab artifacts and utilized by child pipelines in each microservices repository. These pipelines automatically execute application deployment to the Kubernetes cluster using Helm. This phase is significant for enabling automated and repeatable deployment, reducing manual intervention and accelerating application delivery.

G. Operate

The operational phase involves applying the CI/CD pipeline across all microservices repositories. The operations team ensures that deployed applications run in a healthy state and comply with the predefined configuration settings. This phase ensures operational stability by maintaining system health and validating that deployed applications meet expected performance criteria.

H. Monitor

In the monitoring phase, the researchers observe the execution results of the CI/CD pipeline as well as the status of applications running in the Kubernetes cluster. If errors or bugs are identified, the development process returns to the planning stage for corrective actions, in accordance with the principle of continuous improvement in DevOps. This phase supports continuous improvement by providing feedback on system performance and pipeline execution, enabling iterative enhancements in the DevOps lifecycle.

RESULTS AND DISCUSSION

1. CI/CD Automation using Shared-pipeline

The primary outcome of this study is the implementation of a shared-pipeline–based CI/CD automation that can be applied across multiple microservices repositories. The automation is developed based on the requirements analysis and system modeling conducted during the planning phase, enabling the automation of previously manual build, packaging, and deployment activities. The resulting pipeline supports the entire development lifecycle, from source code changes to deployment on a Kubernetes cluster.

Using the shared-pipeline approach, all microservices repositories follow a standardized CI/CD workflow, as illustrated in Figure 8. The workflow diagram demonstrates the orchestration of the build, package, and deployment stages through a consistent and validated process. These findings indicate that the shared-pipeline not only improves automation but also standardizes CI/CD implementation across repositories, thereby reducing configuration inconsistencies caused by individual developers.

2. Core Pseudocode of CI/CD Automation workflow

Based on the results of the use case analysis and pipeline design, the CI/CD automation workflow is formalized into a core pseudocode that represents the main orchestration logic of the shared-pipeline. This pseudocode serves as a blueprint for implementing shell scripts within the GitLab CI/CD environment.

```
INPUT: app_name, namespace, version,
registry_credential
BEGIN
  command_check(mvn)
  command_check(docker)
  command_check(helm)
  build_java()
  login_container_registry(registry_credential)
  build_and_push_docker_image(app_name, version)
  deploy_application(app_name, namespace, version)
END
```

Source: (Research Results, 2026)

Figure 3. Algorithm Core CI/CD Automation Workflow

Figure 3 presents the core pseudocode of CI/CD automation workflow utilized in this study. The pseudocode illustrates the pipeline execution sequence, beginning with dependency checks (Maven, Docker, and Helm), followed by the application build process, authentication to the container registry, Docker image creation and publishing, and concluding with application deployment to Kubernetes. This workflow serves as a blueprint for implementing shared-pipeline–based CI/CD automation.

3. Implementation of Shell Script Automation

The implemented shell scripts are integrated into each stage of the GitLab shared-pipeline to ensure consistent execution across all microservices repositories. As shown in Figure 4, the scripts orchestrate the CI/CD workflow sequentially, enabling automated and standardized build, packaging, and deployment processes. This implementation also improves deployment reliability, reduces manual intervention, and

Accredited Rank 2 (Sinta 2) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No.225/E/KPT/2022, December 07, 2022. Published by LPPM Universitas Nusa Mandiri

VOL. 12. NO. 1 AUGUST 2026
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v12i1.8208

accelerates software delivery within the DevOps environment.

