Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Intelligent and Secure Automation of CI/CD Pipelines for Cloud Infrastructures

Riham Borghol

riham.borghol@esprit.tn

ESPRIT School of Engineering

Rim Zoglami

ESPRIT School of Engineering

Hamza Sarraj

ESPRIT School of Engineering

Article

Keywords: CI/CD, GitOps, DevOps, Cloud-Native, Kubernetes, Microservices, Jenkins, ArgoCD, Secure Automation, Trivy

Posted Date: April 28th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9260975/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at Scientific Reports on October 13th, 2026. See the published version at https://doi.org/10.1038/s41598-026-75804-6.

Abstract

The paradigm shift towards cloud-native applications and microservices architectures has introduced significant complexities in software deployment and management. Traditional manual deployment processes are inadequate, being error-prone, slow, and insecure. This paper presents a comprehensive case study on the design and implementation of a fully automated, secure, and intelligent CI/CD pipeline for a cloud-based infrastructure. Leveraging a suite of modern DevOps tools, we demonstrate an end-to-end workflow for a microservices-based application deployed on Microsoft Azure. Unlike purely theoretical models, this work focuses on practical implementation, this work focuses on a practical implementation carried out in an academic engineering environment. The architecture integrates Jenkins for Continuous Integration, SonarQube for static code analysis, and Trivy for container security scanning. Continuous Deployment is achieved through a GitOps approach, orchestrated by ArgoCD on an Azure Kubernetes Service (AKS) cluster. The results validate the efficacy of this model, showcasing a 95.6% pipeline success rate, an 87.5% reduction in deployment time, and the enforcement of stringent quality and security gates. These results were obtained in an academic environment; performance may vary in large-scale industrial deployments. This study aims to share concrete implementation insights that may support organizations and academic institutions seeking to improve the reliability, traceability, and security of their software delivery lifecycle.

1. Introduction

In the contemporary landscape of software engineering, the adoption of distributed systems, particularly microservices architectures, has become ubiquitous. This architectural style offers enhanced scalability, resilience, and maintainability compared to monolithic systems [1, 2]. However, it also introduces significant challenges related to deployment, configuration management, and operational oversight. In practice, these challenges are often underestimated, especially when systems evolve rapidly or involve multiple independent services. The need to manage dozens or even hundreds of independently deployable services necessitates a departure from traditional, manual operational practices towards highly automated, reliable, and secure workflows.

In response to these challenges, DevOps practices, combined with Continuous Integration (CI) and Continuous Deployment (CD), have emerged as the industry standard [4]. CI/CD pipelines automate the process of building, testing, and deploying software, enabling development teams to deliver value to users faster and more reliably. More recently, the GitOps paradigm has gained prominence as an evolution of CI/CD for cloud-native environments. GitOps leverages Git as the single source of truth for declarative infrastructure and applications [5, 6], using automated tools to ensure the live system state converges towards the state described in the repository. This shift also introduces a cultural change, as operational actions become auditable software changes rather than manual interventions.

This paper details a practical implementation of these modern principles through a case study conducted at ESPRIT School of Engineering. The project's primary objective was to build a secure and automated CI/CD pipeline with a GitOps workflow for a real-world web application. The application, a training management platform, was developed using a microservices architecture and deployed on the Microsoft Azure cloud

platform [7]. The contribution of this work lies not in proposing a new tool, but in demonstrating how existing technologies can be combined into a coherent and operational deployment ecosystem. We document the architecture, implementation methodology, and empirical results, offering valuable insights for practitioners and academics aiming to implement similar systems.

This paper contributes to the field of cloud-native software delivery by presenting an integrated DevSecOps and GitOps-based CI/CD framework tailored for microservices architectures deployed on Kubernetes.

Unlike existing approaches that address automation, security, or deployment independently, this work combines continuous integration, security enforcement, and declarative deployment into a unified and fully automated pipeline.

The proposed approach is experimentally validated through a real-world case study deployed on Microsoft Azure, where quantitative metrics related to deployment time, pipeline reliability, and security compliance are analyzed.

The results demonstrate that embedding quality and security controls early in the pipeline significantly improves deployment efficiency, reliability, and operational transparency.

The remainder of this paper is organized as follows.

Section 2 reviews related work on cloud-native architectures, CI/CD automation, DevSecOps practices, and GitOps-based deployment models.

Section 3 presents the overall system architecture and the design choices adopted for the proposed cloud-native application, including its microservices decomposition and containerization strategy.

Section 4 details the methodology and implementation of the automated CI/CD pipeline, describing the integration of security and quality controls as well as the GitOps-based deployment process.

Section 5 discusses the experimental results and evaluates the effectiveness of the proposed approach using quantitative performance and security metrics.

Finally, Section 6 concludes the paper and outlines directions for future work.

2. Related Work

To the best of our knowledge, no existing work provides a fully integrated CI/CD pipeline that simultaneously combines continuous integration automation, DevSecOps security practices, and GitOps-based deployment for cloud-native applications on Kubernetes. Existing studies typically focus on individual aspects of the software delivery lifecycle, such as CI/CD automation, security integration, or declarative deployment, without addressing their end-to-end integration.

Several studies have investigated CI/CD pipelines in cloud computing environments. Rahman et al. [8] systematically mapped continuous integration practices and demonstrated that automated pipelines reduce

deployment errors and improve release frequency. However, these approaches often rely on imperative deployment scripts and manual operational steps, which can lead to configuration drift, limited traceability, and operational complexity in large-scale distributed systems.

To mitigate security risks, the DevSecOps paradigm has been proposed to integrate security controls early in the software development lifecycle. Myrbakken and Colomo-Palacios [9] highlighted the benefits of shift-left security practices, while Sablotny et al. [10] demonstrated the effectiveness of static code analysis and container vulnerability scanning in CI/CD pipelines. Nevertheless, most DevSecOps solutions still treat deployment as a final stage, lacking continuous security monitoring and automated compliance enforcement after deployment.

More recently, GitOps has emerged as an evolution of CI/CD tailored for cloud-native environments. By using Git repositories as the single source of truth, GitOps frameworks such as ArgoCD enable declarative deployment, automatic rollback, and enhanced auditability [6, 11]. Studies by Hüttermann [11] and Wurster et al. [12] show that GitOps improves deployment reliability and consistency in Kubernetes-based environments. However, GitOps-based solutions often focus primarily on deployment and do not fully integrate CI processes such as build automation, testing, and security scanning.

Table 1 summarizes representative CI/CD pipeline approaches from the literature and highlights their limitations in terms of deployment time, security coverage, and auditability. In contrast to existing works, the approach proposed in this paper provides an end-to-end pipeline that integrates CI/CD automation, DevSecOps practices, and GitOps deployment within a unified and fully auditable framework, validated through a real-world cloud deployment case study.

Table 1
Comparison of Existing CI/CD Pipelines and Limitations

Pipeline / Study	Components	Avg. Deployment Time	Security Checks	Auditability	Limitations	How Our Approach Overcomes
Traditional Jenkins	Jenkins CI, manual scripts	45–120 min	Limited, mostly post-build	Low, manual logging	Long deployment, error-prone, limited traceability	Fully automated, declarative pipeline with audit trail
Jenkins + DevSecOps	Jenkins + SonarQube, Trivy	30–60 min	Improved via static code & container scanning	Medium	Security checks often post-build, limited rollback	Shift-left security integrated, automatic rollback via ArgoCD
GitOps pipelines	GitOps (ArgoCD) + Kubernetes	15–30 min	Basic deployment security	High	Lacks CI integration for building, testing, container scanning	Full CI/CD + DevSecOps + GitOps in unified pipeline
Proposed Approach	Jenkins CI + SonarQube + Trivy + ArgoCD on AKS	12–15 min	Full shift-left security	High, automated & auditable	N/A	Combines CI/CD automation, DevSecOps, and GitOps with real metrics

3. System Architecture and Design

The foundation of this study is a web-based training management platform designed for an educational institution. The system's architecture was deliberately chosen to reflect modern software design patterns, providing a realistic testbed for the DevOps pipeline. The design process followed the Scrum agile framework [13], breaking down the work into manageable sprints.

