

JENKINS JOB FOR AUTOMATED DEPLOYMENT

Micro Project Report

Submitted by

Bharath M

22ITR012

Course Code & Name: 22ITF02-Devops

Programme & Branch: B.TECH IT

Department : Information Technology

Kongu Engineering College,Perundurai

May 2025

BONAFIDE CERTIFICATE

Certified that this micro project documentation of “**JENKINS JOB FOR AUTOMATED DEPLOYMENT**” is the bonafide work of “**BHARATH M(22ITR012)**” who carried out the project under my supervision. Certified further that to the best of my knowledge the work reported here in does not form part of my other thesis or dissertation on the basis of which a degree or awarded was conferred on an earlier occasion on this or any other candidate.

Submitted on _____

SIGNATURE

Mr. D.VIJAY ANAND,
MCA,ME

SUPERVISOR

Assistant Professor
Department of IT,
Kongu Engineering
College, Perundurai

SIGNATURE

Dr.S.ANANDAMURUGAN,
M.E.,Ph.D

HEAD OF THE DEPARTMENT

Professor & HOD,
Department of IT,
Kongu Engineering College,
Perundurai.

JENKINS JOB FOR AUTOMATED DEPLOYMENT

AIM

To automate the deployment process of a Node.js web application to a Kubernetes cluster using Jenkins CI/CD pipeline integrated with Docker and Helm or kubectl.

ABSTRACT

This project focuses on streamlining the software deployment lifecycle by leveraging Jenkins, Docker, Kubernetes, and Helm. A simple Node.js application is containerized using Docker, stored on DockerHub, and deployed to a Kubernetes cluster through a Jenkins pipeline. The pipeline automates stages including cloning the repository, building the Docker image, pushing to DockerHub, and deploying via Helm or kubectl. This solution reduces manual errors, enhances productivity, and ensures consistent deployments across environments.

SCOPE AND OBJECTIVES

Scope:

- CI/CD implementation for containerized applications
- Integration with cloud-native tools (Docker, Kubernetes, Jenkins)
- Deployment in local (Minikube) or cloud Kubernetes clusters

Objectives:

- Automate build, test, and deployment stages using Jenkins
- Containerize a Node.js app using Docker
- Deploy the app to Kubernetes using Helm or kubectl
- Enable rollback and version control in deployments
- Improve reliability and speed of software delivery.

HARDWARE & SOFTWARE REQUIREMENTS

Hardware Requirements:

- A system with at least 8GB RAM
- 2+ GHz Processor
- Internet connectivity

Software Requirements:

- Jenkins
- Docker
- Git
- Kubernetes
- Helm
- GitHub Account
- DockerHub Account

PROJECT DESIGN

The project design follows a streamlined CI/CD pipeline that begins with the developer pushing code to GitHub, which triggers Jenkins to start the automation process. Jenkins first clones the repository, then builds a Docker image of the Node.js application using the provided Dockerfile. This image is pushed to DockerHub, ensuring version control and easy accessibility. Next, Jenkins deploys the application to a Kubernetes cluster using deployment and service YAML files or optionally Helm charts. The project is organized into separate folders for the application code, Kubernetes configurations, and the Jenkinsfile, ensuring a modular and maintainable structure. This design enables fast, consistent, and error-free deployments with minimal manual intervention.

ADVANTAGES

- Fully automated build and deployment process
- Faster time-to-market
- Scalable deployments on Kubernetes
- Integrated version control and rollback
- Improved developer productivity

