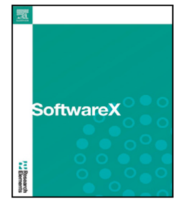
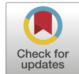
SoftwareX 32 (2025) 102414

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx





Automating Docker image deployment across network-segmented environments

Guillermo Bermejo [a], Ángel Macías [a], Juan Luis Herrera [c], Sergio Laso [a,b,*], Javier Berrocal [a]

[a] Universidad de Extremadura, Badajoz, Spain
[b] Global Process and Product Improvement S.L., Cáceres, Spain
[c] TU Wien, Vienna, Austria

ARTICLE INFO

Keywords:
Docker
Microservices
Orchestration
Automation
DevOps
CI/CD

ABSTRACT

In security-sensitive or regulated environments — such as banking, healthcare, or industrial control systems — strict network segmentation policies prevent direct communication between development and production infrastructure. As a result, software delivery processes in these contexts often rely on manual workflows, including detecting new Docker images, transferring them across isolated domains, and manually applying deployment updates. This paper presents a self-managed, lightweight CI/CD framework specifically designed for such disconnected environments. Rather than managing containers directly, the system automates a critical subset of the DevOps workflow: the detection, transfer, and deployment of updated Docker images across network-isolated zones. It operates from a bastion host with access to both segments, utilizing open-source tools: Diun for monitoring external Docker registries, Skopeo for transferring images securely between registries, and 'kubectl' for updating the corresponding Kubernetes deployments. Notifications are sent via Postfix to maintain traceability at every stage of the process. The main contribution of this work lies in adapting DevOps automation principles to segmented infrastructures without relying on cloud services or central control, a scenario largely unsupported by existing tools. The proposed solution requires no internet access, cloud platforms, or third-party services, making it suitable for environments with strict connectivity restrictions. It is modular, reproducible, and vendor-neutral. Validation in a simulated enterprise scenario confirms the system's reliability across both successful and failure cases. By targeting the image propagation stage of the deployment pipeline, this work contributes a practical, focused automation tool for CI/CD under constrained network conditions. Source code and deployment artifacts are publicly available to facilitate reuse in similarly restricted environments.

Code metadata

Current code version	v1.0.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-25-00557
Permanent link to Reproducible Capsule	N/A
Legal Code License	CC BY-NC 4.0
Code versioning system used	git
Software code languages, tools, and services used	Bash, YAML, Docker, Skopeo, kubectl, Postfix, Diun.
Compilation requirements, operating environments & dependencies	Ubuntu 22.04, Docker, Skopeo, kubectl, Postfix, Diun container, optional Helm, Minikube
If available Link to developer documentation/manual	Documentation
Support email for questions	slasom@unex.es

* Corresponding author at: Universidad de Extremadura, Badajoz, Spain.
E-mail addresses: guillermobb@unex.es (Guillermo Bermejo), amaciba@unex.es (Ángel Macías), juan.gonzalez@tuwien.ac.at (Juan Luis Herrera), slasom@unex.es (Sergio Laso), jberolm@unex.es (Javier Berrocal).

https://doi.org/10.1016/j.softx.2025.102414
Received 30 July 2025; Received in revised form 1 October 2025; Accepted 15 October 2025
Available online 29 October 2025
2352-7110/© 2025 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

1. Motivation and significance

In containerized DevOps pipelines, Docker images [1] are the primary unit of delivery, and orchestration platforms like Kubernetes [2] enable automated deployment at scale [3,4]. Automating the propagation of images from development to production environments is essential for maintaining efficiency, traceability, and agility in continuous integration and deployment workflows [5,6]. However, in many regulated or security-constrained environments—such as finance, healthcare, or critical infrastructure— there are strict network segmentation policies [7]. These policies do not allow direct communication between external development environments and internal infrastructure [8,9]. This creates a major operational challenge: detecting, transferring, and deploying updated Docker images across disconnected environments often requires manual procedures, undermining the automation typically expected in modern Continuous Integration/Continuous Deployment (CI/CD) pipelines.