3.1 Application Architecture

The application follows a three-tier architecture, comprising a presentation layer, an application layer, and a data layer. This separation of concerns is fundamental to building a maintainable and scalable system.

• Presentation Layer: A dynamic Single-Page Application (SPA) built with Angular [14]. It provides a responsive user interface for administrators and teachers to manage and browse training programs.

• Application Layer: The backend is built on a microservices architecture using Spring Boot [15]. This layer contains the core business logic, decomposed into independent, loosely coupled services.

• Data Layer: A MySQL relational database [16] provides persistent storage for all application data, including user information, training details, and registrations.

Figure 1 presents the global architecture of the proposed training management platform. The system follows a three-tier architecture composed of a presentation layer, an application layer, and a data layer, ensuring a clear separation of concerns. This architectural design improves modularity and scalability by isolating user interaction, business logic, and data management. Such separation facilitates independent evolution of each layer and aligns with best practices for cloud-native application development.

3.2 Microservices Decomposition and Communication

The backend was decomposed into two primary microservices, each with a distinct responsibility, to promote modularity and independent scalability:

• Formation Service: Manages the entire lifecycle of training programs, including creation, updates, and teacher registrations.

• User Service: Handles user authentication, authorization, and profile management.

Communication between these services is handled through a combination of synchronous and asynchronous patterns to balance responsiveness and resilience.

• Synchronous Communication: For immediate request-response interactions, such as the frontend fetching data from a service, RESTful APIs are used. To avoid hardcoding service locations, Netflix Eureka is employed as a service discovery registry [17], allowing services to dynamically find and communicate with each other.

• Asynchronous Communication: For decoupling services and handling events that do not require an immediate response, Apache Kafka is used as a distributed event streaming platform [18]. For instance, when a teacher registers for a course via the Formation Service, an event is published to a Kafka topic. The User Service subscribes to this topic and asynchronously updates the teacher's registration history, ensuring eventual consistency without creating a tight coupling between the services.

Figure 2 illustrates the communication patterns between the microservices composing the backend system. Synchronous interactions are handled through RESTful APIs, while asynchronous communication is achieved using Apache Kafka for event-driven processing. The combination of synchronous and asynchronous communication enhances system resilience and reduces tight coupling between services. In particular, event-driven communication improves scalability and fault tolerance, which are critical requirements in distributed cloud environments.

3.3 Containerization and Repository Structure

To ensure consistency across development, testing, and production environments, all application components (frontend, backend microservices, Eureka server) were containerized using Docker [19]. A docker-compose.yml file was created to orchestrate the entire stack for local development.

A critical decision for the GitOps workflow was the separation of source code and deployment configurations into two distinct Git repositories. This practice is a cornerstone of robust GitOps implementations [6].

• Application Repository (espritPFAapp): Contains all application source code (Angular and Spring Boot), Dockerfiles, and the Jenkinsfile defining the CI pipeline.

• Manifests Repository (formationApp-k8s): Contains only the Kubernetes YAML manifests that declaratively define the desired state of the application in the cluster. This repository is the single source of truth for ArgoCD.

Figure 3 shows the separation between the application source code repository and the Kubernetes manifests repository. This separation is a fundamental principle of the GitOps approach adopted in this work. By isolating deployment configurations from application code, this approach improves traceability, simplifies rollback operations, and ensures that the desired system state is fully version-controlled.

4. Methodology: The Automated DevOps Workflow

The implementation of the automated pipeline was divided into three logical phases: infrastructure provisioning, continuous integration setup, and continuous deployment implementation using GitOps.

4.1. Tool Selection Rationale and Alternatives

The effectiveness of a CI/CD pipeline for cloud-native applications strongly depends on the selection of tools that ensure automation, security, scalability, and auditability. In this work, tool selection was guided by four main criteria:
(i) compatibility with Kubernetes-based environments,
(ii) support for automation and declarative configuration,
(iii) integration of security controls within the pipeline (DevSecOps), and
(iv) maturity and adoption in both industrial and academic contexts.

These criteria are widely recognized in cloud-native and DevOps literature as essential for building reliable and reproducible deployment pipelines [19, 8, 11]. The following subsections justify the selection of each tool and discuss the considered alternatives.

4.1.1. Continuous Integration Tool Selection

Jenkins was selected as the primary Continuous Integration (CI) tool due to its maturity, extensibility, and widespread adoption in both industry and research [20]. Jenkins provides extensive plugin support, enabling seamless integration with version control systems, testing frameworks, container registries, and security analysis tools. Its *pipeline-as-code* model promotes reproducibility and version-controlled CI workflows, which are essential for large-scale cloud-native systems.

Alternative CI tools such as GitHub Actions [21] and GitLab CI [22] were considered. GitHub Actions offers tight integration with GitHub repositories but provides limited flexibility for complex, multi-stage pipelines and hybrid cloud deployments. GitLab CI integrates CI/CD into a single platform; however, its strong coupling with GitLab repositories may reduce portability across heterogeneous infrastructures. In contrast, Jenkins offers greater flexibility, extensibility, and vendor neutrality, making it more suitable for cloud-agnostic CI/CD architectures.

4.1.2. Containerization and Orchestration Platform

Docker was adopted for application containerization due to its lightweight execution model, standardized image format, and extensive ecosystem support. Containerization ensures consistent application behavior across development, testing, and production environments, significantly reducing environment-specific issues [19].

For orchestration, Kubernetes [2, 23] was chosen as the deployment platform. Kubernetes has become the de facto standard for managing containerized applications, offering advanced features such as automatic scaling, self-healing, service discovery, and declarative configuration management. Alternative orchestration platforms, including Docker Swarm and Apache Mesos, were evaluated. Docker Swarm provides simpler configuration but lacks advanced scheduling and ecosystem maturity, while Mesos introduces higher operational complexity. Kubernetes provides a balanced trade-off between scalability, flexibility, and community support, making it particularly suitable for cloud-native microservices architectures [2].

4.1.3. Security Integration and DevSecOps Tooling

To implement DevSecOps principles, security controls were embedded directly into the CI pipeline following a *shift-left* security strategy. SonarQube [9] was selected for static code analysis to detect code quality issues, security vulnerabilities, and technical debt early in the software development lifecycle.

For container vulnerability scanning, Trivy [10] was chosen due to its lightweight design, high scanning performance, and comprehensive vulnerability database covering operating systems and application dependencies.

Alternative security tools such as Checkmarx, Fortify, and Clair were considered. While enterprise solutions offer advanced analysis features, they often require proprietary licenses and complex configurations, which may hinder full automation in academic or resource-constrained environments. In contrast, SonarQube and Trivy offer an effective balance between accuracy, performance, and ease of integration, making them well-suited for automated cloud-native CI/CD pipelines.

4.1.4. GitOps Deployment Tool Selection

For continuous deployment, ArgoCD was selected to implement the GitOps paradigm. ArgoCD continuously monitors Git repositories containing Kubernetes manifests and automatically reconciles the desired and actual system states of the cluster. This declarative deployment approach improves reliability, enables automated rollback, and ensures full auditability of configuration changes [6, 11, 12].

Alternative GitOps tools such as FluxCD were evaluated. Although FluxCD offers similar reconciliation capabilities, ArgoCD provides richer visualization, role-based access control, and advanced synchronization features. These characteristics make ArgoCD more suitable for deployment monitoring and operational transparency in complex Kubernetes environments.

4.1.5. Cloud Platform Selection

Microsoft Azure Kubernetes Service (AKS) was selected as the target cloud platform due to its managed Kubernetes control plane, native integration with monitoring and security services, and support for scalable production deployments. While other managed Kubernetes platforms such as Amazon EKS and Google GKE offer comparable functionality, AKS was chosen to demonstrate the portability of the proposed CI/CD and GitOps pipeline across public cloud infrastructures. The proposed methodology remains cloud-agnostic and can be replicated on alternative Kubernetes-based platforms.