PROPOSED SYSTEM

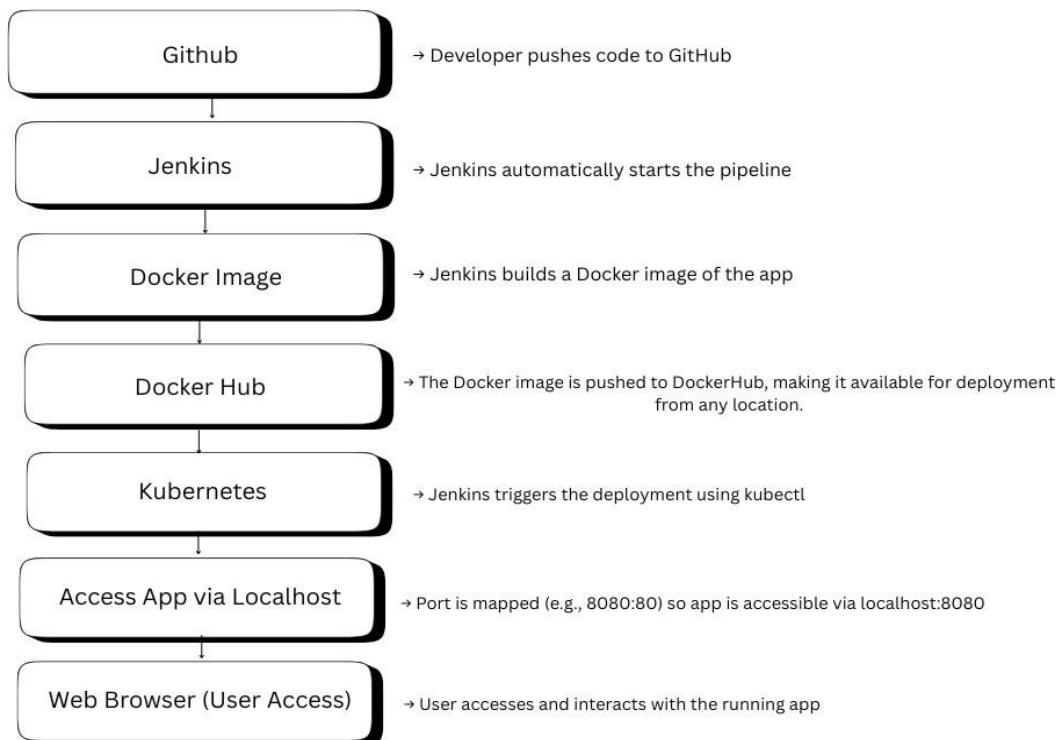
The proposed system introduces a Continuous Integration and Continuous Deployment (CI/CD) pipeline using Jenkins, aimed at automating the end-to-end deployment workflow. It streamlines several critical stages, including pulling the latest source code from GitHub, building a Docker image for the Node.js application, pushing the image to DockerHub, and finally deploying the application to a Kubernetes cluster using either Helm or kubectl. This automation not only accelerates the deployment process but also ensures consistency, minimizes manual errors, and enhances the overall reliability and efficiency of the software delivery pipeline.

IMPLEMENTATION OF THE PROJECT

The implementation of this project involves integrating Continuous Integration and Continuous Deployment (CI/CD) tools to automate the entire software delivery pipeline. First, a simple Node.js application is developed and containerized using Docker with a defined Dockerfile. The application code is stored in a GitHub repository. Jenkins is then configured with a Jenkinsfile to automate multiple stages: cloning the source code from GitHub, building the Docker image, and pushing it to DockerHub using authenticated credentials. Once the image is available on DockerHub, Jenkins executes kubectl commands (or Helm charts, if used) to deploy the application into a Kubernetes cluster. Kubernetes deployment and service YAML files define how the application is launched and exposed. This

CI/CD flow ensures reliable, repeatable, and efficient software deployment with minimal human intervention.

