**CI/CD Pipeline Setup with Git, Jenkins,Docker and Kubernetes**

**Prepared by:** Tarun kumar Shori  
**Email:** Tarunshori801@gmail.com

Problem Statement:

The goal of this project is to automate the process of building artifacts from code commits to specific Git repositories and deploying them to Kubernetes clusters using a CI/CD pipeline. Jenkins will serve as the automation server to trigger builds upon code changes, and Kubernetes will be used for application deployment.

Scenario:

You are tasked with setting up a Continuous Integration (CI) and Continuous Deployment (CD) pipeline that will automatically build, test, and deploy a React application to a Kubernetes cluster whenever a commit is made to a specific Git repository.

## **Overview :**

This document outlines the setup and configuration of a Continuous Integration/Continuous Deployment (CI/CD) pipeline using Git, Jenkins, and Kubernetes (K8s). The goal of this pipeline is to automate the building and deployment process upon code commits to Git repositories, ensuring that artifacts are built and deployed to Kubernetes clusters with minimal manual intervention.

## **Requirements:**

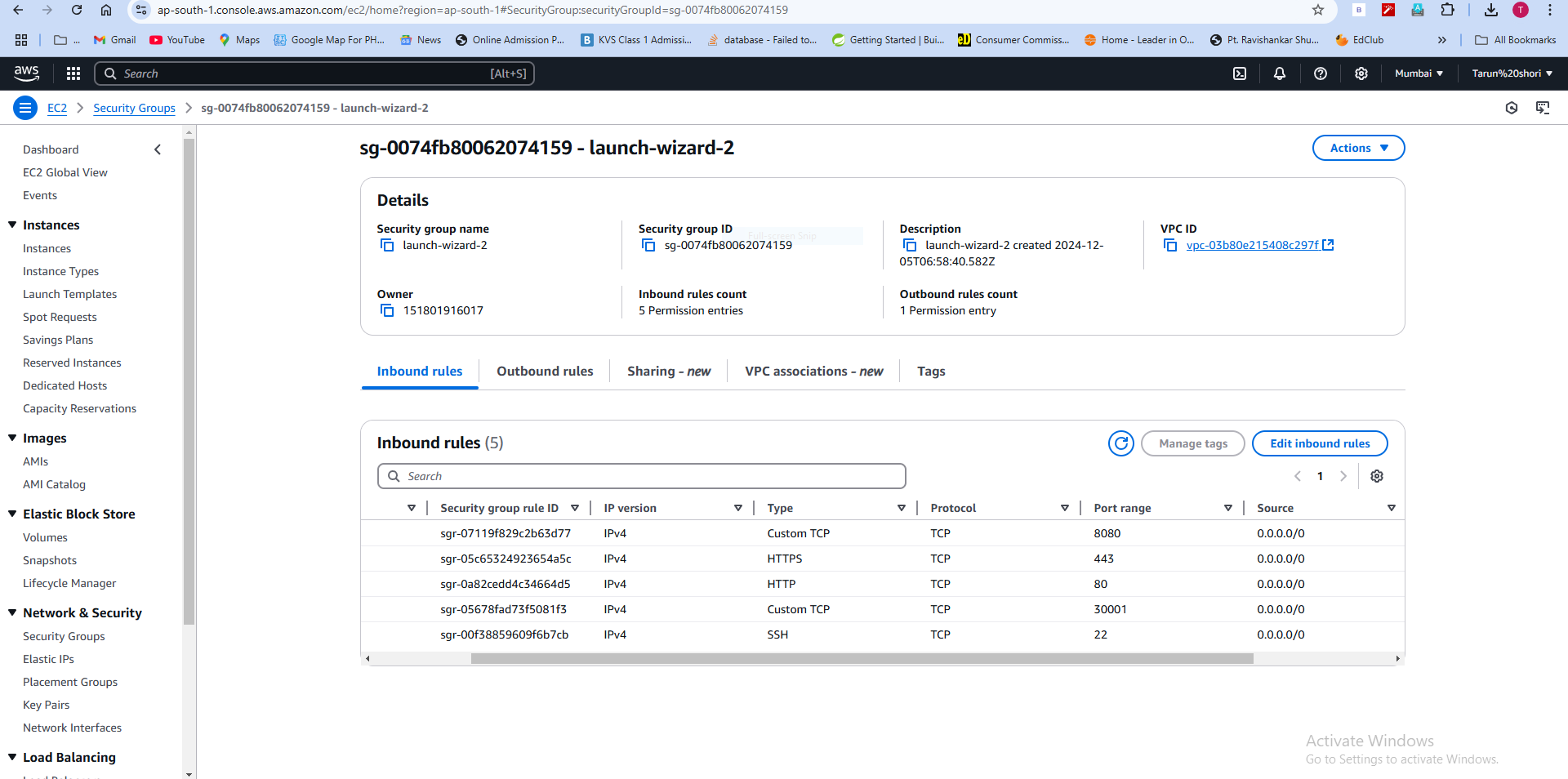
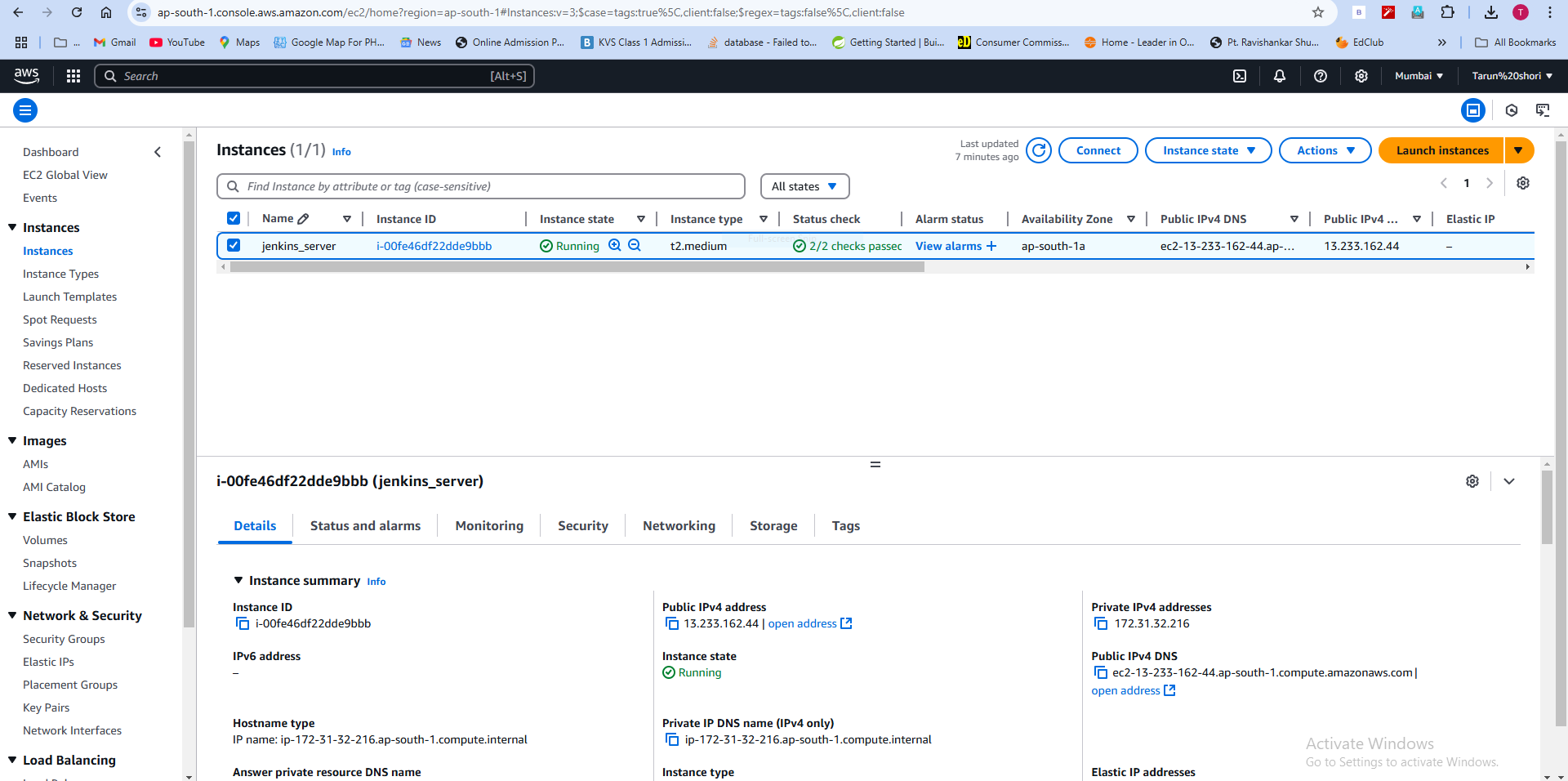
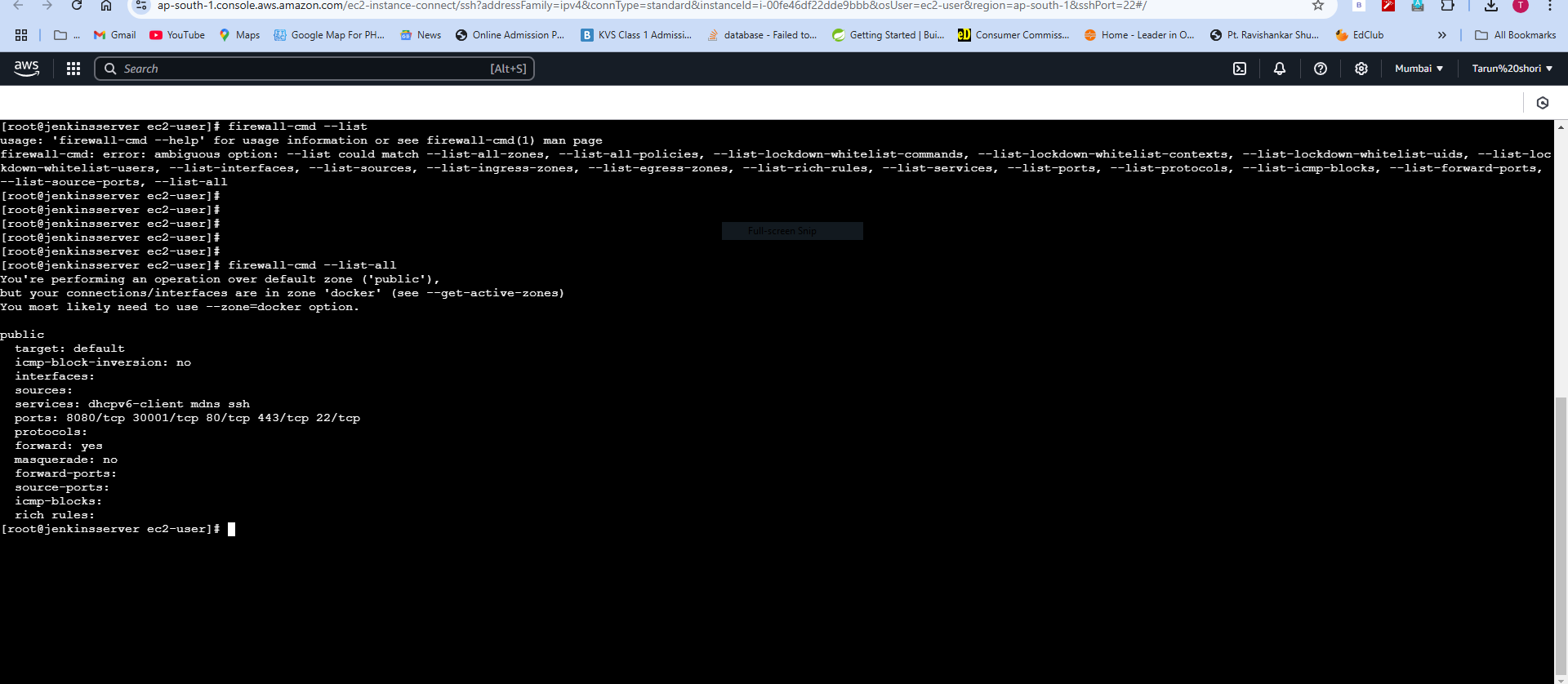
* **Git Integration**: Set up a webhook or polling mechanism so that Jenkins is notified when a commit is made to a Git repository.
* **Automated Build**: Use Jenkins to automate building artifacts (e.g., Docker images, binaries) upon successful commit detection.
* **Kubernetes Integration**: Deploy the built artifact to a specified Kubernetes cluster after a successful build.
* **Groovy Scripting**: Implement automation logic using Groovy scripting for flexibility and scalability.