4.2. Cloud Infrastructure on Microsoft Azure

A secure and scalable foundation was provisioned on Microsoft Azure. All resources were organized within a single resource group for simplified management and billing. The core components were :

- Virtual Machines for DevOps Tools: Two Ubuntu VMs were deployed. The first (Standard_B2s) hosted the Jenkins CI server and the Trivy security scanner. The second (Standard_B2ms) hosted the SonarQube server for code quality analysis. Network Security Groups (NSGs) were configured to restrict access to necessary ports (e.g., 8080 for Jenkins, 9000 for SonarQube).

- Azure Kubernetes Service (AKS): A production-grade AKS cluster with three worker nodes (Standard_D2s_v3) was deployed to host the containerized application. This managed Kubernetes service handles control plane operations, health monitoring, and scaling, significantly reducing operational overhead.

- Container Registry: Docker Hub was used as the public container registry to store the Docker images built by the CI pipeline. Credentials were securely stored in Jenkins for authenticated pushes.

Figure 4 illustrates the configuration of the virtual machine hosting the Jenkins server on Microsoft Azure. The VM is secured using Network Security Groups to restrict access to essential services.

4.3. The Continuous Integration Pipeline

The CI pipeline, orchestrated by Jenkins, is the engine of the workflow. It is defined as code in a Jenkinsfile and automatically triggered by commits to the application repository. The pipeline executes a series of stages designed to build, test, and secure the application before a deployment artifact is created. During the initial iterations, several pipeline failures were encountered, mainly due to dependency conflicts and unstable test configurations. These issues were progressively resolved, highlighting the importance of incremental pipeline refinement.

Figure 5 presents the declarative Jenkins pipeline used for continuous integration. The pipeline is composed of multiple stages, including build, testing, static code analysis, containerization, and security scanning. This

pipeline design enables early detection of defects and vulnerabilities by enforcing quality and security checks before deployment artifacts are generated. The use of declarative pipelines also improves maintainability and reproducibility.

To improve the reproducibility and measurability of the proposed CI/CD pipeline, the execution time and validation metrics of each pipeline stage were systematically recorded. Table 2 summarizes the pipeline stages, their average execution time, and the quality and security metrics enforced at each step.

Table 2
CI/CD pipeline stages, execution time, and enforced metrics

Pipeline Stage	Description	Avg. Time	Metrics / Checks Enforced
Checkout	Clone application source repository	~ 1 min	Repository access, success rate
Build	Compile backend and frontend components	~ 3 min	Build success, dependency resolution
Unit Tests	Execute automated unit tests	~ 2 min	Test coverage ≥ 70%
SonarQube Analysis	Static code quality and security analysis	~ 2 min	No critical bugs or vulnerabilities
Docker Build	Build optimized container images	~ 1 min	Image size optimization
Trivy Scan	Container vulnerability scanning	~ 1 min	CVEs: 0 critical / high severity
Push Docker Images	Push validated images to registry	~ 1 min	Successful push logs
Update Manifests	Update Kubernetes manifests (GitOps trigger)	~ 1 min	ArgoCD synchronization success

As shown in Table 2, the full CI pipeline executes in approximately 12–15 minutes, while enforcing strict quality and security gates at each stage. This structured and metric-driven workflow ensures both reproducibility and reliability, enabling consistent deployment behavior across multiple execution cycles.

The key stages of the CI pipeline include:

1. Checkout: Clones the source code from the Git repository.

2. Build: Compiles the Spring Boot microservices using Maven and builds the production version of the Angular frontend.

3. Unit Tests: Executes automated unit tests for both backend and frontend to ensure code correctness.

4. SonarQube Analysis: The code is analyzed by SonarQube for bugs, vulnerabilities, and code smells. A Quality Gate is enforced; if the code fails to meet predefined thresholds (e.g., >70% test coverage, 0 critical vulnerabilities), the pipeline fails. This stage acts as an intelligent quality checkpoint.

5. Build Docker Images: Docker images are built for each component using multi-stage builds to optimize for size and security.

6. Trivy Security Scan: Each newly built Docker image is scanned by Trivy for known Common Vulnerabilities and Exposures (CVEs). The pipeline fails if critical or high-severity vulnerabilities are detected, preventing insecure images from being pushed to the registry.

7. Push Docker Images: Upon passing all previous checks, the validated Docker images are tagged with the build number and pushed to Docker Hub.

8. Update Manifests: The final stage automatically checks out the formationApp-k8s repository, updates the image tags in the relevant Kubernetes Deployment manifests, and pushes the change back to Git. This commit is the trigger for the GitOps workflow.

Figure 6 shows an example of a Trivy security scan report generated during the CI process. The scan identifies known vulnerabilities in container images before they are pushed to the registry. Integrating container vulnerability scanning into the CI pipeline enforces a shift-left security strategy, preventing insecure images from reaching production environments.

4.4. Continuous Deployment with GitOps and ArgoCD

The final link in the chain is the Continuous Deployment process, managed by ArgoCD according to GitOps principles. ArgoCD was installed in the AKS cluster and configured to monitor the formationApp-k8s repository.

The workflow is as follows:

1. Detection: ArgoCD continuously monitors the manifests repository. When the CI pipeline pushes an updated image tag, ArgoCD detects that the live state of the cluster has drifted from the desired state in Git.

2. Synchronization: ArgoCD automatically initiates a synchronization process. It pulls the new manifests from Git and applies them to the AKS cluster.

3. Rolling Update: Kubernetes handles the update gracefully using a rolling update strategy. It incrementally replaces old application pods with new ones, ensuring zero downtime for the end-user.

4. Self-Healing: A key benefit of GitOps is self-healing. If any manual, out-of-band changes are made to the cluster (e.g., via kubectl), ArgoCD detects this drift and can automatically revert the changes to match the state defined in Git, ensuring consistency and auditability.

Figure 7 illustrates the complete end-to-end CI/CD and GitOps workflow, from source code commit to deployment on the Kubernetes cluster. This workflow highlights the indirect deployment mechanism enabled by GitOps, where changes are applied through version-controlled manifests rather than direct cluster access. This approach enhances auditability, consistency, and operational safety.

5. Results and Discussion

The implemented system was rigorously tested and validated to measure its effectiveness, performance, and reliability. The results confirm the significant benefits of adopting an automated, secure, and GitOps-driven deployment model.

5.1. Deployment Verification and Monitoring

Following the first successful pipeline run, ArgoCD synchronized the application to the AKS cluster. Verification using kubectl confirmed that all pods for the microservices, database, and frontend were in a $Ru \cap \in g$ state with high availability (2 replicas for stateless services). The frontend was accessible via a public IP address assigned by the Azure Load Balancer, and all backend functionalities were operational.

The ArgoCD dashboard provided a real-time, comprehensive view of the application's health and sync status. The resource tree view allowed for intuitive visualization of the entire application hierarchy, from high-level services down to individual pods, making troubleshooting and monitoring highly efficient.

Figure 8 presents the ArgoCD resource tree view, providing a real-time visualization of the deployed application and its associated Kubernetes resources. This visualization facilitates monitoring and troubleshooting by offering a clear overview of application health and synchronization status.

Figure 9 shows the fully deployed and functional training management application accessed through the public interface.

5.2. Performance and Quality Metrics

The automated pipeline yielded significant quantitative improvements :

- Pipeline Performance: The end-to-end CI pipeline execution time averaged between 12-15 minutes. This represents an 87.5% reduction compared to an estimated manual process of 2 hours. Over 45 builds were executed during the project, with a success rate of 95.6%, demonstrating the pipeline's reliability.

- Code Quality: SonarQube's quality gates were consistently met. For example, the Formation Service achieved 78.3% test coverage with zero bugs or vulnerabilities and an 'A' rating for maintainability. This automated quality enforcement prevented technical debt from accumulating.

