**Project: Enterprise Java CI/CD with Jenkins, GitOps, and Kubernetes Observability**

**Ensuring Modern, Automated, and Observable Java Deployments on AWS**

**Submitted By – Heena**

**1. Introduction**

**1.1 Problem Statement**

Modern Java applications require robust, scalable, and observable deployment pipelines. Manual deployments are error-prone, slow, and lack traceability. There is a need for a fully automated, auditable, and observable CI/CD workflow that covers everything from code commit to production, enforces code quality, and enables real-time monitoring.

**1.2 Scope**

This project implements a cloud-native, automated CI/CD pipeline for a Java microservice on AWS. It leverages best-in-class DevOps tools for build, test, analysis, containerization, GitOps-based deployment, and observability. The solution ensures reliability, maintainability, and operational excellence.

**1.3 Objectives**

* Automate the entire build, test, and deployment process using Jenkins, GitHub, and Argo CD.
* Enforce code quality gates with SonarQube.
* Package the application as a Docker container and store it in Amazon ECR.
* Deploy to Amazon EKS using GitOps (Argo CD).
* Enable real-time application and cluster monitoring with Prometheus and Grafana.

**2. Technology Stack: Components & Purpose**

|  |  |  |
| --- | --- | --- |
| **S. No.** | **Component** | **Purpose** |
| 1. | **Java + Maven** | Application code, dependency management, and build tool |
| 2. | **Jenkins** | CI/CD orchestration: automates build, test, analysis, deploy |
| 3. | **SonarQube** | Static code analysis and quality gates |
| 4. | **Docker** | Containerization of the application |
| 5. | **Kubernetes (EKS)** | Application deployment and orchestration platform |
| 6. | **Argo CD** | GitOps-based continuous deployment to Kubernetes |
| 7. | **Prometheus** | Metrics collection from app and cluster |
| 8. | **Grafana** | Metrics visualization and alerting |
| 9. | **GitHub** | Source code and GitOps manifests version control |
| 10. | **AWS ECR** | Docker image registry |

**3. End-to-End Workflow**

A diagram of a blockchain

AI-generated content may be incorrect.

* + 1. Architectural Diagram of Enterprise Java CI/CD with Jenkins, GitOps, and Kubernetes Observability.
* **Source Control:** GitHub hosts both application code and Kubernetes manifests (GitOps repo).
* **CI/CD Orchestration:** Jenkins automates build, test, code analysis, and Docker image creation.
* **Quality Gate:** SonarQube is integrated for static code analysis.
* **Containerization:** Docker packages the Java application.
* **Artifact Store:** Amazon ECR stores Docker images.
* **Deployment:** Argo CD handles GitOps-based deployment to EKS.
* **Observability:** Prometheus and Grafana provide monitoring and alerting.

**4. GitHub Repositories**

* **Application & CI/CD:**  
  <https://github.com/HeenaDania/enterprise-java-gitops.git>
* **GitOps Manifests:**  
  <https://github.com/HeenaDania/enterprise-java-cicd.git>

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Directory structure for both GitHub repositories.

Inside enterprise-java-cicd repository, app folder contains the Java Spring Boot application while src folder contains pom.xml, Dockerfile, cicd folder holds Jenkinsfile for pipeline and automation scripts

Inside enterprise-java-gitops repository there are Kubernetes manifests and Argo CD configs while observability folder contains the Monitoring configurations.

**5. Detailed Implementation Steps**

**5.1 Initial Setup**

**5.1.1 Create GitHub Repositories**

* Created two repositories:
  + enterprise-java-cicd for application code and CI/CD pipeline
  + enterprise-java-gitops for Kubernetes manifests and GitOps

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. enterprise-java-cicd repository on Github.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. enterprise-java-gitops repository on Github.

**5.1.2 Prepare Local Development Environment**

* Installed Java 17+, Maven, Docker, and Git.
* Generated a Simple Spring Boot App using Spring Initializr with: Project: Maven, Language: Java, Spring Boot: 3.5.3, Group: com.example, Artifact: demo, Dependencies: Spring Web, Actuator.
* Verified Java build with Maven and Docker image creation.