Steps Followed:

Launch EC2 Instance:

Created an EC2 instance with 4 GB RAM on AWS.

Configured the security group to allow necessary inbound rules like SSH (port 22), port 30001 and HTTP (port 8080 for Jenkins).



Install Jenkins on Amazon Linux :

Step 1: Install Java

Jenkins requires Java (OpenJDK 11 is recommended).

sudo yum update -y

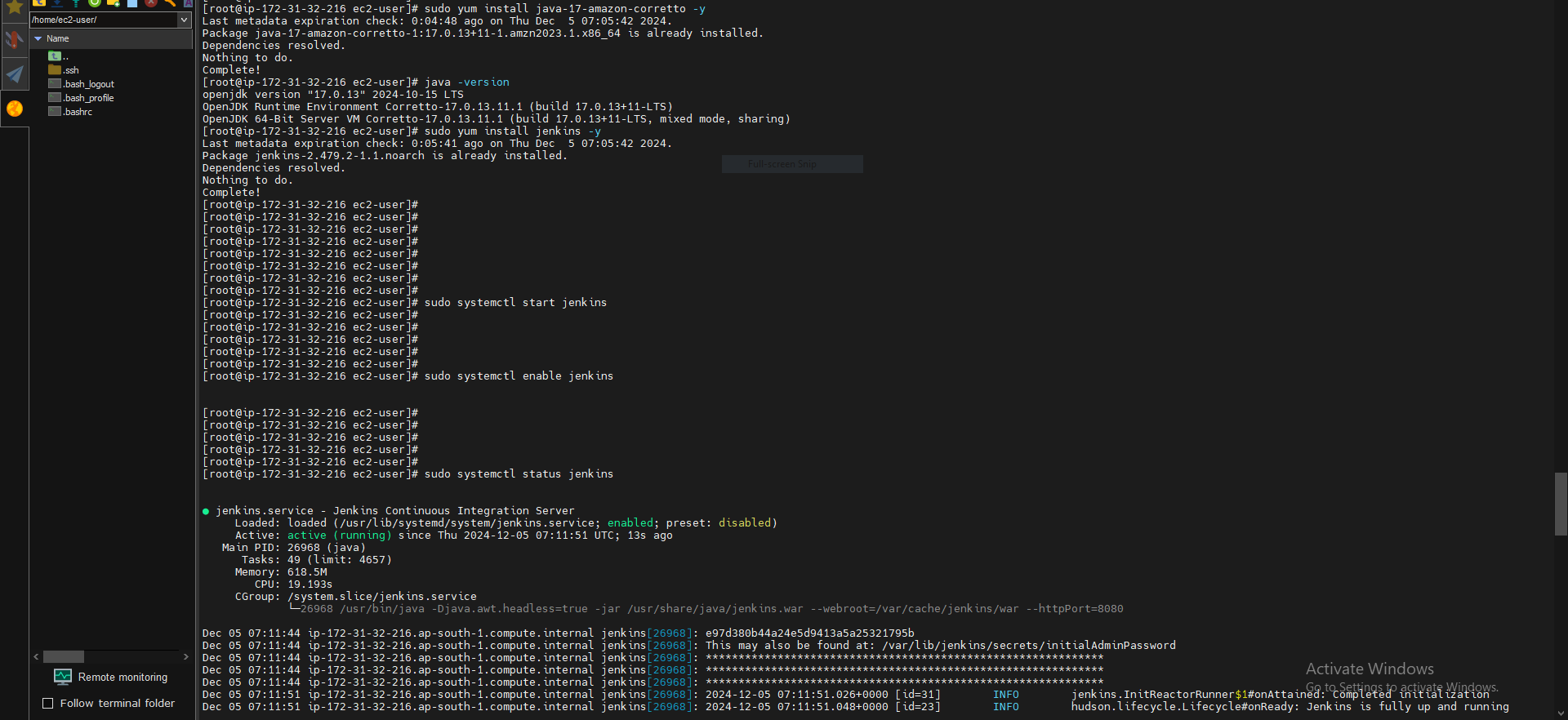
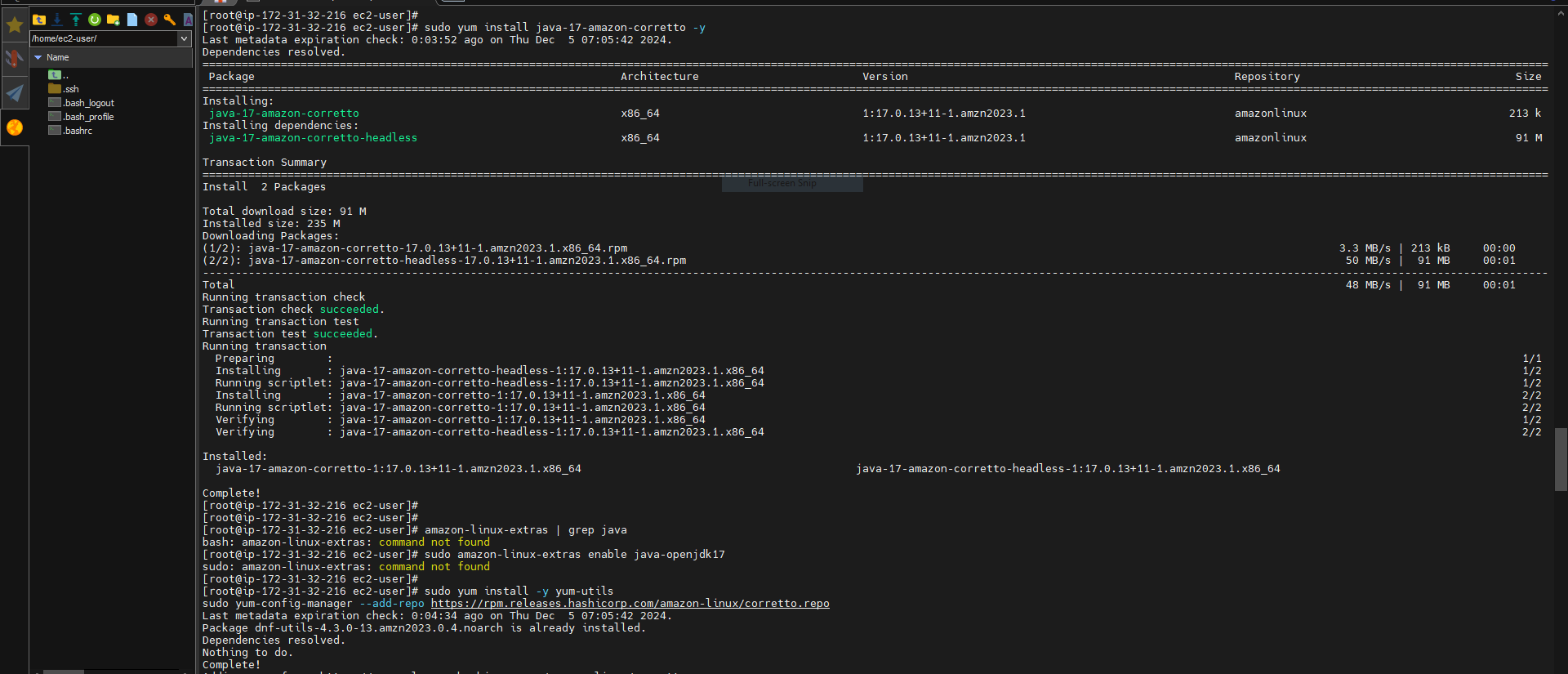
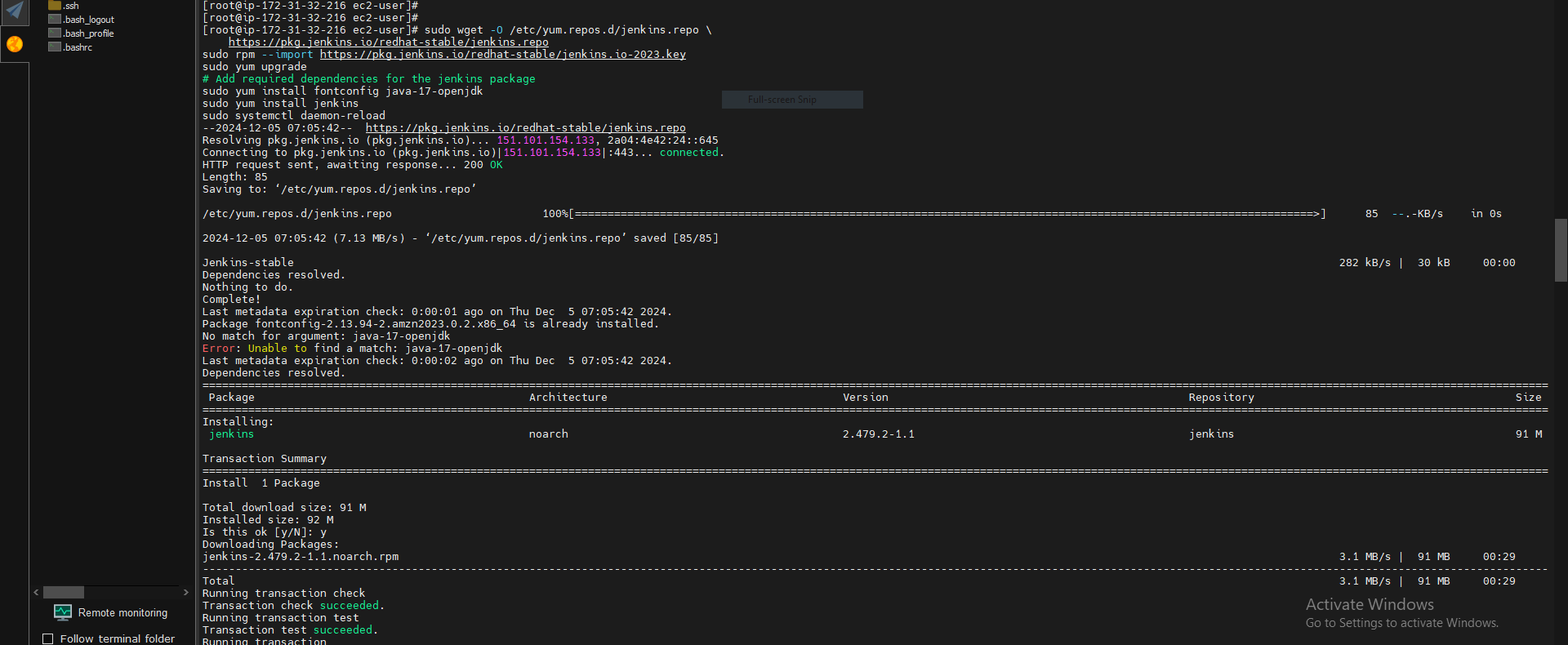
sudo yum install -y java-11-openjdk-devel