- Security Posture: Trivy scans confirmed that all final Docker images pushed to the registry had zero critical or high-severity vulnerabilities. This "shift-left" approach to security ensures that vulnerabilities are caught early in the development cycle, not in production.

5.3. Analysis of the GitOps Workflow

The GitOps workflow, orchestrated by ArgoCD, proved to be a robust and transparent method for continuous deployment. The validation of a complete commit-to-production cycle took approximately 16 minutes with zero manual intervention. Note that these measurements were obtained in a controlled academic environment; execution times may differ in larger production deployments. A key test involved simulating a

bug introduction and then reverting the commit in the manifests repository. ArgoCD detected the revert and automatically rolled the application back to the previous stable version within 3 minutes, demonstrating the effectiveness of Git-based rollback mechanisms.

The primary benefits observed from this model include:

- Traceability: Every change to the production environment is tied to a Git commit, providing a complete and immutable audit trail.

- Consistency: The single source of truth in Git ensures that all environments are configured consistently, eliminating configuration drift.

- Developer Experience: Developers can trigger deployments by simply merging code, without needing direct access to the Kubernetes cluster, which also improves the security posture.

- Reliability: Automated rollbacks and zero-downtime rolling updates significantly increase the reliability of the deployment process.

While this implementation demonstrates a practical and effective approach for automating CI/CD pipelines with integrated security and GitOps practices, it is limited by the scale of the academic environment, the number of microservices, and the specific cloud platform used (Microsoft Azure). Further studies are required to validate scalability, cost efficiency, and performance in industrial settings with larger deployments.

6. Conclusion and Future Work

6.1 Conclusion

This paper has successfully demonstrated the design, implementation, and validation of an intelligent and secure CI/CD pipeline for a cloud-native microservices application. By integrating Jenkins, SonarQube, Trivy, and ArgoCD on the Microsoft Azure platform, we constructed a fully automated workflow that bridges the gap between development and operations. The case study confirms that a GitOps-centric approach not only accelerates the software delivery lifecycle but also embeds quality and security checks directly into the process. The empirical results indicate significant reductions in deployment time and demonstrate the practical benefits observed in this case study. The presented architecture serves as a practical and replicable model for organizations aiming to achieve true agility and reliability in their cloud infrastructure management.

6.2 Future Work

While the current implementation provides a robust foundation, several avenues for future enhancement exist. These perspectives aim to extend the proposed framework rather than represent limitations of the current model

• Enhanced Monitoring and Observability: Integrating Prometheus, Grafana, and the ELK stack
• Advanced Security: Kubernetes Network Policies, Azure Key Vault, OAuth2/OIDC
• Progressive Delivery: Canary or blue-green deployments with Argo Rollouts
• Infrastructure as Code: Full automation using Terraform to achieve complete infrastructure reproducibility

Declarations

Author contributions

Riham Borghol contributed to the methodology design, the conceptualization of the study, supervision of the research activities, validation of results, and critical revision of the manuscript.

Rim Zoglami contributed to the implementation of the CI/CD pipeline, experimental setup.

Hamza Sarraj contributed to the formal analysis, investigation, and interpretation of experimental results.

Funding

No funding received for this study

Data availability

No datasets were generated or analysed during this study.

Competing interests

The authors declare no competing interests

References

1. Newman S (2015) Building Microservices. O'Reilly Media
2. Zhang Q, Chen M, Li L, Li Y (2021) Kubernetes-based cloud-native application deployment: architecture and challenges. J Cloud Comput 10(1). https://doi.org/10.1186/s13677-021-00251-x
3. Humble J, Farley D (2010) Continuous Delivery. Addison-Wesley
4. Fowler M, Humble J (2010) Continuous Integration. ThoughtWorks
5. Weaveworks (2017) Guide to GitOps. https://weave.works/technologies/gitops/
6. The Argo Project (2025) Argo CD: Declarative GitOps CD for Kubernetes. https://argoproj.github.io/argo-cd/
7. Microsoft Corporation (2025) Azure Kubernetes Service (AKS). https://azure.microsoft.com/
8. Rahman AA, Helms E, Williams L (2019) A systematic mapping study of continuous integration. J Syst Softw 155:78–94. https://doi.org/10.1016/j.jss.2019.05.003
9. Myrbakken H, Colomo-Palacios R (2017) DevSecOps: A multivocal literature review. Softw Qual J 25(3):701–739. https://doi.org/10.1007/s11219-016-9336-1

10. Sablotny M, Biallas S, Schmid K (2020) Security testing in CI/CD pipelines for cloud-native applications. IEEE Softw 37(6):42–49. https://doi.org/10.1109/MS.2020.2991187
11. Hüttermann M (2022) GitOps and Kubernetes: Continuous deployment with declarative infrastructure. IEEE Cloud Comput 9(1):60–67. https://doi.org/10.1109/MCC.2022.3169632
12. Wurster M et al (2021) Improving deployment reliability using GitOps. J Cloud Comput 10(1). https://doi.org/10.1186/s13677-021-00234-x
13. Sutherland J, Schwaber K (2017) The Scrum Guide. Scrum.org
14. Google (2025) Angular documentation. https://angular.io
15. Spring (2025) Spring Boot documentation. https://spring.io/projects/spring-boot
16. Oracle Corporation (2025) MySQL reference manual. https://dev.mysql.com/doc/
17. Netflix (2012) Eureka at Netflix. https://netflixtechblog.com
18. Kreps J, Narkhede N, Rao J (2011) Kafka: A distributed messaging system. NetDB
19. Pahl C (2015) Containerization and the PaaS cloud. IEEE Cloud Comput 2(3):24–31
20. Jenkins Project (2025) Jenkins documentation. https://www.jenkins.io
21. GitHub (2024) GitHub Actions documentation. https://docs.github.com/actions
22. GitLab (2024) GitLab CI/CD documentation. https://docs.gitlab.com/ee/ci/
23. Burns B et al (2016) Borg, Omega, and Kubernetes. ACM Queue 14(1)

