Project Report

Solving Kubernetes Configuration Drift with a Secure GitOps Pipeline

Author:Vasu Desai (22070122050), Dhaval Bhimani(22070122051), Dhruv Patel(22070122054), Harsh Kharwar(22070122074)

Date: October 3, 2025

Course: DevOps

Table of Contents

Abstract

Traditional Kubernetes management, often reliant on manual kubectl commands, is highly susceptible to configuration drift—a state where the live cluster configuration diverges from the intended state stored in version control. This divergence introduces significant risks, including unreliable deployments, failed rollbacks, and security vulnerabilities.

This project addresses this critical challenge by designing and implementing a secure, fully automated CI/CD pipeline on Amazon Web Services (AWS). The core of the solution is a GitOps workflow orchestrated by Argo CD, which establishes a Git repository as the single source of truth for the application's desired state. This ensures that the Kubernetes cluster is always in sync, eliminating configuration drift.

To further enhance the pipeline's integrity, a robust Continuous Integration (CI) process was implemented using Jenkins. This process integrates "shift-left" security principles by enforcing a SonarQube Quality Gate, ensuring that only high-quality, secure code is containerized and pushed to the Amazon Elastic Container Registry (ECR). The deployed application's health and performance are continuously monitored using Prometheus and Grafana, providing complete observability over the system.

The result is a resilient, auditable, and efficient application delivery lifecycle that significantly improves deployment reliability and security posture.

1. Introduction

1.1. Background

In modern software development, Kubernetes has become the de facto standard for container orchestration. While it offers immense power and scalability, managing its declarative configurations presents significant operational challenges. The imperative nature of tools like kubectl allows for direct, on-the-fly changes to a live cluster, a practice that often leads to inconsistencies.

1.2. Problem Statement

Manually managing Kubernetes configurations with kubectl often leads to **configuration**

drift, where the live cluster's state no longer matches the version-controlled files. This divergence is typically caused by urgent hotfixes, manual debugging, or inconsistent team practices. The consequences are severe:

- **Unreliable Deployments**: Pushing a new update can have unintended side effects because the starting state of the cluster is unknown.
- **Failed Rollbacks**: Reverting to a "last known good" version in Git may not work if out-of-band changes were made to the cluster.
- **Lack of Auditability**: It becomes impossible to track who changed what, when, and why.
- **Security Risks**: Manual changes can inadvertently expose security vulnerabilities that are not captured in version control.

1.3. Project Scope and Objectives

The primary objective of this project is to design and implement a CI/CD pipeline that solves the problem of configuration drift by adopting a GitOps methodology.

The key objectives are:

1. **Automate the entire build-to-deploy lifecycle** using Jenkins.
2. **Integrate automated security checks** using SonarQube and Trivy to "shift left".
3. **Implement a GitOps workflow using Argo CD** to make Git the single source of truth.
4. **Deploy a sample application** (an Amazon Prime Video clone) to a managed Kubernetes cluster on AWS EKS.
5. **Establish a monitoring stack** using Prometheus and Grafana for real-time observability.

1.4. Report Structure

This report details the design, implementation, and results of the project. It begins with the overall system architecture, followed by a detailed breakdown of the infrastructure setup, the CI pipeline, the CD pipeline, and the monitoring stack. It concludes with an analysis of the results and suggestions for future work.

2. System Architecture and Design

2.1. High-Level Architecture

As shown in the figure below, the architecture is designed as a cohesive, event-driven workflow that begins with a code commit and ends with a monitored application running in Kubernetes.

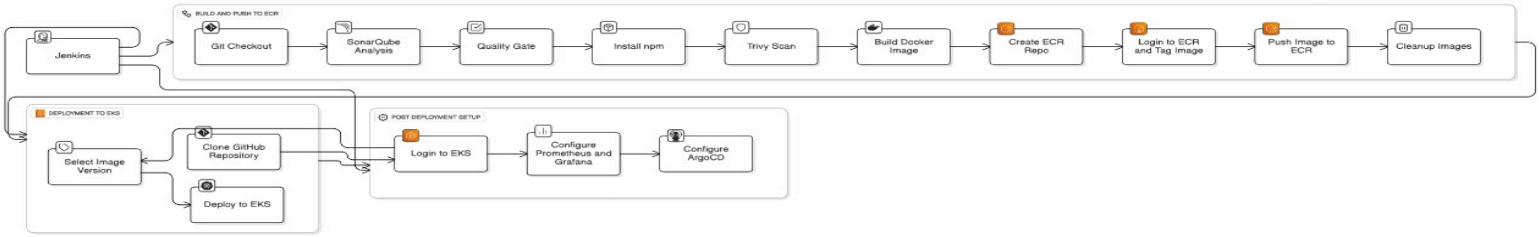

Figure 1: High-Level Workflow Diagram

The workflow is divided into three main phases:

1. **CI (Continuous Integration)**: Triggered by a git push, Jenkins fetches the code, runs security and quality checks, builds a container image, and pushes it to a registry.
2. **CD (Continuous Deployment)**: Argo CD detects the new image tag (or any other configuration change) in the Git repository and automatically syncs the Kubernetes cluster to match this new desired state.
3. **Monitoring**: Prometheus continuously scrapes metrics from the cluster, which are then visualized in Grafana.

2.2. Core Concepts: GitOps

GitOps is the foundational principle of this project's deployment strategy. It is a paradigm for managing infrastructure and applications where Git is the **single source of truth**. In this model:

- The desired state of the entire system is declared and version-controlled in Git.
- An automated agent (Argo CD) ensures that the live environment converges to the desired state described in Git.
- All changes are made via pull requests, providing a clear, auditable log of modifications.

2.3. Tools and Technologies

The following tools were selected to build the pipeline:

Tool	Category	Purpose
AWS EKS	Orchestration	Managed Kubernetes Service.
AWS EC2	Compute	Virtual servers for Jenkins and K8s nodes.
AWS ECR	Registry	Secure storage for Docker

		container images.
Jenkins	CI Orchestrator	Manages the entire build and test pipeline.
Docker	Containerization	Packages the application into a portable image.
SonarQube	Code Quality	Static analysis and security scanning.
Trivy	Security	Scans container images for vulnerabilities.
Argo CD	CD Orchestrator	Implements the GitOps workflow.
Prometheus	Monitoring	Collects time-series metrics from the cluster.
Grafana	Visualization	Creates dashboards from Prometheus data.

3. Infrastructure Setup on AWS

3.1. Overview of Cloud Infrastructure

All components of this project are hosted on Amazon Web Services, providing a scalable and reliable foundation. The infrastructure is logically separated into an orchestration layer, a compute layer, and a storage layer.

3.2. Orchestration Layer: Amazon EKS

The core of the deployment environment is an Amazon Elastic Kubernetes Service (EKS) cluster named amazon-prime-cluster. EKS manages the Kubernetes control plane, automating updates and patching, which simplifies cluster management.