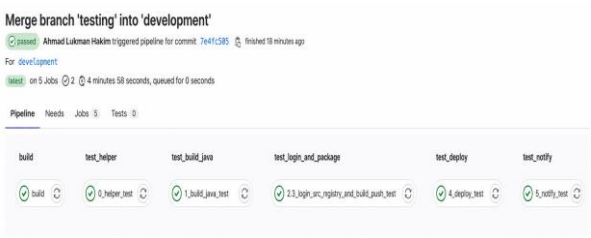
```
#!/bin/bash

function command_check {
 if ! command -v "$1" > /dev/null 2>&1; then
  echo "command not found, please install first"
  return 1
 else
  return 0
 fi
}

function build_java {
 command_check mvn
 mvn package -B
}

function login_ecr_registry {
 local password
 password=$1

 command_check docker

 docker login --username AWS --password $password
001.dkr.ecr.ap-southeast-1.amazonaws.com
}

function build_push_docker_image {
 local image_full_name
 image_full_name=$1

 command_check docker

 docker build -t $image_full_name .
 docker push $image_full_name
}

function deploy {
 local app_name
 local namespace
 local version

 app_name=$1
 namespace=$2
 version=$3

 command_check helm

 helm upgrade -i ${app_name} ./mono-java-apps -n
${namespace} \
 -f ./mono-manifest/mono-${app_name}/values-
${namespace}.yml \
 --set image.tag="${version}" \
 --wait \
 --atomic \
 --timeout 4m
}
```

Source: (Research Results, 2026)

Figure 4. Algorithm Core CI/CD Shell Script Automation

4. Automated Testing Results

The automated testing results show that all shell script functions operate successfully according to the predefined scenarios. In addition, the fail-fast mechanism that automatically stops the pipeline when errors are detected prevents invalid processes from reaching the deployment stage, thereby improving pipeline stability and minimizing the risk of production failures. To support reproducibility, the CI/CD testing pipeline used in this study is also publicly available through a GitLab repository.



Source: (Research Results, 2026)

Figure 5. Pipeline of Shell script Development

Figure 5 illustrates the successful execution of the CI/CD pipeline in GitLab after the branch merge process. The figure shows that all pipeline jobs, including build and test stages, were completed sequentially without errors, confirming that the proposed automation workflow functions correctly and is ready for application release activities.

Table 2. List and Scope of Shell Script Job.

No	Job Name	Scope
1	Build	Package the Shell script to .zip
2	Test helper	A testing script to test helper function
3	Test Build Java	A testing script to test build java command
4	Test Login and Package	A testing script to test login and package docker image command
5	Test Deploy	A testing script to test deployment

Source: (Research Results, 2026)

Table 2 describes the list of jobs and the scope of testing in the shell script development pipeline, demonstrating that each core function within the scripts is tested independently, from the build process to deployment. This approach ensures that every CI/CD automation component is validated prior to its use in the application development pipeline. All test cases were successfully executed during the pipeline process, indicating that the automation scripts meet the expected functional requirements.

5. Shared Pipeline Integration Results

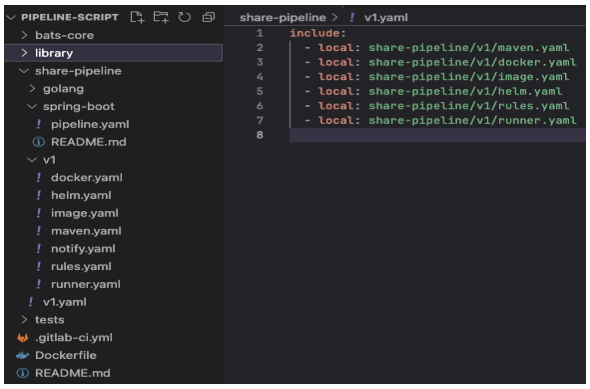
The implementation results indicate that the child pipelines in each microservices repository

successfully invoke the shared-pipeline as the parent pipeline. The child pipelines contain only application-specific variable configurations, such as application name, namespace, and version, while the entire CI/CD logic is centrally managed within the shared pipeline.

```
include:
  - project: mono-pipline/devops/pipeline-script
    ref: development
    file: share-pipeline/v1.yaml
  - project: mono-pipline/devops/pipeline-script
    ref: development
    file: share-pipeline/spring-boot/pipeline.yaml

variables:
  VERSION: "${CI_COMMIT_REF_NAME}-
${CI_COMMIT_SHORT_SHA}"
  DOCKER_IMAGE_FULL_NAME: "001.dkr.ecr.ap-
southeast-1.amazonaws.com/infrastructure:$VERSION"
  APP_NAME: "infrastructure"
  NAMESPACE: "demo"
  SCRIPT_VERSION: "1.0.11
```