Figures

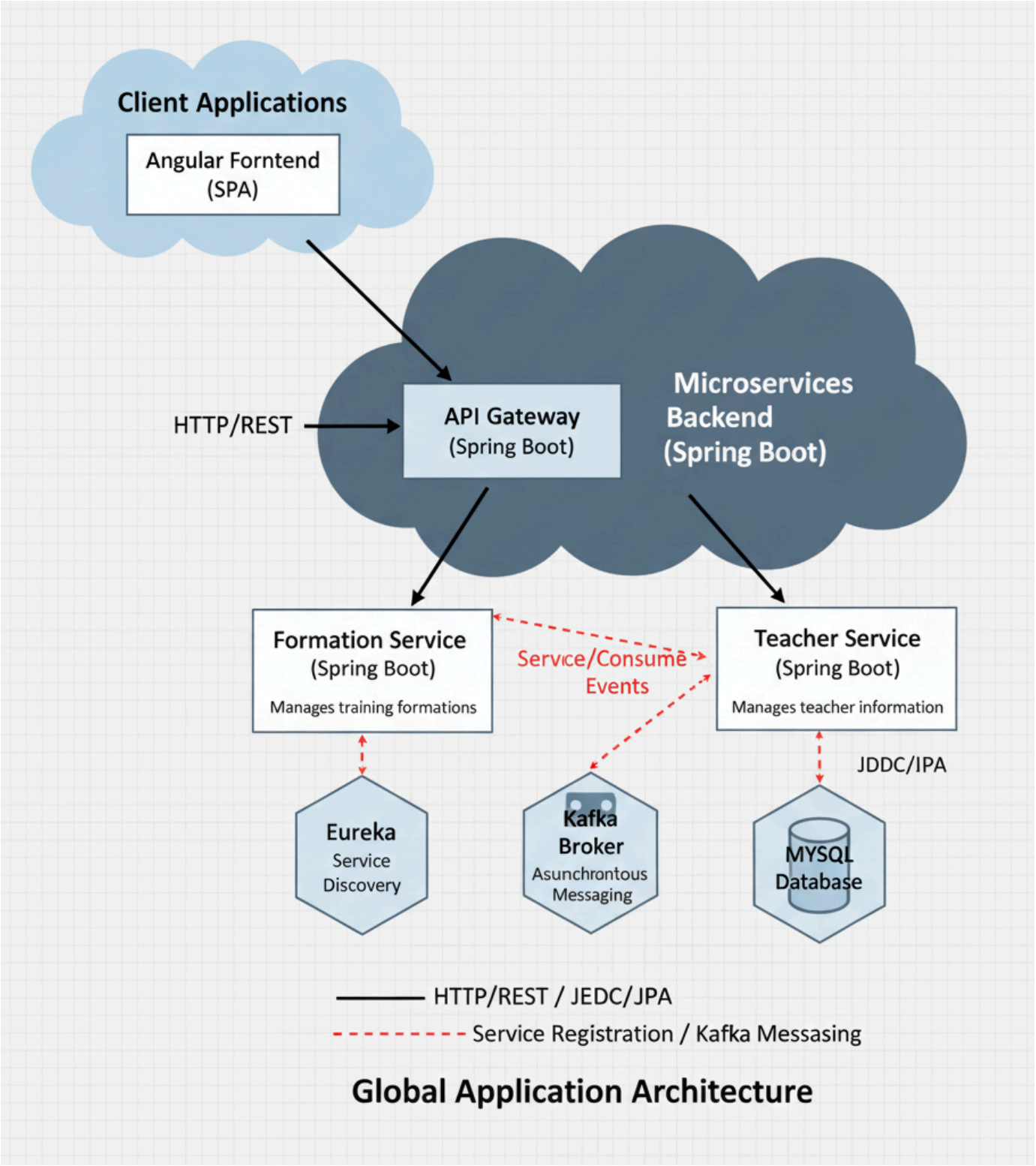


Figure 1

Global System Architecture.

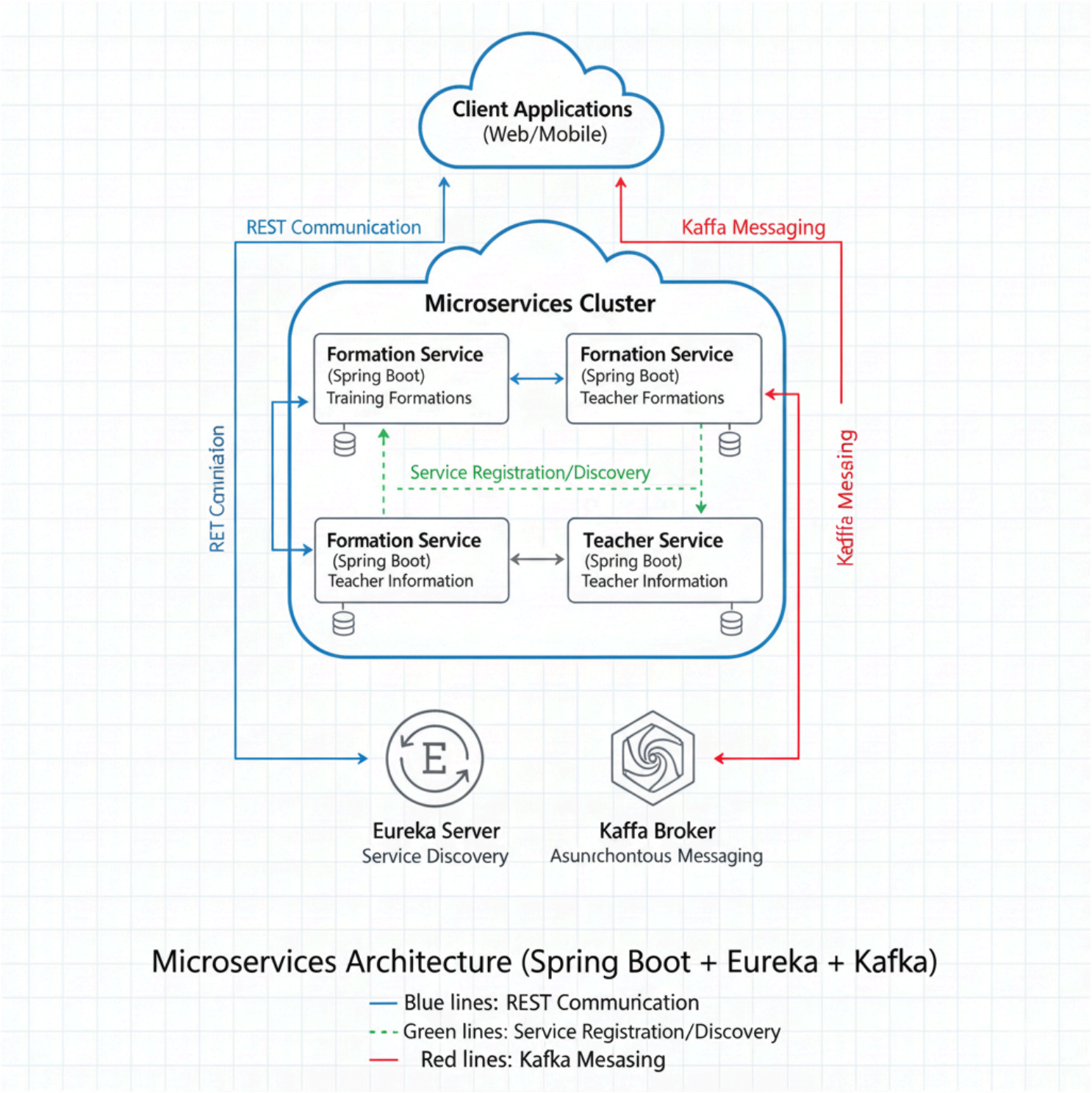


Figure 2

Microservices Communication Patterns with Eureka and Kafka.