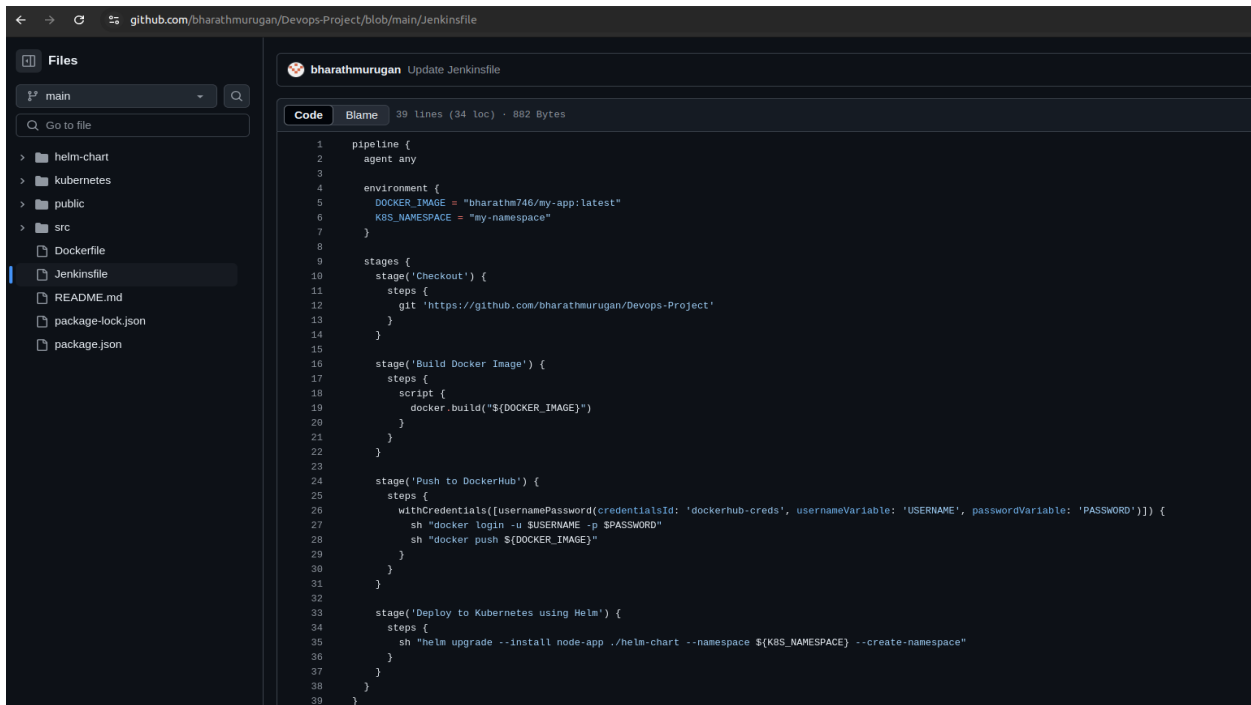
Flow of Design:



Flow Chart

As shown in the flow chart, the process starts when a developer pushes code to GitHub, which triggers Jenkins to run an automated pipeline that builds a Docker image of the application. Jenkins then pushes this image to Docker Hub, making it available for deployment anywhere. Next, Jenkins deploys the image to a Kubernetes cluster using kubectl, after which the application becomes accessible to users through a web browser, completing the end-to-end CI/CD workflow.

OUTPUT:

A screenshot of a GitHub web interface showing a Jenkinsfile configuration. The left sidebar displays a file explorer with folders like 'helm-chart', 'kubernetes', 'public', and 'src', and files like 'Dockerfile', 'Jenkinsfile', 'README.md', 'package-lock.json', and 'package.json'. The main area shows the 'Jenkinsfile' with a 'Code' tab selected. The file content is a YAML pipeline configuration with stages for checkout, building a Docker image, pushing to Docker Hub, and deploying to Kubernetes using Helm. The pipeline is named 'pipeline' and uses the 'any' agent. Environment variables are set for 'DOCKER_IMAGE' and 'K8S_NAMESPACE'. The 'Checkout' stage clones the repository. The 'Build Docker Image' stage uses a Docker build script. The 'Push to DockerHub' stage uses 'docker login' and 'docker push' with credentials. The 'Deploy to Kubernetes using Helm' stage uses 'helm upgrade' to install or upgrade the application.

```
1 pipeline {
2   agent any
3
4   environment {
5     DOCKER_IMAGE = "bharath746/my-app:latest"
6     K8S_NAMESPACE = "my-namespace"
7   }
8
9   stages {
10    stage('Checkout') {
11      steps {
12        git 'https://github.com/bharathmurugan/Devops-Project'
13      }
14    }
15
16    stage('Build Docker Image') {
17      steps {
18        script {
19          docker.build("${DOCKER_IMAGE}")
20        }
21      }
22    }
23
24    stage('Push to DockerHub') {
25      steps {
26        withCredentials([usernamePassword(credentialsId: 'dockerhub-creds', usernameVariable: 'USERNAME', passwordVariable: 'PASSWORD')]) {
27          sh "docker login -u $USERNAME -p $PASSWORD"
28          sh "docker push ${DOCKER_IMAGE}"
29        }
30      }
31    }
32
33    stage('Deploy to Kubernetes using Helm') {
34      steps {
35        sh "helm upgrade --install node-app ./helm-chart --namespace ${K8S_NAMESPACE} --create-namespace"
36      }
37    }
38  }
39 }
```

FIGURE 1: Automated CI/CD Pipeline Configuration Using Jenkins

As shown in the figure 1, the pipeline includes stages to clone a GitHub repository, build a Docker image, and push it to Docker Hub using credentials securely stored in Jenkins. It also sets environment variables for image naming and Kubernetes configuration, enabling smooth deployment processes in Kubernetes clusters. This setup demonstrates the integration of source control, containerization, and automated deployments through Jenkins.

