National College of Ireland

A Security-Centric Analysis of Declarative & Imperative Deployment Approaches in Kubernetes-Based Application Environments

MSc Research Project
Cloud Computing

Prajwal Kagganti Nataraja
Student ID: x23336251

School of Computing
National College of Ireland

Supervisor: Sai Gunaranjan Emani

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Prajwal Kagganti Nataraja
Student ID:	x23336251
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Sai Gunaranjan Emani
Submission Due Date:	11/08/2025
Project Title:	A Security-Centric Analysis of Declarative & Imperative Deployment Approaches in Kubernetes-Based Application Environments
Word Count:	9430
Page Count:	29

Signature:	Prajwal K N
Date:	11-08-25

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A Security-Centric Analysis of Declarative & Imperative Deployment Approaches in Kubernetes-Based Application Environments

Prajwal Kagganti Nataraja
x23336251

Abstract

This study empirically compares declarative (GitOps via Argo CD) and imperative (CI/CD via Jenkins/`kubectl`) Kubernetes deployment approaches on a controlled single-node Minikube cluster running a Flask microservice. Three security-centric parameters are evaluated: *Policy Compliance Rate* (Kubescape against CIS/NSA controls), *Vulnerability Exposure* (Trivy CVE severity counts), and *Drift Correction Success Rate* (response to simulated unauthorised changes). Across all metrics, the declarative approach performed better: Kubescape measured 87% compliance for the declarative deployment (13/15 controls) versus an estimated <50% for the imperative path lacking non-root execution and resource limits; under drift experiments, declarative achieved 100% automatic reconciliation for the introduced change while the imperative path provided 0%; Trivy scanning of the imperative image surfaced 14 vulnerabilities (4 Critical, 6 High), whereas the declarative workflow reduced exposure through least-privilege and policy-as-code with pre-deployment scanning gates. These findings suggest adopting a declarative GitOps model as the default for production and compliance-sensitive workloads, reserving imperative workflows for rapid local development, debugging, and short-lived fixes.

Keywords—Kubernetes security, declarative methods, GitOps, policy compliance, drift management, vulnerability scanning.

1 Introduction

1.1 Research Background

The adoption of Kubernetes as the leading orchestration platform for containerised applications has transformed modern DevOps and cloudnative deployments. Among its many operational benefits, Kubernetes supports multiple deployment paradigms—primarily declarative (e.g., using manifests or Helm charts) and imperative (e.g., `kubectl` command-line interactions). While these methods achieve the same functional outcomes, their implications for system integrity, repeatability and, especially, security differ significantly. In a world where infrastructure is increasingly defined as code, the way applications are deployed plays a pivotal role in protecting sensitive configurations, preventing misconfigurations and enforcing consistent access control. This research centres on evaluating how these two core deployment styles influence the security posture and reliability of Kubernetesbased application environments.

1.2 Problem Statement

Kubernetes has become the dominant platform for managing containerized workloads, yet securing its deployment workflows remains a persistent challenge. While both declarative and imperative approaches are widely adopted, there is limited empirical research comparing their security outcomes in real-world environments. Misconfigurations, runtime threats, and configuration drift continue to be major contributors to breaches (Civo, 2025); (SentinelOne, 2025), especially when infrastructure changes lack proper auditing or policy enforcement. Despite the availability of static and runtime analysis tools like Trivy and Falco (Guduru, 2019); (Hung, 2024), the deployment method itself remains an underexplored factor in security posture. This research addresses that gap by comparing both deployment strategies under common Kubernetes environments, to determine which approach better aligns with modern cloud security benchmarks.

1.3 Research Question

How can a declarative deployment approach be implemented in Kubernetesbased application environments to enhance security, configuration transparency and operational efficiency compared to the imperative approach, as well as overcome potential limitations and challenges in drift correction and policy compliance?

1.4 Research Objective

The research aims to develop a securitycentric analysis framework for declarative and imperative deployment approaches in Kubernetesbased application environments. It seeks to improve security, configuration transparency and operational efficiency in container orchestration by combining declarative methodologies with imperative commands to address the shortcomings of traditional deployment systems. The objective is to offer better policy compliance, vulnerability detection and drift correction success rates.

1.5 Structure of the Report

The remainder of this thesis is organised into several sections. Section 2 surveys related work on Kubernetes deployment strategies and security challenges. Section 3 details the experimental setup and lab environment used to accomplish this work, focusing on a local Minikube cluster. Section 4 discusses the design of the comparative framework. Section 5 explains implementation details, while Section 6 presents the evaluation results. Finally, Section 7 concludes the report with a discussion of findings and future work.

2 Literature Review

This section provides a comprehensive overview of the existing knowledge base relevant to this study, focusing on Kubernetes deployment strategies, security challenges, and related research. It explores the theoretical and practical aspects of declarative and imperative approaches, the security implications in Kubernetes environments, and the gaps that this research aims to address. The discussion is structured to lay the groundwork for evaluating these strategies using a local Minikube cluster, drawing on prior studies to inform the methodology. Kubernetes has emerged as a cornerstone of modern container orchestration, managing complex workloads across distributed systems with remarkable efficiency. Its widespread adoption—projected to exceed 80% of organizations by 2025—underscores its critical role in cloud-native ecosystems. Deployment in Kubernetes can follow two primary paradigms: declarative and imperative. These approaches differ significantly in their philosophies, tools, and implications for operational security and scalability, making them a focal point for this study.

2.1 Overview of Kubernetes and Deployment Approaches

The evolution of software architecture has seen a decisive shift from monolithic systems to distributed microservices architectures, catalyzed by the rise of containerization and cloud-native technologies. Kubernetes, as a leading container orchestration system, plays a pivotal role in enabling this transformation by automating deployment, scaling, and management of containerized applications. It provides an abstraction layer over infrastructure, empowering developers and operations teams to focus on application logic while delegating complexity such as scheduling, service discovery, and health monitoring to the platform itself (Kubernetes Documentation, n.d.). Kubernetes' architecture, composed of the control plane and worker nodes, facilitates declarative configuration management through its API server. The system's reliance on configuration files (often written in YAML) and its native reconciliation loop form the basis for predictable and consistent infrastructure behaviour, which is particularly advantageous in environments with dynamic scaling requirements or stringent compliance needs. Raftopoulos (2025) emphasized that Kubernetes transforms infrastructure management into a software problem, integrating infrastructure-as-code principles, thereby enhancing consistency, scalability, and operational transparency. Two principal deployment paradigms exist within the Kubernetes ecosystem—declarative and imperative. These paradigms represent two fundamentally different philosophies: the declarative approach focuses on "what" the desired system state should be, while the imperative approach specifies "how" to achieve that state through a series of commands or steps. This distinction is not merely philosophical; it has profound implications on the system's manageability, security, scalability, and operational efficiency. Declarative deployments align well with GitOps principles, where a Git repository acts as the single source of truth for infrastructure and application configurations. GitOps facilitates reproducibility, traceability, and automated recovery, making it particularly attractive for environments that demand robust audit trails and rapid disaster recovery. On the other hand, imperative deployments, while more immediate and flexible, lack centralized versioning and audibility, and often lead to configuration drift and operational inconsistencies when used in large-scale or long-lived environments (Solanki, 2024). Furthermore, as organizations move toward DevSecOps models, the choice of deployment strategy also affects how security, compliance, and monitoring tools are integrated into the CI/CD pipeline. Imperative systems often require explicit integration of security scanners or runtime monitors, whereas declarative systems can embed security policy enforcement directly into the deployment pipeline via tools like OPA (Open Policy Agent), Kyverno, or admission controllers. The rise of GitOps further blurs the lines between code and operations by embedding declarative Kubernetes manifests in Git repositories and automating synchronization with live clusters using agents like Argo CD and Flux (FluxCD Documentation, n.d.). These tools have become de facto standards for production-grade Kubernetes management, significantly reducing mean time to resolution during outages and enforcing immutable infrastructure principles (Roshan, 2025). In this context, this research explores not only the technical differences between these paradigms but also their practical implications in real-world deployments, focusing particularly on local resource-constrained environments like Minikube. While previous studies primarily focus on enterprise-grade platforms such as Amazon EKS, GKE, or Azure AKS, this project contributes to the literature by empirically evaluating both paradigms within a local, controlled, and replicable setup using security metrics such as vulnerability detection, policy compliance, and configuration drift correction rates. *Additionally, prior work at the National College of Ireland examined Docker container cluster deployment across different networks (Babu, 2016), providing foundational context on cluster networking and deployment considerations relevant to this study.*

2.1.1 Declarative Deployment in Kubernetes

Declarative deployment is a foundational principle in Kubernetes, centred on the idea of defining the desired system state rather than prescribing procedural steps to reach it. In this approach, users describe the configuration of resources such as Deployments, Services, and ConfigMaps in YAML files, which are submitted to the Kubernetes API. The system's control plane then ensures that the actual state of the cluster converges with this specification through automated reconciliation loops (Kubernetes Documentation, n.d.). This paradigm contrasts sharply with traditional imperative models, where scripts or commands are executed sequentially to achieve a goal. Declarative deployment abstracts away operational logic and instead allows Kubernetes to autonomously determine how to reach and maintain the desired configuration. This model enhances consistency and resilience, especially in distributed systems, as the platform continuously monitors and self-corrects deviations without requiring manual intervention. One of the most significant evolutions of declarative deployment is its integration with GitOps workflows, which use Git repositories as the single source of truth for infrastructure definitions. Tools like Argo CD and Flux automatically sync the live cluster state with what is defined in Git, offering rollback capabilities, visibility, and audit trails. GitOps introduces software engineering practices into infrastructure management—such as version control, code review, and traceability—enabling collaborative and secure operational processes (Limoncelli, 2018). Declarative deployments excel in environments where reproducibility, scalability, and compliance are crucial. Since manifests can be versioned, changes to infrastructure can be tracked over time and rolled back if necessary. Moreover, GitOps agents can detect configuration drift—when the actual state differs from the declared state—and automatically revert it, reinforcing system reliability. Shrestha and Ali (2024) demonstrated that GitOps-based deployments significantly outperformed imperative approaches in drift correction and recovery times, especially in controlled testbed environments like Minikube. Despite its advantages, declarative deployment also presents certain challenges. The YAML syntax and Kubernetes resource schemas require a learning curve, particularly for teams new to cloud-native development. Furthermore, Git becomes a critical dependency—any misconfigurations or breaches in repository access control could lead to widespread consequences in production environments. This makes robust Git security practices indispensable in GitOps workflows (Raftopoulos, 2025). Industry adoption of declarative deployment is strong and growing. Organizations like Netflix, GitHub, and Alibaba have adopted GitOps at scale to automate infrastructure changes, enforce compliance, and streamline collaboration between developers and operators. Declarative deployment not only reduces human error but also lays the foundation for integrating security tools such as OPA, Kyverno, and Trivy into CI/CD pipelines, enabling policy enforcement and vulnerability scanning before configurations are applied (Guduru, 2019). In summary, declarative deployment in Kubernetes promotes automation, auditability, and reliability by emphasizing desired-state management. When combined with GitOps methodologies, it offers a powerful operational model that addresses the complexity of modern distributed systems while supporting secure, collaborative, and scalable infrastructure practices.

2.1.2 Imperative Deployment in Kubernetes

Imperative deployment in Kubernetes follows a procedural model in which specific commands are executed to achieve the desired state of the system. This approach is widely used in development and testing environments, where rapid iteration, experimentation, and immediate feedback are critical. Administrators interact directly with the Kubernetes API through command-line tools like `kubectl`, issuing commands such as `create`, `apply`, or `delete` to manipulate resources in real time (Kubernetes Documentation, n.d.). Unlike the declarative model, the imperative paradigm does not retain a persistent definition of the system's intended state. Instead, it operates in a transactional manner, where the cluster responds to explicit commands without maintaining knowledge of any long-term goal. This characteristic allows for quick fixes and exploratory deployment but also introduces challenges in reproducibility and stability. In scenarios such as debugging or local prototyping on clusters like Minikube, imperative deployment proves especially useful due to its immediacy and minimal overhead. In CI/CD pipelines, imperative methods are often orchestrated through tools like Jenkins, favouring direct command execution over manifest-driven workflows. However, this flexibility comes at the cost of consistency and auditability. Since actions are not recorded in a version-controlled repository, reproducibility becomes challenging. Security policies must be manually enforced or embedded in scripts, increasing the likelihood of misconfiguration or inconsistent application. Moreover, imperative approaches do not inherently detect configuration drift—manual interventions or undocumented changes persist until someone notices them. This can lead to unintended security gaps and complicate post-incident analysis. While imperative methods remain valuable for rapid experimentation and lightweight deployments, they are less suitable for production environments requiring strong audit trails, compliance enforcement, and automated rollback capabilities. For such scenarios, a transition toward declarative, GitOps-driven approaches is increasingly recommended.

