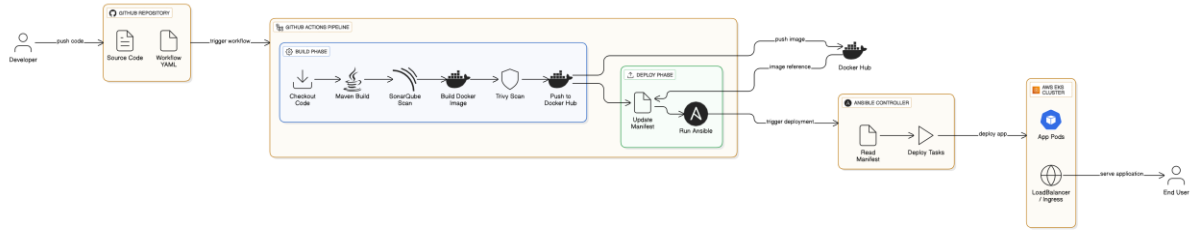
Project Overview

Purpose: A CI/CD pipeline to build, test, containerize, and deploy a Java application to an AWS EKS (Kubernetes) cluster using GitHub Actions + Ansible.



1. Developer Stage

- The process begins with a **developer pushing source code** and the corresponding **workflow YAML file** to the GitHub repository.
- This push action **automatically triggers** the CI/CD workflow.

2. GitHub Repository

- **Source Code**: Java application with build configuration (e.g., Maven pom.xml).
- **Workflow YAML**: Defines the pipeline steps for build, test, image creation, and deployment.

3. GitHub Actions Pipeline

This is the heart of the automation.

Build Phase

1. **Checkout Code** – Pulls the latest source from GitHub.
2. **Maven Build** – Compiles the Java project and creates an artifact (e.g., JAR).
3. **SonarQube Scan** – Performs static code analysis for code quality.
4. **Docker Build** – Packages the app into a container image.
5. **Trivy Scan** – Scans the Docker image for vulnerabilities.
6. **Push to Docker Hub** – Publishes the built image to Docker Hub.

Deploy Phase

7. **Update Manifest** – The Kubernetes deployment manifest file (k8s_deployment.yaml) is updated with the **new image tag** pushed to Docker Hub.
8. **Run Ansible** – The pipeline triggers Ansible to apply the updated manifest to the cluster.

4. Ansible Controller

- Acts as the **bridge between GitHub Actions and Kubernetes**.
- Reads the updated manifest file.
- Executes deployment tasks such as applying the manifest via kubectl or kubernetes.core.k8s.
- This ensures the latest container image is deployed.

5. AWS EKS Cluster

- The deployment is applied to Amazon Elastic Kubernetes Service.
- **App Pods** are created or updated using the new image from Docker Hub.
- A **LoadBalancer/Ingress** exposes the service to the external network.

6. End User

- The application becomes accessible to end users through the **public LoadBalancer or Ingress endpoint**.
- Requests are routed to the running pods inside the cluster.

Flow Summary

Developer → GitHub Repo → GitHub Actions Pipeline

→ Docker Hub + Ansible → EKS Cluster → End User

- **Build phase** ensures application is compiled, scanned, and containerized.
- **Deploy phase** ensures seamless delivery to the production cluster.
- **Automation** minimizes manual intervention and ensures consistent deployments.

Key Benefits of This Architecture

- Full CI/CD automation from code to deployment.
- Integrated security and quality checks (SonarQube, Trivy).
- Easy rollback by managing image tags and manifests.
- Scalable and reliable hosting on Kubernetes.
- Developer-friendly workflow using Git push as the trigger.

The pipeline covers:

- Code checkout

- Build (Maven)
- Static analysis (SonarQube)
- Container image build
- Image security scan (Trivy)
- Push to Docker Hub
- Update Kubernetes deployment manifest
- Deploy to EKS via Ansible

The project is structured as a “real-time” demo for doing CI/CD to Kubernetes using Ansible as the deployment tool.