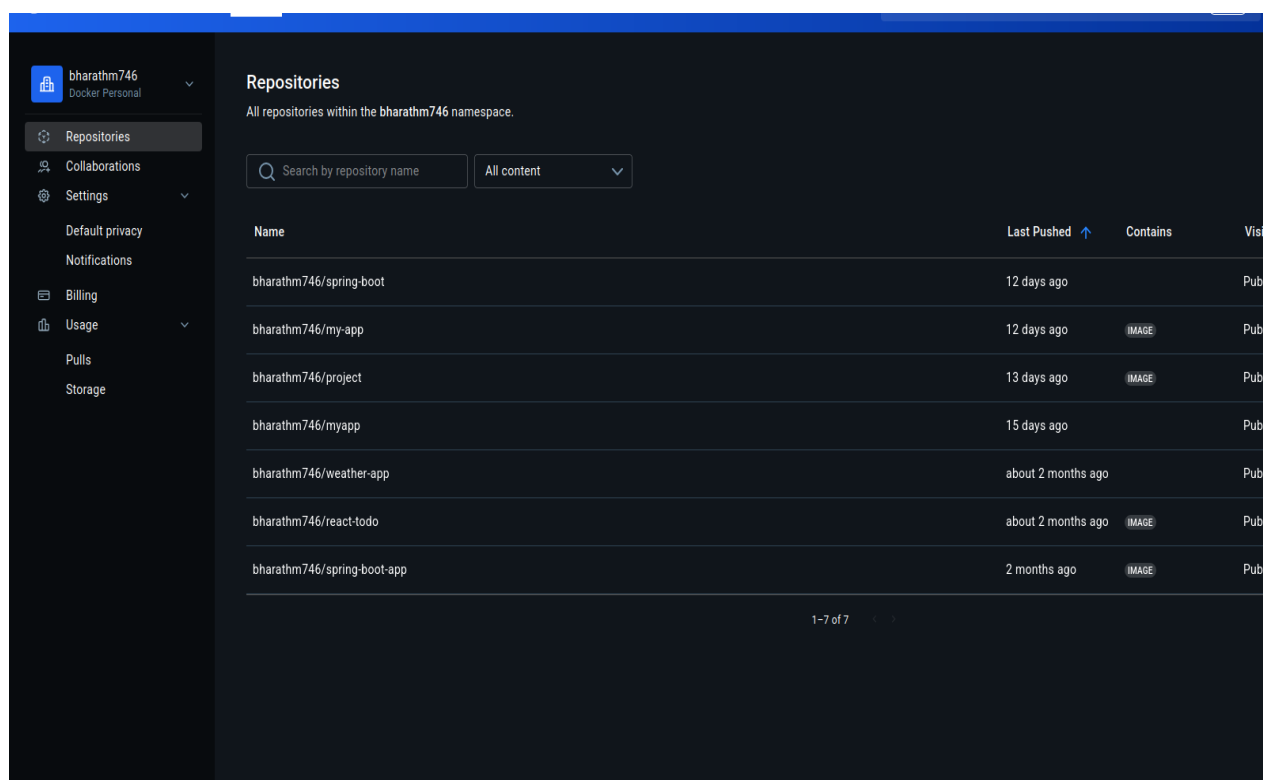


FIGURE 2: Docker Hub Repository View After Jenkins Pipeline Deployment

As shown in Figure 2, the Docker Hub dashboard for the user **krishna728** highlights successfully pushed Docker images. These repositories represent projects that were built and deployed through automated Jenkins pipelines. The timestamps under "Last Pushed" confirm recent activity, demonstrating that the CI/CD pipeline is operating correctly and integrating seamlessly with Docker Hub for container image management