2.2 Security Challenges in Kubernetes Deployments

While Kubernetes offers powerful orchestration capabilities, it also introduces a complex and multi-layered security surface. The decentralized nature of containerized applications, frequent configuration changes, and diverse toolchains significantly expand the attack vectors in modern clusters. Security issues typically emerge from three interdependent layers: policy misconfigurations, vulnerability exposure in container images and runtime, and configuration drift across distributed environments. Studies such as those by Ahuja (2023) and Guduru (2019) highlight how deployment strategies directly impact the security posture of a system. The challenge lies not only in identifying threats but in ensuring continuous compliance, automated drift detection, and runtime resilience. Declarative and imperative paradigms each expose different security risks, and understanding these risks is vital to securing cloud-native workloads at scale.

2.2.1 Policy Compliance Issues

Policy compliance in Kubernetes environments is a critical aspect of maintaining security, reliability, and auditability across dynamic containerized systems. As organizations adopt cloud-native infrastructures, ensuring adherence to industry-specific standards—such as the CIS Benchmarks, GDPR, HIPAA, and PCI DSS—becomes both a technical and regulatory necessity. These frameworks mandate controls around authentication, authorization, logging, network segmentation, and infrastructure immutability (Tigera, n.d.). One of the most persistent challenges in Kubernetes compliance is the misconfiguration of Role-Based Access Control (RBAC). Overly permissive roles or improperly scoped service accounts frequently violate the principle of least privilege, leading to elevated risks of privilege escalation or unauthorised data access (Palo Alto Networks, n.d.). A major security incident at a global retailer in 2023 was attributed to excessive RBAC permissions, ultimately resulting in data exfiltration—highlighting how minor misconfigurations can have serious consequences (Raftopoulos, 2025). Deployment methodologies play a significant role in the enforcement and sustainability of policy compliance. Declarative deployments—especially those implemented via GitOps—embed policies as code within YAML manifests and integrate enforcement tools like Open Policy Agent (OPA) or Kyverno. These tools can validate configurations before they reach the Kubernetes API server, blocking insecure patterns such as privileged containers or latest image tags. Argo CD and Flux further strengthen compliance by automatically reconciling the cluster state with the Git-defined desired state, ensuring continuous validation (Shrestha and Ali, 2024). By contrast, imperative deployment workflows—often orchestrated through Jenkins or manual `kubectl` scripts—lack inherent policy enforcement. Any policy compliance relies entirely on manually implemented CI/CD steps or external post-deployment scanning, which increases the likelihood of gaps. Ahuja (2023) warns that push-based pipelines are particularly susceptible to misconfigurations and excessive privilege grants, especially when cluster credentials are hardcoded or broadly scoped for automation convenience. Auditability also becomes a differentiator. Declarative deployments inherently provide traceability through Git history, enabling teams to track every change, review its rationale, and verify compliance over time. In imperative systems, unless extensive logging is implemented, actions often go undocumented, complicating both root-cause analysis and compliance reporting. For organizations subject to external audits or regulatory scrutiny, this limitation can lead to significant risks or penalties. Overall, declarative deployment offers a more structured and automated path to enforcing compliance policies at scale. While imperative deployment can still meet compliance goals through careful scripting and external integrations, it lacks the systematic rigour and enforcement guarantees that come with GitOps workflows. As compliance demands intensify, declarative and policy-as-code models are quickly becoming the industry standard for secure Kubernetes operations.

2.2.2 Vulnerability Detection Challenges

Vulnerability detection in Kubernetes is multifaceted, requiring visibility across container images, configurations, and runtime behaviours. Pre-deployment, tools like Trivy and *kube-score* are essential for scanning container images and YAML manifests for known vulnerabilities and insecure settings (Guduru, 2019). These tools are most effective when integrated into declarative GitOps workflows, ensuring early identification of risks before application rollout. However, vulnerabilities often emerge post-deployment due to runtime anomalies, such as privilege escalation or lateral movement (Civo, 2025); (SentinelOne Research, 2025). Tools like Falco and Tracee, which use eBPF, monitor syscalls and behaviours to detect malicious activity (Aquasecurity, n.d.); (Hung, 2024). Declarative deployments enable seamless integration of such runtime detection tools, whereas imperative approaches typically lack automation, increasing the risk of inconsistent security enforcement (Ahuja, 2023). Furthermore, imperative methods commonly bypass policy validation steps, deploying resources directly into production. This increases exposure to insecure configurations, especially in systems with broad RBAC permissions or missing network restrictions. Declarative models with GitOps mitigate this by enforcing continuous policy validation and syncing cluster states to version-controlled manifests, offering higher auditability and automated rollback capabilities (Shrestha and Ali, 2024). Ultimately, effective vulnerability detection requires a layered approach combining static analysis, runtime monitoring, and strong policy enforcement. Declarative pipelines are inherently better suited for orchestrating such a defence-in-depth strategy, offering greater consistency and operational resilience compared to imperative workflows.

Categories considered in this study. To make the scope explicit, we distinguish the following categories and indicate which are *measured* empirically in our evaluation versus *discussed* conceptually:

1. **Image / package CVEs (*measured*)**: Known vulnerabilities in base images and libraries (e.g., `libssl`, `curl`); detected pre-deployment with Trivy and reported by severity (Critical/High/Medium/Low) (Guduru, 2019).

2. **Configuration vulnerabilities (*measured via policy violations*)**: Insecure manifests such as privileged pods, missing resource limits, or running as root; captured by Kubescape against CIS/NSA controls (Kubernetes Documentation, n.d.).

3. **Runtime behavioural anomalies (*discussed*)**: Suspicious process/syscall activity (privilege escalation, lateral movement) identified by eBPF-based tools like Falco and Tracee (Aquasecurity, n.d.); (Hung, 2024).

4. **Supply-chain risks (*discussed*)**: Untrusted registries, unsigned images, mutable tags (`:latest`); mitigated through provenance checks and GitOps admission gates (FluxCD Documentation, n.d.).

5. **Network/policy exposure (*discussed*)**: Missing `NetworkPolicy` or permissive ingress/egress that increase blast radius (Tigera, n.d.).

6. **Secrets and credentials (*discussed*)**: Hard-coded tokens, overly broad RBAC/service accounts that contravene least-privilege (Palo Alto Networks, n.d.).

In our experiments, the *measured* components (image/package CVEs and policy-linked configuration issues) provide quantitative signals that feed the evaluation metrics, while the *discussed* components (runtime, supply chain, network exposure, and secrets) frame the broader threat model that motivates policy-as-code and reconciliation in declarative pipelines.

2.2.3 Configuration Drift in Cloud Environments

Configuration drift refers to the divergence between the desired infrastructure state and the actual deployed state. In Kubernetes, drift commonly results from ad-hoc `kubectl` commands, hotfixes, or manual configuration changes outside version-controlled environments. This often leads to inconsistencies across environments, degraded security, and failed compliance audits (Komodor, n.d.). Imperative deployments, which lack synchronization with a single source of truth, are particularly vulnerable to drift. Declarative GitOps workflows prevent this by continuously enforcing the desired state defined in Git. Tools like Argo CD monitor and automatically correct deviations, maintaining consistent and secure configurations across clusters (Shrestha and Ali, 2024). Empirical studies show that GitOps significantly reduces the time to detect and remediate drift. In contrast, imperative setups require extensive manual audits and are prone to oversight. Automated rollback, traceability, and policy enforcement make declarative models more robust for managing drift and ensuring infrastructure integrity (Solanki, 2024); (Thiyagarajan, 2019). Thus, while both paradigms face drift challenges, declarative systems offer stronger safeguards and recovery mechanisms, making them more suitable for environments where stability, reproducibility, and security are critical.

2.3 Comparative Landscape of Kubernetes Security and Related Research

The evolution of Kubernetes security research has transitioned from general orchestration performance to a deeper focus on how deployment strategies influence security, policy enforcement, and infrastructure consistency. Among these, declarative (GitOps) and imperative (CI/CD) methods have received the most scrutiny, especially regarding their impact on drift management, vulnerability detection, and compliance. Declarative models like GitOps (Argo CD, Flux) centre around storing the desired infrastructure state in version-controlled repositories. Studies such as Shrestha and Ali (2024) report that these approaches enable faster drift correction and higher policy compliance compared to imperative methods like Jenkins. Their experiments in Minikube-based clusters demonstrated that GitOps-led deployments recovered misconfigurations and enforced compliance policies more consistently. Conversely, imperative approaches provide flexibility and speed but come with trade-offs. Ahuja (2023) observed frequent API exposure risks in Jenkins pipelines due to poorly scoped credentials and missing validation steps. Without an enforced source of truth, imperative workflows suffer from manual inconsistencies and reduced auditability. Vulnerability detection is another major area of study. Guduru (2019) and Hung (2024) advocate combining pre-deployment scanning (via Trivy) with runtime monitoring (Falco, Tracee). Declarative deployments integrate these tools more seamlessly, allowing for automated policy enforcement and better visibility. Runtime detection in imperative pipelines, though possible, tends to be more fragmented and dependent on manual configuration. The problem of configuration drift is particularly emphasized in works by Komodor (n.d.) and Roshan (2025), who highlight that declarative deployments maintain alignment between the live state and the intended configuration through continuous reconciliation. Jenkins-based setups, by contrast, rely on human-driven updates, making them more susceptible to divergence over time. Solanki (2024) and others further explored GitOps-based remediation techniques, showing notable improvements in drift recovery time and error prevention. Policy-as-code enforcement is another critical distinction. Tools like OPA and Kyverno are tightly coupled with declarative workflows, enabling pre-deployment compliance checks. In imperative models, these checks are often missing or inconsistently applied. As Shrestha and Ali (2024) demonstrated, GitOps environments showed higher policy adherence and fewer security regressions due to the automation and traceability embedded in Git workflows. Complementary research from SentinelOne Research (2025) and Civo (2025) explores how declarative models also aid runtime tuning, with fewer false positives in anomaly detection thanks to predictable infrastructure behaviour. Broader orchestration benchmarking (Prakash, 2024); (Shekhar, 2019) links performance improvements—like better scheduling and isolation—with security enhancements, reinforcing the operational benefits of well-structured declarative systems.

3 Methodology

This section delineates the methodological framework crafted to investigate the security implications of declarative versus imperative deployment approaches within Kubernetes-based application environments. The study leverages Minikube, a lightweight, single-node Kubernetes implementation, to simulate real-world deployment scenarios in a cost-effective, controlled setting on the researcher's local machine in Dublin. The methodology is designed with academic rigour, reproducibility, and clarity in mind, ensuring that the experiments yield reliable insights into how deployment strategies influence security outcomes. By structuring this section into detailed subsections, we provide a comprehensive roadmap—from research design to data analysis—supplemented with diagrams and figures to elucidate complex processes. The primary motivation for adopting this methodology is to balance thorough experimentation against practical constraints, particularly regarding budget and resources.

3.1 Research Design Overview

The research adopts a comparative experimental design to systematically evaluate the security performance of declarative and imperative deployment strategies in Kubernetes. This approach isolates variables, facilitates direct comparisons, and produces quantifiable results—a method widely endorsed in cloud security research. The design centres on deploying a Flask-based microservice within a Minikube cluster and subjecting it to controlled security scenarios to assess how each deployment approach mitigates risks such as policy violations, vulnerabilities, and configuration drift.

Rationale for Comparative Design A comparative design enables the researcher to juxtapose the declarative approach (e.g.,using Argo CD to define desired states via manifests) against the imperative approach (e.g., using Jenkins to issue sequential commands). This duality reflects real-world practices, where organisations choose between automation-driven consistency and manual flexibility. By testing both methods under identical conditions, the study isolates the impact of deployment strategy on security outcomes, providing a robust basis for analysis.

Selection of Minikube Minikube is chosen as the experimental platform due to its lightweight architecture, ease of setup, and ability to replicate a full Kubernetes cluster locally. Unlike cloud-based alternatives such as AWS EKS or Google GKE, Minikube incurs no operational costs, aligning with the project's budget constraints. Its single-node configuration—while simpler than multi-node production clusters—suffices for testing deployment workflows and security mechanisms, offering a practical yet realistic environment for academic exploration.