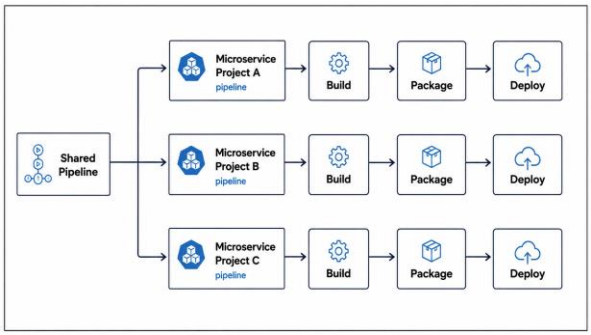
Source: (Research Results, 2026)
Figure 6. Child Pipeline Integration

Figure 6 illustrates the configuration of the child pipeline in GitLab that invokes the shared-pipeline through the include mechanism. The child pipeline defines only application-specific variables, such as version, application name, namespace, and Docker image, while the entire CI/CD logic is centrally managed within the shared-pipeline.



Source: (Research Results, 2026)
Figure 7. Shared-pipeline of GitLab

As shown in Figure 7, the modular repository structure supports better maintainability and scalability by separating CI/CD functions into reusable configuration files. This design allows operations teams to manage pipeline components independently while preserving consistency across repositories. In addition, the centralized shared-pipeline architecture accelerates configuration updates, minimizes human error, and improves the efficiency of CI/CD implementation in microservices-based application development.



Source: (Research Results, 2026)
Figure 8. How Shared-pipeline Works

Figure 8 illustrates the working mechanism of the shared-pipeline, in which a single parent pipeline is shared across multiple microservices pipelines. Each microservice executes the same build, package, and deploy stages through the shared-pipeline, thereby ensuring that the CI/CD process is standardized, consistent, and easy to maintain.

For each microservices repository, the researchers add a required dependency in the form of a Dockerfile. This Dockerfile is used by Docker to enable the GitLab Runner to build an image, which is subsequently executed as a container within the Kubernetes environment:

```
FROM eclipse-temurin:17-jre-alpine
COPY target/*.jar app.jar
ENTRYPOINT java -jar app.jar
```

Source: (Research Results, 2026)
Figure 9. Dockerfile for Java-Based Web Service Deployment

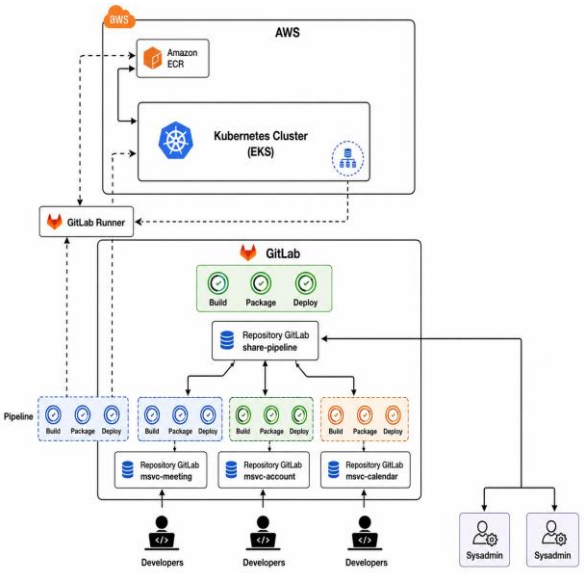
Figure 9 presents the Dockerfile configuration used to package a Java application into a Docker image based on the Eclipse Temurin JRE. The Dockerfile copies the application binary (.jar) into the container and defines the main execution command, enabling the application to run consistently within a containerized environment and be deployed through the CI/CD pipeline.