In recent years, several works have proposed CI/CD architectures aimed at optimizing the deployment of containerized applications using Kubernetes in cloud-connected environments. Kokku [10] presents a conceptual overview of integrating CI/CD pipelines into Kubernetes-managed infrastructures, highlighting benefits in deployment speed, scalability, and operational resilience. Mercy [11] complements this with a comparative analysis of Kubernetes-native automation tools, evaluating their performance in terms of deployment time, resource efficiency, and fault recovery. Saleh et al. [12], through a systematic literature review, identify security, image management, and automation as critical challenges in modern CI/CD pipelines. Recent works address automation in diverse domains, such as deep learning–based caching for intelligent transportation [13], multi-objective optimization in the food–energy–water nexus [14], and energy-efficient clustering in wireless sensor networks [15]. Although not based on automated synchronization and deployment of Docker images across segmented networked environments, it highlights the importance of these aspects.

However, all these contributions address scenarios where CI/CD systems and Kubernetes clusters are interconnected through unrestricted networks. In these conventional infrastructures, automated pipelines operate by directly pushing container images to internal registries and triggering deployments in the target clusters.

Unlike these cloud-connected approaches, segmented network architectures in regulated environments prevent any direct communication between CI/CD systems and production clusters, requiring manual operation from the infrastructure team for detecting updates to images, transferring them, and updating existing deployments [16]. These repetitive tasks are time-consuming, error-prone, and incompatible with agile development methodologies.

Existing automation tools, such as Watchtower [17] or Keel [18], are designed for connected environments and rely on direct registry-to-cluster communication. Similarly, platforms like CloudStack [19] focus on virtual machine orchestration and do not address the specific problem of synchronizing Docker images across isolated networks for automated deployment in Kubernetes clusters.

To address these operational constraints, we present a fully self-managed CI/CD automation framework specifically designed for isolated Kubernetes environments. This framework enables automated detection, transfer, and deployment of Docker images without requiring external connectivity or third-party services. The system integrates several open-source tools: Diun [20] for monitoring external registries, Skopeo [21] for transferring Docker images between disconnected registries, and kubectl for updating deployments within a private Kubernetes cluster (emulated with Minikube [22]). All components are executed on a bastion host with access to both registries, and notification emails are sent via Postfix to ensure full traceability of the update process.

The proposed solution provides a lightweight, reproducible, and vendor-neutral alternative to commercial CI/CD platforms. It is particularly suitable for organizations that require deployment automation while adhering to strict network segmentation or offline policies. Its modular structure allows integration into broader DevOps pipelines, and its configuration can be adapted to suit different infrastructures without relying on external cloud services. The approach is validated through a simulated production environment and contributes a practical tool to support automation in constrained enterprise contexts. Unlike general-purpose CI/CD automation tools, our solution is optimized for disconnected environments and prioritizes secure, deterministic operations over cloud-centric assumptions.

The contribution of this work lies in adapting DevOps automation principles to segmented infrastructures through a fully offline and self-managed pipeline. While existing solutions assume continuous connectivity and centralized orchestration, our framework provides a reproducible and traceable deployment workflow tailored to air-gapped and policy-constrained environments. By combining open-source components under a unified architecture, we address a deployment scenario that remains underserved in the current literature and unsupported by mainstream CI/CD tools. To our knowledge, this is the first open-source implementation focused on the automated propagation of Docker images across disconnected domains using only local infrastructure.

The remainder of this paper is organized as follows. Section 2 describes the software architecture and workflow of the proposed automation framework. Section 3 presents illustrative examples, including a simulated enterprise scenario, environment configuration, and validation tests. Section 4 discusses the potential impact and applicability of the system in different domains. Finally, Section 5 concludes the work and outlines possible directions for future research and extensions.

2. Software description

The proposed software is a self-managed CI/CD automation framework designed to operate in isolated Kubernetes environments with restricted network connectivity. It enables the automatic synchronization and deployment of Docker images across decoupled infrastructures, where development and production environments cannot communicate directly. The solution combines several open-source tools within a modular, lightweight system that runs entirely from a bastion host.

2.1. System architecture

The system architecture consists of four main components: an external Docker registry (Harbor), managed by the development team; a bastion host acting as the CI/CD engine; an internal Docker registry deployed within the Kubernetes cluster; and the cluster itself. For this work, we use a cluster hosted locally via Minikube as a running example, although the framework also supports distributed Kubernetes clusters. These components are distributed across two isolated network zones, as illustrated in Fig. 1.