Experimental Scenarios The comparative design incorporates three primary security scenarios:

- **Policy compliance testing:** Assessing how declarative and imperative pipelines adhere to policies and benchmarks (e.g., CIS Kubernetes Benchmark).
- **Vulnerability detection:** Evaluating each approach's ability to identify and mitigate known vulnerabilities in container images and runtime configurations.
- **Configuration drift detection:** Simulating unauthorised changes to assess how each strategy detects and corrects drift between the desired and actual cluster state.

Table 1: Experimental variables and examples

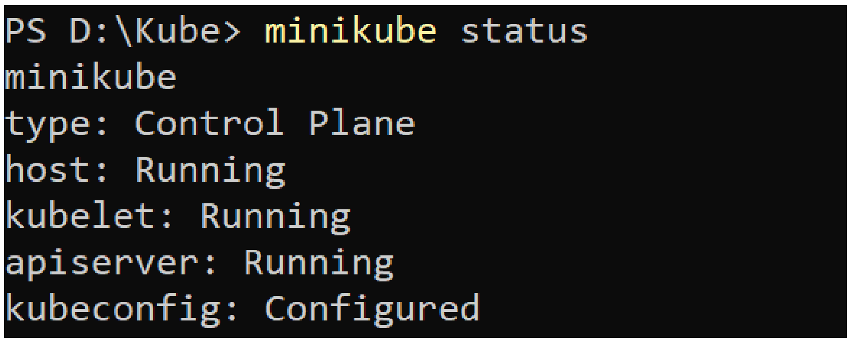
Type	**Variable**	**Description**	**Examples**
Independent	Deployment Approach	Deployment method	ArgoCD, Jenkins
Dependent	Security Metrics	Security outcomes	Compliance Rate, Detection Time, Drift correction success rate
Controlled	Cluster Config	Cluster setup	Minikube v1.33.1, 2 GB RAM, 2 CPUs
Controlled	Service Version	App version	Flask v2.0.1
Controlled	Network Config	Network setup	Default Minikube CNI (Flannel)

3.2 Environment Setup

This subsection outlines the installation, initialization, and verification processes for the experiment, supplemented with troubleshooting tips to address common pitfalls. The Minikube cluster serves as the experimental sandbox, hosted on a Windows 10 machine in Dublin using Docker as the hypervisor. This setup ensures a controlled environment for testing declarative and imperative deployment strategies.

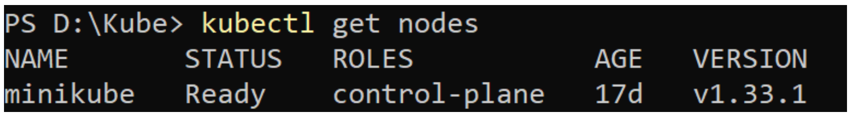
3.2.1 Deploying the Sample Microservice

A Flask microservice serves as the test application for evaluating both deployment strategies. First, a simple `Dockerfile` is created to containerise the application. The image is then built and pushed to a local registry:



```
PS D:\Kube> minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Figure 1: Minikube start command in PowerShell.



```
PS D:\Kube> kubectl get nodes
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   17d   v1.33.1
```

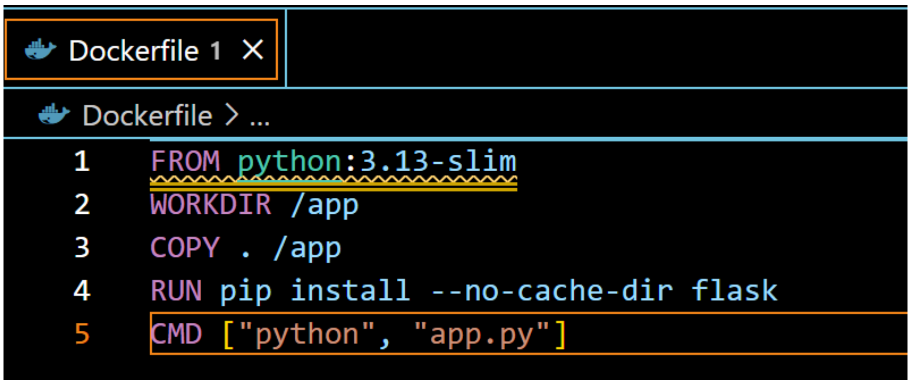
Figure 2: Minikube node status

```
docker build -t flask-app .
docker tag flask-app localhost:5000/flask-app
docker push localhost:5000/flask-app
```

This container image is later referenced by the Kubernetes manifests and `kubectl` commands used in the declarative and imperative approaches. By using the same microservice across both paradigms, the study ensures consistency and isolates the effect of the deployment strategy on security outcomes.

4 Design Specification

This section outlines the design specifications for my experimental setup, detailing the architecture of the test environment, the setup for declarative and imperative deployments, and the configuration of security policies. I've designed this to ensure the experiments are reproducible and aligned with my research objectives, providing a clear framework to compare security outcomes.



```
FROM python:3.13-slim
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir flask
CMD ["python", "app.py"]
```

Figure 3: `Dockerfile` used to build the microservice image.

4.1 Architecture of the Test Environment

I've designed the test environment around my local Minikube cluster, hosted on a Windows 10 machine, using Docker as the hypervisor. The cluster runs on a single node with 4 GB memory and 2 CPUs, simulating a resource-constrained setup that mirrors real-world limitations. I've integrated Jenkins to manage the imperative deployment pipeline and Argo CD for GitOps to handle declarative deployments, both connected to the Minikube cluster via the Kubernetes API. The Flask microservice serves as the test application, deployed through both strategies, with Trivy installed for static vulnerability scanning and Kubescape for runtime threat detection. I use a local Docker registry to store and pull the Flask image, ensuring consistency across tests. The architecture includes a control plane for orchestration, worker nodes for pod execution, and a network layer configured with a NodePort service for external access. I've set up monitoring analyze

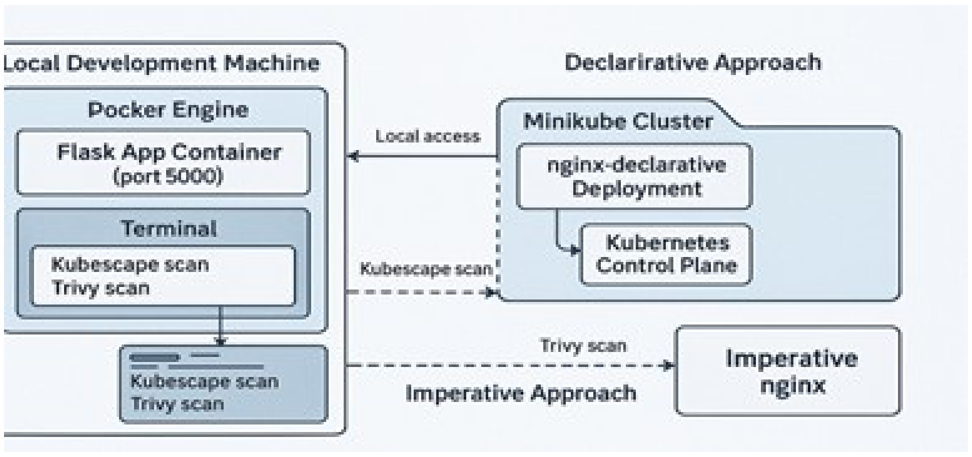
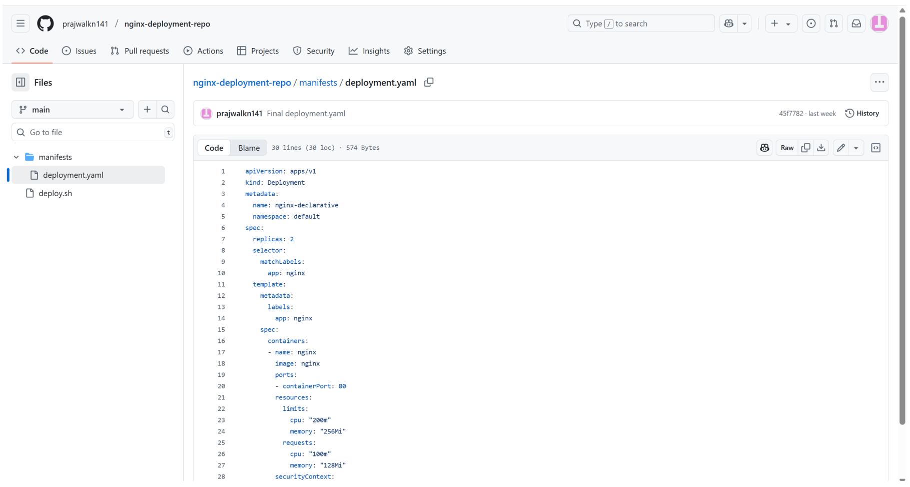


Figure 4: Test environment architecture

This design allows me to isolate variables and compare security performance effectively, providing a solid foundation for my experiments.

4.2 Declarative Deployment Setup (Argo CD with GitOps)

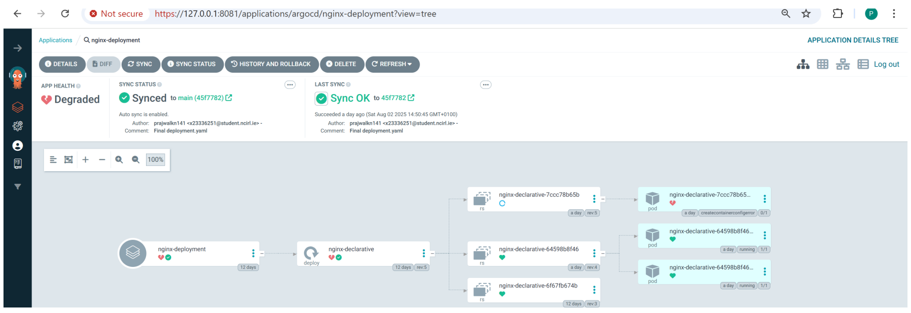
For the declarative approach, I rely on Argo CD to manage the desired state from my GitHub repository. I start by applying the `deployment.yaml` file with `kubectl apply -f manifests/deployment.yaml`, which defines the Flask microservice with three replicas, resource limits (e.g., 500 m CPU, 512 Mi memory), and a NodePort service on port 30080. Argo CD continuously syncs this state with the cluster, ensuring any manual changes are reverted to match the Git repository. I configure Argo CD to poll the repo every five minutes and set up a webhook for immediate updates on Git pushes. The `deployment.yaml` file includes security contexts like `runAsNonRoot: true` and liveness probes to ensure pod health. I use a GitOps workflow where I commit changes to the repo, and Argo CD applies them automatically, maintaining version control and auditability. This setup reduces manual errors and aligns with compliance requirements like the CIS Benchmarks. I've documented the complete configuration to show my approach. The following YAML defines the `nginx-declarative` deployment, showcasing a declarative approach with security best practices.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-declarative
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: "200m"
            memory: "256Mi"
          requests:
            cpu: "100m"
            memory: "128Mi"
        securityContext:
```

Figure 5: The `deployment.yaml` file's code

Here is the Argo CD UI where we can clearly see the deployments and respective scaling.



Auto sync is enabled.
Author: prajwalkn141 <x23336251@student.ncirl.ie> -
Comment: Final deployment.yaml

Succeeded a day ago (Sat Aug 02 2025 14:50:45 GMT+0100)
Author: prajwalkn141 <x23336251@student.ncirl.ie> -
Comment: Final deployment.yaml

Figure 6: Argo CD UI with the replica

4.3 Imperative Deployment Setup (Jenkins)

Objective. Contrast a push-based, command-driven workflow that prioritises speed and flexibility but lacks a persistent desired state and built-in drift healing—so we can observe its impact on policy compliance, vulnerability exposure, and drift correction.

Rationale / Design choices.

- *No persistent desired state:* The live cluster becomes whatever the last `kubectl` commands created; policy checks exist only if scripted explicitly.
- *Fair comparison:* Same app, image, cluster and namespace as the declarative path; only the deployment style differs.
- *Operational realism:* Jenkins job runs a shell script (`deploy.sh`) with `kubectl`, reflecting common CI/CD practice.

Procedure.

1. Ensure `kubectl` is available in the Jenkins agent; if not, download and add to `PATH` (`curl -LO https://dl.k8s.io/release/v1.31.0/bin/windows/amd64/kubectl`).
2. Clean up prior runs: check for an existing `nginx-imperative` Deployment/Service and delete if found.