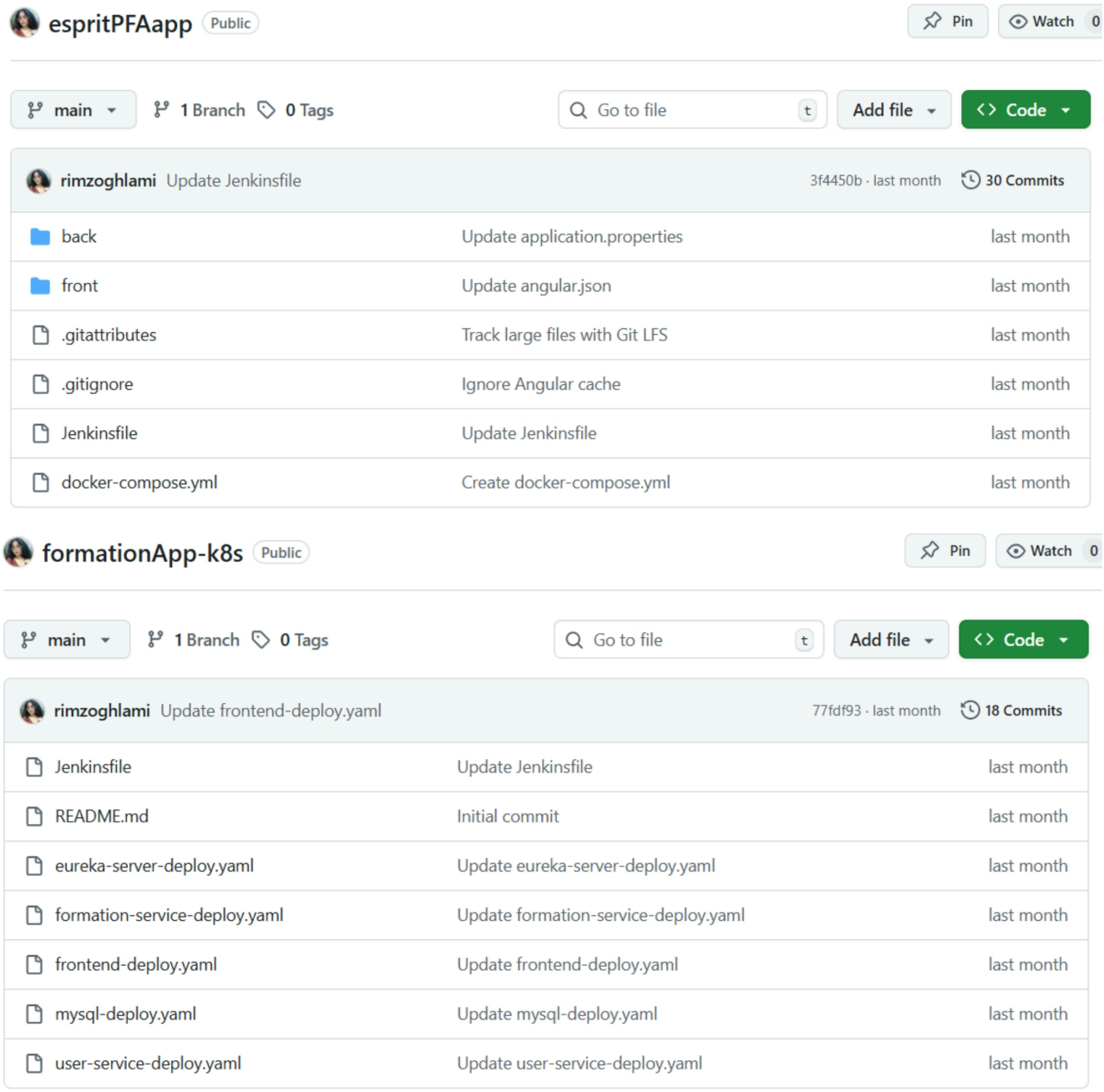


Figure 3

Separation of Application Code and Kubernetes Manifests into Two Git Repositories.

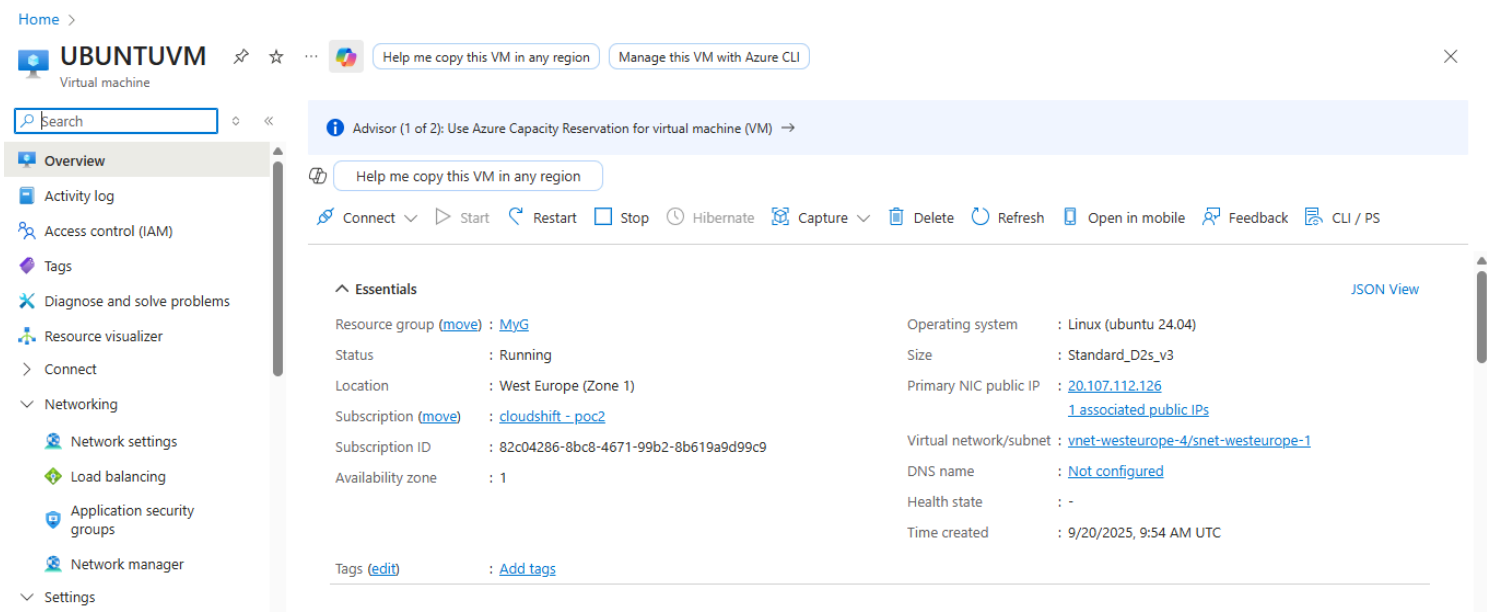


Figure 4

Azure VM Configuration for the Jenkins Server.

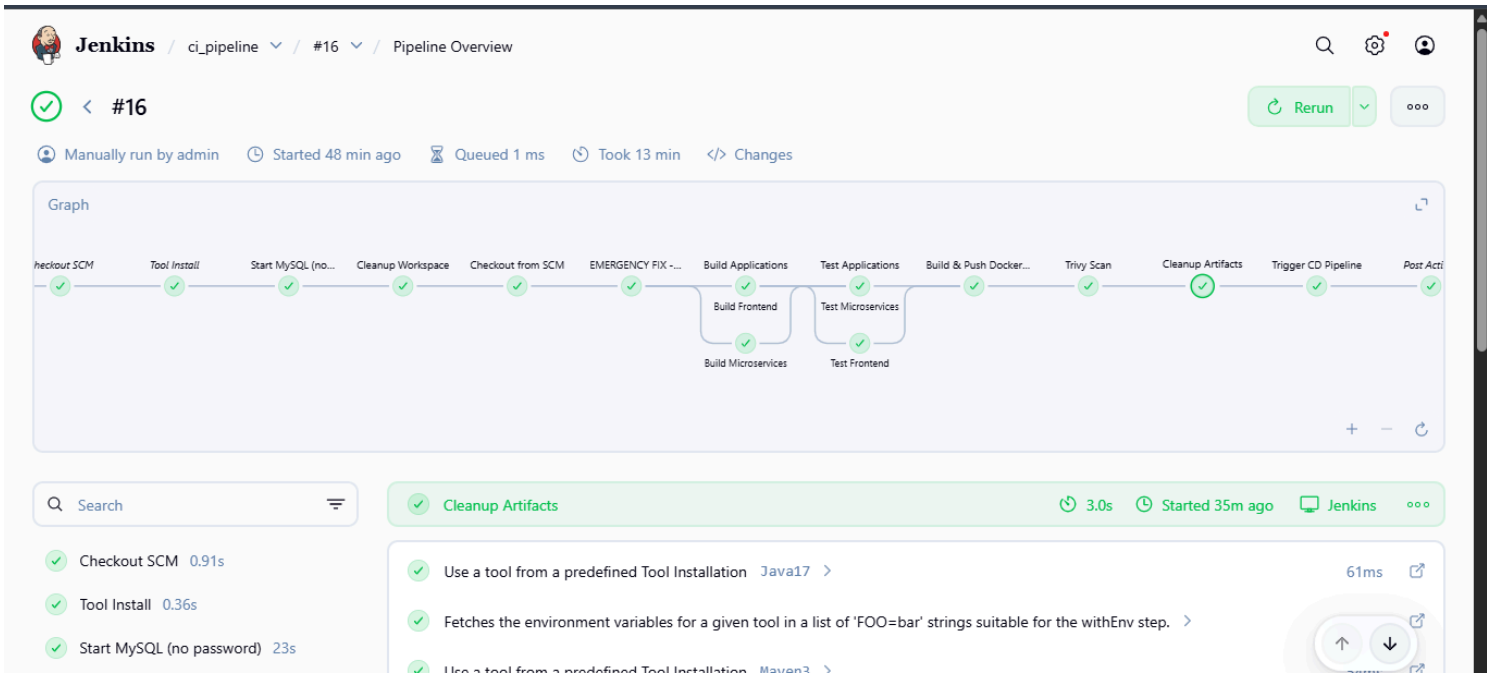


Figure 5

Jenkins Declarative Pipeline Architecture with Parallel Stages.