The bastion host is physically deployed within the internal network (network 1), where the Kubernetes cluster is located. It acts as the only controlled communication point for operations initiated from the external development network (network 2). It runs a containerized instance of Diun (Docker Image Update Notifier), responsible for monitoring the external registry for new versions of a given image. When a new version is detected, Diun triggers a shell script that transfers the image to the internal registry using Skopeo and updates the Kubernetes deployment via `kubectl`. Finally, a Postfix service sends automated email notifications to the infrastructure team, enabling full traceability.

This host constitutes the only controlled communication channel between the isolated environments, ensuring that no direct connections are established from the internal cluster to the external registry or to the internet. All monitoring, transfer, and deployment actions are coordinated exclusively from this host, providing a single point of control and simplifying auditing. The containerized nature of Diun allows for flexible deployment and easy customization of the monitoring

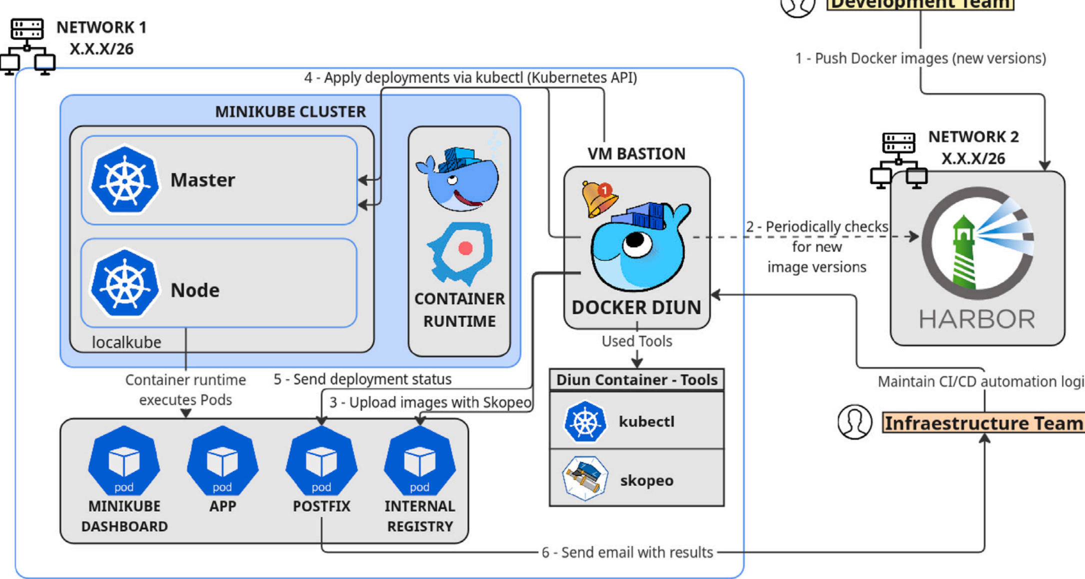


Fig. 1. System architecture and component interactions.

process, while the integration of `Skopeo` and `kubectl` in the same execution environment ensures that image transfer and deployment updates can be performed atomically. The email notification mechanism, implemented via Postfix, guarantees that infrastructure administrators are continuously informed of system activity, without requiring direct access to the internal Kubernetes cluster or the bastion host itself.

This architecture guarantees that the internal cluster never directly accesses the external registry or the internet. All communications are controlled and traceable via the bastion host, ensuring compliance with network segmentation and security policies.

The framework assumes a baseline of operational trust in the development team, particularly regarding the integrity and quality of published Docker images. While the bastion host is a critical infrastructure element, the system itself focuses on automating image transfer and deployment workflows. In real-world deployments, standard hardening practices such as SSH key-only access, container isolation, firewall restrictions, and audit logging should be applied to reduce the risk surface.

The operation of the system can be understood as a six-step sequential workflow, matching the interactions illustrated in Fig. 1.

1. Push Docker images (new versions): The external development team builds and pushes new versions of the application image to the Harbor registry. This registry is located in a separate network zone, inaccessible to the internal infrastructure.

2. Periodic image check with Diun: The bastion host runs Diun, which periodically checks Harbor for new image tags. Diun operates based on scheduled polling intervals, using its YAML configuration file to define the monitored image, tag filters, and the command to execute upon change detection.