Step 2: Install Jenkins

Add Jenkins repository and install the GPG key:

sudo curl -fsSL https://pkg.jenkins.io/redhat/jenkins.repo | sudo tee /etc/yum.repos.d/jenkins.repo

sudo rpm --import <https://pkg.jenkins.io/redhat/jenkins.io.key>



Install Jenkins:

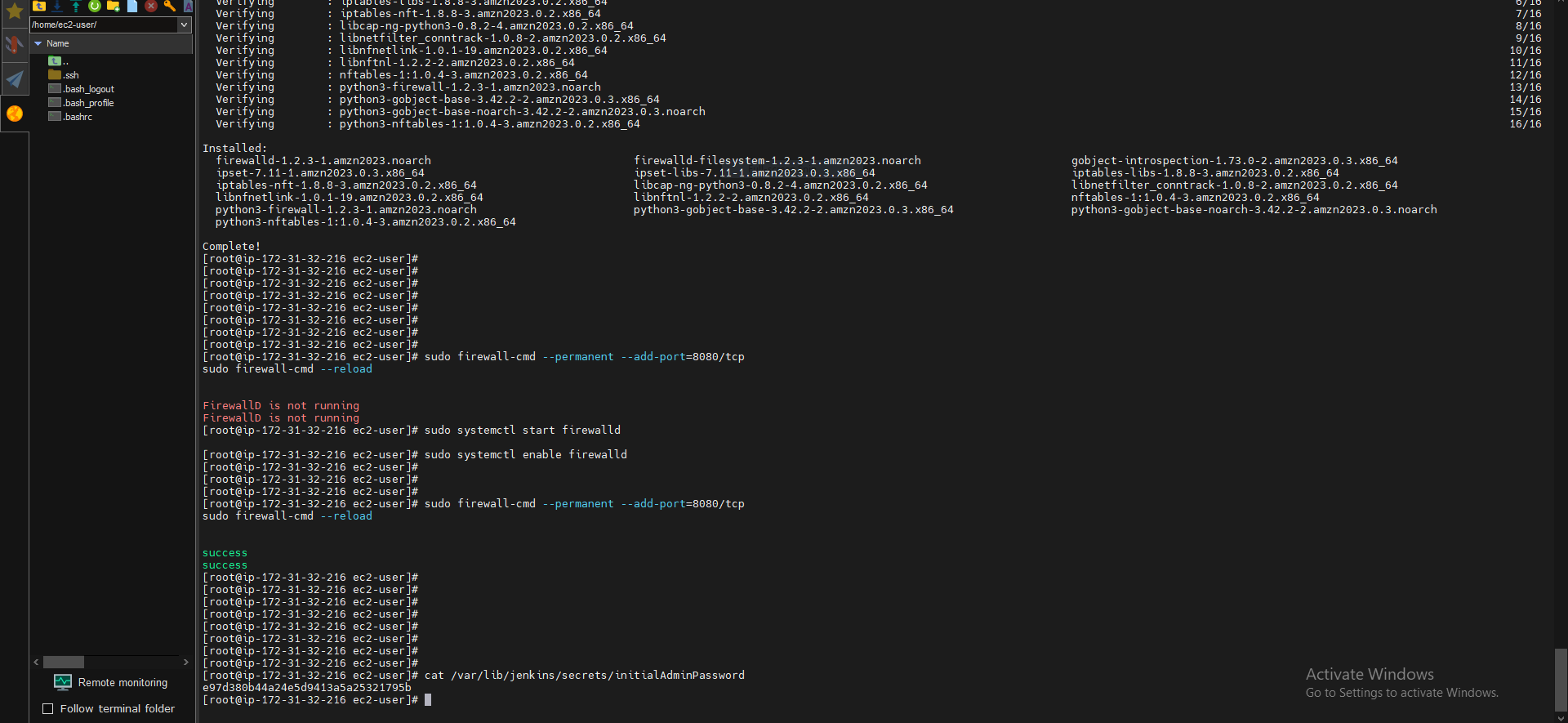
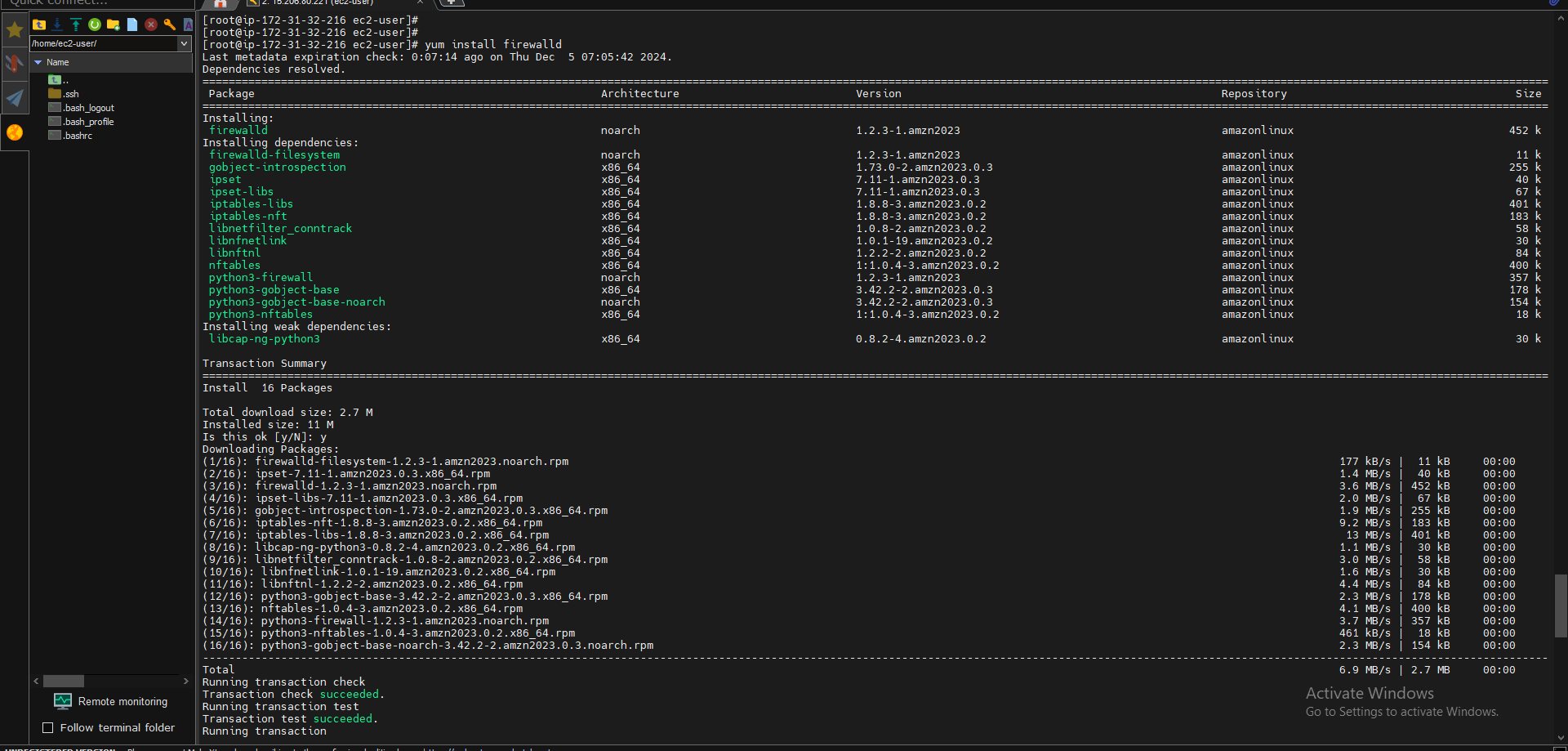
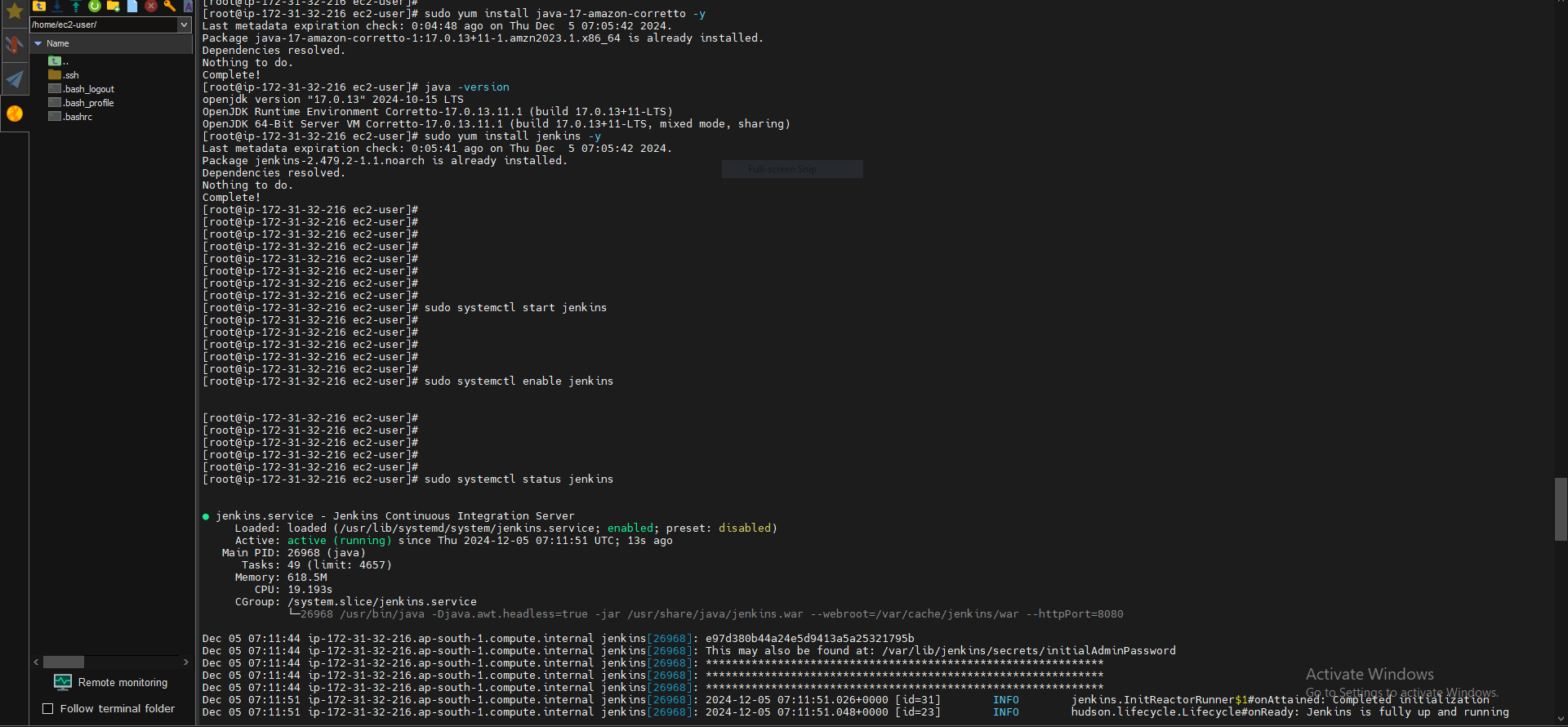
sudo yum install -y jenkins