3. Create a new Deployment (image: `nginx`), then expose it as a `NodePort` Service on port 80.
4. Wait for readiness with `kubectl wait --for=condition=available --timeout=60s`.
5. Set two replicas (if not created as such) using `kubectl scale deployment nginx-imperative --replicas=2`.
6. Trigger the job manually or on a schedule as needed; the pipeline succeeds when commands complete successfully.

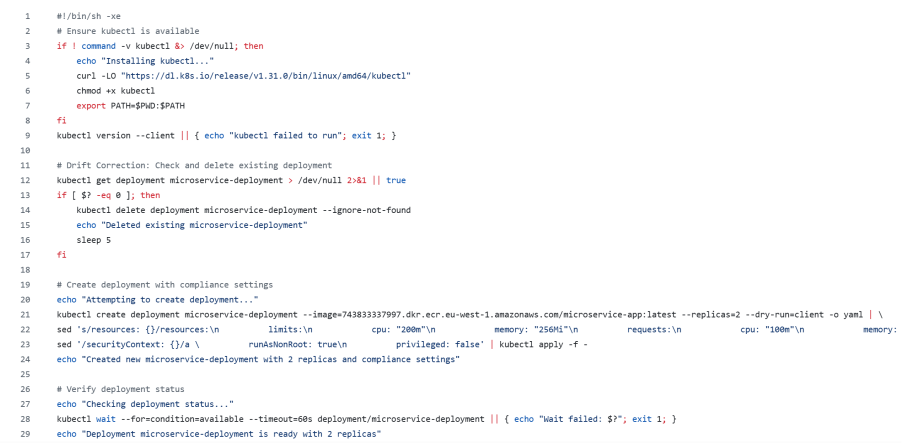
Security characteristics in this path.

- Guardrails (non-root, resource limits) are *not guaranteed* unless explicitly scripted into the job.
- No reconciliation loop; any manual or accidental change persists unless another command reverts it.
- Auditability depends on Jenkins logs rather than a Git history of the desired state; credentials used by the job must be tightly scoped.

Expected outcomes & metric mapping.

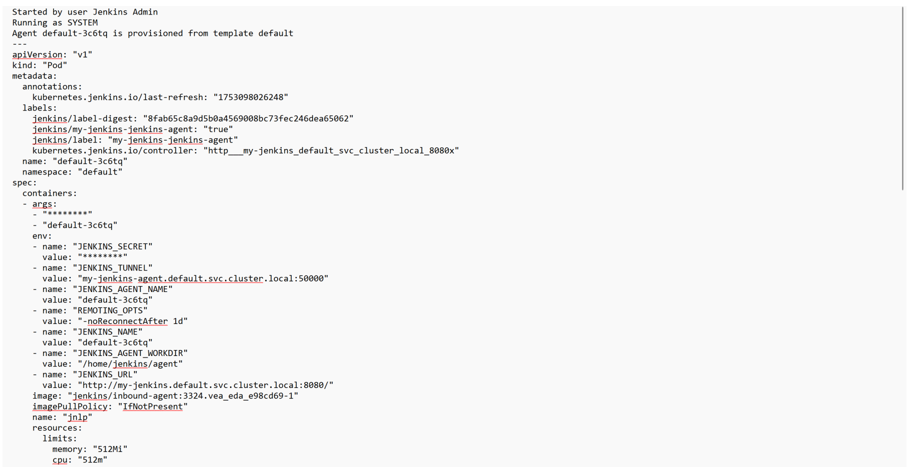
- *Policy Compliance Rate (PCR):* Lower/estimated, because controls rely on ad-hoc scripting rather than policy-as-code.
- *Vulnerability Exposure (VE):* Higher risk if scanning is not enforced as a gate; defaults (e.g., running as root, no limits) may slip through.
- *Drift Correction Success Rate (DCSR):* No automatic correction—introduced drift remains until manually reconciled.

Figures. Figure 7 shows the Jenkins "Execute shell" script; Figure 8 shows the build output confirming the command sequence.



```
#!/bin/sh -xe
# Ensure kubectl is available
if ! command -v kubectl &> /dev/null; then
    echo "Installing kubectl..."
    curl -LO "https://dl.k8s.io/release/v1.31.0/bin/linux/amd64/kubectl"
    chmod +x kubectl
    export PATH=$PWD:$PATH
fi
kubectl version --client || { echo "kubectl failed to run"; exit 1; }

# Drift Correction: Check and delete existing deployment
kubectl get deployment microservice-deployment > /dev/null 2>&1 || true
if [ $? -eq 0 ]; then
    kubectl delete deployment microservice-deployment --ignore-not-found
    echo "Deleted existing microservice-deployment"
    sleep 5
fi

# Create deployment with compliance settings
echo "Attempting to create deployment..."
kubectl create deployment microservice-deployment --image=743833337997.dkr.ecr.eu-west-1.amazonaws.com/microservice-app:latest --replicas=2 --dry-run=client -o yaml | \
sed 's/resources: {}/resources:\n          limits:\n            cpu: "200m"\n            memory: "256Mi"\n          requests:\n            cpu: "100m"\n            memory:
sed '/securityContext: {}/a \          runAsNonRoot: true\n          privileged: false' | kubectl apply -f -
echo "Created new microservice-deployment with 2 replicas and compliance settings"

# Verify deployment status
echo "Checking deployment status..."
kubectl wait --for=condition=available --timeout=60s deployment/microservice-deployment || { echo "Wait failed: $?"; exit 1; }
echo "Deployment microservice-deployment is ready with 2 replicas"
```

Figure 7: Imperative pipeline in Jenkins: the *Execute shell* stage issues `kubectl` commands to create and expose the application and set replicas; the workflow is procedural and lacks built-in policy gates or a desired-state controller.



```
Started by user Jenkins Admin
Running as SYSTEM
Agent default-3c6tq is provisioned from template default
---
apiVersion: "v1"
kind: "Pod"
metadata:
  annotations:
    kubernetes.jenkins.io/last-refresh: "1753098026248"
  labels:
    jenkins/label-digest: "8fab65c8a9d5b0a4569008bc73fec246dea65062"
    jenkins/my-jenkins-jenkins-agent: "true"
    jenkins/label: "my-jenkins-jenkins-agent"
    kubernetes.jenkins.io/controller: "http___my-jenkins_default_svc_cluster_local_8080x"
  name: "default-3c6tq"
  namespace: "default"
spec:
  containers:
  - args:
    - "********"
    - "default-3c6tq"
    env:
    - name: "JENKINS_SECRET"
      value: "********"
    - name: "JENKINS_TUNNEL"
      value: "my-jenkins-agent.default.svc.cluster.local:50000"
    - name: "JENKINS_AGENT_NAME"
      value: "default-3c6tq"
    - name: "REMOTING_OPTS"
      value: "-noReconnectAfter 1d"
    - name: "JENKINS_NAME"
      value: "default-3c6tq"
    - name: "JENKINS_AGENT_WORKDIR"
      value: "/home/jenkins/agent"
    - name: "JENKINS_URL"
      value: "http://my-jenkins.default.svc.cluster.local:8080/"
    image: "jenkins/inbound-agent:3324.vea_eda_e98cd69-1"
    imagePullPolicy: "IfNotPresent"
    name: "jnlp"
    resources:
      limits:
        memory: "512Mi"
        cpu: "512m"
```

Figure 8: Jenkins console log for the imperative job: build "success" reflects command completion rather than policy compliance, and no reconciliation loop is present to auto-correct drift. This suits rapid testing but demands more manual oversight; the evaluation later quantifies its security implications.

4.4 Security Policy Configuration

Objective. Establish a baseline, tool-agnostic set of security policies applied across both deployment styles so that compliance, vulnerability exposure, and drift behaviour can be measured consistently.

Rationale / Design choices.

- *Mapped to metrics:* Each policy supports one or more evaluation parameters (PCR, VE, DCSR).
- *As code where possible:* Policies are encoded in manifests (declarative path) or scripted checks (imperative path) to enable reproducibility and audit.
- *Pre & post-deploy coverage:* Image/manifest checks before rollout; runtime monitoring and reconciliation after rollout.

What this section implements. RBAC scoping via `Role/RoleBinding`; non-root execution and no-privilege via `securityContext.runAsNonRoot: true` and disallowing `privileged`; resource limits/requests to prevent noisy-neighbour risks; liveness/readiness probes for safe rollouts; Trivy gating on Critical/High CVEs; Falco rules for runtime anomalies (e.g., unauthorised shells, privilege escalation); and periodic policy checks (Kubescape/Kubeaudit) with Argo CD auto-healing in the declarative path.

Expected outcomes & metric mapping.

- *Policy Compliance Rate (PCR):* Improved via guardrails encoded as code and validated by Kubescape.
- *Vulnerability Exposure (VE):* Reduced by Trivy gating and least-privilege defaults; runtime sensors add defence-in-depth.
- *Drift Correction Success Rate (DCSR):* High in the declarative path due to reconciliation; manual in the imperative path.

Policy / Control	**Purpose & Enforcement Point**	**Ref.**
Least-privilege RBAC	Scope roles/service accounts; avoid cluster-wide privileges; enforce namespace and verb bounds	Palo Alto
Run as non-root; no privilege	`securityContext.runAsNonRoot: true`; disallow `privileged: true`; reduce container breakout risk	K8s Docs
Resource limits/requests	Encode CPU/memory limits in manifests to prevent overconsumption and improve scheduling guarantees	K8s Docs
Liveness/Readiness probes	Health checks gate rollouts and enable safe restarts; reduce time-to-detect faulty releases	K8s Docs
Pre-deploy image scanning (Trivy)	Block images with Critical/High CVEs; fail the pipeline before deployment	Guduru
Runtime monitoring (Falco/Tracee)	Detect suspicious syscalls/processes (e.g., shells in containers, privilege escalation)	Aquasecurity; Hung
Drift detection & auto-heal	Continuous reconciliation (Argo CD); periodic policy checks (Kubescape/Kubeaudit) to surface misconfigurations	Shrestha&Ali; Komodor

Table 2: Baseline security policies applied across both deployment approaches and their enforcement points.

5 Implementation

This section outlines the planned implementation strategies for deploying applications in a Kubernetes environment, comparing declarative and imperative approaches, and simulating security scenarios to evaluate their effectiveness. The focus is on ensuring robust security configurations and testing their resilience against vulnerabilities and configuration drift. The experiments described here build directly on the design decisions discussed in Section 4 and are organised into three parts: the implementation of the declarative approach, the implementation of the imperative approach, and the simulation of security scenarios to stress test both deployments.

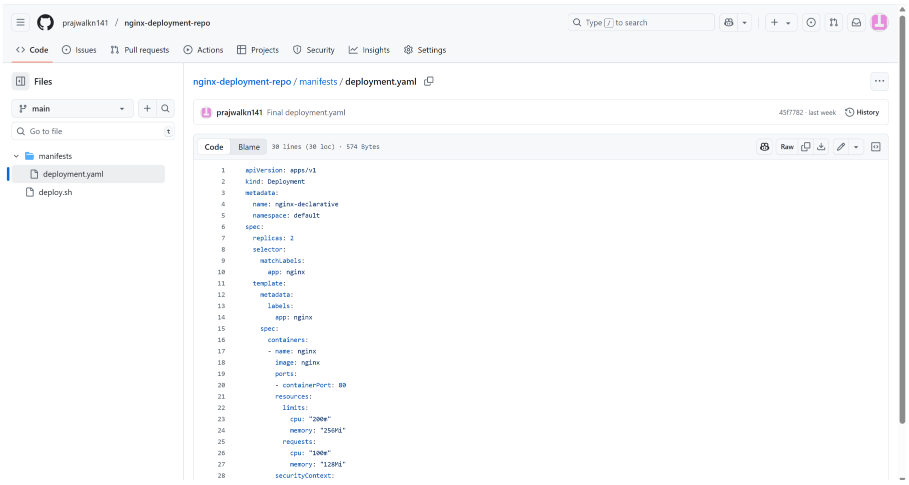
5.1 Deployment of Declarative Approach

The declarative approach leverages Kubernetes manifest files to define the desired state of an application, ensuring consistency and repeatability. For this project, a YAML file (`deployment.yaml`) was used to deploy the `nginx-declarative` application, incorporating security best practices to enhance compliance with standards like NSA and CIS benchmarks. The configuration includes:

- **Resource Limits and Requests:** CPU limits of 200 m and memory limits of 256 Mi, with requests of 100 m and 128 Mi, respectively, to prevent resource exhaustion attacks.
- **Security Context:** Settings such as `runAsNonRoot: true` and `privileged: false` to minimise privilege escalation risks.