Although the transfer process is automated, the system assumes that internal quality assurance (QA) procedures are followed before tagging and publishing Docker images in the external registry. This assumption is common in segmented environments where development and production are maintained by different teams or vendors. The framework focuses on automating the transfer and deployment actions, but it does not replace or bypass validation steps. In scenarios requiring stronger guarantees of trust and traceability, the workflow can be extended with container image signing tools such as `cosign`, allowing operators to verify image provenance before execution.

3. Image transfer using `Skopeo`: When a new image is detected in the external registry, Diun triggers a shell script on the bastion host. The first operation of this script uses `Skopeo` to copy the image directly from the external Harbor registry to the internal Docker registry deployed in the Kubernetes cluster. This registry-to-registry transfer avoids intermediary storage, ensuring secure and efficient synchronization across isolated environments.

4. Deployment update using `kubectl`: Once the image has been transferred, the script updates the target deployment within the Kubernetes cluster. If the deployment already exists, its container image is replaced using `kubectl set image`, triggering a rolling update. If it does not exist, the script creates a new deployment and exposes it via a NodePort service. This logic guarantees idempotent behavior regardless of the current state of the cluster.

5. Deployment status logging: After the deployment command is issued, the script captures the outcome and generates a short status report. Success and error messages are written to local log files stored on the bastion host. This information is later used to populate the body of the email notification sent in the final step.

6. Send email notification using Postfix: Finally, the script sends an automated email to the infrastructure team using `Postfix`, providing a summary of the operation and reporting success or failure.

Listing 1 shows an excerpt of the shell script executed by Diun on the bastion host. It consolidates the logic described in steps 3 to 6: transferring the image (step 3), updating the Kubernetes deployment (step 4), logging the outcome (step 5), and sending the final notification (step 6).

```
1  # Transfer the image using Skopeo
2  skopeo copy --src-tls-verify=false --dest-tls-verify=false \
3    docker://$DIUN_ENTRY_IMAGE \
4    docker://192.168.49.2:30500/${DIUN_ENTRY_IMAGE:28}
5
6  # Update or create the Kubernetes deployment
7  if kubectl get deployment "$DEPLOY_NAME" -n "$PROJECT"; then
8    kubectl set image deployment/$DEPLOY_NAME $DEPLOY_NAME=$IMAGE -
       n $PROJECT
9  else
10   kubectl create deployment $DEPLOY_NAME --image=$IMAGE -n
       $PROJECT
11 fi
12
13 # Send notification via Postfix
14 echo "Deployment completed: $DEPLOY_NAME" | \
15   sendmail -f "notifier@example.com" -t "admin@example.com"
```

Listing 1: Excerpt of the update script corresponding to steps 3 to 6: image transfer, deployment logic, and notification.