A screenshot of a computer program

AI-generated content may be incorrect.

* + 1. Installed and checked all Prerequisites on Windows.

A screen shot of a computer

AI-generated content may be incorrect.

* + 1. Locally build the application successfully to check its working.

A computer screen with white text

AI-generated content may be incorrect.

* + 1. Created a Docker image and moved it to my account using DockerDesktop app.

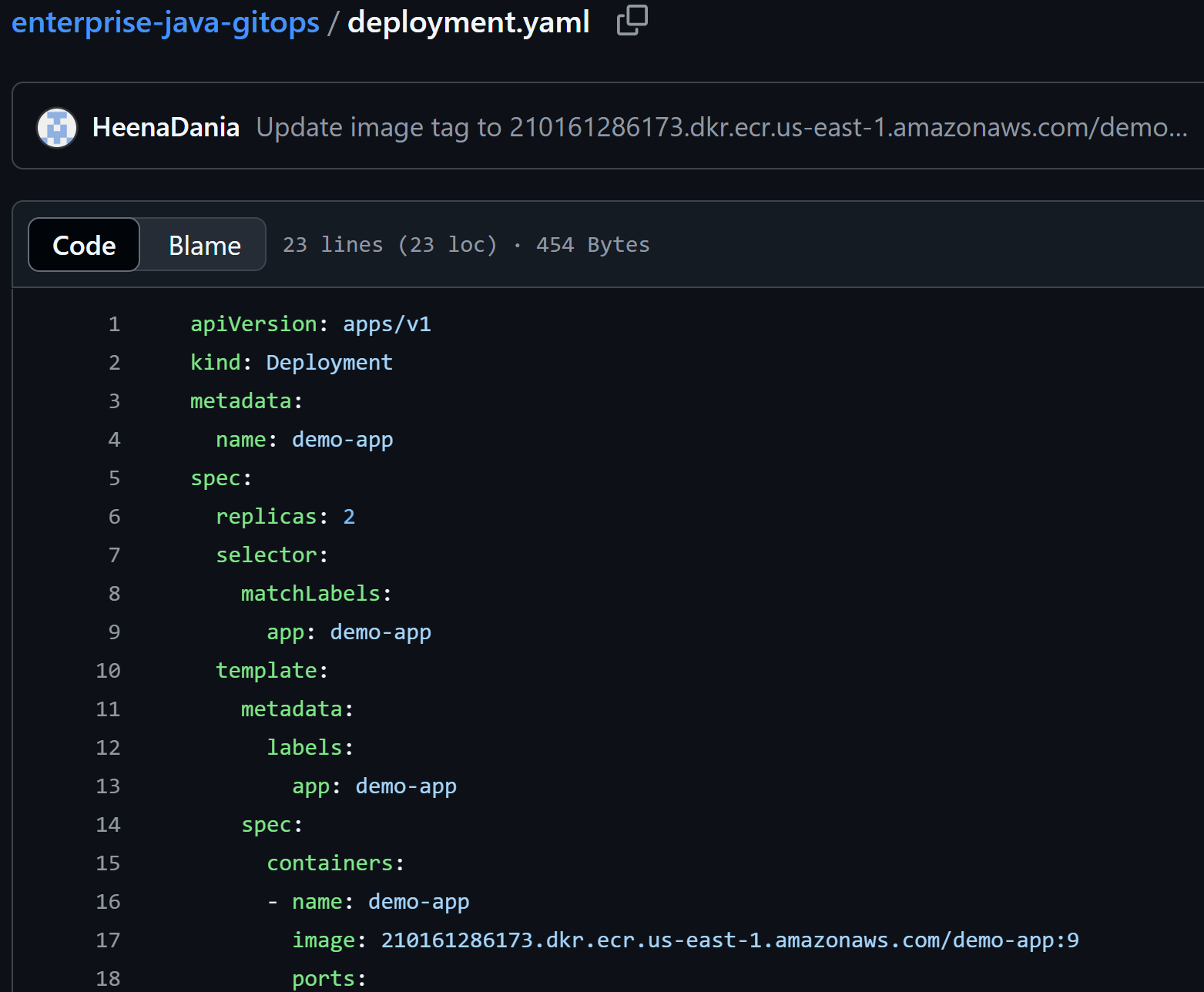
**5.2 Jenkins Setup (CI/CD Orchestration)**