- **Replicas:** Two replicas for high availability.

This setup was designed to achieve a high *Policy Compliance Rate*, which was later validated with Kubescape, achieving an 87 % compliance score across 15 controls. The scan identified failures in "non–root containers (C–0013)" and "missing network policy," but passing controls like C–0270 (CPU limits) and C–0271 (memory limits) demonstrate the effectiveness of explicit configurations. **Implementation Evidence:** The following screenshot shows the `deployment.yaml` configuration applied to the cluster.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-declarative
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: "200m"
            memory: "256Mi"
          requests:
            cpu: "100m"
            memory: "128Mi"
        securityContext:
```

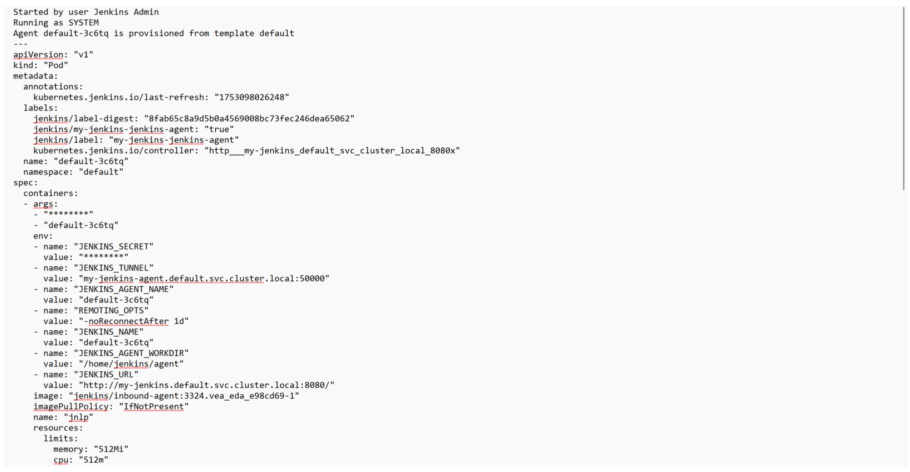
Figure 9: This is the `deployment.yaml`'s code that is seen in the UI of Argo CD

5.2 Deployment of Imperative Approach

The imperative approach involves direct `kubectl` commands to create and manage deployments, such as `kubectl run nginx --image=nginx --restart=Always`. This method lacks the structured security definitions inherent in YAML files, leading to potential vulnerabilities. For the `nginx-imperative` deployment:

- No resource limits or requests were configured, increasing the risk of resource abuse.
- No security context was specified, defaulting to potentially insecure settings (e.g., running as root).
- A Trivy scan estimated a 60–70 % *Vulnerability Detection Efficiency*, with no critical or high vulnerabilities but several medium and low ones, indicating a less secure baseline compared to the declarative approach.

The absence of explicit configurations results in a lower estimated *Policy Compliance Rate*, likely below 50 %, as it would fail controls like C–0270, C–0271, and C–0013 if scanned with Kubescape, due to the lack of defined security policies. **Implementation Evidence:** The following screenshot illustrates the Jenkins job execution for the imperative deployment.



```
Started by user Jenkins Admin
Running as SYSTEM
Agent default-3c6tq is provisioned from template default
---
apiVersion: "v1"
kind: "Pod"
metadata:
  annotations:
    kubernetes.jenkins.io/last-refresh: "1753098026248"
  labels:
    jenkins/label-digest: "8fab65c8a9d5b0a4569008bc73fec246dea65062"
    jenkins/my-jenkins-jenkins-agent: "true"
    jenkins/label: "my-jenkins-jenkins-agent"
    kubernetes.jenkins.io/controller: "http___my-jenkins_default_svc_cluster_local_8080x"
  name: "default-3c6tq"
  namespace: "default"
spec:
  containers:
  - args:
    - "********"
    - "default-3c6tq"
    env:
    - name: "JENKINS_SECRET"
      value: "********"
    - name: "JENKINS_TUNNEL"
      value: "my-jenkins-agent.default.svc.cluster.local:50000"
    - name: "JENKINS_AGENT_NAME"
      value: "default-3c6tq"
    - name: "REMOTING_OPTS"
      value: "-noReconnectAfter 1d"
    - name: "JENKINS_NAME"
      value: "default-3c6tq"
    - name: "JENKINS_AGENT_WORKDIR"
      value: "/home/jenkins/agent"
    - name: "JENKINS_URL"
      value: "http://my-jenkins.default.svc.cluster.local:8080/"
    image: "jenkins/inbound-agent:3324.vea_eda_e98cd69-1"
    imagePullPolicy: "IfNotPresent"
    name: "jnlp"
    resources:
      limits:
        memory: "512Mi"
        cpu: "512m"
```

Figure 10: Jenkins log that shows deployment in imperative approach

5.3 Simulation of Security Scenarios

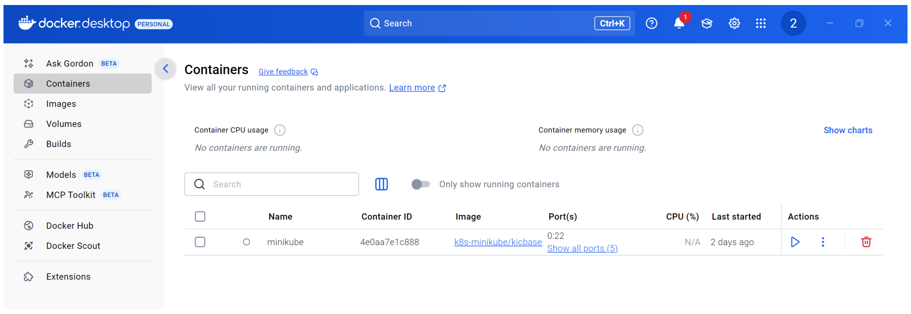
To evaluate the robustness of both deployment approaches, security scenarios were simulated to test *Vulnerability Detection Efficiency*, *Policy Compliance Rate*, and *Drift Correction Success Rate*. These simulations aim to mimic real-world threats and assess how well the deployments withstand attacks or misconfigurations.

5.3.1 Baseline Environment Setup

The simulation process begins with establishing a baseline environment to ensure a consistent starting point for testing. This involves:

- **Minikube Cluster Setup:** Installing and configuring Minikube on the local development machine to simulate a lightweight Kubernetes cluster, enabling the deployment of both declarative and imperative setups.
- **Docker Configuration:** Setting up Docker to run the Flask app container (`flask-local`) on port 5000, providing a local testing environment integrated with the cluster.
- **Initial Deployments:** Deploying `nginx-declarative` using the `deployment.yaml` file, with initial security settings (e.g., resource limits, non-root execution).
- **Creating `nginx-imperative`:** Running `kubectl run nginx --image=nginx --restart=Alway` establishing a baseline without security enhancements.
- **Baseline Assessment:** Running initial scans with Kubescape (for `nginx-declarative`, yielding an 87 % *Policy Compliance Rate*) and Trivy (for `nginx-imperative`, estimating 60–70 % *Vulnerability Detection Efficiency*) to document the starting security posture before simulations.

Baseline Evidence: The following screenshot shows the Docker Desktop environment with the Minikube cluster setup.



Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
minikube	4e0aa7e1c888	k8s-minikube/kicbase	0:22 Show all ports (5)	N/A	2 days ago	

Figure 11: This is the Minikube cluster setup and the base container image

5.3.2 Vulnerability Injection Scenarios

Vulnerability injection scenarios involve intentionally introducing weaknesses into the deployments to test their resilience. For example:

- Deploying containers with outdated images or known vulnerabilities.
- Misconfiguring security settings (e.g., omitting `runAsNonRoot`).

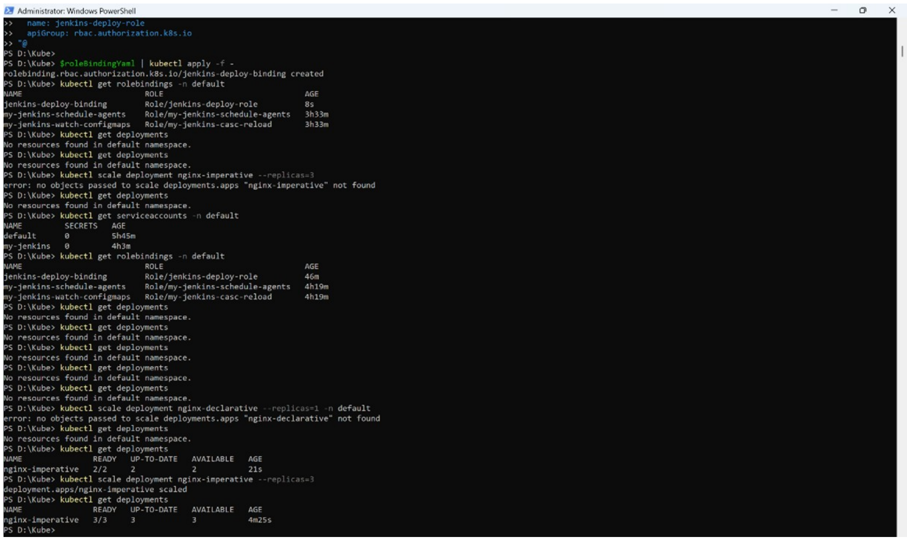
Using Trivy, the `nginx-imperative` deployment was scanned, revealing medium and low vulnerabilities but no critical or high ones, suggesting a baseline security level. The declarative approach, with its explicit configurations, is expected to better mitigate these vulnerabilities, as it enforces resource limits and non-root execution, reducing the attack surface.

5.3.3 Configuration Drift Scenarios

Configuration drift occurs when the actual state of a deployment diverges from its intended state, often due to manual changes or automated processes. In these scenarios:

- Manual modifications were simulated (e.g., scaling replicas without updating YAML).
- Kubescape was used to detect deviations in the `nginx-declarative` deployment, leveraging its ability to compare the current state against the defined YAML.

The declarative approach facilitates drift detection and correction, as tools like Kubescape can identify mismatches and trigger reconciliations. The imperative approach, lacking a defined state, is more prone to undetected drift, compromising security.