6. System Architecture and Deployment Results

The resulting system architecture demonstrates a clear separation of responsibilities between the development and operations teams. The development team focuses on implementing application source code, while the operations team provides a tested and ready-to-use shared-pipeline. Each code change pushed to the GitLab repository automatically triggers the CI/CD pipeline until the application is successfully deployed to the Kubernetes cluster.

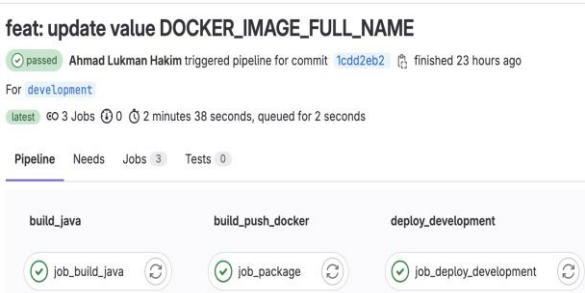
Accredited Rank 2 (Sinta 2) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No.225/E/KPT/2022, December 07, 2022. Published by LPPM Universitas Nusa Mandiri

VOL. 12. NO. 1 AUGUST 2026
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v12i1.8208



Source: (Research Results, 2026)
Figure 10. System Architecture Design

Figure 10 presents the shared-pipeline–based CI/CD architecture, where multiple GitLab microservices repositories use a centralized parent pipeline executed by GitLab Runner. The pipeline integrates with ECR and Elastic Kubernetes Service (EKS) to automate build, packaging, and deployment processes in a standardized manner. The implementation results show that applications can be deployed consistently across development and production environments. This automation replaces manual deployment activities with repeatable and reliable processes using a unified configuration.



Source: (Research Results, 2026)
Figure 11. Pipeline of Web service Development

Figure 11 presents the successful execution results of the CI/CD pipeline in GitLab following an update to the Docker image variable configuration. All major stages—including application build, Docker image creation and publish, then deployment to the development environment—were executed without failure, confirming the reliability of the implemented CI/CD pipeline.

Table 3. List and Scope of Job Web service development

No	Job Name	Scope
1	Build Java	Build Java to a .jar
2	Build and Push	Package and push the .jar as a Docker image
3	Deploy	Deploy the application using Helm to Kubernetes cluster

Source: (Research Results, 2026)

Table 3 describes the list of jobs and the scope of processes in the web service development pipeline, which include building the Java application into a .jar file, creating and pushing the Docker image to the container registry, then deploying the application to Kubernetes using Helm.

The structure of these jobs illustrates a concise and standardized CI/CD workflow that supports automated web service application releases. The improvements in CI/CD performance are influenced by key implementation decisions in this study. The shared-pipeline architecture centralizes CI/CD configuration, reducing duplication and ensuring consistent pipeline execution, which contributes to faster deployment and fewer errors.

In addition, modular shell script automation improves maintainability and enables easier testing, thereby reducing failure rates. The integration within GitLab further enhances process efficiency by combining source code management and pipeline execution. These decisions collectively lead to improved deployment speed, release frequency, and system reliability.

7. Discussion

This study aligns with DevOps maturity models and software automation theory, which emphasize progressive improvement through automation, standardization, and continuous integration practices. The implementation of a shared-pipeline reflects a higher level of DevOps maturity by enabling centralized control, reusable automation, and reduced process variability across microservices environments. Furthermore, these findings support software automation theory, where standardized automation reduces human-induced errors and enhances process repeatability, as reflected in the significant improvements in deployment time, release frequency, and failure rate observed in this study. The results demonstrate that the shared-pipeline approach with shell script automation successfully improves the consistency and standardization of CI/CD processes in a microservices-based environment. By centralizing pipeline logic in a single parent repository, operations teams can manage, maintain, and update CI/CD configurations more efficiently without

modifying individual microservices repositories. This centralized approach also reduces configuration inconsistencies previously caused by different developer practices. In addition, the use of modular shell scripts provides flexibility in customizing build, packaging, and deployment processes, while automated testing with Bats-core increases pipeline reliability before deployment. These findings support previous studies highlighting the benefits of DevOps and CI/CD adoption in improving software delivery and system stability.