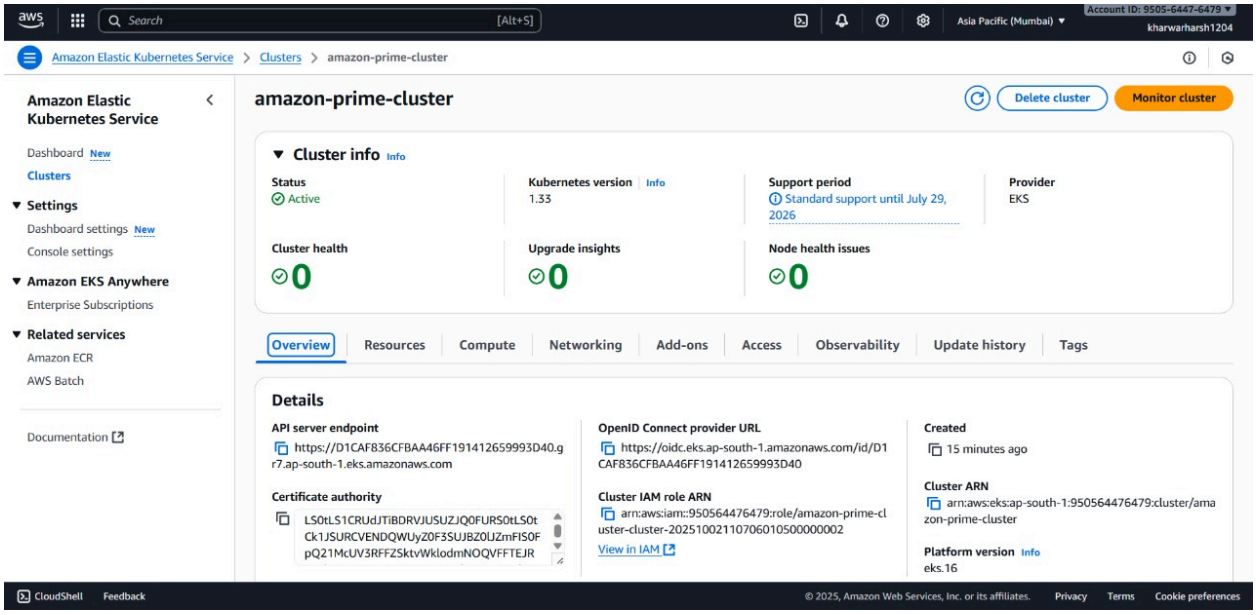

Figure 2: EKS Cluster Details

3.3. Compute Layer: Amazon EC2

The EKS cluster utilizes Amazon Elastic Compute Cloud (EC2) instances as worker nodes. These nodes are where the application pods are scheduled and run.

- **Jenkins Server**: A dedicated t2.large EC2 instance runs the Jenkins automation server.
- **EKS Node Group**: A group of c7i-flex.large EC2 instances serves as the worker nodes for the Kubernetes cluster.

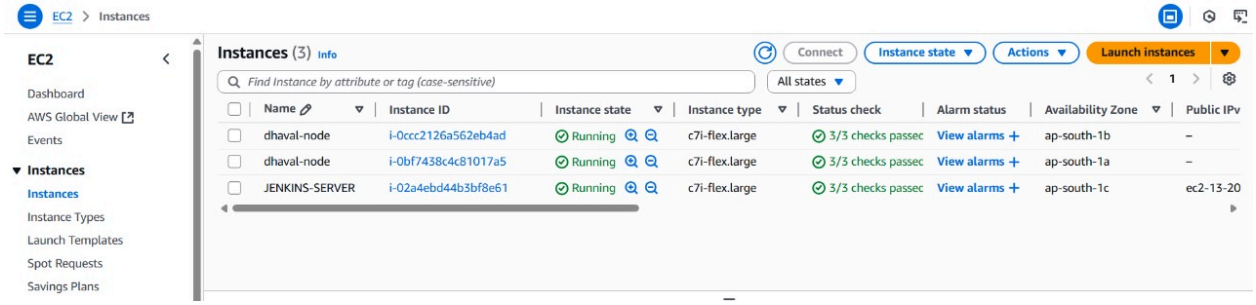

Figure 3: EC2 Instance Configuration

3.4. Security and Networking

- **Security Groups**: AWS Security Groups are configured to act as virtual firewalls, allowing traffic only on necessary ports (e.g., 8080 for Jenkins, 9000 for SonarQube, and 22 for SSH).
- **IAM Roles**: AWS Identity and Access Management (IAM) roles are used to grant the Jenkins pipeline secure, keyless access to other AWS services like ECR.

4. Continuous Integration (CI) Pipeline Implementation

4.1. Orchestration with Jenkins

The CI pipeline is defined as a Jenkinsfile and orchestrates all the steps required to prepare a secure, deployable artifact.

Figure 4: Jenkins Build Pipeline Stages

4.2. Code Quality and Security with SonarQube

Upon code checkout, the pipeline triggers a SonarQube analysis. This performs static analysis to detect bugs, code smells, and security vulnerabilities. A critical component is the **Quality Gate**, a set of conditions that the code must meet. If the Quality Gate fails, the pipeline is immediately stopped, preventing low-quality code from progressing.

4.3. Vulnerability Scanning with Trivy

After dependencies are installed, Trivy performs a vulnerability scan on the application's packages. This "shift-left" approach ensures that known vulnerabilities are caught early in the development cycle, long before they can reach production.

4.4. Containerization and Storage with Docker & ECR

If all quality and security checks pass, the pipeline proceeds to:

1. **Build a Docker Image**: Packages the application and its dependencies into a standardized Docker image.
2. **Tag the Image**: The image is tagged with a unique identifier (e.g., the build number).
3. **Push to ECR**: The tagged image is pushed to a private Amazon Elastic Container Registry (ECR) repository, where it is securely stored and versioned.

5. Continuous Deployment (CD) Pipeline Implementation

5.1. Git as the Single Source of Truth

The deployment process is governed by a separate Git repository that contains the Kubernetes manifest files (e.g., deployment.yaml, service.yaml). This repository declaratively defines the desired state of the application, including the Docker image tag to be used.