Step 3: Start Jenkins

Enable and start the Jenkins service:

sudo systemctl start jenkins

sudo systemctl enable Jenkins

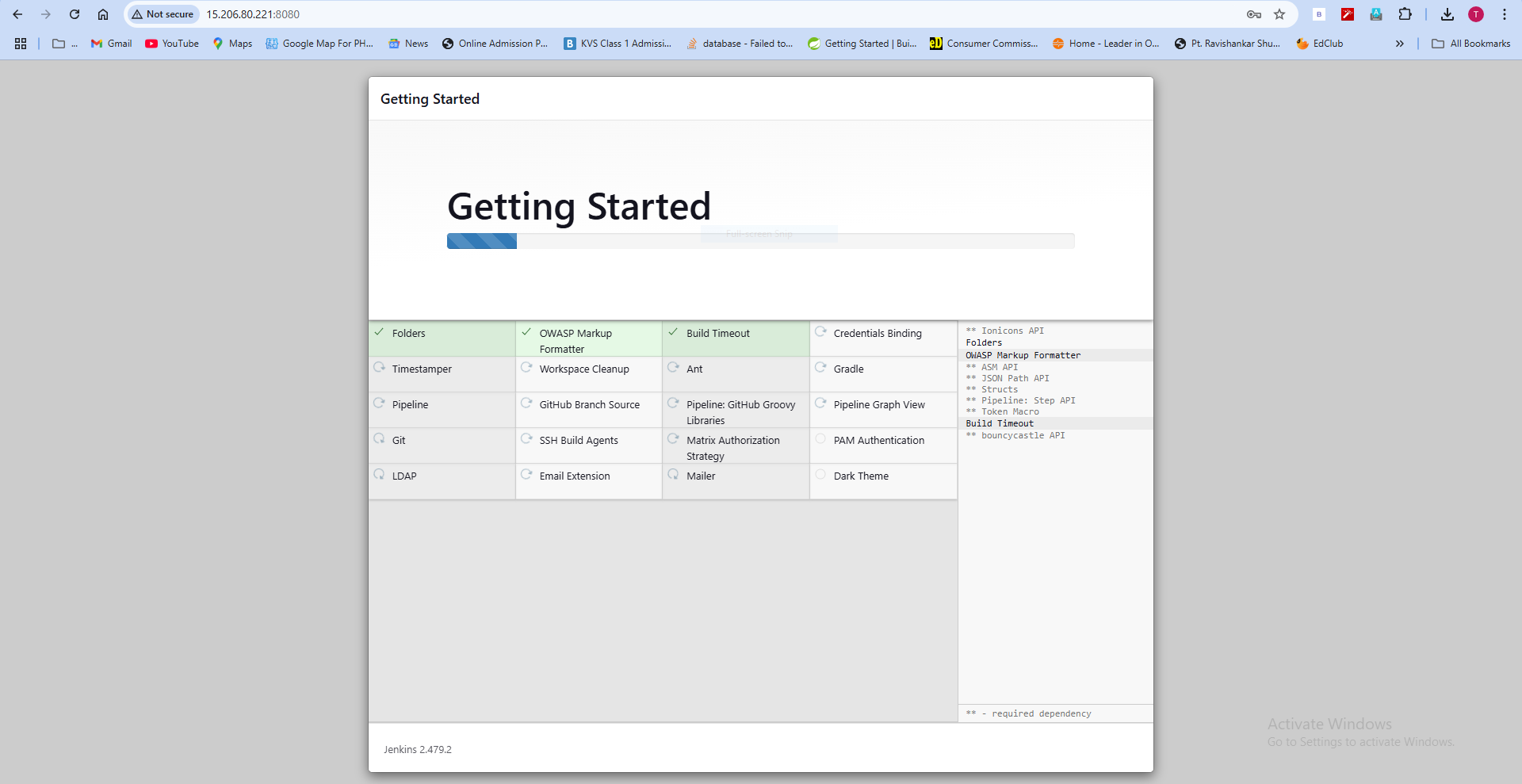


Step 4: Access Jenkins

Find the Jenkins initial admin password:

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

Access Jenkins by navigating to http://<EC2\_PUBLIC\_IP>:8080 and use the password to unlock Jenkins.



### **Essential Jenkins Plugins for CI/CD Pipeline with Kubernetes:**

To set up a robust CI/CD pipeline using **Git**, **Jenkins**, and **Kubernetes**, the following Jenkins plugins are highly recommended:

 Git Plugin

 GitHub Integration Plugin

 Pipeline Plugin

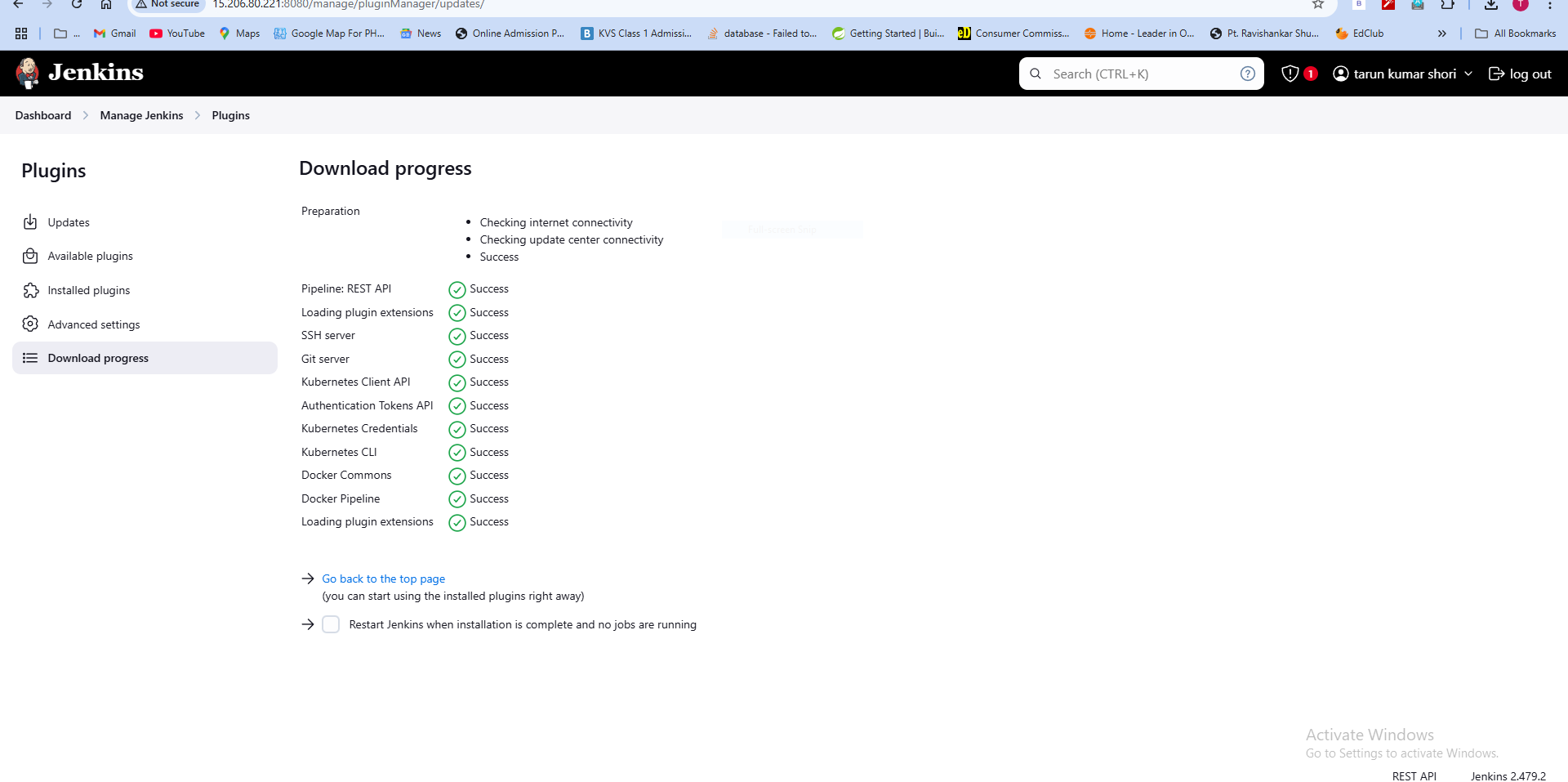
 Docker Pipeline Plugin

 Kubernetes CLI Plugin

 Kubernetes Plugin

### **Installation Tip:**

Go to **Jenkins Dashboard > Manage Jenkins > Manage Plugins**, search for these plugins, and install them. Ensure Jenkins is restarted after installation to apply changes.



Install Docker on Amazon Linux :

Docker is essential for building and running containers in the CI/CD pipeline.