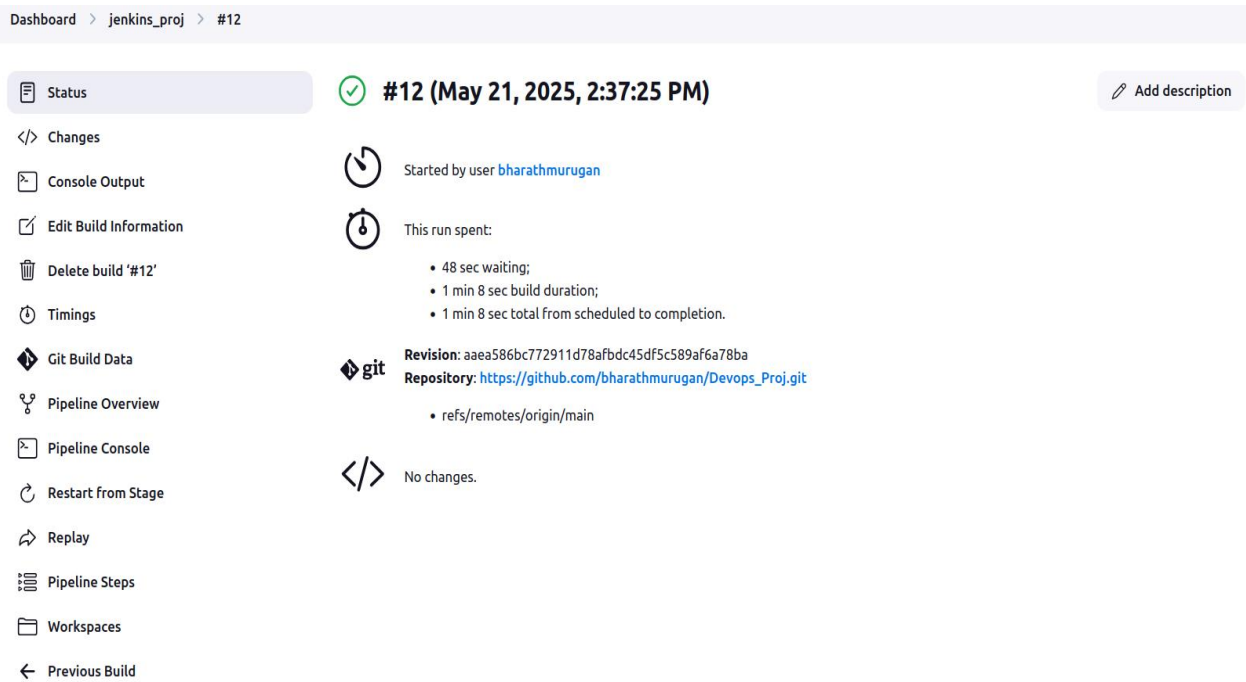


FIGURE 3: Jenkins Build Execution Summary

As shown in Figure 3, the Jenkins CI/CD pipeline initiates automatically when changes are detected in the linked GitHub repository, typically via a webhook or scheduled polling. The process begins with Jenkins cloning the updated code, followed by the execution of a predefined Jenkinsfile that outlines stages such as building the application, running tests, creating a Docker image, and pushing that image to Docker Hub. This seamless pipeline ensures that every code change is automatically built, tested, and deployed as a Docker container, with detailed logs and status updates provided throughout the process enabling efficient, automated, and continuous delivery of applications.