5.2. Argo CD: The GitOps Reconciler

Argo CD is the engine that drives the GitOps workflow. It is deployed inside the EKS cluster and configured to monitor the manifest repository. Its sole job is to ensure that the live state of the cluster matches the state defined in Git.

5.3. The Deployment and Sync Process

When a developer wants to deploy a new version, they do not interact with the cluster directly. Instead, they update the image tag in the deployment.yaml file in the Git repository.

1. Argo CD detects this change in Git.
2. It compares the new desired state with the current live state and identifies the difference (a new image tag).
3. It automatically applies the change to the cluster, triggering a rolling update of the application pods.
   The Argo CD dashboard below shows the application in a Synced and Healthy state, confirming that configuration drift has been eliminated.

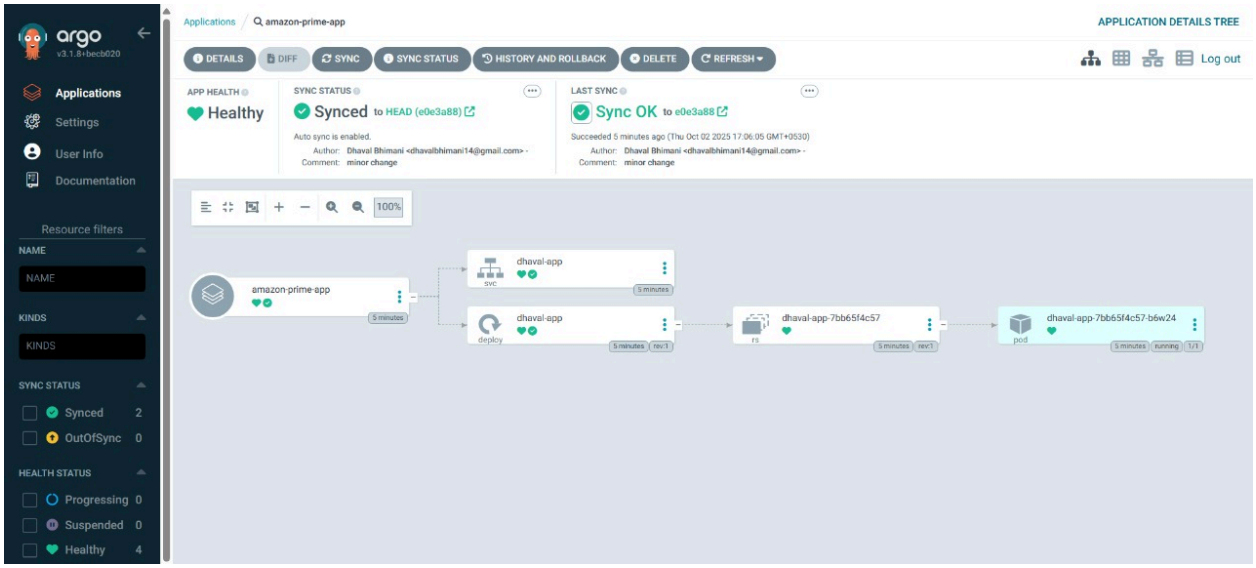

Figure 5: Argo CD Dashboard Showing a Synced Application

6. Monitoring and Observability

6.1. The Prometheus and Grafana Stack

To ensure the deployed application is running optimally, a monitoring stack was implemented using Prometheus and Grafana.

- **Prometheus**: Deployed in the cluster, it automatically discovers and scrapes metrics from pods and nodes.
- **Grafana**: Connects to Prometheus as a data source and provides a powerful interface for building visualization dashboards.

6.2. Key Metrics for Observability

The configured dashboard tracks critical infrastructure metrics in real-time, including:

- CPU Usage and Load
- Memory Consumption
- Disk Space Usage
- Network Traffic

This data is essential for debugging, performance tuning, and capacity planning.