Step 1: Install Docker

Run the following commands to install Docker:

sudo amazon-linux-extras install docker

sudo yum install -y docker

Step 2: Start Docker

Start and enable the Docker service:

sudo systemctl start docker

sudo systemctl enable docker

Step 3: Add the User to Docker Group (Optional)

Allow the Jenkins user to run Docker commands without sudo:

sudo usermod -aG docker ec2-user

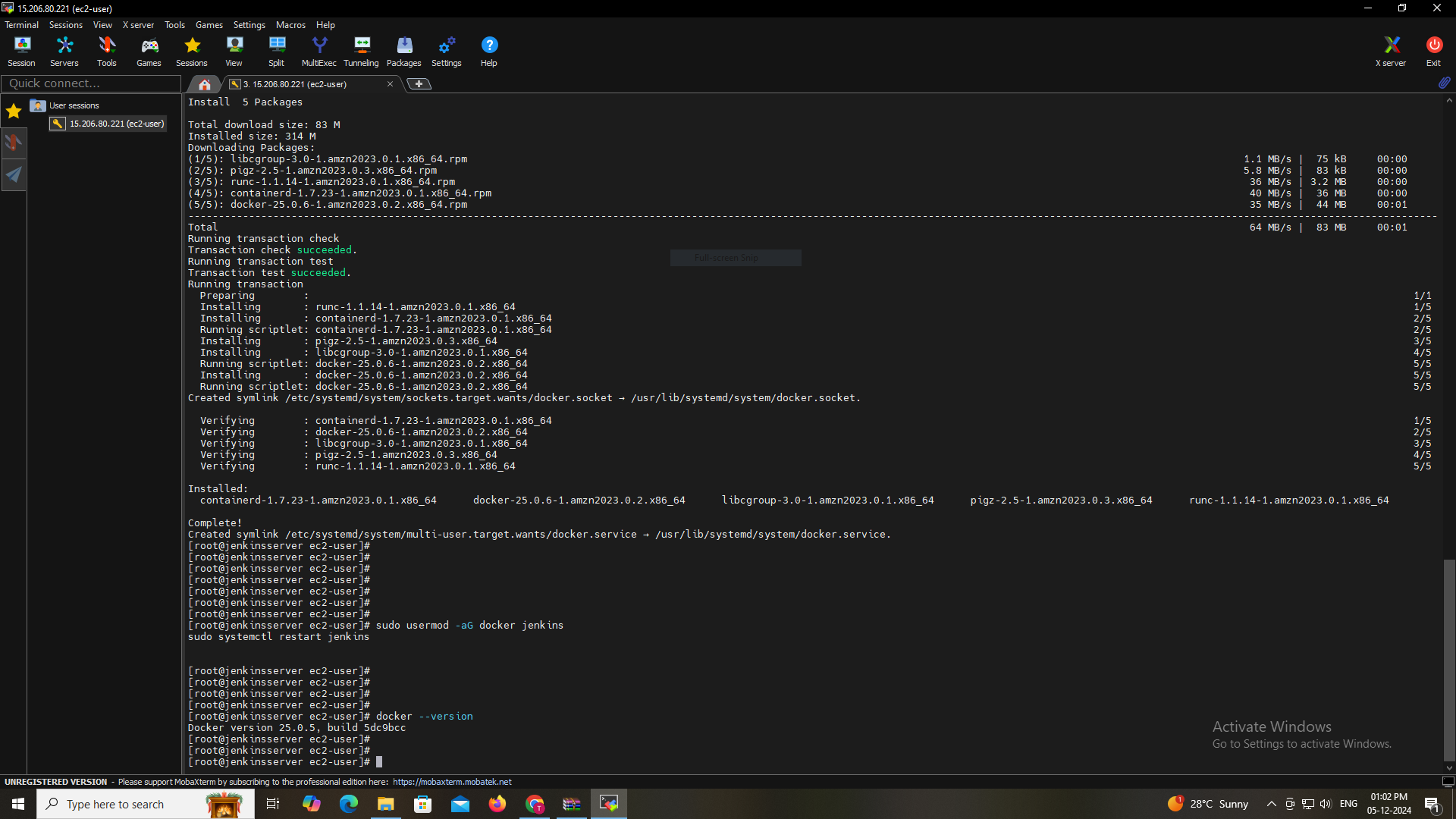
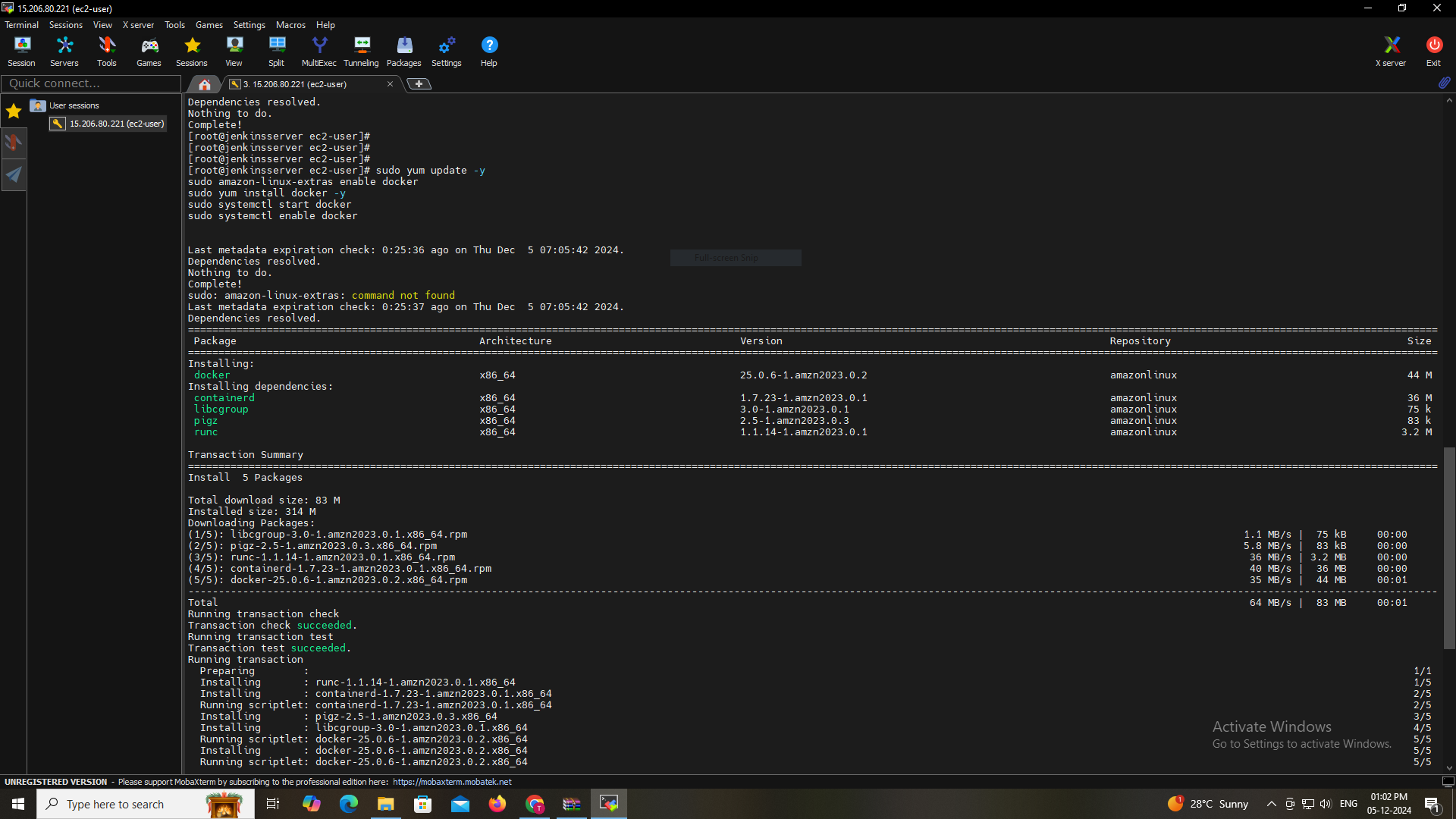
Log out and back in for changes to take effect, or use:

newgrp docker

Step 4: Verify Docker Installation

Check if Docker is installed correctly:

docker –version



Install Minikube on Amazon Linux :

You can use **Kubernetes (K8s) clusters with kubeadm control plane** or **Amazon EKS (Elastic Kubernetes Service)**. EKS is a managed Kubernetes service provided by AWS that simplifies deploying, managing, and scaling Kubernetes clusters in the cloud.

Minikube allows me to run a local Kubernetes cluster, and Docker will be the driver to manage containers.

Step 1: Install Minikube Dependencies

Minikube requires kubectl and VirtualBox (or Docker) for running the Kubernetes cluster.

Install dependencies:

sudo yum install -y conntrack

Step 2: Install Minikube

Download and install Minikube using the following command:

curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

sudo chmod +x minikube

sudo mv minikube /usr/local/bin/

Step 3: Install kubectl

Minikube requires kubectl to interact with the Kubernetes cluster.

curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl"

sudo chmod +x kubectl

sudo mv kubectl /usr/local/bin/

Step 4: Start Minikube with Docker Driver

Start Minikube with Docker as the driver:

minikube start --driver=docker

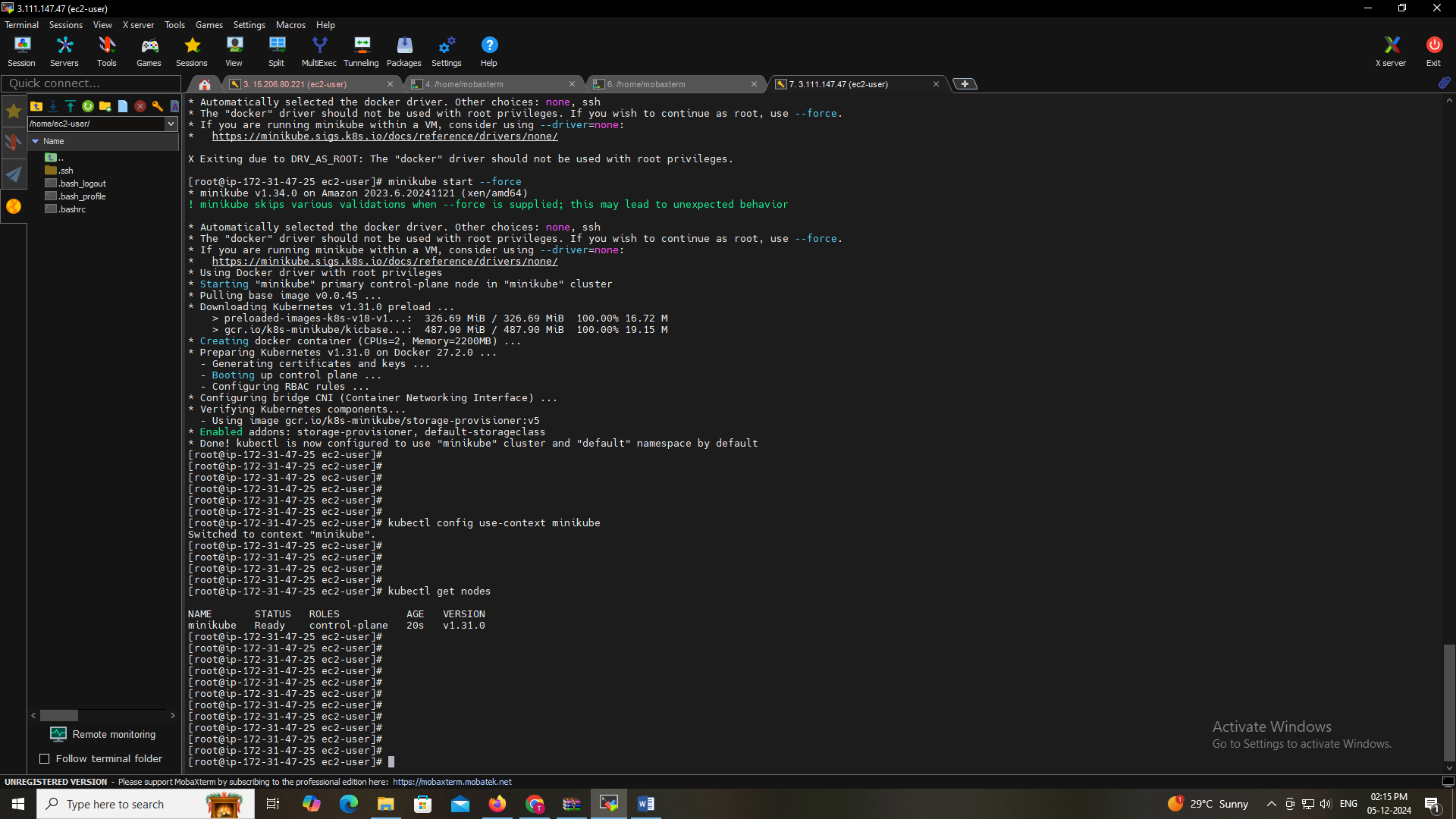
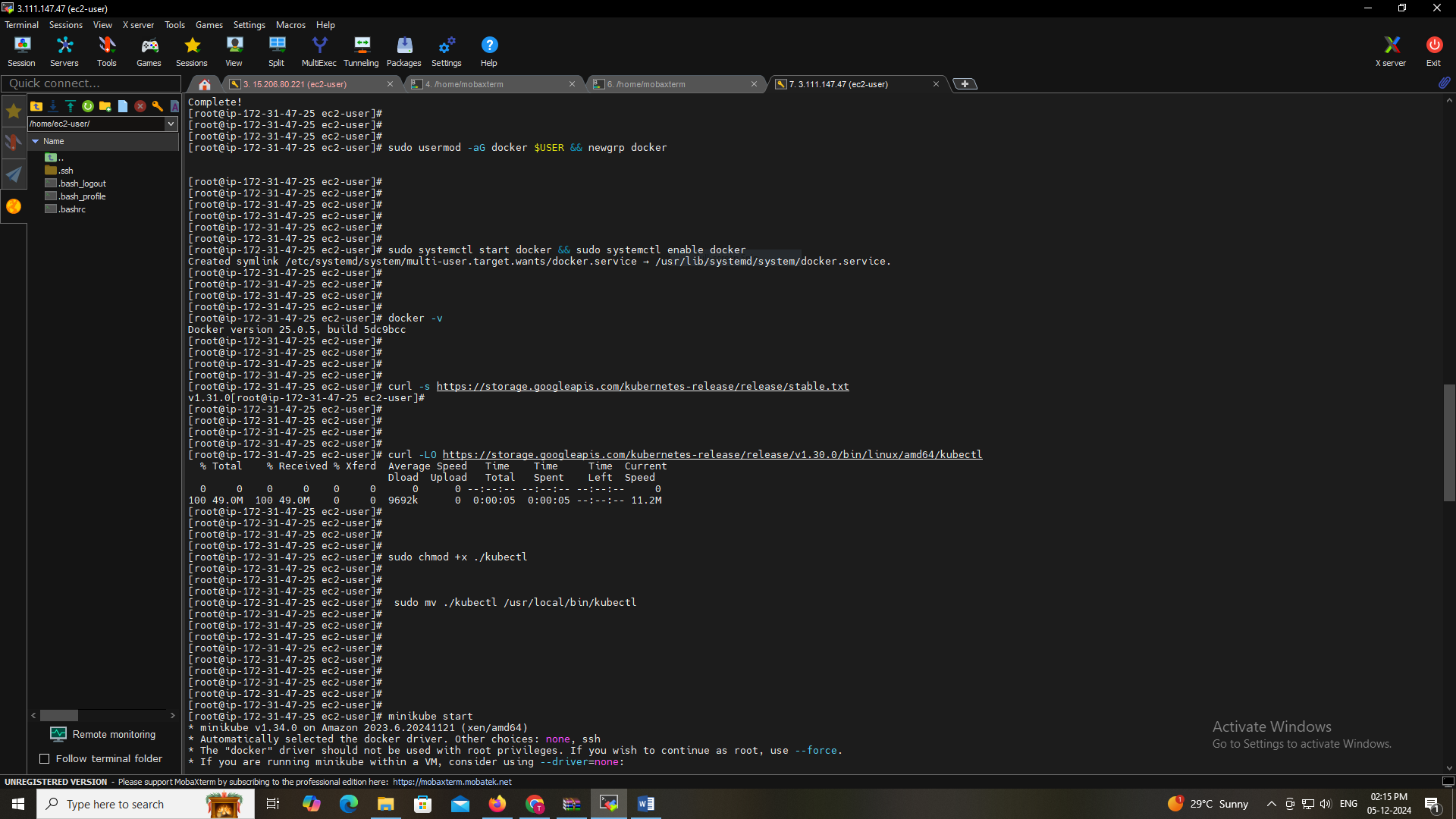
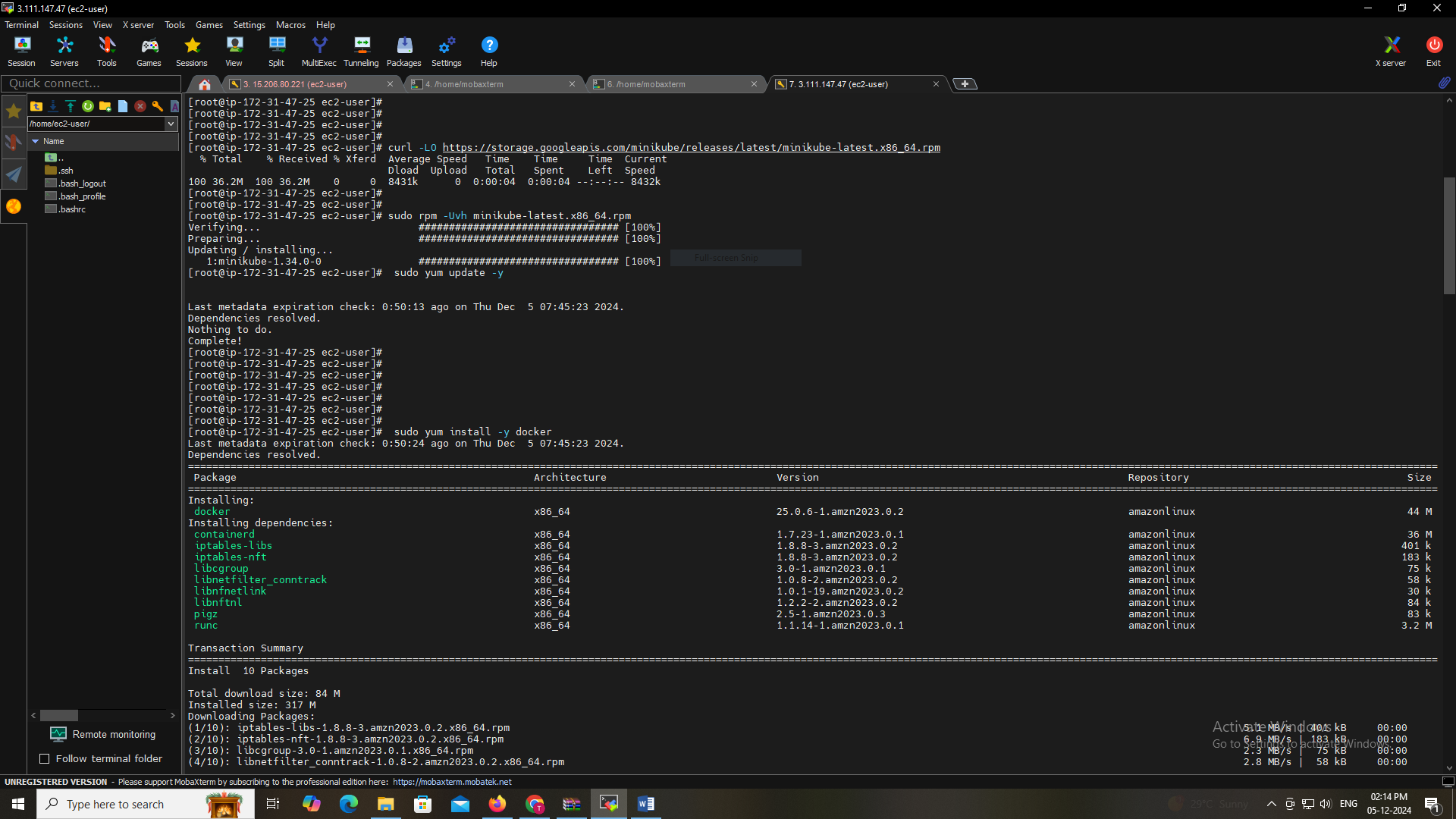
Step 5: Verify Minikube Installation

Check if Minikube is running successfully:

minikube status

You can also test if kubectl is working with Minikube:

kubectl get nodes



Create a Dockerfile for React App:

Build the React app

FROM node:18 AS build

# Set the working directory

WORKDIR /app

# Copy the package.json and package-lock.json

COPY package.json package-lock.json ./

# Install dependencies

RUN npm install

# Copy the rest of the application files

COPY . ./

# Build the React app

RUN npm run build

# Step 2: Serve the app using Nginx

FROM nginx:alpine

# Copy the build files from the build stage to the Nginx server

COPY --from=build /app/build /usr/share/nginx/html

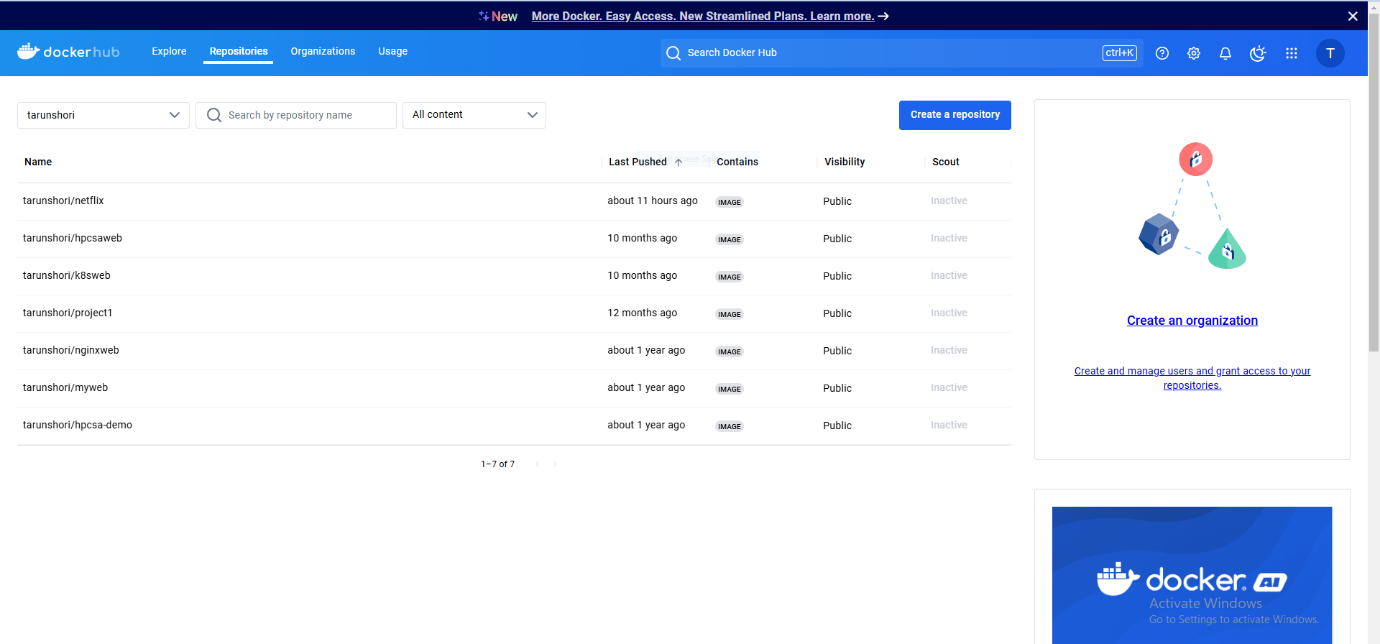
# Expose port 80 for the Nginx server

EXPOSE 80

# Start Nginx server

CMD ["nginx", "-g", "daemon off;"]

Pushed the Dockerfile to the GitHub repository.



Jenkins Pipeline Configuration (Groovy Script):

Jenkins Pipeline (Groovy Script) for automating the build and deployment of a React application. The pipeline includes steps for cloning the repository, building and pushing a Docker image, and deploying to a Kubernetes cluster.

Jenkins Pipeline Configuration (Jenkinsfile)

pipeline {

agent any

environment {

// Define environment variables for Docker image and Kubernetes config file location

DOCKER\_IMAGE = 'tarunshori/netflix:latest' // Docker image name and tag

KUBECONFIG\_FILE = '/tmp/kubeconfig' // Path to the kubeconfig file for Minikube

}

stages {

// Stage to clone the repository from GitHub

stage('Clone Repository') {

steps {

// Use Jenkins credentials to securely access the GitHub repository

withCredentials([string(credentialsId: 'git-token', variable: 'GIT\_TOKEN')]) {

sh '''

// Clean up any previous clones, then clone the repository

rm -rf Netflix-Clone

git clone --branch master https://${GIT\_TOKEN}@github.com/tarun-code/Netflix-Clone.git

cd Netflix-Clone

'''

}

}

}

// Stage to build the Docker image for the React application

stage('Build Docker Image') {

steps {

sh '''

// Change directory to the project folder and build the Docker image

cd Netflix-Clone

docker build -t ${DOCKER\_IMAGE} .

'''

}

}

// Stage to push the built Docker image to Docker Hub

stage('Push Docker Image') {

steps {

// Use Jenkins credentials to securely log into Docker Hub

withCredentials([usernamePassword(credentialsId: 'docker-credentials', usernameVariable: 'DOCKER\_USER', passwordVariable: 'DOCKER\_PASS')]) {

sh '''

// Log in to Docker Hub and push the Docker image

echo ${DOCKER\_PASS} | docker login -u ${DOCKER\_USER} --password-stdin

docker push ${DOCKER\_IMAGE}

'''

}

}

}

// Stage to deploy the React application to the Kubernetes (Minikube) cluster

stage('Deploy to Kubernetes') {

steps {

// Use credentials for the Minikube Kubernetes config file

withCredentials([file(credentialsId: 'minikube-kubeconfig', variable: 'KUBECONFIG\_FILE')]) {

sh '''

// Copy the kubeconfig file to the correct location and set appropriate permissions

cp ${KUBECONFIG\_FILE} /tmp/kubeconfig

chmod 600 /tmp/kubeconfig

// Update the deployment with the new Docker image

kubectl --kubeconfig=/tmp/kubeconfig set image deployment/netflix-clone-deployment netflix-clone=${DOCKER\_IMAGE} --record

// Wait for the rollout to complete

kubectl --kubeconfig=/tmp/kubeconfig rollout status deployment/netflix-clone-deployment

'''

}

}

}

}

// Post-actions to clean up and provide feedback on the pipeline status

post {

always {

// Clean up the kubeconfig file to ensure security

sh 'rm -f /tmp/kubeconfig'

}

success {

// Print success message if the pipeline completes without errors

echo 'Pipeline completed successfully!'

}

failure {

// Print failure message if any stage fails

echo 'Pipeline failed.'

}

}

}

Breakdown of the Pipeline:

Environment Variables:

DOCKER\_IMAGE: Specifies the Docker image name and tag (in this case, tarunshori/netflix:latest).

KUBECONFIG\_FILE: The location of the kubeconfig file for connecting to the Minikube Kubernetes cluster.

Stages:

Clone Repository: This stage clones the repository from GitHub using a GitHub token stored securely in Jenkins credentials. It pulls the latest master branch of the Netflix-Clone repository.

Build Docker Image: This stage builds a Docker image from the Dockerfile present in the project directory (Netflix-Clone).

Push Docker Image: Pushes the built Docker image to Docker Hub. Credentials for Docker Hub are stored in Jenkins and used securely to log in.

Deploy to Kubernetes: Deploys the newly built Docker image to a Kubernetes cluster (in this case, Minikube). The kubeconfig file for Minikube is used to authenticate and set the correct Kubernetes context for deployment. The image is updated in the netflix-clone-deployment and the status of the deployment rollout is checked.

Post-Actions:

Always: The kubeconfig file is removed to ensure security.

Success: If all stages succeed, a success message is logged.

Failure: If any stage fails, a failure message is logged.

Prerequisites:

Jenkins Credentials:

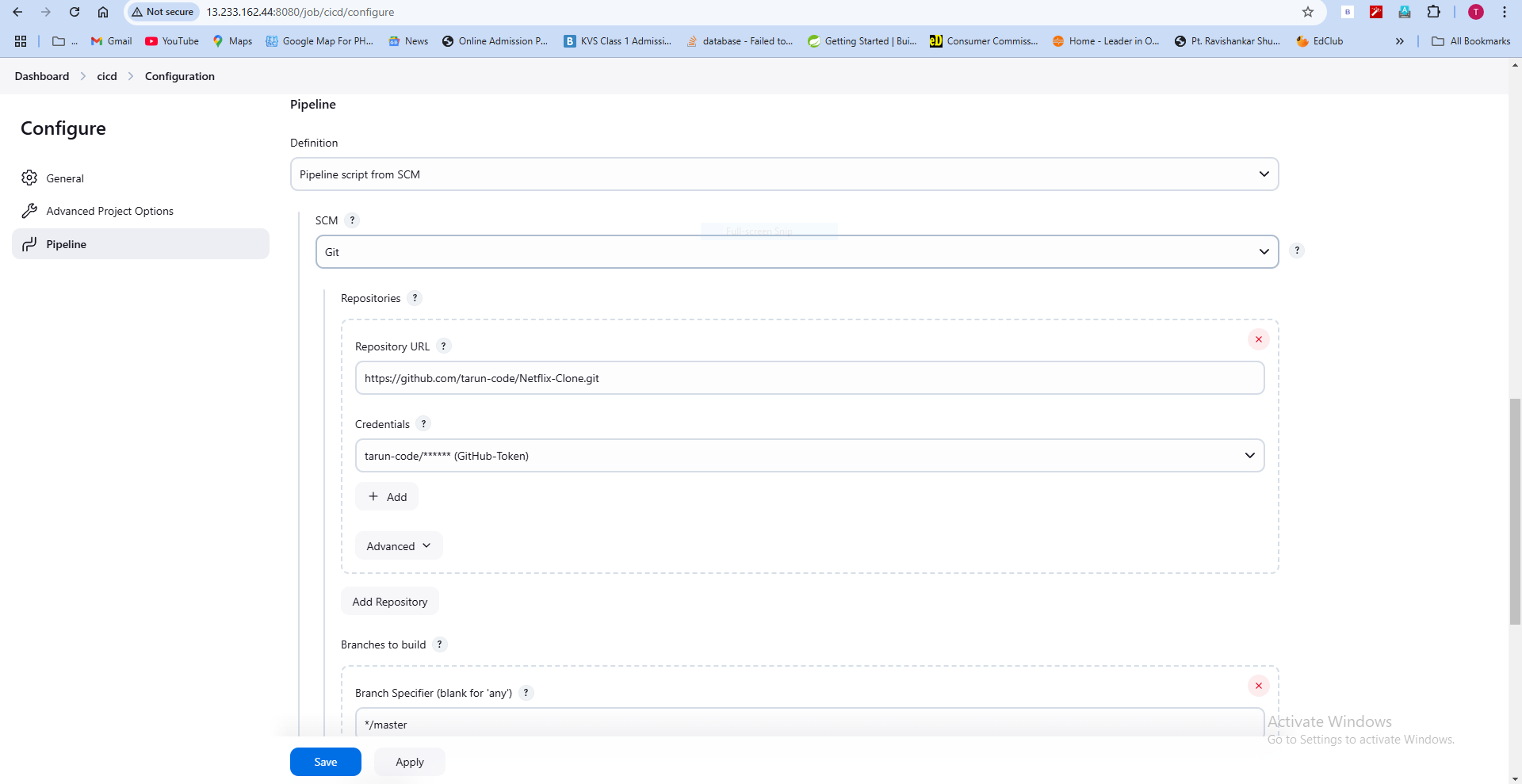
git-token: A GitHub token for cloning repositories.