The main contribution of this study is its detailed implementation of a reusable shared-pipeline architecture integrated with shell script automation and testing mechanisms, which remains less explored in existing DevOps research. A quantitative evaluation was conducted by comparing key performance indicators (KPIs) from the manual deployment era at JEL Tech Inc against the newly implemented shared-pipeline scheme. This evaluation focuses on critical metrics including deployment time, release frequency, and failure rates to objectively measure the impact of CI/CD automation on the backend microservices environment. The percentage of improvement for each metric was calculated using the following standard efficiency formula:

$$Improvement\ (\%) = \frac{Value_{manual} - Value_{CI/CD}}{Value_{manual}}\ x\ 100 \qquad (1)$$

Table 4. Definition of CI/CD Performance Metrics

Metric	Description
Deployment Time (average)	Total time from code merge to successful production running.
Release Frequency	How often new code or updates are successfully deployed
Pipeline/Manual Failure Rate	Percentage of deployments that result in errors or manual rollback.
Setup Time for New Service	Time required to configure CI/CD for new microservice repository

Source: (Research Results, 2026)

Table 5. Performance Comparison Before and After Implementation

Metric	Manual Process (before)	Shared-Pipeline (after)	Improvement (%)
Deployment Time (average)	45 - 60 Minutes	3 - 5 minutes	~90%
Release Frequency	1 -2 times per week	5 - 10 times per day	~400%
Pipeline/Manual Failure Rate	20 - 30 %	< 2%	~90%
Setup Time for New Service	3 - 4 hours	20 - 30 minutes	~97%

Source: (Research Results, 2026)

The implementation of the shared-pipeline scheme at JEL Tech Inc significantly reduced the Deployment Time from an average of 60 minutes (manual) to just 5 minutes (automated), representing a 91.6% efficiency improvement. These findings are consistent with DevOps performance theory, which emphasizes automation, standardization, and continuous feedback as key factors in improving deployment frequency, reducing lead time, and enhancing system reliability. The significant improvements observed in this study, particularly in deployment time (~90%) and release frequency (~400%), indicate that the adoption of automated and standardized CI/CD processes effectively reduces process variability and supports high-performance software delivery in microservices environments.

CONCLUSION

This study implemented a DevOps-based CI/CD approach using a reusable shared-pipeline architecture in GitLab combined with modular shell script automation to support microservices-based web service development. The proposed approach successfully standardized CI/CD processes across multiple repositories, reduced manual deployment activities, and improved the consistency and reliability of application release processes through automated build, testing, packaging, and deployment mechanisms. The findings demonstrate that the shared-pipeline model improves collaboration between development and operations teams by centralizing orchestration logic and simplifying CI/CD maintenance. In addition, the integration of automated shell script testing using the Bats-core library increased the reliability of automation workflows before deployment into production environments. This research contributes to DevOps implementation studies from three perspectives. Theoretically, it extends existing CI/CD research through a reusable and centralized shared-pipeline architecture. Practically, it presents a real-world implementation model for microservices environments with improved operational consistency and reduced human error. Methodologically, the study applies an applied research approach based on the DevOps lifecycle to validate CI/CD implementation in an industrial

Accredited Rank 2 (Sinta 2) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No.225/E/KPT/2022, December 07, 2022. Published by LPPM Universitas Nusa Mandiri

VOL. 12. NO. 1 AUGUST 2026
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v12i1.8208

context. Furthermore, the proposed approach can be adapted to multi-cloud environments and applied across different programming languages and microservices platforms. This study is limited to a single case study at JEL Tech Inc and a specific technology stack consisting of GitLab, Docker, and Amazon EKS, which may affect the generalizability of the findings. Future research may explore the integration of DevSecOps practices and intelligent CI/CD automation approaches to further improve scalability, security, and operational efficiency in complex software systems.