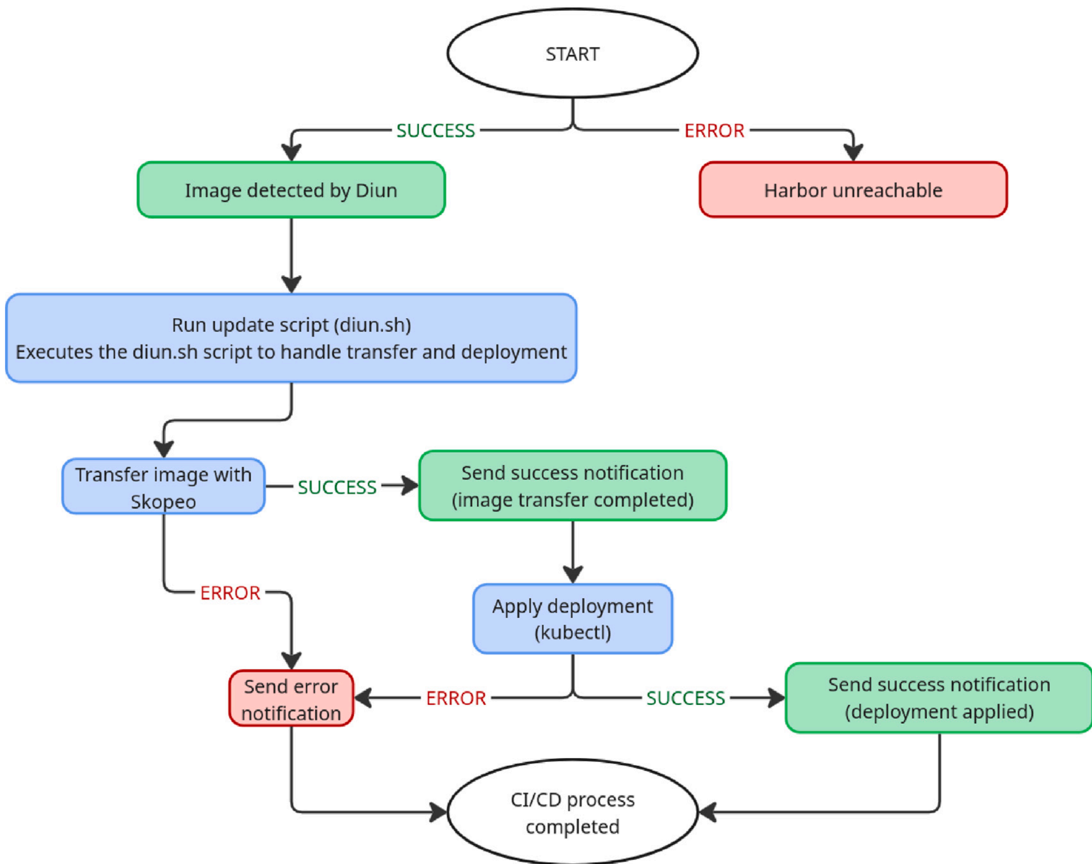


Fig. 2. Automated CI/CD workflow with error handling and notifications.

All actions are executed locally on the bastion host and logged for traceability. The architecture supports extension to multiple clusters or alternative notification mechanisms without modifying the core workflow.

The email notifications ensure that infrastructure administrators are kept informed of the deployment status without requiring direct access to the cluster or the bastion host.

In addition to these steps, all actions are logged locally on the bastion host for traceability and auditing purposes. The modularity of this approach allows the system to be extended to multiple clusters or adapted to alternative notification mechanisms if required.

2.2. Workflow description

The operational workflow is illustrated in Fig. 2. The process begins when `Diun`, running on the bastion host, detects the publication of a new Docker image in the external Harbor registry. This event triggers the execution of the `diun.sh` script, which automates both the secure image transfer using `Skopeo` and the deployment update in the internal Kubernetes cluster via `kubectl`. Success or error notifications are sent at each critical stage of the process to ensure traceability.

The workflow accounts for possible failure scenarios:

- If Harbor is unreachable, Diun aborts the process and logs the failure.

- If the image transfer fails, the script halts and sends an error notification.

- If the deployment fails, a separate error notification is issued.

- If all stages are completed successfully, two success notifications are sent: one upon image transfer completion and one after deployment.

This sequential process ensures full automation while providing transparent monitoring and traceability through email notifications and local logging, both of which enable administrators to track system activity without requiring direct access to the production cluster. The current design assumes the presence of a manually configured bastion host with access to both network segments, acting as the sole point of control for the CI/CD workflow.

3. Illustrative examples

3.1. Example scenario

We simulated a realistic enterprise context in which a company operates an internal Kubernetes cluster that is fully isolated from the internet and external resources. The application development is outsourced to an external team that publishes new versions of a Docker image to a Harbor registry, which is not accessible from the internal infrastructure.

The infrastructure team must deploy these application versions in the development environment without any direct access to the external registry and without relying on manual intervention.

To demonstrate the use case, we defined a fictional application representing a back-end service used for internal billing and accounting operations. The goal is to ensure that the deployment environment always runs the latest available image version, automatically and securely. The source code of this example application is available in the public repository accompanying this article.

3.2. Environment configuration and workflow

The internal cluster is simulated using Minikube, with a private internal registry enabled via addons. Harbor is deployed as the external registry and can only be accessed from a bastion host that connects both network zones.

tfg-dev.core.harbor.dev.lab/tfg-app/nginx:0.0.3 is available

Diun 🔔 notification

Docker tag **tfg-dev.core.harbor.dev.lab/tfg-app/nginx:0.0.3** which you subscribed to through file provider is available on **tfg-dev.core.harbor.dev.lab** registry (triggered by *ubuntu* host).