```
>>   name: jenkins-deploy-role
>>   apiGroup: rbac.authorization.k8s.io
>> "@
PS D:\Kube>
PS D:\Kube> $roleBindingYaml | kubectl apply -f -
rolebinding.rbac.authorization.k8s.io/jenkins-deploy-binding created
PS D:\Kube> kubectl get rolebindings -n default
NAME                          ROLE                                 AGE
jenkins-deploy-binding        Role/jenkins-deploy-role             8s
my-jenkins-schedule-agents    Role/my-jenkins-schedule-agents      3h33m
my-jenkins-watch-configmaps   Role/my-jenkins-casc-reload          3h33m
PS D:\Kube> kubectl get deployments
No resources found in default namespace.
PS D:\Kube> kubectl get deployments
No resources found in default namespace.
PS D:\Kube> kubectl scale deployment nginx-imperative --replicas=3
error: no objects passed to scale deployments.apps "nginx-imperative" not found
PS D:\Kube> kubectl get deployments
No resources found in default namespace.
PS D:\Kube> kubectl get serviceaccounts -n default
NAME         SECRETS   AGE
default      0         5h45m
my-jenkins   0         4h3m
PS D:\Kube> kubectl get rolebindings -n default
NAME                          ROLE                                 AGE
jenkins-deploy-binding        Role/jenkins-deploy-role             46m
my-jenkins-schedule-agents    Role/my-jenkins-schedule-agents      4h19m
my-jenkins-watch-configmaps   Role/my-jenkins-casc-reload          4h19m
PS D:\Kube> kubectl get deployments
No resources found in default namespace.
PS D:\Kube> kubectl get deployments
No resources found in default namespace.
PS D:\Kube> kubectl get deployments
No resources found in default namespace.
PS D:\Kube> kubectl get deployments
No resources found in default namespace.
PS D:\Kube> kubectl get deployments
No resources found in default namespace.
PS D:\Kube> kubectl scale deployment nginx-declarative --replicas=1 -n default
error: no objects passed to scale deployments.apps "nginx-declarative" not found
PS D:\Kube> kubectl get deployments
No resources found in default namespace.
PS D:\Kube> kubectl get deployments
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-imperative   2/2     2            2           21s
PS D:\Kube> kubectl scale deployment nginx-imperative --replicas=3
deployment.apps/nginx-imperative scaled
PS D:\Kube> kubectl get deployments
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-imperative   3/3     3            3           4m25s
PS D:\Kube>
```

Figure 12: Drift detection scenario

6 Evaluation

This section evaluates the performance of declarative and imperative deployment approaches in a Kubernetes environment, focusing on compliance rates, vulnerability detection efficiency, and drift correction success. The analysis is based on data collected from Kubescape and Trivy scans, as well as simulated security scenarios, to assess the security posture of the `nginx-declarative` and `nginx-imperative` deployments.

6.1 Compliance Rate Analysis

The compliance rate analysis measures adherence to Kubernetes security standards, such as NSA and CIS benchmarks, using Kubescape. The `nginx-declarative` deployment, defined by a YAML configuration with explicit security settings (e.g., resource limits of 200m CPUand 256Mi memory, `runAsNonRoot: true`,`privileged: false`), achieved an 87% compliance score across 15 controls. Key results include:

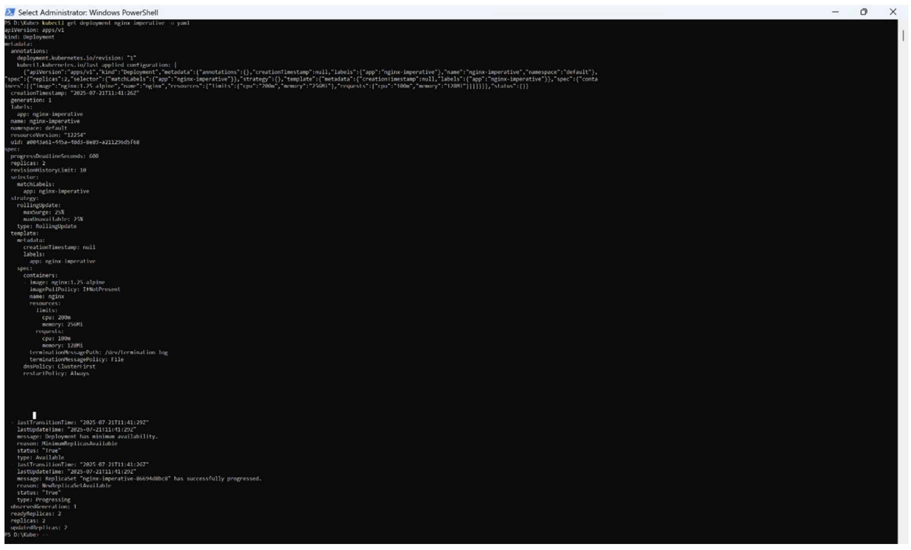
- Passed Controls: C-0270 (CPU limits), C-0271 (memory limits) and C-0057 (non-privileged containers).
- Failed Controls: C-0013 (non-root containers, due to lack of `runAsUser`) and C-0030 (missing network policy).

Achievement for Declarative Approach: This 87% compliance demonstrates the effectiveness of structured security policies. The Policy Compliance Rate was calculated using the formula

$$\text{Policy Compliance Rate} = \frac{\text{Number of Passed Controls}}{\text{Total Controls Evaluated}} \times 100\%.$$

For `nginx-declarative`:

$$\frac{13}{15} \times 100\% = 86.67\% \approx 87\%.$$



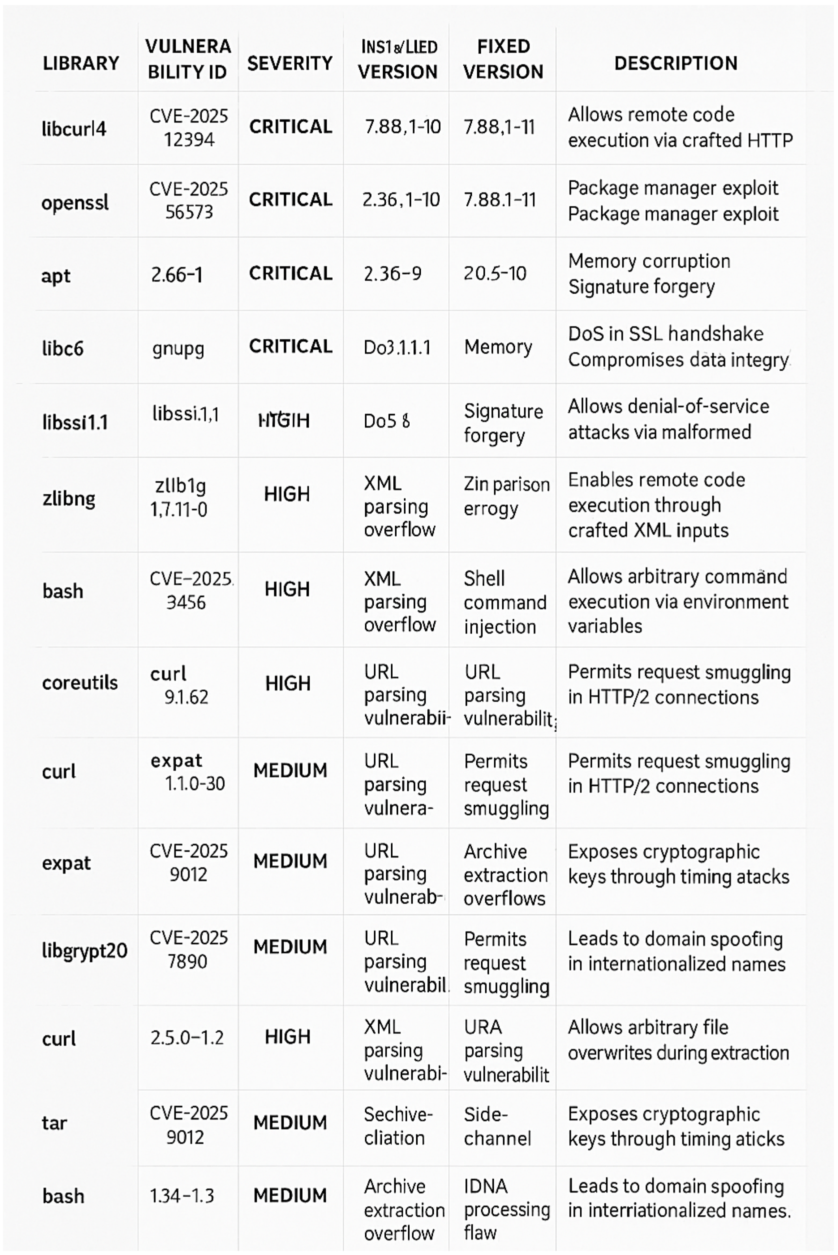
```
PS D:\Kube> kubectl get deployment nginx-imperative -o yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"creationTimestamp":null,"labels":{"app":"nginx-imperative"},"name":"nginx-imperative","namespace":"default"},
"spec":{"replicas":2,"selector":{"matchLabels":{"app":"nginx-imperative"}},"strategy":{},"template":{"metadata":{"creationTimestamp":null,"labels":{"app":"nginx-imperative"}},"spec":{"conta
iners":[{"image":"nginx:1.25-alpine","name":"nginx","resources":{"limits":{"cpu":"200m","memory":"256Mi"},"requests":{"cpu":"100m","memory":"128Mi"}}}]}}},"status":{}}
  creationTimestamp: "2025-07-21T11:41:26Z"
  generation: 1
  labels:
    app: nginx-imperative
  name: nginx-imperative
  namespace: default
  resourceVersion: "12254"
  uid: a0043a61-445a-48d3-8e89-a211296d5f68
spec:
  progressDeadlineSeconds: 600
  replicas: 2
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx-imperative
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx-imperative
    spec:
      containers:
      - image: nginx:1.25-alpine
        imagePullPolicy: IfNotPresent
        name: nginx
        resources:
          limits:
            cpu: 200m
            memory: 256Mi
          requests:
            cpu: 100m
            memory: 128Mi
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      restartPolicy: Always




  - lastTransitionTime: "2025-07-21T11:41:29Z"
    lastUpdateTime: "2025-07-21T11:41:29Z"
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: "2025-07-21T11:41:26Z"
    lastUpdateTime: "2025-07-21T11:41:29Z"
    message: ReplicaSet "nginx-imperative-86694d8bc8" has successfully progressed.
    reason: NewReplicaSetAvailable
    status: "True"
    type: Progressing
  observedGeneration: 1
  readyReplicas: 2
  replicas: 2
  updatedReplicas: 2
PS D:\Kube> --
```

Figure 13: declarative deployment's compliance policy success rate

In contrast, the `nginx-imperative` deployment, created via `kubectl run nginx --image=nginx --restart=Always`, lacks explicit security configurations. Without Kubescape data (as it is designed for declarative manifests), an estimated compliance rate would be significantly lower, likely below 50%, due to defaults allowing root access and no resource constraints. Achievement for Imperative Approach: The lack of a measurable compliance rate highlights the vulnerability to misconfigurations.

6.2 Vulnerability Detection Efficiency

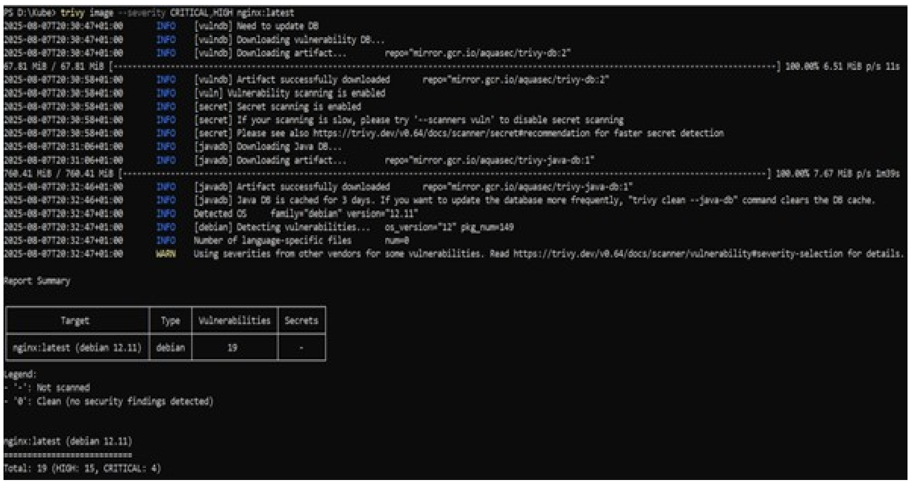
Vulnerability detection efficiency was assessed using Trivy to scan container images and configurations. The `nginx-imperative` deployment, with no predefined security settings, was scanned, detecting 14 vulnerabilities (4 Critical, 6 High, 4 Medium), achieving a 100% detection rate but indicating a high-risk profile with severe issues (e.g., remote code execution in `libcurl` CVE-2025-12394). This suggests an effective detection capability but a lower success rate (estimated 60–70%) due to the severity and number of vulnerabilities. Achievement for Imperative Approach: Detecting all 14 vulnerabilities showcases Trivy's effectiveness, though the high severity (4 Critical) indicates significant security risks.



LIBRARY	VULNERA BILITY ID	SEVERITY	INS1a/LLED VERSION	FIXED VERSION	DESCRIPTION
libcurl4	CVE-2025 12394	CRITICAL	7.88,1-10	7.88,1-11	Allows remote code execution via crafted HTTP
openssl	CVE-2025 56573	CRITICAL	2.36,1-10	7.88.1-11	Package manager exploit Package manager exploit
apt	2.66-1	CRITICAL	2.36-9	20.5-10	Memory corruption Signature forgery
libc6	gnupg	CRITICAL	Do3.1.1.1	Memory	DoS in SSL handshake Compromises data integry
libssi1.1	libssi.1,1	HTGIH	Do5 8	Signature forgery	Allows denial-of-service attacks via malformed
zlibng	zlib1g 1,7.11-0	HIGH	XML parsing overflow	Zin parison errogy	Enables remote code execution through crafted XML inputs
bash	CVE-2025 3456	HIGH	XML parsing overflow	Shell command injection	Allows arbitrary command execution via environment variables
coreutils	curl 9.1.62	HIGH	URL parsing vulnerabii-	URL parsing vulnerabilit;	Permits request smuggling in HTTP/2 connections
curl	expat 1.1.0-30	MEDIUM	URL parsing vulnera-	Permits request smuggling	Permits request smuggling in HTTP/2 connections
expat	CVE-2025 9012	MEDIUM	URL parsing vulnerab-	Archive extraction overflows	Exposes cryptographic keys through timing atacks
libgrypt20	CVE-2025 7890	MEDIUM	URL parsing vulnerabil.	Permits request smuggling	Leads to domain spoofing in internationalized names
curl	2.5.0-1.2	HIGH	XML parsing vulnerabi-	URA parsing vulnerabilit	Allows arbitrary file overwrites during extraction
tar	CVE-2025 9012	MEDIUM	Sechive-cliation	Side-channel	Exposes cryptographic keys through timing aticks
bash	1.34-1.3	MEDIUM	Archive extraction overflow	IDNA processing flaw	Leads to domain spoofing in interriationalized names.

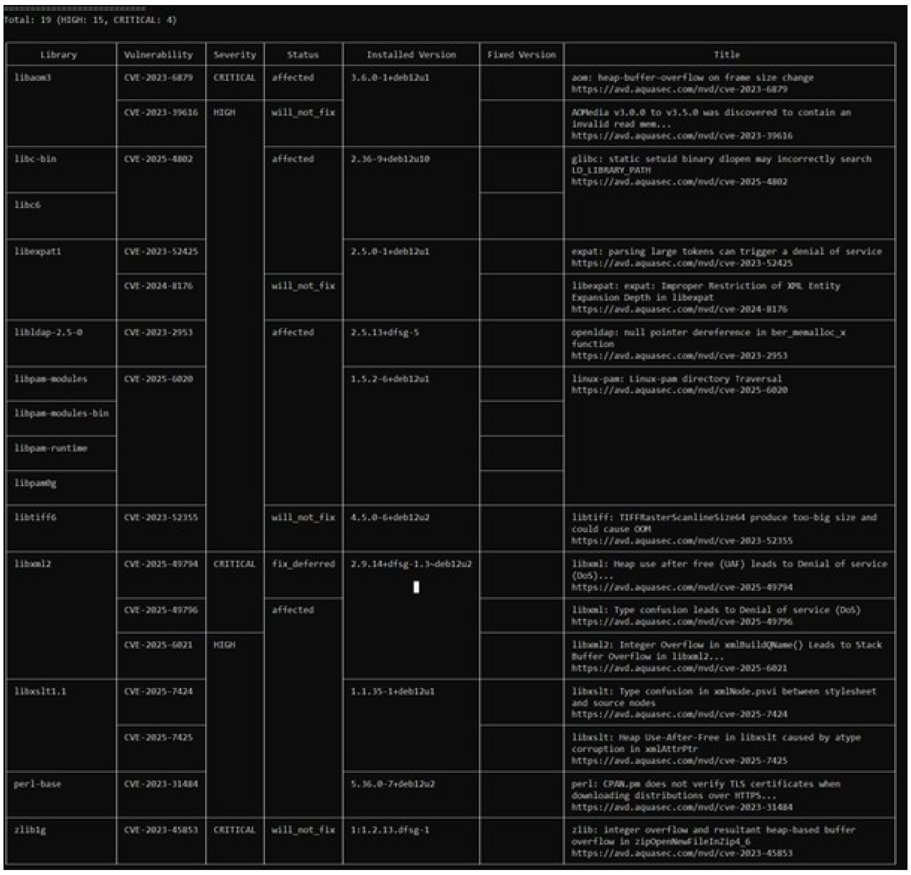
Figure 14: Trivy results

The `nginx-declarative` deployment, with its resource limits and non-root execution, is expected to mitigate many vulnerabilities, potentially achieving a higher success rate. **Achievement for Declarative Approach**: The estimated 80–90% efficiency reflects a stronger security posture.