REFERENCE

[1] T. P. Böttcher, S. Empelmann, J. Weking, A. Hein, and H. Krcmar, "Digital sustainable business models: Using digital technology to integrate ecological sustainability into the core of business models," *Information Systems Journal*, vol. 34, no. 3, pp. 736–761, 2024, doi: 10.1111/isj.12436.

[2] J. Xu, S. She, and W. Liu, "Role of digitalization in environment, social and governance, and sustainability: Review-based study for implications," *Front. Psychol.*, vol. 13, Nov. 2022, doi: 10.3389/fpsyg.2022.961057.

[3] A. Alam and A. Mohanty, "Business Models, Business Strategies, and Innovations in EdTech Companies: Integration of Learning Analytics and Artificial Intelligence in Higher Education," in 2022 IEEE 6th Conference on Information and Communication Technology (CICT), Nov. 2022, pp. 1–6. doi: 10.1109/CICT56698.2022.9997887.

[4] F. M. Mulaydinov, "Integrated Information Systems in Industrial Enterprises and Business Activities," Journal of Computer Science Engineering and Information Technology Research (JCSEITR), vol. 13, no. 1, pp. 33–44, 2024.

[5] V. Velepucha and P. Flores, "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges," IEEE Access, vol. 11, pp. 88339–88358, 2023, doi: 10.1109/ACCESS.2023.3305687.

[6] F. E. Aouni, K. Moumane, A. Idri, M. Najib, and S. U. Jan, "A systematic literature review on Agile, Cloud, and DevOps integration: Challenges, benefits," Information and Software Technology, vol. 177, p. 107569, Jan. 2025, doi: 10.1016/j.infsof.2024.107569.

[7] Thalary, S. and Katipelly, A., Platform Engineering for Distributed Systems: How Cloud DevOps Enables Scalable, Policy-Driven Software Architectures. "International Journal of AI, BigData, Computational and Management Studies", 2025, 6(1), pp.189-197, doi. 10.63282/3050-9416.IJAIBDCMS-V6I1P119.

[8] M. Moeez, R. Mahmood, H. Asif, M.W Iqbal, K. Hamid, U. Ali, N. Khan., "Comprehensive Analysis of DevOps: Integration, Automation, Collaboration, and Continuous Delivery," *Bulletin of Business and Economics (BB*E), vol. 13, no. 1, Mar. 2024, doi: 10.61506/01.00253.

[9] E. M. Arvanitou, A. Ampatzoglou, S. Bibi, A. Chatzigeorgiou, and I. Deligiannis, "Applying and Researching DevOps: A Tertiary Study," IEEE Access, vol. 10, pp. 61585–61600, 2022, doi: 10.1109/ACCESS.2022.3171803.

[10] J. Zhou, P. Brereton, and K. Campbell, "Building an intelligent food assurance system based on DevOps: A review," Future Foods, vol. 12, p. 100847, Dec. 2025, doi: 10.1016/j.fufo.2025.100847.

[11] "Improving Software Development with Continuous Integration and Deployment for Agile DevOps in Engineering Practices," IJCATR, Jan. 2025, doi: 10.7753/IJCATR1401.1002.

[12] A. Nayanajith and R. Wickramarachchi, "Challenges Affecting the Successful Adoption of DevOps Practices: A Systematic Literature Review," in 2024 4th International Conference on Advanced Research in Computing (ICARC), Feb. 2024, pp. 311–315. doi: 10.1109/ICARC61713.2024.10499735.

[13] R. Grande, A. Vizcaíno, and F. O. García, "Is it worth adopting DevOps practices in Global Software Engineering? Possible challenges and benefits," Computer Standards & Interfaces, vol. 87, p. 103767, Jan. 2024, doi: 10.1016/j.csi.2023.103767.