This image has been created at `Apr 08, 2025 00:19:58 UTC` with digest `sha256:7874069815c7813ebe098cc62530aa0ef66dee8bf4b21cd2f9706dc0923f3507` for `linux/amd64` platform.

Need help, or have questions? Go to https://github.com/crazy-max/diun and leave an issue.

Thanks for your support,
Diun

Fig. 3. Email notification automatically generated by Diun upon detection of a new image version published in the external Harbor registry.

To reinforce network isolation, Calico is used to apply custom network policies that explicitly block traffic between the cluster and Harbor.

On the bastion host, Diun continuously monitors the external Harbor registry. When a new version of the `InternalBillingApp` image is detected, a script is triggered that uses Skopeo to transfer the image to the internal registry. The same script then updates the Kubernetes deployment using `kubectl`.

Finally, a notification email is sent using Postfix to inform the infrastructure team of the deployment status.

3.3. Tests and validation

The system was evaluated through a series of functional scenarios designed to assess its stability, reliability, and error handling capabilities. Rather than relying on performance benchmarks or stress tests, the focus was placed on observing the system's behavior under both normal operating conditions and controlled failure situations.

In the ideal operational scenario, a new image was published by the *external development team*. Diun, running on the bastion host, detected the new version and triggered the update sequence (see Fig. 3). The system successfully executed each automated stage: Skopeo transferred the image to the internal registry, Kubernetes applied the updated deployment, and three separate notification emails were generated at each key point: after image detection, after transfer completion, and upon successful deployment. This validated the end-to-end operation of the system without any manual intervention.

Beyond normal operation, the system's response to failure cases was also examined:

When the external Harbor registry was intentionally made unreachable by misconfiguring the URL, Diun logged a connection error and refrained from triggering any further actions. No emails were sent, correctly preventing unnecessary alerts.

If the internal registry was made unavailable, the image transfer failed during the Skopeo operation. This failure was logged, and an error notification email was sent to the *infrastructure team*, detailing the problem.

Another failure mode tested involved configuring Diun to monitor a non-existent image. In this case, Diun logged the absence of the image but did not trigger any downstream actions. The *infrastructure team* received no email, as expected, confirming that no unnecessary operations were performed under misconfigured monitoring.

When an incorrect tag or invalid parameters were introduced during image transfer, Skopeo failed gracefully. The system logged the error and issued an appropriate failure notification via email, preventing any incomplete or corrupted deployments.

Deployment failures were simulated by specifying non-existent namespaces or incorrect deployment names in the Kubernetes update command. As expected, the system logged the error during the `kubectl` operation and sent an error notification to the infrastructure team. This type of failure is illustrated in Fig. 4.b, which shows an automatically generated email reporting the unsuccessful deployment. Crucially, no partial deployment occurred, and the cluster state remained unaffected. Conversely, when the update was successful, a confirmation message was issued as shown in Fig. 4.a, including both deployment status and application URL.

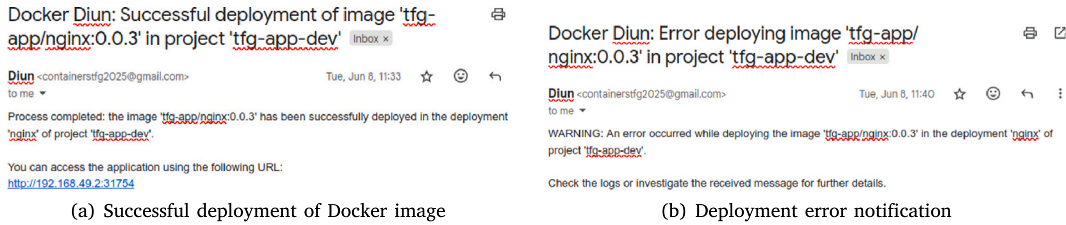
Finally, the notification layer itself was tested by disabling the Postfix service. While email delivery naturally failed, this issue was logged locally without affecting the execution of the previous stages. This confirmed that the notification system, while informative, does not introduce critical dependencies.

Table 1 summarizes the system's responses to the different failure scenarios tested.

In all cases, the system exhibited robust and predictable behavior, reliably communicating both success and failure outcomes. Logs were consistently updated, and notifications ensured traceability throughout the process. Most importantly, once the system was deployed and configured, no manual action was required from the *infrastructure team*, fulfilling the core objective of providing a fully automated and autonomous deployment solution.