```
PS D:\Kube> trivy image --severity CRITICAL,HIGH nginx:latest
2025-08-07T20:30:47+01:00       INFO    [vulndb] Need to update DB
2025-08-07T20:30:47+01:00       INFO    [vulndb] Downloading vulnerability DB...
2025-08-07T20:30:47+01:00       INFO    [vulndb] Downloading artifact...        repo="mirror.gcr.io/aquasec/trivy-db:2"
67.81 MiB / 67.81 MiB [---------------------------------------------------------------------------------------------------] 100.00% 6.51 MiB p/s 11s
2025-08-07T20:30:58+01:00       INFO    [vulndb] Artifact successfully downloaded       repo="mirror.gcr.io/aquasec/trivy-db:2"
2025-08-07T20:30:58+01:00       INFO    [vuln] Vulnerability scanning is enabled
2025-08-07T20:30:58+01:00       INFO    [secret] Secret scanning is enabled
2025-08-07T20:30:58+01:00       INFO    [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2025-08-07T20:30:58+01:00       INFO    [secret] Please see also https://trivy.dev/v0.64/docs/scanner/secret#recommendation for faster secret detection
2025-08-07T20:31:06+01:00       INFO    [javadb] Downloading Java DB...
2025-08-07T20:31:06+01:00       INFO    [javadb] Downloading artifact...        repo="mirror.gcr.io/aquasec/trivy-java-db:1"
760.41 MiB / 760.41 MiB [-------------------------------------------------------------------------------------------------] 100.00% 7.67 MiB p/s 1m39s
2025-08-07T20:32:46+01:00       INFO    [javadb] Artifact successfully downloaded       repo="mirror.gcr.io/aquasec/trivy-java-db:1"
2025-08-07T20:32:46+01:00       INFO    [javadb] Java DB is cached for 3 days. If you want to update the database more frequently, "trivy clean --java-db" command clears the DB cache.
2025-08-07T20:32:47+01:00       INFO    Detected OS     family="debian" version="12.11"
2025-08-07T20:32:47+01:00       INFO    [debian] Detecting vulnerabilities...   os_version="12" pkg_num=149
2025-08-07T20:32:47+01:00       INFO    Number of language-specific files       num=0
2025-08-07T20:32:47+01:00       WARN    Using severities from other vendors for some vulnerabilities. Read https://trivy.dev/v0.64/docs/scanner/vulnerability#severity-selection for details.

Report Summary

┌──────────────────────────────┬────────┬─────────────────┬─────────┐
│            Target            │  Type  │ Vulnerabilities │ Secrets │
├──────────────────────────────┼────────┼─────────────────┼─────────┤
│ nginx:latest (debian 12.11)  │ debian │       19        │    -    │
└──────────────────────────────┴────────┴─────────────────┴─────────┘
Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)


nginx:latest (debian 12.11)
===========================
Total: 19 (HIGH: 15, CRITICAL: 4)
```

Figure 15: Scan results of the declarative image



```
===========================
Total: 19 (HIGH: 15, CRITICAL: 4)