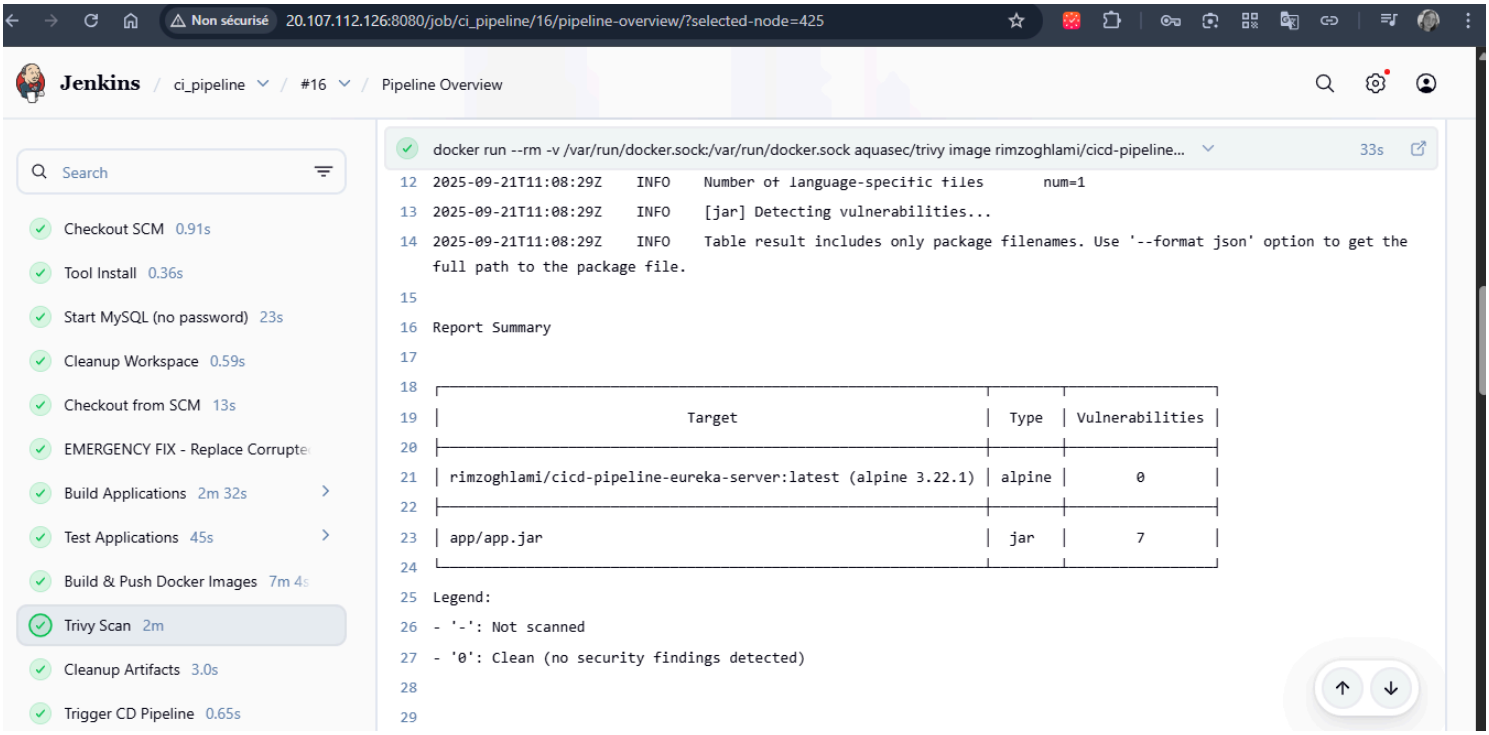


Figure 6

Trivy Security Scan Report Integrated into the Jenkins Pipeline Output.

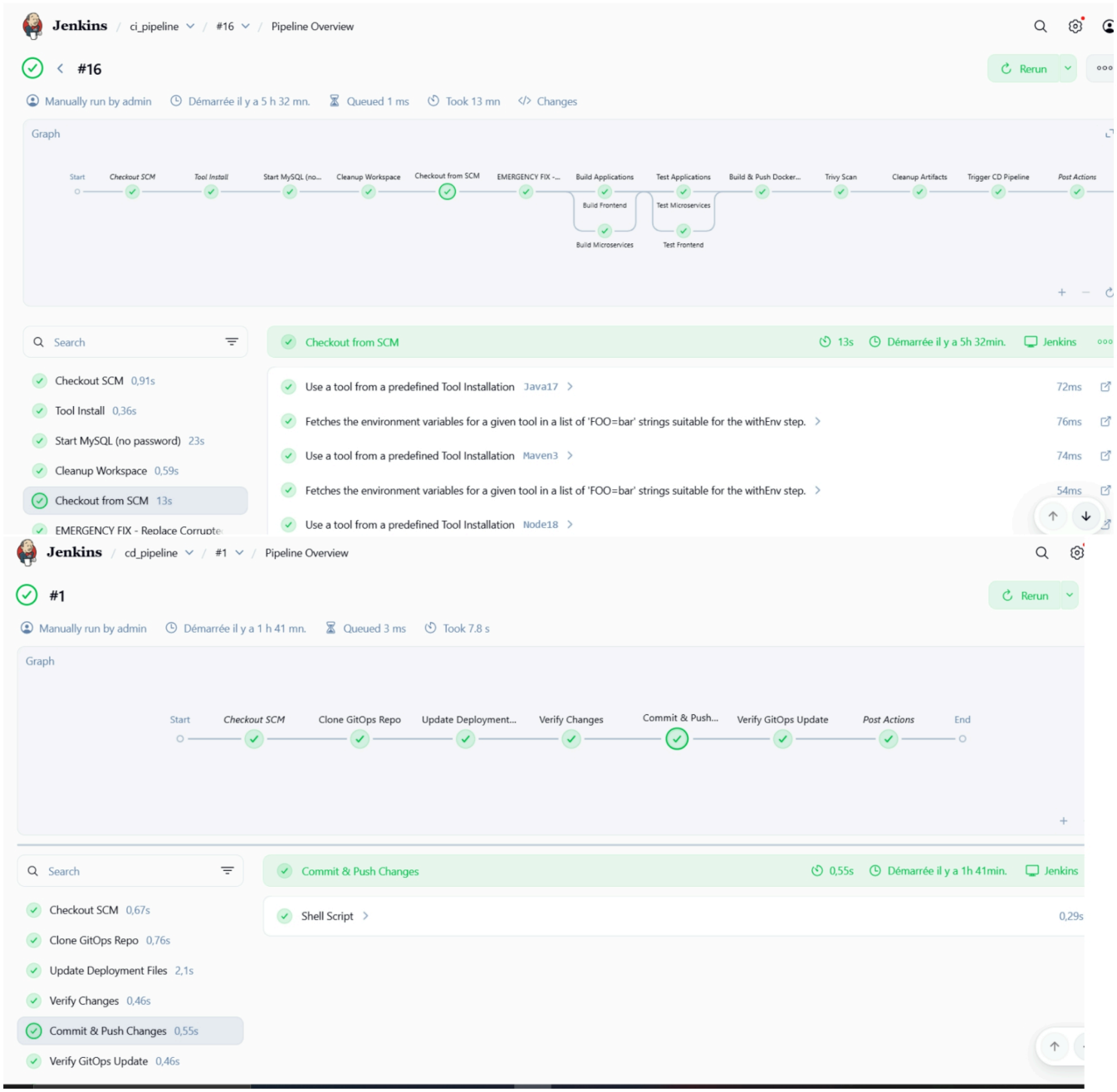


Figure 7

The End-to-End Automated CI/CD and GitOps Workflow.