* Deployed Jenkins locally accessible through url: <http://localhost:8080>.
* Created a Jenkinsfile for project: <https://github.com/HeenaDania/enterprise-java-cicd/blob/main/cicd/Jenkinsfile>
* Installed plugins: Git, Maven, Docker, SonarQube, Kubernetes, Blue Ocean from Manage Jenkins.
* Configured credentials for GitHub and AWS ECR and SonarQube.
* Created a Jenkins pipeline (Jenkinsfile) to automate:
  + Code checkout from GitHub.
  + Maven build and test.
  + SonarQube static code analysis.
  + Docker image build and push to ECR.
  + Update of image tag in deployment.yaml in GitOps repo.
* **How Used:** Jenkins triggers on code push, runs the pipeline, and updates the GitOps repo for deployment.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Successful Jenkins run triggered from update in enterprise-java-cicd repository on Github.



* + 1. Image tag changed by Jenkins for Deployment file in enterprise-java-gitops repository on Github.

* 1. **SonarQube Setup (Static Code Analysis)**
* Deployed SonarQube locally by downloading SonarQube from Community Edition and then running “StartSonar.bat” file from “cd sonarqube-<version>\bin\windows-x86-xx” folder.
* We can access it at: [http://localhost:9000](http://localhost:9000/).
* In Jenkins, configured SonarQube server and credentials.
* Created a project for Java code analysis.
* Integrated SonarQube with Jenkins pipeline for automated code quality checks.
* **How Used:** Every Jenkins build runs SonarQube analysis; build fails if quality gate is not met.

**A screenshot of a computer

AI-generated content may be incorrect.**

* + 1. SonarQube displaying analysis result for Java application code.

**5.4 Docker & Amazon ECR (Containerization & Registry)**

* Wrote a Dockerfile for the Java Spring Boot app.
* Built Docker images via Jenkins pipeline.
* Created an ECR repository on AWS.
* Configured Jenkins to authenticate and push images to ECR.
* **How Used:** Jenkins builds and pushes a new image to ECR on every successful build.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Elastic Container Registry storing the latest images build by Jenkins under Docker stage.

**5.5 Kubernetes (EKS) Setup**

* Outlined EKS cluster provisioning via AWS CLI: eksctl create cluster --name demo-cluster --region your-region --nodes 2
* Configured kubectl to connect to EKS.

A black screen with white text

AI-generated content may be incorrect.

* + 1. Elastic Kubernetes Service “demo-cluster” Cluster Nodes.

**5.6 Kubernetes Deployment & Service**

* Wrote “deployment.yaml” and “service.yaml” for the app.
* Ensured correct port configuration (used 8081 for app, 8080 for Jenkins).
* Used LoadBalancer service for external access.
* **How Used:** Deployment and service YAMLs are updated in the GitOps repo and applied by Argo CD.

A computer screen with many white text

AI-generated content may be incorrect.

* + 1. Successfully deployed the app on cluster “demo-cluster”

**5.7 Argo CD Setup (GitOps Deployment)**

* Installed Argo CD in the argocd namespace on EKS by using following commands:

kubectl create namespace argocd

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

* Port-forwarded Argo CD UI by using: kubectl port-forward svc/argocd-server -n argocd 8082:443, making it accessible through url: <http://localhost:8082/>
* Logged in with admin credentials.
* Created an Argo CD application pointing to the GitOps repo and manifest path.
* Set sync policy to automatic for continuous deployment.
* **How Used:** Argo CD watches the GitOps repo; any change (like a new image tag) triggers a deployment to EKS.