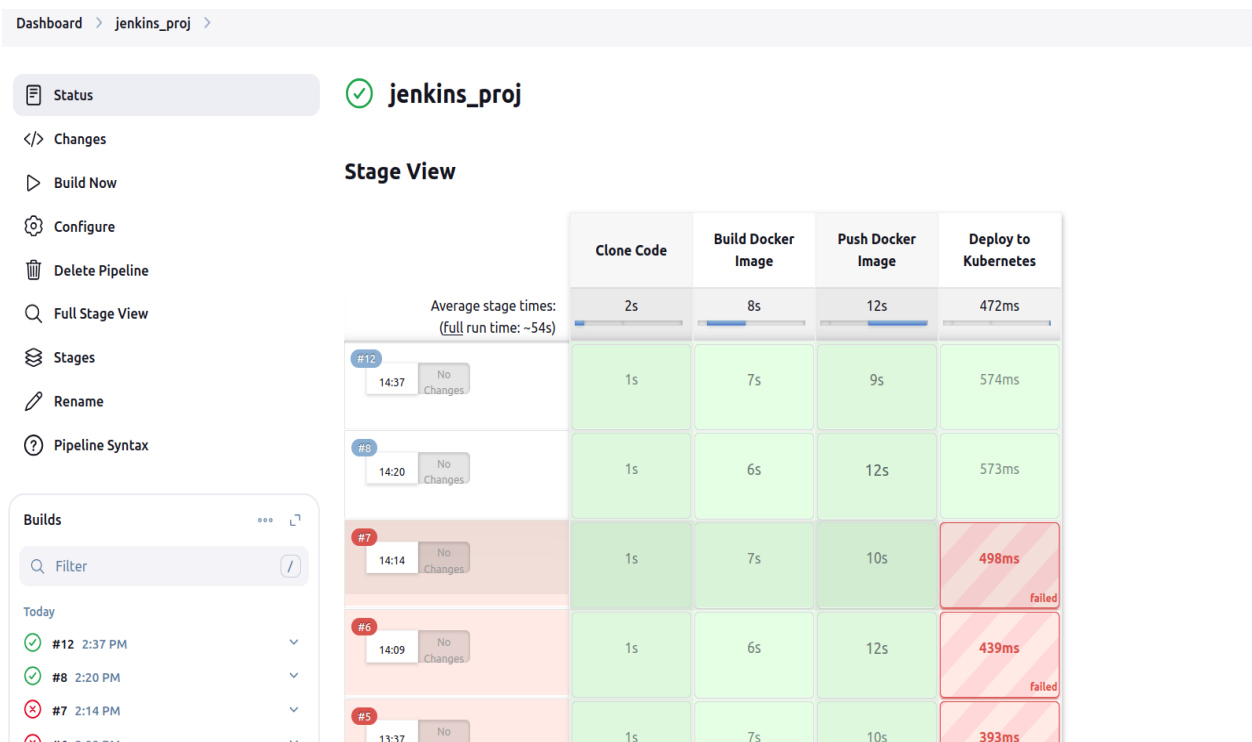


FIGURE 4: Jenkins Pipeline Stages with Kubernetes Deployment

As shown in Figure 4, the Jenkins pipeline automates the entire CI/CD workflow through clearly defined stages—starting with source code checkout and repository cloning, followed by building a Docker image, pushing the image to Docker Hub, and finally deploying it to a Kubernetes cluster. Each stage completes successfully, indicating a robust and functional pipeline. The final step executes `kubectl apply` commands using predefined Kubernetes manifests (`deployment.yaml` and `service.yaml`), resulting in the deployment and exposure of the application as Kubernetes services, thereby demonstrating an end-to-end automated delivery pipeline from code commit to production deployment.

COMMANDS:

- **git clone repo_link**
→ Clone the source code from GitHub repository.
- **docker build -t your-dockerhub-username/node-app:latest ./app**
→ Build a Docker image for the Node.js application.
- **docker login**
→ Log in to DockerHub to allow pushing images.
- **docker push your-dockerhub-username/node-app:latest**
→ Push the Docker image to DockerHub.
- **kubectrl apply -f kubernetes/deployment.yaml**
→ Deploy the application to Kubernetes using the deployment file.
- **kubectrl apply -f kubernetes/service.yaml**
→ Expose the app using a NodePort service in Kubernetes.
- **kubectrl get pods**
→ Check if the application pods are running.
- **kubectrl get services**
→ Get the service details including the NodePort to access the app.
- **minikube service node-service**
→ Open the deployed Node.js application in a browser.
- **kubectrl delete -f kubernetes**
→ Remove all Kubernetes resources related to this app.

CONCLUSION:

This project successfully demonstrates how to automate the deployment of a Node.js application using a CI/CD pipeline powered by Jenkins, Docker, and Kubernetes. By streamlining the development-to-deployment process, it reduces manual errors, speeds up release cycles, and ensures consistent deployments across environments. The integration of Jenkins with DockerHub and Kubernetes allows teams to build, test, and deploy applications seamlessly, making the overall DevOps workflow more efficient and scalable for modern software delivery needs.

FUTURE WORK:

To further enhance the CI/CD pipeline, several improvements can be implemented. First, incorporating automated testing stages into the Jenkins pipeline will ensure code reliability and reduce bugs before deployment. Configuring Slack or email notifications for build failures can improve team awareness and response time. Advanced deployment strategies like Blue-Green or Canary deployments can be introduced to minimize downtime and risks during updates. Integrating dynamic environment provisioning using Terraform would enable scalable and reproducible infrastructure. Finally, adopting GitOps practices with tools like ArgoCD will support declarative delivery and continuous synchronization between Git repositories and Kubernetes clusters, promoting more efficient and reliable deployments.