Automated Secure Software Delivery Using GitOps on AWS EKS

Built using Git, Jenkins, Maven, SonarQube, Docker, Trivy, ArgoCD & Kubernetes(EKS)

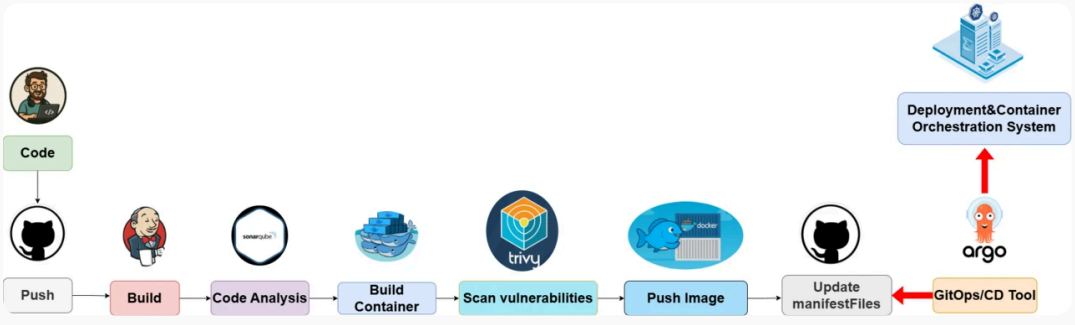
---

1. Project Introduction

This document describes a complete DevOps project implementing a CI/CD (Continuous Integration/Continuous Delivery) pipeline for a containerized application. The pipeline automates the process of building, testing, and deploying a Java web application, with a focus on modern DevOps practices such as GitOps and containerization. Key components include a source code repository, a CI server, build automation, static code analysis, containerization, image scanning, a container registry, a GitOps deployment manager, and a Kubernetes cluster for orchestration. The project leverages open-source tools like Jenkins, Maven, SonarQube, Docker, Trivy, Argo CD, and Kubernetes to achieve end to-end automation and reliability in the software delivery lifecycle.

2. Pipeline Architecture Overview

The entire project is based on the following CI/CD architecture, which illustrates the flow of an application from a developer's commit to a running container in a Kubernetes cluster. Each stage is handled by a specialized tool, creating a powerful and cohesive toolchain.



This diagram illustrates a CI/CD pipeline for containerized applications. The process begins with "Source Code" stored in "GitHub," which is then processed by a "CI/CD Server" like "Jenkins." The code goes through a "Build Stage" using "Maven," followed by "Static Code Analysis" with tools like "SonarQube." Next, the code is used to "Build & scan (Trivy) docker image." The built image is then pushed to "Docker Hub." Subsequently, "CD manifest files repo for Argo CD" are updated. Finally, a "GitOps/CD Tool" like "Argo CD" is used for "Container App Deployment & Orchestration" on "Kubernetes."

Core Components and Workflow:

1. Source Code Management: Developers push code to a GitHub repository.
2. CI Trigger: A webhook from GitHub triggers a pipeline job on the Jenkins CI/CD server.
3. Build & Analyze: Jenkins checks out the code, compiles it using Maven, and runs a static code analysis scan with SonarQube.
4. Containerize & Scan: A Docker image is built from the application code. This image is then scanned for known vulnerabilities using Trivy.
5. Store Artifact: If the scans pass, the validated Docker image is pushed to a container registry, such as Docker Hub.
6. GitOps Trigger: The CI pipeline automatically updates a Kubernetes manifest file in a separate Git repository (the "Config Repo"), pointing to the new Docker image tag.
7. Deploy & Orchestrate: Argo CD, a GitOps tool, detects the change in the Config Repo and automatically synchronizes the state of the Kubernetes cluster to match the desired state defined in the manifests, deploying the new version of the application.

3. Phase 1: Prerequisites and Environment Setup

Before beginning the implementation, it is essential to set up the necessary accounts, infrastructure, and tools. This foundational phase ensures that all components of the pipeline can communicate and operate correctly.