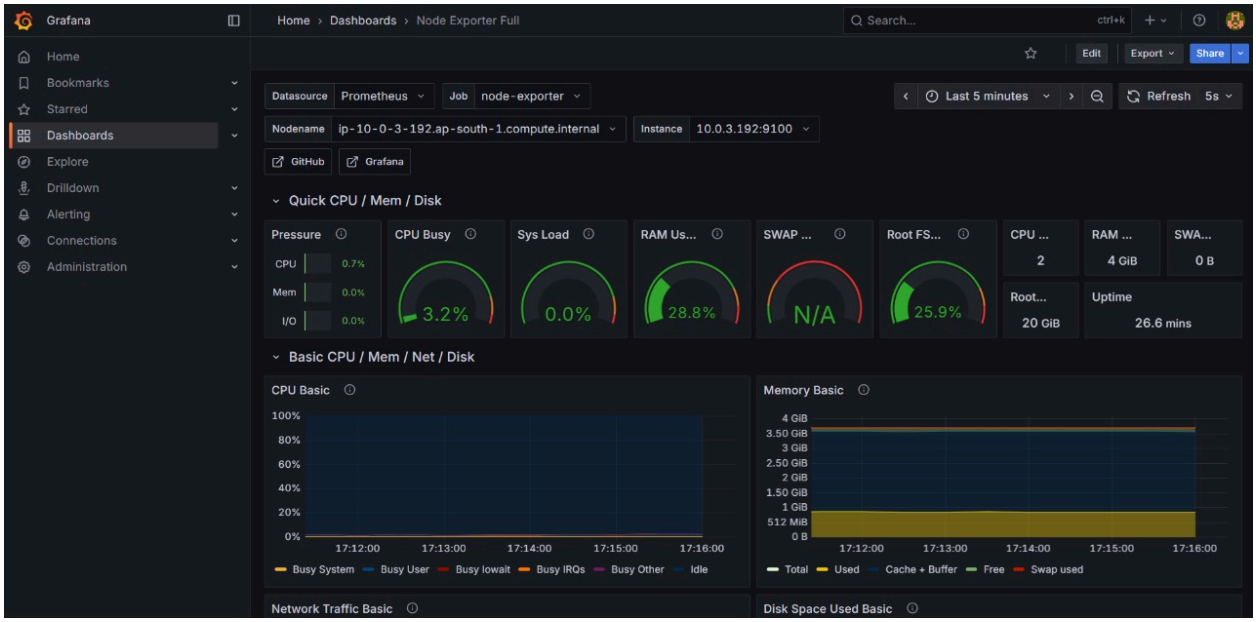

Figure 6: Grafana Dashboard Monitoring a Cluster Node

7. Results and Discussion

7.1. Elimination of Configuration Drift

The primary goal of the project was successfully achieved. By implementing a GitOps workflow with Argo CD, the possibility of configuration drift is virtually eliminated. All changes are auditable through Git history, and the cluster state is self-healing, as Argo CD will automatically revert any manual, out-of-band changes.

7.2. Enhanced Security and Code Quality

The integration of SonarQube and Trivy into the CI pipeline provides a robust security gate. The automated Quality Gate ensures that code with known vulnerabilities or quality issues is never allowed to be containerized or deployed, significantly improving the application's security posture.

7.3. Improved Deployment Velocity and Reliability

The fully automated pipeline dramatically reduces the time and effort required to deploy new versions. What was once a manual, error-prone process is now a streamlined, reliable workflow. This allows developers to release features faster and with greater confidence.

8. Conclusion and Future Work

8.1. Summary of Achievements

This project successfully designed and implemented an end-to-end, secure CI/CD pipeline

that solves the critical problem of Kubernetes configuration drift. By leveraging modern DevOps principles like GitOps and "shift-left" security, the solution provides a reliable, auditable, and efficient framework for cloud-native application delivery.

8.2. Future Enhancements

While the current pipeline is robust, it can be extended with several advanced features:

- **Automated Testing**: Integrating unit, integration, and end-to-end tests into the Jenkins pipeline.
- **Canary or Blue-Green Deployments**: Using a tool like Argo Rollouts to implement progressive delivery strategies and minimize deployment risk..
- **Secret Management**: Integrating a dedicated secrets management tool like HashiCorp Vault or AWS Secrets Manager.
- **Cost Optimization**: Implementing auto-scaling policies for the EKS node group based on Prometheus metrics.

9. References

1. Argo CD Documentation. (2025). *Argo CD - Declarative GitOps CD for Kubernetes*. Retrieved from https://argoproj.github.io/argo-cd/
2. Stack Overflow. (2018). *Best practices for storing and using lots of Kubernetes YAML files?* Retrieved from https://www.google.com/search?q=https://stackoverflow.com/questions/49211029/best-practices-for-storing-and-using-lots-of-kubernetes-yaml-files
3. Atlassian. *What is GitOps?* Retrieved from https://www.google.com/search?q=https://www.atlassian.com/gitops
4. AWS Documentation. *What is Amazon EKS?* Retrieved from https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html