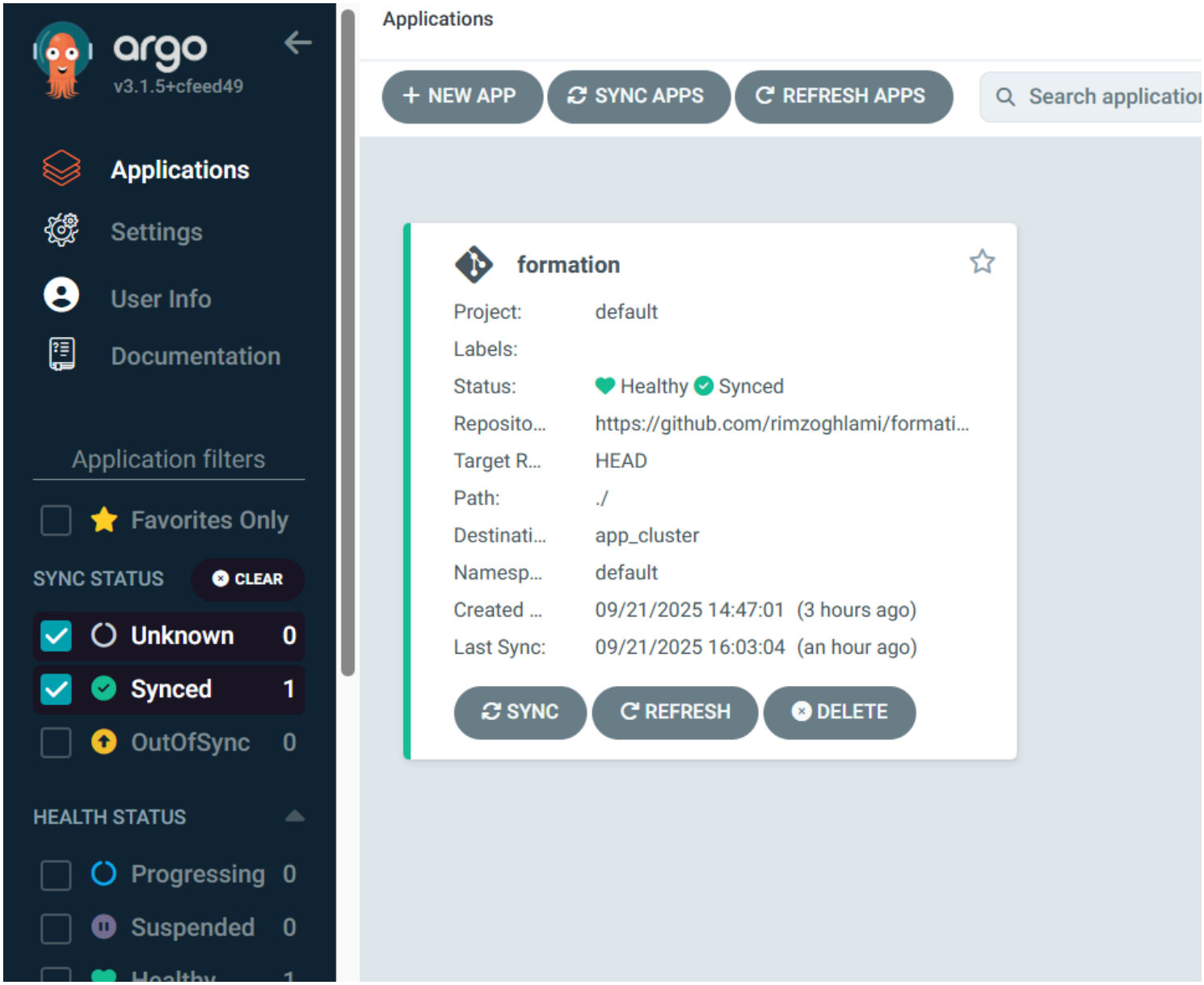


Figure 8

ArgoCD's Resource Tree View, Visualizing the Deployed Application Hierarchy and Health Status.

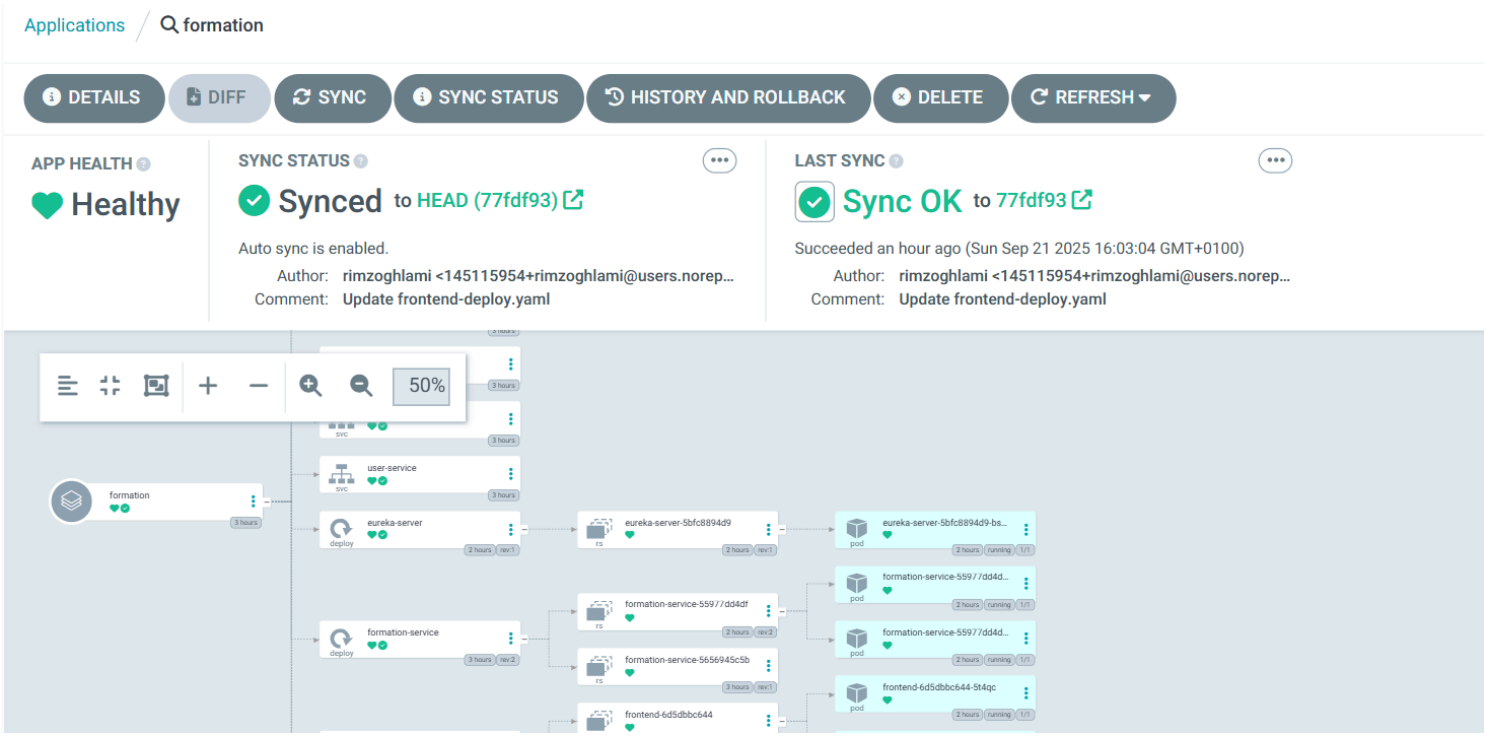


Figure 9

The Fully Deployed and Functional Training Management Application.