3.1. Required Accounts and Services

- GitHub Account: To host the application source code and the Kubernetes configuration repositories. A free account is sufficient for public repositories.
- Docker Hub Account: To store and distribute the container images. A free account provides public repositories.
- Cloud Provider / Kubernetes Environment: Access to a Kubernetes cluster. Options include:
  - Local Development: gitbash or visual studio for local testing.
  - Cloud-Managed: Amazon Elastic Kubernetes Service (EKS) for more robust, production-like environments.

3.2. Infrastructure and Tool Installation

A Linux server for Jenkins , Docker, Sonarqube setup with an instance type "t2.Xlarge" and a EKS Cluster to deploy container application.

3.2.1. Jenkins Server Setup

```
#Install jenkins on your linux system
sudo yum update –y
sudo wget -O /etc/yum.repos.d/jenkins.repo \
    https://pkg.jenkins.io/redhat-stable/jenkins.repo
sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io-2023.key
sudo yum upgrade
sudo yum install java-17-amazon-corretto -y
sudo yum install jenkins -y
sudo systemctl enable jenkins
sudo systemctl start jenkins
sudo systemctl status jenkins
```

3.2.2. Installation & configuration of docker

```
#Install Docker and configure it to work with Jenkins
sudo yum install docker -y
sudo systemctl start docker
sudo usermod -aG docker jenkins
sudo usermod -aG docker ec2-user
```

```
sudo systemctl restart docker
sudo chmod 666 /var/run/docker.sock
```

3.2.3. AWS CLI configuration

```
#Configure AWS CLI to interact with AWS services
aws configure
# Enter your AWS
Access Key ID, Secret Access Key, region, and output format
when prompted
```

3.2.4. Kubectl and EKSCTL installation

```
#Install kubectl and eksctl to manage kubernetes cluster:

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/
release/stable.txt)/bin/linux/amd64/kubectl"

curl-LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io
/release/stable.txt)/bin/linux/amd64/kubectl.sha256"

echo "$(cat kubectl.sha256) kubectl" | sha256sum --check

sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

# Install eksctl

curl --silent --location
"https://github.com/weaveworks/eksctl/releases/latest/download
/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp

sudo mv /tmp/eksctl/usr/local/bin
```

3.2.5. Create EKS Cluster

```
#Create an EKS cluster using eksctl
eksctl create cluster --name myappcluster --nodegroup-name
mygroup --node-type t3.micro --nodes 8 --managed
```

3.2.6. Install required jenkins plugins

Install the Docker plugin in Jenkins:
1. Go to Jenkins Dashboard > Manage Jenkins > Manage Plugins > Available.
2. Install the Docker plugin,Pipeline stage view,blue ocean..etc

3.2.7. Install and configure SonarQube as a docker container

1. Docker run --itd --name sonar -p 9000:9000 sonarqube

2. Check if the SonarQube container is running:

docker ps

3. Access the SonarQube web interface:

- Access SonarQube at `http://<your-server-ip>:9000`. The default login is `admin` / `admin`. You will be prompted to change the password.

- The default login credentials are:

1. Username: admin

2. Password: admin

4. Log in to SonarQube using the default credentials.
- Change the default password

5. Create Sonar token for Jenkins: Sonar Dashboard -> Administration -> MyAccount -> Security -> Create token

3.2.8. Generate Github Token

Generate a GitHub token for Jenkins to access your GIT repositories:

1. Go to GitHub > Settings > Developer settings > Personal access tokens.

2. Generate a new token with the necessary scopes (e.g., repo, admin:repo_hook).

3.2.9. Kubernetes Deployment and Service Files

1. Installation of argocd:

```
# Create a namespace for Argo CD
kubectl create namespace argocd

# Apply the Argo CD installation manifests
```

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj
/argo-cd/stable/manifests/install.yaml
```

2. Edit the Argo CD server service to type LoadBalancer

```
kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}

kubectl get svc argocd-server -n argocd

kubectl -n argocd get secret argocd-initial-admin-secret -o
jsonpath=" {.data.password}" |base64 -d ; echo
```

4. Phase 2: Source Code and Configuration Management

This phase involves setting up the Git repositories that will be the single source of truth for our application and its deployment configuration.

4.1. Application Source Code Repository ("App Repo")

Create a new repository on GitHub for your application. For this project, we'll assume a simple Java application built with Maven.

Recommended Repository Structure:

```
my-app/
├── .github/
│   └── workflows/          # (Optional for GitHub Actions)
├── k8s/                    # Kubernetes manifest templates
│   ├── deployment.yaml
│   └── service.yaml
├── src/
│   ├── main/
│   │   ├── java/
│   │   └── resources/
│   └── test/
├── Dockerfile              # Instructions to build the container image
├── Jenkinsfile             # The CI/CD pipeline definition for Jenkins
├── pom.xml                 # Maven project configuration
└── README.md
```

4.2. Kubernetes Manifest Repository ("Config Repo")

Create a second, separate repository on GitHub. This repository will hold the definitive Kubernetes manifests that Argo CD will use to deploy the application. It is the core of the GitOps workflow.

Recommended Repository Structure:

```
my-app-config/
├── my-app/
│   ├── deployment.yaml
│   ├── service.yaml
└── README.md
```

Create deployment.yaml and service.yaml in your repository to define the Kubernetes resources

deployment.yml:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: devopshubg333/batch13:tag
          ports:
            - containerPort: 8080
```

service.yml:

```
---
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: 8080
    protocol: TCP
selector:
app: my-app
```

> Why two repositories? Separating application code from deployment configuration is a best practice. It provides a clear separation of concerns: developers work in the App Repo, while the CI system and operations team manage the Config Repo. This enhances security and stability.

5. Phase 3: Continuous Integration (CI) Implementation with Jenkins

The CI pipeline is defined as code using a `Jenkinsfile`. This file lives in the root of the App Repo and orchestrates all the steps from code checkout to pushing the container image.

5.1. The Jenkinsfile: Pipeline as Code

Below is a comprehensive, commented `Jenkinsfile` that implements the CI stages shown in the architecture diagram. This file should be created in the root of your App Repo.

```
pipeline {
    agent any

    // Specify the Maven installation configured in Jenkins
    tools {
        maven 'maven3'
```

```
    }

    stages {

        stage('Checkout') {
            steps {
                echo 'Cloning GIT HUB Repo'
                git branch: 'main',
                    url: 'https://github.com/kadivetikavya/
                          mindcircuit13'
            }
        }

        stage('SonarQube Scan') {
            steps {
                echo 'Scanning project'
                sh 'ls -ltr'

                sh '''
                    mvn sonar:sonar \
                    -Dsonar.host.url=http://100.26.227.191:9000 \
                    -Dsonar.login=squ_19733ad4e43d54992ef61923b91447e2d
                '''
            }
        }

        stage('Build Artifact') {
            steps {
                echo 'Build Artifact'
                sh 'mvn clean package'
            }
        }

        stage('Build Docker Image') {
            steps {
                echo 'Build Docker Image'
                sh 'docker build -t kadivetikavya/kadi-repo:
                ${BUILD_NUMBER} .'
            }
        }

        stage('Push to Docker Hub') {
            steps {
                script {
                    withCredentials([
                        string(credentialsId: 'dockerhub',
                        variable: 'dockerhub')
                    ]) {
                        sh 'docker login -u usernamehere -p
                        ${dockerhub}'
```

```
                    }

                    sh 'docker push kadivetikavya/kadi-repo:
                    ${BUILD_NUMBER}'
                    echo 'Pushed to Docker Hub'
                }
            }
        }

        stage('Update Deployment File') {
            environment {
                GIT_REPO_NAME = 'kadivetikavya'
                GIT_USER_NAME = 'kavyakadiveti'
            }

            steps {
                echo 'Update Deployment File'

                withCredentials([
                    string(credentialsId: 'githubtocken',
                    variable: 'githubtoken')
                ]) {
                    sh '''
                        git config user.email "kaxxxx123@gmail.com"
                        git config user.name "kavya"

                        sed -i "s/kadi-repo:.*/kadi-repo:
                        ${BUILD_NUMBER}/g" deploymentfiles/deployment.ym

                        git add .
                        git commit -m "Update deployment image to
                        version ${BUILD_NUMBER}"

                        git push https://${githubtoken}@github.com/
                        ${GIT_USER_NAME}/${GIT_REPO_NAME} HEAD:main
                    '''
                }
            }
        }

    }
}
```

Each stage explanation:

1. Checkout: This stage clones the source code from the GitHub repository. It ensures Jenkins always works with the latest code from the main branch.