**A screenshot of a computer

AI-generated content may be incorrect.**

* + 1. Argo CD showing synced and healthy state reflecting that manifest files has been successfully deployed on Kubernetes.

**5.8 Prometheus & Grafana Setup (Observability)**

**5.8.1 Prometheus**

* Installed kube-prometheus-stack via Helm.
* Added spring-boot-starter-actuator and micrometer-registry-prometheus to pom.xml.
* Configured application.properties, Exposed metrics at /actuator/prometheus.
* Created a ServiceMonitor for Prometheus to scrape app metrics.

A screenshot of a computer screen

AI-generated content may be incorrect.

* + 1. Successfully installed Prometheus and Grafana on local system.
* Verified Prometheus UI.
* **How Used:** Prometheus scrapes metrics from the app and cluster.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Prometheus scrapes metrics from the app and cluster.

**5.8.2 Grafana**

* Port-forwarded Grafana service, making it accessible through url: <http://localhost:3000/>

A screen shot of a computer

AI-generated content may be incorrect.

* + 1. Forwarding to port 3000 for Grafana service.
* Logged in with admin credentials.
* Verified Prometheus as a data source.
* Imported dashboard via JSON or ID (e.g., 315).
* **How Used:** Grafana visualizes metrics and enables alerting.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Grafana Dashboard showing the CPU usage as well as the Alert created for it.

**6. Testing & Validation**

**6.1 Application Validation**

* Verified pods and services via kubectl get pods and kubectl get svc.
* Confirmed app accessible via LoadBalancer DNS.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Java App is accessible through LoadBalancer Domain name.

**6.2 Observability Validation**

* Confirmed Prometheus scraping targets.
* Verified Grafana dashboards show live metrics.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Grafana dashboard showing the CPU & Memory usage by pods running the app.

**6.3 CI/CD Validation**

* Triggered Jenkins builds on code push.
* Confirmed new images in ECR, updated manifests in GitOps repo, and automatic deployment via Argo CD.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

* + 1. Successful Jenkins builds and Argo CD syncing to the updated deployment file, deploying changes to Kubernetes.

**7. Challenges & Solutions**

| **Error/Issue** | **Cause/Diagnosis** | **Solution/Resolution** |
| --- | --- | --- |
| Jenkins push rejected | Remote repo had new commits | Pulled latest, resolved conflicts, then pushed |
| Pods not ready (0/1 READY) | Readiness probe failed: port mismatch (app on 8081, probe on 8080) | Updated all configs to use port 8081 |
| Spring Boot 404 Whitelabel error | /actuator/prometheus not enabled or missing dependencies | Added actuator & micrometer dependencies, updated properties |
| LoadBalancer DNS not resolving | AWS ELB not yet provisioned | Waited for provisioning, verified pod readiness |
| Prometheus metrics not scraped | No ServiceMonitor for app | Added ServiceMonitor with correct port/path |

**8. Lessons Learned**

* **Port Consistency:** All config files (app, deployment, service, probes) must use the same port.
* **GitOps Discipline:** All changes should flow through Git for traceability and automation.
* **CI/CD Automation:** Jenkins and Argo CD provide robust, hands-off deployments.
* **Observability:** Exposing and scraping custom app metrics is crucial for monitoring.
* **Troubleshooting:** Using kubectl describe, kubectl logs, and dashboards is vital for rapid issue resolution.

**9. Future Scope**

* Implement blue/green or canary deployments with Argo CD.
* Integrate advanced alerting and notification channels in Grafana.
* Expand observability with distributed tracing (e.g., Jaeger).
* Automate EKS provisioning with Terraform.

**10. Conclusion**

This project demonstrates a robust, automated, and observable CI/CD pipeline for Java microservices on AWS. By integrating Jenkins, GitHub, Argo CD, EKS, Prometheus, and Grafana, we achieved a modern DevOps workflow that is scalable, maintainable, and production-ready.