DevOps End-to-End Lifecycle Project: User Manual

Project Overview

This project demonstrates the practical implementation of a complete DevOps lifecycle workflow. It transforms a monolithic application into a containerized, automated, and highly available system using Jenkins, Docker, Kubernetes, Terraform, and AWS. The aim is to enable continuous integration, continuous delivery, and automated infrastructure provisioning.

This manual serves as a step-by-step guide for users to understand, operate, and maintain the project.

Architecture Overview

The architecture consists of four EC2 instances (Ubuntu 22.04) on AWS, each dedicated to specific roles:

- Jenkins for CI/CD automation.
- Docker for containerization.
- Kubernetes for orchestration.
- Terraform and Ansible for provisioning and configuration.

Communication and deployment flow is orchestrated through these components, with DockerHub acting as the container registry.

System Components and Tools

Tool	Purpose
GitHub	Source Control and Repository

Jenkins	CI/CD Automation
Docker	Containerization
Kubernetes	Container Orchestration
Terraform	Infrastructure as Code
AWS EC2	Cloud Infrastructure
Ansible	Configuration Management

Infrastructure Setup (Provisioned via Terraform)

Instances and Configuration

- **Worker 1:** Jenkins, Java
- **Worker 2:** Docker, Kubernetes
- **Worker 3:** Java, Docker, Kubernetes
- **Worker 4:** Docker, Kubernetes

All EC2 instances run on **Ubuntu 22.04**.

Step-by-Step Guide

1. Version Control & Git Workflow

- Utilize **GitHub** for maintaining the source code repository.
- Adopt the **GitFlow branching model** suitable for monolithic projects:
    - Feature branches for development.
    - `master/main` branch for production-ready code.
    - All merge requests should trigger automated CI/CD pipelines.

---

2. CI/CD Pipeline with Jenkins

- **Jenkins** is configured to automate the pipeline through a declarative `Jenkinsfile`.
- **Trigger Mechanism:**
    1. Automatically triggers builds on every commit to the master branch via webhook.
- **Pipeline Stages:**
    1. Pull source code from GitHub.
    2. Build Docker Image using Dockerfile.
    3. Push Docker Image to **DockerHub**.
    4. Deploy updated application on Kubernetes.
    5. Roll out updates via Kubernetes Deployment.

---

3. Containerization with Docker

- **Dockerfile** is created to package the monolithic application into a Docker container.
- Every commit triggers a build, resulting in a new image version pushed to **DockerHub**.
- Ensure Docker images are tagged properly for traceability (e.g., `app:v1`, `app:v2`, etc.).

---

4. Kubernetes Deployment

- Kubernetes cluster is deployed on AWS EC2 instances.
- **NodePort Service** is configured on port `30008` to expose the application externally.
- Kubernetes YAML configurations:

- `deployment.yaml`: Defines the desired state for pods, deployments, and container images pulled from DockerHub.
- `service.yaml`: Exposes the application using NodePort for browser accessibility.

Scaling Configuration:

- Deployment is set with **2 replicas** to ensure high availability.

---

5. Infrastructure as Code (Terraform)

- Terraform scripts manage the entire AWS EC2 infrastructure lifecycle.
- Each EC2 instance is provisioned and configured automatically.

Resources Managed by Terraform:

- EC2 Instances
- Security Groups
- VPC and Networking
- IAM Roles for Jenkins and EC2 access

---

6. Configuration Management with Ansible

- Ansible is utilized post-provisioning for:
  - Installing dependencies like Docker, Kubernetes, Jenkins, Java.
  - Ensuring consistent environment setup across all EC2 instances.

Workflow Summary

1. Developer pushes code to GitHub.
2. Webhook triggers Jenkins pipeline.
3. Jenkins builds Docker Image and pushes it to DockerHub.
4. Kubernetes cluster pulls the latest Docker Image.
5. Kubernetes Deployment rolls out the update.
6. Service exposes application at `NodePort 30008`.
7. Application is accessible via EC2 Public IP and port 30008.

Important Notes

- **DockerHub Credentials:** Jenkins must be configured with DockerHub credentials for secure image push.
- **Kubernetes Cluster Role:** Jenkins can be configured as a slave node to the Kubernetes master for seamless deployment.
- **Terraform State Management:** Ensure Terraform state files are preserved securely to avoid conflicts during re-provisioning.

Deployment Access

To access the deployed application:

1. Obtain the public IP of the Kubernetes master EC2 instance.
2. Access the application via browser at: `http://<EC2-Public-IP>:30008`

Maintenance Guidelines

- Monitor Jenkins, Docker, Kubernetes regularly for updates and security patches.

- Backup Terraform state files periodically.
- Regularly clean up unused Docker images and Kubernetes resources.
- Validate infrastructure using Terraform plan before applying changes.

Common Issues and Solutions

Issue	Solution
Jenkins failing to trigger build	Check webhook configuration on GitHub
Docker push failure	Validate DockerHub credentials on Jenkins
Kubernetes pods not updating	Ensure new image tag is referenced in YAML
NodePort not accessible	Verify security group and EC2 port settings

---

If you want, I can provide this in a PDF or Markdown format for your documentation as well. Let me know.