2. SonarQube Scan: The source code is analyzed using SonarQube for code quality and security issues. It checks bugs, vulnerabilities, and code smells before the build continues.

3. Build Artifact: Maven compiles the source code and packages it into a deployable artifact (JAR/WAR). This verifies that the application builds successfully without errors.

4. Build Docker Image: A Docker image is created using the Dockerfile present in the project. The image is tagged with the Jenkins build number for versioning.

5. Push to Docker Hub: The Docker image is pushed to Docker Hub using secured credentials. This makes the image available for deployment in any environment.

6. Update Deployment File: The Kubernetes deployment YAML file is updated with the new Docker image tag. Changes are committed and pushed back to GitHub to trigger GitOps-based deployment.

6. Phase 4: Continuous Deployment (CD) with Argo CD

With the CI pipeline pushing configuration changes to the Config Repo, the final step is to configure Argo CD to watch this repository and apply the changes to our Kubernetes cluster.

6.1. Creating the Application in Argo CD

You can define an Argo CD application declaratively with a YAML manifest or through the Argo CD UI.

6.1.1. Using the Argo CD UI

1. Log in to your Argo CD UI.
2. Click on "+ NEW APP".
3. Fill in the application details:
    - Application Name: `my-app`
    - Project: `default`
    - Sync Policy: `Automatic`

- Check Prune Resources and Self Heal for a fully automated GitOps experience.

4. Configure the Source:
    - Repository URL: The URL of your Config Repo (e.g., `https://github.com/your-github-username/my-app-config.git`).
    - Revision: `HEAD` (or your main branch name).
    - Path: The path to the manifests for this specific application (e.g., `my-app`).
5. Configure the Destination:
    - Cluster URL: `https://kubernetes.default.svc` (for the in-cluster deployment).
    - Namespace: The Kubernetes namespace where you want to deploy the app (e.g., `production`). Ensure this namespace exists.
6. Click "CREATE".button at the bottom of the page.
7. Sync the Application

 After the application is created, it will appear in the Argo CD dashboard with a status of OutOfSync.

 To sync the application, click on the application name (my-app).

 Click the Sync button at the top right of the application details page.

 In the sync dialog, review the resources to be synchronized and click Synchronize to start the sync process.

 The status will change to Healthy once the sync is complete and the application is successfully deployed.

> Deployment Complete! Once the Argo CD application is created, it will immediately fetch the manifests from your Config Repo and deploy the application to the specified namespace. From this point on, any commit to the `main` branch of your App Repo will trigger the entire CI/CD pipeline, resulting in a new, validated image being automatically deployed to Kubernetes within minutes.

7. Project Conclusion and Next Steps

This project plan has detailed the end-to-end implementation of a modern, GitOps-driven CI/CD pipeline. By following these phases, an organization can establish a highly automated, secure, and efficient workflow for developing and deploying containerized applications on Kubernetes.

Potential Enhancements and Further Steps:

- Advanced Branching and Environments: Extend the pipeline to support multiple environments (e.g., `staging`, `production`) based on Git branches. Use Kustomize or Helm in the Config Repo to manage environment-specific configurations.
- Secrets Management: Integrate a dedicated secrets management tool like HashiCorp Vault or External Secrets Operator instead of storing sensitive information directly in Jenkins.
- Dynamic Test Environments: Configure the pipeline to automatically spin up ephemeral preview environments for each pull request, allowing for better testing and review before merging to the main branch.
- Monitoring and Observability: Deploy monitoring tools like Prometheus and Grafana, and logging solutions like the EFK stack (Elasticsearch, Fluentd, Kibana) to gain insights into the deployed application's performance and health.
- Cost Optimization: Implement tools like Kubecost to monitor and optimize the resource utilization and cost of the Kubernetes cluster.