Repository Structure

```
.
├── Application/
│   ├── pom.xml
│   └── src/
│
├── Ansible/
│   ├── ansible_k8s_deploy_playbook.yaml
│   └── k8s_deployment.yaml
│
└── .github/
    └── workflows/
        └── deploy.yaml   (or ci-cd.yaml)
```

- **Application/** — Java / Maven application’s source code
- **Ansible/** — The Ansible playbook and Kubernetes manifest
- **.github/workflows/** — GitHub Actions workflow definition

In the README of the repo, there's a more detailed description: (GitHub)
It explains how GitHub Actions maps to the traditional Jenkins pipeline:

Stage	Description
Checkout	actions/checkout
Build Jar	mvn clean package
SonarQube scan	via sonar-maven-plugin with SONAR_TOKEN

Stage	Description
Build Docker image	docker build
Scan image	via aquasecurity/trivy-action
Push image	docker push (to Docker Hub)
Update manifest	sed to replace image tag, commit back
Deploy via Ansible	ansible-playbook step (runner needs kube access)

Prerequisites & Setup

Before using the pipeline, the following are required: (GitHub)

1. **Existing AWS EKS cluster** (with worker nodes running)
2. **Docker Hub account** for hosting images
3. **GitHub repository** with Secrets configured:

Secret Name	Purpose
AWS_ACCESS_KEY_ID	AWS IAM key with EKS / ECR permissions
AWS_SECRET_ACCESS_KEY	AWS IAM secret key
KUBECONFIG_DATA	Base64-encoded kubeconfig for the EKS cluster
GITHUB_TOKEN	(auto-provided by GitHub)
DOCKER_USERNAME	Docker Hub user
DOCKER_PASSWORD	Docker Hub token / password
SONAR_HOST_URL	URL of SonarQube server
SONAR_TOKEN	Token for SonarQube

4. The GitHub Actions runner must have ability to run Ansible and have access to the target kubeconfig (decoded from KUBECONFIG_DATA) so that kubectl / Ansible modules can talk to the EKS cluster.
5. The repository assumes connection: local (i.e. playbooks run locally) rather than targeting remote hosts (i.e. no inventory hosts except localhost). (GitHub)
6. Tools used in the pipeline include:
    - Maven
    - Docker
    - Trivy
    - Ansible
    - kubectl

- sed / shell tooling

Pipeline Flow & Architecture

Here's the step-by-step flow:

1. **Checkout code** — via actions/checkout
2. **Configure AWS credentials** — use GitHub Secrets to allow EKS/ECR access
3. **Configure kubeconfig** — decode KUBECONFIG_DATA into $HOME/.kube/config
4. **Build & scan Docker image**
   - Use Maven to build application (packaging as JAR)
   - Build Docker image
   - Use Trivy to scan the image for vulnerabilities
5. **Push image to Docker Hub**
6. **Update Kubernetes manifest**
   - Use sed within the action to replace image tag with the new tag (e.g. using GITHUB_RUN_NUMBER)
   - Commit that change back to the same repo
7. **Deploy with Ansible**
   - Run ansible-playbook pointing to the playbook in Ansible/ansible_k8s_deploy_playbook.yaml
   - The playbook uses the updated manifest file Ansible/k8s_deployment.yaml
   - Apply the manifest to EKS (using kubectl/Ansible modules)

Success criteria:

- Docker image built, scanned, and pushed successfully
- Kubernetes manifest updated reflect new image tag
- Ansible step applies the manifest to EKS
- GitHub Actions job ends with success

Troubleshooting tips

- "No inventory was parsed" warning — can be ignored when using connection: local
- localhost:8080 connection refused — indicates kubeconfig or AWS creds are incorrect
- Push rejected on GitHub — remedy by adding fetch-depth: 0 to checkout and doing a git pull --rebase
- Manifest path not found — use relative path {{ playbook_dir }}/k8s_deployment.yaml rather than absolute

---

Ansible Playbook & Kubernetes YAML

Ansible/ansible_k8s_deploy_playbook.yaml

This is the playbook invoked in the pipeline.

- Read the Kubernetes manifest file k8s_deployment.yaml
- Use Ansible (or kubectl via tasks) to apply / update the deployment in EKS
- It likely uses the kubernetes.core.k8s module or kubectl commands under the hood

Ansible/k8s_deployment.yaml

This is the Kubernetes Deployment manifest template. The pipeline modifies the container image tag in this manifest (via sed) before running Ansible.

The playbook will then deploy this manifest to the target EKS cluster.

How to Use / Run Locally

If you wanted to run this pipeline outside GitHub Actions or locally, here is a rough sequence:

1. Ensure your local environment (or runner) has:
    - AWS credentials (with EKS / ECR permissions)
    - kubeconfig file that can access the EKS cluster
    - Docker, Ansible, kubectl installed
2. Build & tag the Docker image manually:
3. mvn clean package
4. docker build -t yourname/app:TAG .
5. docker push yourname/app:TAG
6. Modify the Ansible/k8s_deployment.yaml to use yourname/app:TAG
7. Run the Ansible playbook:
8. ansible-playbook Ansible/ansible_k8s_deploy_playbook.yaml -c local
9. Verify on EKS:
10. kubectl get deployments
11. kubectl get pods