By isolating detection, transfer, deployment, and notification as independent but sequential stages, the system also demonstrated its modularity and maintainability. The clear separation of concerns simplifies future extensions or modifications, such as introducing rollback mechanisms or integrating graphical monitoring interfaces.

Overall, these tests confirm that the proposed system is sufficiently robust and practical for real-world use in isolated environments requiring automated CI/CD pipelines.



(a) Successful deployment of Docker image (b) Deployment error notification

Fig. 4. Diun notifications following the deployment attempt of image `nginx:0.0.3`.

Table 1
Failure scenarios and corresponding system responses.

Failure scenario	System response	Notification sent?
Harbor unreachable	Logs connection error and aborts process.	No
Internal registry unavailable	Logs transfer error, halts execution, and sends failure report.	Yes (error)
Non-existent image monitored	Logs absence, prevents further execution.	No
Invalid tag or parameters	Logs transfer error, aborts process, sends error report.	Yes (error)
Deployment failure	Logs error during `kubectl`, no changes applied, sends failure report.	Yes (error)
Postfix disabled	Logs notification error, prior stages unaffected.	No

4. Impact

The proposed software addresses a common operational challenge faced by organizations operating in restricted or security-sensitive environments: the lack of automated deployment workflows due to strict network isolation policies. Traditionally, the absence of connectivity between external development teams and internal infrastructure requires manual detection of new Docker images, manual transfers across network segments, and manual deployment updates—all of which introduce operational delays, inefficiencies, and an increased risk of human error.

By automating these processes through a modular and open-source framework, this tool enables the implementation of CI/CD pipelines in environments that were previously limited to manual workflows. It allows organizations to achieve continuous delivery of containerized applications while maintaining full network isolation and complying with internal security policies.

Potential use cases span a wide range of industries where security and segmentation are critical, including banking, healthcare, industrial control systems, and academic environments used for secure software development training. From a community perspective, the solution offers value as a lightweight, reproducible, and adaptable framework. It relies exclusively on open-source technologies, is fully self-managed, and is designed to be easily extended to meet specific deployment needs. By providing a practical approach to CI/CD in constrained environments, the tool fills a gap left by conventional cloud-based orchestration systems, which typically require full network connectivity.

The software is publicly available as open-source and may serve as a starting point for infrastructure teams seeking to automate similar workflows under restrictive network policies.

5. Conclusions

This paper presented a modular CI/CD automation framework designed specifically for isolated Kubernetes environments. The system enables automatic detection, transfer, and deployment of Docker images in infrastructures segmented by strict network isolation policies, removing the need for manual intervention in the deployment process. By combining lightweight, open-source tools within a simple yet effective architecture, the solution delivers a fully self-managed, vendor-neutral alternative to commercial CI/CD platforms. Once configured, the system operates autonomously from a bastion host, ensuring that development updates can be propagated reliably without compromising the network segregation of the production cluster. Validation tests conducted in a simulated production environment confirmed the reliability of the approach across both normal and failure scenarios. The architecture's modularity, combined with its use of standard tools such as Diun, Skopeo, and kubectl, guarantees reproducibility and ease of adaptation for different operational contexts.

Future work could focus on extending the current system with features such as automated provisioning through Ansible, integration with observability tools like K8sGPT, multi-cluster support, and the development of a graphical dashboard for real-time monitoring and management.

CRediT authorship contribution statement

Guillermo Bermejo: Writing – review & editing, Writing – original draft, Validation, Software. **Ángel Macías:** Writing – review & editing, Writing – original draft, Validation, Methodology. **Juan Luis Herrera:** Writing – review & editing, Writing – original draft, Validation, Supervision, Investigation. **Sergio Laso:** Writing – review & editing, Writing – original draft, Investigation, Funding acquisition, Conceptualization. **Javier Berrocal:** Writing – review & editing, Writing – original draft, Visualization, Investigation, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially funded by the grant DIN2020-011586 funded by MICIU/AEI/10.13039/50100011033 and the "European Union NextGenerationEU/PRTR", by the project 0289_SER65_PLUS_6_P co-financed by the European Union through the Interreg Spain-Portugal POCTEP programme, by the Department of Education, Science and Professional Training of the Government of Extremadura (GR24099), and by the European Regional Development Fund.