[14] N. Azad and S. Hyrynsalmi, "DevOps critical success factors — A systematic literature review," Information and Software Technology, vol. 157, p. 107150, May 2023, doi: 10.1016/j.infsof.2023.107150.

[15] El Aouni, F., Moumane, K., Idri, A., Najib, M. and Jan, S.U., A systematic literature review on Agile, Cloud, and DevOps integration: Challenges, benefits. "Information and Software Technology", vol. 177, p.107569, 2025, doi. 10.1016/j.infsof.2024.107569.

[16] Z. Alamin, Dahlan, Khaeruddin, and S. Ramadhan, "Evolving DevOps Practices in Modern Software Engineering: Trends, Challenges, and Impacts on Quality and Delivery Performance," Journix: Journal of

Informatics and Computing, vol. 1, no. 1, pp. 21–29, Mar. 2025, doi: 10.63866/journix.v1i1.4.

[17] R. Hernández, B. Moros, and J. Nicolás, "Requirements management in DevOps environments: a multivocal mapping study," Requirements Eng, vol. 28, no. 3, pp. 317–346, Sep. 2023, doi: 10.1007/s00766-023-00396-w.

[18] M. L. Gupta, R. Puppala, V. V. Vadapalli, H. Gundu, and C. V. S. S. Karthikeyan, "Continuous Integration, Delivery and Deployment: A Systematic Review of Approaches, Tools, Challenges and Practices," in *Recent Trends in AI Enabled Technologies*, G. Paidi, S. V. Gangashetty, and A. K. Varma, Eds., Cham: Springer Nature Switzerland, 2024, pp. 76–89. doi: 10.1007/978-3-031-59114-3_7.

[19] 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 (Basel), vol. 24, no. 18, p. 6002, Sep. 2024, doi: 10.3390/s24186002.

[20] J. J. López-Jiménez, J. Pérez-Sánchez, J. M. Carrillo-de-Gea, J. Nicolás Ros, and J. L. Fernández-Alemán, "Human DevOps: A tool for measuring and enhancing human factors in DevOps adoption," SoftwareX, vol. 33, p. 102515, Feb. 2026, doi: 10.1016/j.softx.2026.102515.

[21] L. Giamattei, A. Guerriero, R. Pietrantuono, et al., "Monitoring tools for DevOps and microservices: A systematic grey literature review," Journal of Systems and Software, vol. 208, p. 111906, Feb. 2024, doi: 10.1016/j.jss.2023.111906.

[22] Gomes, F., Rego, P. and Trinta, F., A systematic mapping study on observability of microservices-based applications: fundamentals, classifications, and challenges. "Computing", 2025, 107(9), p.183. doi. 10.1007/s00607-025-01540-w.

[23] R. K. Mahimalur, M. Vasagam, and D. Manoharan, "Devops Lifecycle Management And Cloud Migration Assessments: A Security-Driven CICD Perspective," Journal of International Crisis and Risk Communication Research, vol. 7, no. S10, pp. 3314–3322, Oct. 2024, doi: 10.63278/jicrcr.vi.3670.

[24] A. Kumar, M. Nadeem, and M. Shameem, "Assessment of DevOps Lifecycle Phases and their Role in DevOps Implementation using Best–Worst MCDM," Int. j. inf. tecnol., vol. 16, no. 4, pp. 2139–2147, Apr. 2024, doi: 10.1007/s41870-023-01566-3.

[25] A. Saxena, S. Singh, S. Prakash, T. Yang, and R. S. Rathore, "DevOps Automation Pipeline Deployment with IaC (Infrastructure as Code)," in *2024 IEEE Silchar Subsection Conference (SILCON 2024)*, Nov. 2024, pp. 1–6. doi: 10.1109/SILCON63976.2024.10910699

Accredited Rank 2 (Sinta 2) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No.225/E/KPT/2022, December 07, 2022. Published by LPPM Universitas Nusa Mandiri