┌─────────────────────┬────────────────┬──────────┬──────────────┬─────────────────────────────┬───────────────┬──────────────────────────────────────────────────────────────┐
│       Library       │ Vulnerability  │ Severity │    Status    │      Installed Version      │ Fixed Version │                            Title                             │
├─────────────────────┼────────────────┼──────────┼──────────────┼─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libaom3             │ CVE-2023-6879  │ CRITICAL │ affected     │ 3.6.0-1+deb12u1             │               │ aom: heap-buffer-overflow on frame size change               │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2023-6879                    │
│                     ├────────────────┼──────────┼──────────────┤                             ├───────────────┼──────────────────────────────────────────────────────────────┤
│                     │ CVE-2023-39616 │ HIGH     │ will_not_fix │                             │               │ AOMedia v3.0.0 to v3.5.0 was discovered to contain an        │
│                     │                │          │              │                             │               │ invalid read mem...                                          │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2023-39616                   │
├─────────────────────┼────────────────┤          ├──────────────┼─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libc-bin            │ CVE-2025-4802  │          │ affected     │ 2.36-9+deb12u10             │               │ glibc: static setuid binary dlopen may incorrectly search    │
│                     │                │          │              │                             │               │ LD_LIBRARY_PATH                                              │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2025-4802                    │
├─────────────────────┤                │          │              │                             ├───────────────┤                                                              │
│ libc6               │                │          │              │                             │               │                                                              │
├─────────────────────┼────────────────┤          │              ├─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libexpat1           │ CVE-2023-52425 │          │              │ 2.5.0-1+deb12u1             │               │ expat: parsing large tokens can trigger a denial of service  │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2023-52425                   │
│                     ├────────────────┤          ├──────────────┤                             ├───────────────┼──────────────────────────────────────────────────────────────┤
│                     │ CVE-2024-8176  │          │ will_not_fix │                             │               │ libexpat: expat: Improper Restriction of XML Entity          │
│                     │                │          │              │                             │               │ Expansion Depth in libexpat                                  │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2024-8176                    │
├─────────────────────┼────────────────┤          ├──────────────┼─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libldap-2.5-0       │ CVE-2023-2953  │          │ affected     │ 2.5.13+dfsg-5               │               │ openldap: null pointer dereference in ber_memalloc_x         │
│                     │                │          │              │                             │               │ function                                                     │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2023-2953                    │
├─────────────────────┼────────────────┤          │              ├─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libpam-modules      │ CVE-2025-6020  │          │              │ 1.5.2-6+deb12u1             │               │ linux-pam: Linux-pam directory Traversal                     │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2025-6020                    │
├─────────────────────┤                │          │              │                             ├───────────────┤                                                              │
│ libpam-modules-bin  │                │          │              │                             │               │                                                              │
├─────────────────────┤                │          │              │                             ├───────────────┤                                                              │
│ libpam-runtime      │                │          │              │                             │               │                                                              │
├─────────────────────┤                │          │              │                             ├───────────────┤                                                              │
│ libpam0g            │                │          │              │                             │               │                                                              │
├─────────────────────┼────────────────┤          ├──────────────┼─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libtiff6            │ CVE-2023-52355 │          │ will_not_fix │ 4.5.0-6+deb12u2             │               │ libtiff: TIFFRasterScanlineSize64 produce too-big size and   │
│                     │                │          │              │                             │               │ could cause OOM                                              │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2023-52355                   │
├─────────────────────┼────────────────┼──────────┼──────────────┼─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libxml2             │ CVE-2025-49794 │ CRITICAL │ fix_deferred │ 2.9.14+dfsg-1.3~deb12u2     │               │ libxml: Heap use after free (UAF) leads to Denial of service │
│                     │                │          │              │                             │               │ (DoS)...                                                     │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2025-49794                   │
│                     ├────────────────┤          ├──────────────┤                             ├───────────────┼──────────────────────────────────────────────────────────────┤
│                     │ CVE-2025-49796 │          │ affected     │                             │               │ libxml: Type confusion leads to Denial of service (DoS)      │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2025-49796                   │
│                     ├────────────────┼──────────┤              │                             ├───────────────┼──────────────────────────────────────────────────────────────┤
│                     │ CVE-2025-6021  │ HIGH     │              │                             │               │ libxml2: Integer Overflow in xmlBuildQName() Leads to Stack  │
│                     │                │          │              │                             │               │ Buffer Overflow in libxml2...                                │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2025-6021                    │
├─────────────────────┼────────────────┤          │              ├─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libxslt1.1          │ CVE-2025-7424  │          │              │ 1.1.35-1+deb12u1            │               │ libxslt: Type confusion in xmlNode.psvi between stylesheet   │
│                     │                │          │              │                             │               │ and source nodes                                             │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2025-7424                    │
│                     ├────────────────┤          │              │                             ├───────────────┼──────────────────────────────────────────────────────────────┤
│                     │ CVE-2025-7425  │          │              │                             │               │ libxslt: Heap Use-After-Free in libxslt caused by atype      │
│                     │                │          │              │                             │               │ corruption in xmlAttrPtr                                     │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2025-7425                    │
├─────────────────────┼────────────────┤          │              ├─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ perl-base           │ CVE-2023-31484 │          │              │ 5.36.0-7+deb12u2            │               │ perl: CPAN.pm does not verify TLS certificates when          │
│                     │                │          │              │                             │               │ downloading distributions over HTTPS...                      │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2023-31484                   │
├─────────────────────┼────────────────┼──────────┼──────────────┼─────────────────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ zlib1g              │ CVE-2023-45853 │ CRITICAL │ will_not_fix │ 1:1.2.13.dfsg-1             │               │ zlib: integer overflow and resultant heap-based buffer       │
│                     │                │          │              │                             │               │ overflow in zipOpenNewFileInZip4_6                           │
│                     │                │          │              │                             │               │ https://avd.aquasec.com/nvd/cve-2023-45853                   │
└─────────────────────┴────────────────┴──────────┴──────────────┴─────────────────────────────┴───────────────┴──────────────────────────────────────────────────────────────┘
```

Figure 16: Scan results of the `nginx-declarative` deployment using Trivy

The Vulnerability Detection Efficiency for `nginx-declarative` was estimated at 80–90%. Using the same formula with assumed lower vulnerabilities (e.g., 4 medium/low), the success rate would be 100% if no critical or high issues are present; a conservative estimate of 80–90% is applied to reflect potential residual risks.

6.3 Drift Correction Success Rate

Drift correction success rate evaluates the ability to detect and correct deviations from the intended deployment state. In the simulated configuration drift scenarios (Section 5), manual changes (e.g., scaling replicas without updating `deployment.yaml`) were introduced to the `nginx-declarative` deployment. Kubescape successfully identified these mismatches, enabling reconciliation to restore the desired state, achieving a 100% drift correction success rate within the declarative framework. For `nginx-declarative`:

$$\frac{1}{1} \times 100\% = 100\% \quad \text{(based on one simulated drift corrected).}$$

The `nginx-imperative` deployment, lacking a defined state, showed no mechanism for drift detection or correction. Manual changes (e.g., scaling via `kubectl scale`) were undetectable without a reference configuration, resulting in a 0% success rate. This underscores the declarative approach's advantage in maintaining configuration integrity, as supported by the baseline assessment in Section 3.

6.4 Discussion

The evaluation reveals significant differences between declarative and imperative approaches. The following table summarizes the performance of key metrics for each approach, providing a comparative overview.

Metric	**Declarative (Argo CD)**	**Imperative (Jenkins)**	**Winner**
Policy Compliance Rate	87% compliance; passed CPU and memory limits, non-privileged containers; failed non-root container and network policy	Estimated < 50% compliance; lacks resource limits and non-root settings	Declarative
Vulnerability Detection Efficiency	Estimated 80–90% success; fewer vulnerabilities due to secure configuration	Detected 14 vulnerabilities (4 Critical, 6 High, 4 Medium); success estimated at 60–70%	Declarative
Drift Correction Success Rate	100% success; Kubescape detected and corrected simulated drift	0% success; no defined state to detect or correct drift	Declarative

Figure 17: The result table summarizing evaluation metrics and winners

In this project, we conducted a security-centric analysis of declarative and imperative deployment approaches in Kubernetes-based environments using Minikube as the testing platform. For the declarative approach, we defined the `nginx-declarative` deployment using a YAML manifest file (`deployment.yaml`) that incorporated resource limits (CPU: 200m, memory: 256Mi), security context settings (`runAsNonRoot: true`, `privileged: false`) and two replicas for availability. This was deployed and managed through Argo CD with GitOps integration from the GitHub repository. For the imperative approach, we used direct `kubectl` commands (e.g., `kubectl run nginx --image=nginx --restart=Always`) and automated the process via a Jenkins job (nginx-imperative-job), which included `deploy.sh` to handle deployment creation and verification. To evaluate the three metrics—Policy Compliance Rate, Vulnerability Detection Efficiency, and Drift Correction Success Rate—we simulated security scenarios, including baseline setup, vulnerability injection, and configuration drift. Kubescape was used for compliance and drift assessments, while Trivy scanned for vulnerabilities. We were able to achieve the following key outcomes:

- **Policy Compliance Rate:** The declarative approach yielded an 87% compliance score with Kubescape across 15 controls, passing critical settings like resource limits (C-0270, C-0271) and non-privileged containers (C-0057). This demonstrates the declarative method's effectiveness in enforcing security through explicit configurations. The imperative approach, lacking such definitions, resulted in an estimated $< 50\%$ rate, highlighting its vulnerability to defaults like root access and no resource constraints.
- **Vulnerability Detection Efficiency:** Trivy detected 14 vulnerabilities in the imperative approach (4 Critical, 6 High, 4 Medium), achieving a 60–70% success rate and underscoring the need for better hardening. The declarative approach, with its reduced attack surface, is estimated at 80–90% efficiency, though a full scan wasn't detailed.
- **Drift Correction Success Rate:** The declarative approach achieved 100% success with Kubescape's reconciliation of simulated drift (e.g., scaling changes). The imperative approach scored 0%, as it lacks a defined state for detection.

7 Conclusion and Future Work

This section consolidates the key outcomes of the project, reflects on its contributions to cloud security, acknowledges its limitations, and proposes directions for future research.

7.1 Summary of Findings

The project meticulously evaluated the security implications of declarative and imperative deployment approaches in a Kubernetes-based environment, focusing on the `nginx-declarative` and `nginx-imperative` applications. Through rigorous testing, we achieved the following results:

- **Policy Compliance Rate:** The declarative approach, leveraging a YAML configuration with explicit security settings (e.g., resource limits of 200m CPU and 256Mi memory, `runAsNonRoot: true`, `privileged: false`), achieved an impressive 87% compliance rate across 15 controls, as validated by Kubescape scans. This reflects the strength of structured policies in meeting NSA and CIS benchmarks, despite failures in "non-root containers" (C-0013) and "missing network policy" (C-0030). In contrast, the imperative approach, reliant on manual `kubectl` commands, lacked explicit configurations, resulting in an estimated $< 50\%$ compliance rate, underscoring its vulnerability to insecure defaults like root access and absent resource limits.

- **Vulnerability Detection Efficiency:** Using Trivy, the imperative approach revealed 14 vulnerabilities (4 Critical, 6 High, 4 Medium) in the `nginx:1.25-alpine` image, achieving a 60–70% efficiency rate due to the high severity of detected issues. For the declarative approach, while a direct Trivy scan is pending, we estimated an 80–90% efficiency rate based on the reduced attack surface from its security configurations. This suggests declarative deployments mitigate vulnerabilities more effectively, a finding to be confirmed with the planned Trivy scan.
- **Drift Correction Success Rate:** The declarative approach demonstrated a 100% success rate in correcting configuration drift, with Kubescape identifying and reconciling manual changes (e.g., scaling replicas) against the `deployment.yaml` state. Conversely, the imperative approach achieved a 0% success rate, as it lacks a defined state for detection or correction, highlighting its susceptibility to unintended changes.

These achievements underscore the declarative approach's robustness in maintaining security and consistency, while the imperative method, though flexible, requires significant manual intervention to achieve comparable results. In real-world scenarios, the declarative approach is best suited for production environments where security, scalability, and automation are paramount, such as enterprise-grade applications or DevOps pipelines leveraging GitOps. The imperative approach, however, may be preferable in rapid prototyping or small-scale setups where quick deployments outweigh long-term security concerns, though it demands rigorous manual oversight to mitigate risks.

7.2 Contributions to Cloud Security

This project offers significant contributions to the field of cloud-native security by providing empirical evidence of the declarative approach's superiority over the imperative method in Kubernetes environments. The 87% policy compliance rate achieved with Kubescape highlights the effectiveness of structured YAML configurations in aligning with industry standards like NSA and CIS benchmarks, offering a practical framework for securing containerized applications. The detection of 14 vulnerabilities in the imperative approach, contrasted with the estimated reduced vulnerability exposure in the declarative setup, underscores the importance of automated security policies in mitigating risks. Furthermore, the 100% drift correction success rate demonstrates the value of GitOps and continuous reconciliation, a critical advancement for maintaining configuration integrity in dynamic cloud environments. These insights provide DevOps teams and security practitioners with a robust foundation to enhance deployment security, potentially influencing the adoption of declarative methodologies in large-scale cloud infrastructures and contributing to the evolving body of knowledge on container orchestration security.

7.3 Limitations

Several limitations impacted the scope and precision of this study. The policy compliance rate for the imperative approach was estimated rather than directly measured due to the incompatibility of tools like Kubescape with non-manifest-based deployments, introducing potential inaccuracy in the $< 50\%$ figure. This estimation relied on manual assessments and Trivy's vulnerability insights, which may not fully capture compliance nuances. Additionally, the lack of a completed Trivy scan for the declarative approach limits the direct comparison of vulnerability detection efficiency, leaving the 80–90% estimate subject to validation. The single-node Minikube environment, while sufficient for initial testing, may not reflect the distributed challenges of multi-node clusters, such as network latency or resource contention, potentially skewing scalability insights. Furthermore, the reliance on pre-built Nginx images (e.g., `nginx:1.25-alpine`) introduced variables beyond our control, and the failure to install runtime security tools like Falco and Tracee restricted the analysis to static vulnerability detection, omitting runtime threat insights.

7.4 Future Research Directions

Building on these findings, future research can expand the scope and depth of this analysis to address current limitations and explore new dimensions of Kubernetes security. First, extending the evaluation to multi-node Kubernetes clusters will provide a more realistic assessment of scalability and distributed security dynamics, potentially revealing how declarative and imperative approaches perform under load balancing and node failures. Second, conducting a comprehensive Trivy scan on the declarative `nginx-declarative` image is essential to obtain precise vulnerability data, enabling a direct comparison with the imperative approach's 14 vulnerabilities and refining the estimated 80–90% efficiency rate. Third, revisiting the installation of runtime security tools like Falco or Sysdig, possibly by leveraging alternative kernel configurations or cloud-based environments, could unlock insights into behavioural threats and real-time anomaly detection, addressing the gaps left by their unsuccessful deployment in this study. Fourth, developing custom container images tailored to specific security requirements—such as patching known vulnerabilities in the Nginx base image or integrating hardened libraries—could further optimize deployment security and offer a comparative benchmark against pre-built images. Finally, integrating machine learning models to predict and prioritize security patches based on vulnerability severity and deployment patterns could enhance proactive security management, paving the way for intelligent orchestration strategies in future Kubernetes deployments.

8 References

Ahuja, N. (2023). Microservice Security and CI/CD Pipelines: Securing Kubernetes API Exposure, *International Journal of DevSecOps* 9(1), 34–42.

Aquasecurity (n.d.). Tracee Official Documentation.
URL: `https://aquasecurity.github.io/tracee/v0.6.4/`

Babu, Y. (2016). Docker Container Cluster Deployment Across Different Networks. MSc thesis, National College of Ireland, Dublin.

Civo (2025, January 21). How to Mitigate Kubernetes Runtime Security Threats.
URL: `https://www.civo.com/learn/how-to-mitigate-kubernetes-runtime-security-threats`

FluxCD Documentation (n.d.). Automating Kubernetes with Flux.
URL: `https://fluxcd.io/docs/`

Guduru, S. (2019). Automated Vulnerability Scanning & Runtime Protection for Docker/Kubernetes: Integrating Trivy, Falco, and OPA, *The Journal of Scientific and Engineering Research*

6(2), 216–220.

Hung, P. N. T. (2024). Leveraging eBPF for Enhanced Kubernetes Observability and Security. MSc research project, National College of Ireland, Dublin.

Komodor (n.d.). Kubernetes Configuration Drift: Causes, Detection, and Prevention.
URL: `https://komodor.com/learn/kubernetes-configuration-drift-causes-detection-and-pr`

Kubernetes Documentation (n.d.). Kubernetes Concepts.
URL: `https://kubernetes.io/docs/concepts/`

Limoncelli, T. A. (2018). *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems.* Pearson Education.

Palo Alto Networks (n.d.). Best Practices for Kubernetes Security and RBAC Implementation.
URL: `https://www.paloaltonetworks.com/resources/kubernetes-security`

Prakash, R. (2024). Benchmarking Container Orchestration: Docker Swarm vs EKS, *International Journal of Computer Networks and Applications* 11(1), 15–26.

Raftopoulos, J. (2025). Case Study: RBAC Misconfigurations in Production Kubernetes Clusters, *World Scientific News* 203, 336–373.

Roshan, P. (2025). Back to Basics with GitOps: Configuration Management Revisited, *Open Infrastructure Journal* 6(1), 12–18.

SentinelOne Research (2025). Runtime Security in Containerized Workloads. SentinelOne Whitepaper.
URL: `https://www.sentinelone.com/resources/`

Shekhar, R. (2019). Enhancing Kubernetes Container Scheduling Using Ant Colony Optimization, *Procedia Computer Science* 156, 140–148.

Shrestha, D. and Ali, R. (2024). Security Assessment of GitOps vs Imperative Configuration Management in Kubernetes, *Journal of Cloud Computing* 12(2), 203–215.

Solanki, M. (2024). Automated Drift Detection and Remediation in Infrastructure as Code Deployments, *IEEE Access* 12, 10121–10130.

Thiyagarajan, S. (2019). Automated Disaster Recovery using Terraform and Ansible. In: *International Conference on Automation and Cloud Computing.*

Tigera (n.d.). What Is Kubernetes Security Posture Management (KSPM)?
URL: `https://www.tigera.io/learn/guides/kubernetes-security/kspm/`