References

[1] Docker I. Docker documentation. 2025, https://docs.docker.com/ (Accessed 1 July 2025).
[2] Authors TK. Kubernetes documentation. 2025, https://kubernetes.io/docs/.
[3] Khan HH, Zubair S, Nasim F, Akhter S, Ghazanfar MN, Azeem S. Role of kubernetes in DevOps technology for the effective software product management. J Comput Biomed Informatics 2024;7(01):313–27.
[4] Agrawal S, Singh D. Study containerization technologies like docker and kubernetes and their role in modern cloud deployments. In: 2024 IEEE 9th international conference for convergence in technology (i2CT). IEEE; 2024, p. 1–5.
[5] Ponnanna M, Nagasundari S. Leveraging machine learning techniques for the identification of trojans in container images. In: AIP conference proceedings. vol. 3122, (1):AIP Publishing LLC; 2024, 080021.
[6] Hat R. What is CI/CD?. 2025, https://www.redhat.com/en/topics/devops/what-is-ci-cd (Accessed 1 July 2025).
[7] Subramanyam SV. Cloud-based enterprise systems: Bridging scalability and security in healthcare and finance. IJSAT-International J Sci Technol 2025;16(1).
[8] Morales JA, Yasar H, Volkman A. Implementing DevOps practices in highly regulated environments. In: Proceedings of the 19th international conference on agile software development (XP 2018). XP '18, Association for Computing Machinery; 2018, p. 9. http://dx.doi.org/10.1145/3234152.3234188.
[9] Anumandla SKR. Automating container orchestration: Innovations and challenges in kubernetes implementation. Robot Xplore: USA Tech Dig 2024;1(1):29–43.
[10] Kokku R. Transforming enterprise DevOps: Kubernetes and CI/CD pipelines for scalable cloud environments. Int J Curr Sci 2021;11(4):343–8, URL https://rjpn.org/ijcspub/papers/IJCSP21D1035.pdf.
[11] Mercy O. CI/CD pipelines in kubernetes for scalable cloud deployments. ResearchGate 2022. URL https://www.researchgate.net/publication/392529219.
[12] Saleh S, Madhavji N, Steinbacher J. A systematic literature review on continuous integration and deployment (CI/CD) for secure cloud computing. 2025, http://dx.doi.org/10.48550/arXiv.2506.08055, arXiv URL https://arxiv.org/abs/2506.08055.
[13] Ashraf MWA, Raza A, Singh AR, Rathore RS, Damaj IW, Song HH. Intelligent caching based on popular content in vehicular networks: A deep transfer learning approach. IEEE Trans Intell Transp Syst 2024.
[14] Agrawal A, Bakshi BR, Kodamana H, Ramteke M. Multi-objective optimization of food-energy-water nexus via crops land allocation. Comput Chem Eng 2024;183:108610.
[15] Rathore RS, Sangwan S, Prakash S, Adhikari K, Kharel R, Cao Y. Hybrid WGWO: whale grey wolf optimization-based novel energy-efficient clustering for EH-WSNs. EURASIP J Wirel Commun Netw 2020;2020(101):1–16. http://dx.doi.org/10.1186/s13638-020-01721-5.
[16] Shi H, Ying L, Chen L, Duan H, Liu M, Xue Z. Dr. Docker: A large-scale security measurement of docker image ecosystem. In: Proceedings of the ACM on web conference 2025. 2025, p. 2813–23.
[17] Containrrr. Watchtower: Automatically update running docker containers. 2025, https://containrrr.dev/watchtower/ (Accessed 1 July 2025).
[18] Keel. Keel - kubernetes operator for automating helm, DaemonSet, StatefulSet & deployment updates. 2025, https://keel.sh/docs/ (Accessed 1 July 2025).
[19] Foundation A. Apache CloudStack. 2024, https://cloudstack.apache.org/ (Accessed 27 June 2025).
[20] CrazyMax. Diun – Docker image update notifier. 2025, https://crazymax.dev/diun/ (Accessed 1 July 2025).
[21] Organization C. Skopeo: Work with remote image registries - GitHub. 2025, https://github.com/containers/skopeo (Accessed 1 July 2025).
[22] Authors TM. Minikube documentation. 2025, https://minikube.sigs.k8s.io/docs/ (Accessed 1 July 2025).