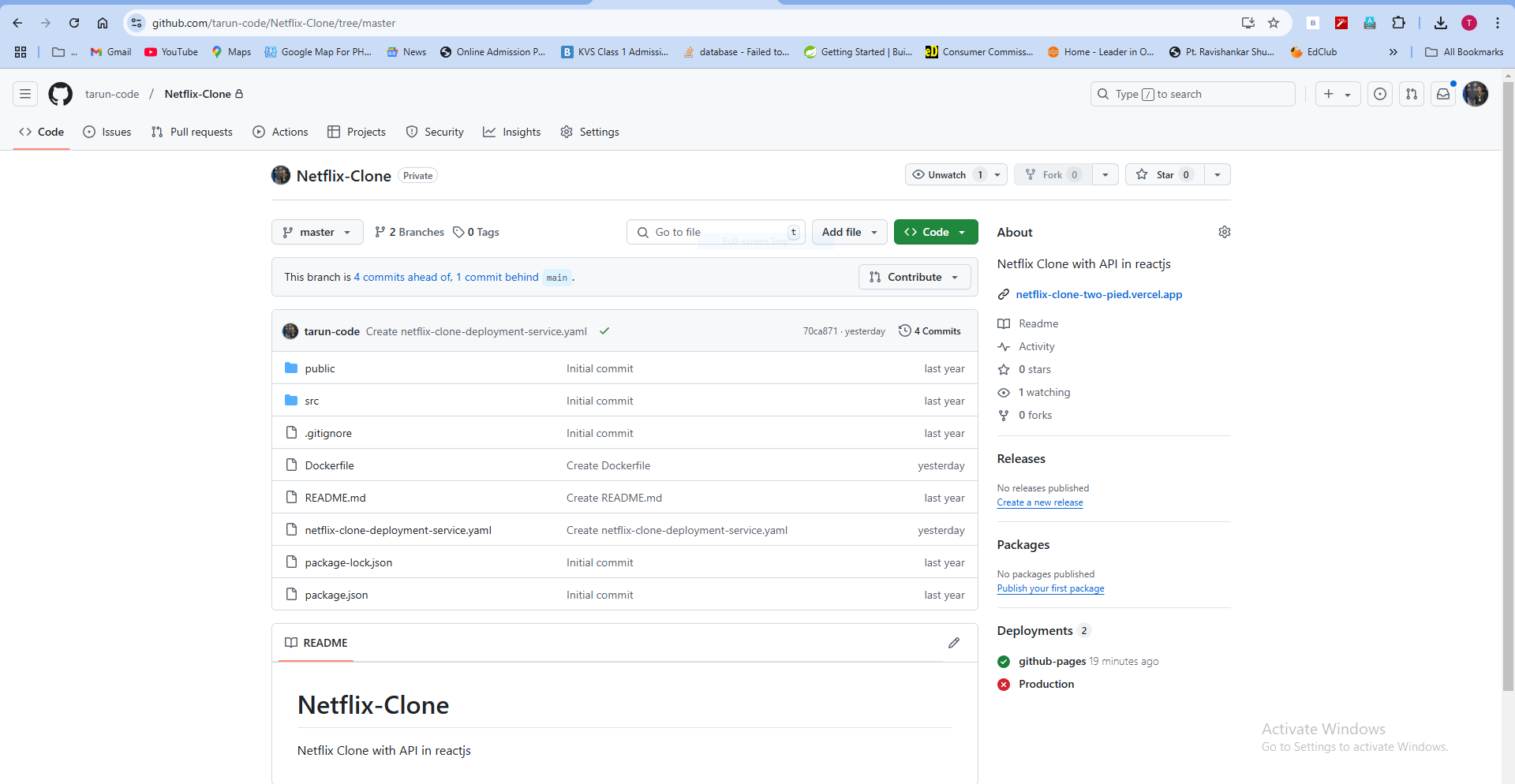
docker-credentials: Docker Hub username and password.

minikube-kubeconfig: The kubeconfig file for accessing the Minikube Kubernetes cluster.

Minikube Setup: Minikube should be running on the Jenkins machine with the proper kubeconfig file set up.

Docker Hub Account: You should have a Docker Hub account and a repository to push the image.





GitHub Webhook Integration :

Test Case 1: Validate Webhook Trigger

Test Steps:

Make a commit and push to the GitHub repository.

Check Jenkins for an automatic build trigger.

Expected Result: Jenkins build should trigger automatically on every push.

Security and Credentials Management

Test Case 1: Validate GitHub Credentials Security

Test Steps:

Check GitHub credentials stored in Jenkins.

Expected Result: GitHub credentials should be securely stored in Jenkins.

Test Case 2: Validate Docker Hub Login

Test Steps:

Check Docker Hub login using Jenkins pipeline credentials.

Expected Result: Docker Hub login should succeed, and the image should be pushed.

Test Case 3: Validate Kubernetes Kubeconfig Security

Test Steps:

Verify if the kubeconfig file is removed after deployment.

Expected Result: Jenkins should delete the kubeconfig file after deployment.

Create Kubernetes Deployment YAML:

Created a Kubernetes deployment YAML file for deploying the React application:

apiVersion: apps/v1

kind: Deployment

metadata:

name: netflix-clone-deployment

labels:

app: netflix-clone

spec:

replicas: 2

selector:

matchLabels:

app: netflix-clone

template:

metadata:

labels:

app: netflix-clone

spec:

imagePullSecrets:

name: my-docker-registry-secret

containers:

name: netflix-clone

image: tarunshori/netflix:latest

ports:

containerPort: 80

resources:

requests:

memory: "256Mi"

cpu: "500m"

limits:

memory: "512Mi"

cpu: "1"

---

apiVersion: v1

kind: Service

metadata:

name: netflix-clone-service

spec:

externalTrafficPolicy: Local

type: NodePort # Change this line from ClusterIP to NodePort

selector:

app: netflix-clone

ports:

protocol: TCP

port: 80

targetPort: 80

nodePort: 30001 # Specify the NodePort range (usually 30000-32767)

To apply a YAML file in Kubernetes, use the following kubectl apply command:

kubectl apply -f <filename>.yaml



The command you provided creates a Kubernetes Docker registry secret:

kubectl create secret docker-registry my-docker-registry-secret \

--docker-server=https://my-private-repo \

--docker-username=your-username \

--docker-password=your-password \

--docker-email=your-email

Configure GitHub Webhook:

Configured GitHub webhook to trigger Jenkins builds on code changes in the Git repository.

In GitHub, navigate to the repository, go to Settings > Webhooks, and add the Jenkins URL with /github-webhook/ as the payload URL.

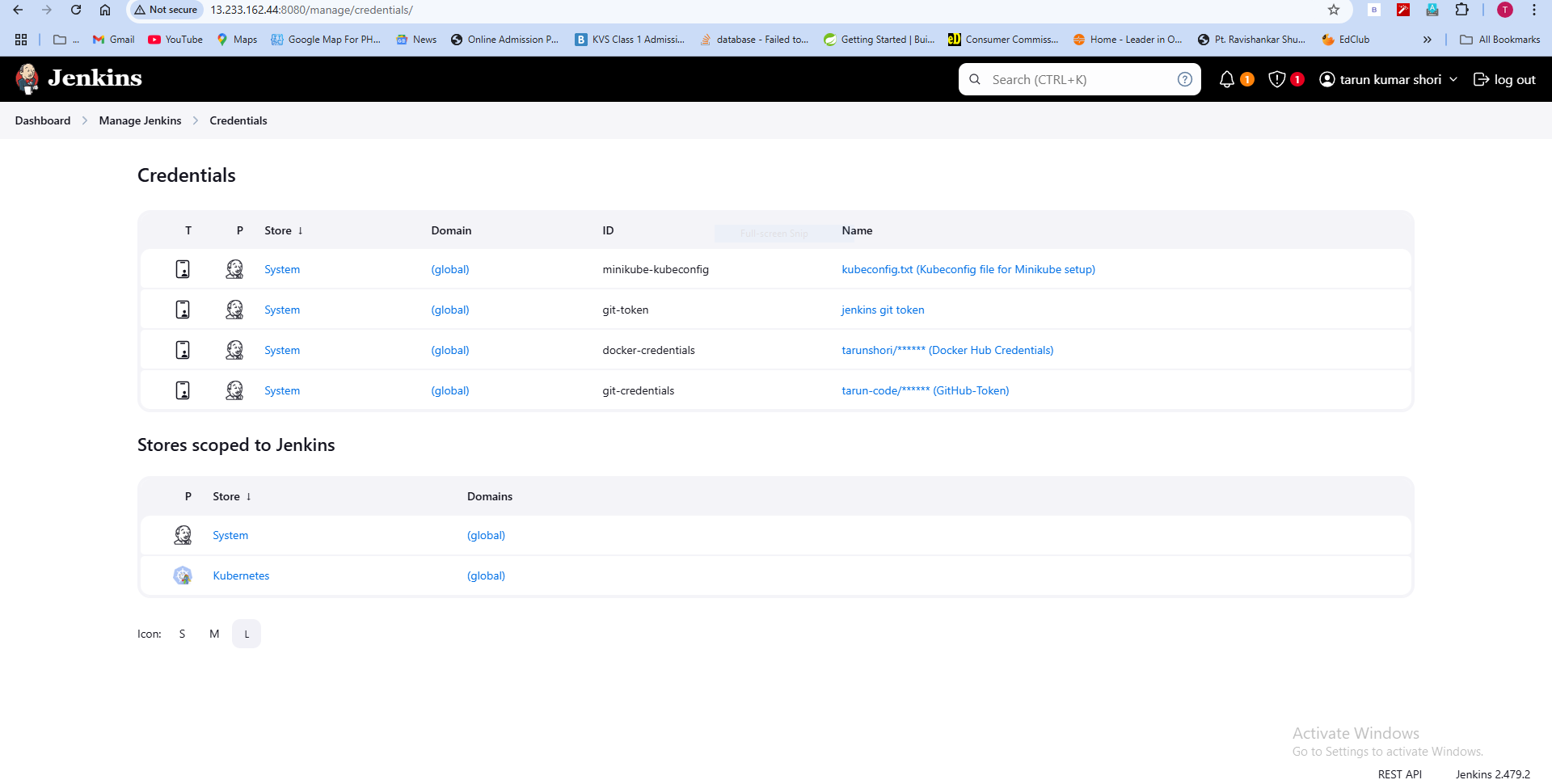
Pipeline Security with Credentials:

Secured Jenkins with credentials:

GitHub credentials: For cloning the Git repository.

Docker Hub credentials: For pushing the Docker image.

Kubeconfig credentials: For accessing the Kubernetes cluster.



Jenkins Pipeline Execution :

Test Case 1: Validate Jenkins Build Trigger

Test Steps:

Push a code change to the GitHub repository.

Check if Jenkins automatically triggers a build.

Expected Result: The build should start automatically.

Test Case 2: Validate Build Stages Execution

Test Steps:

Monitor the Jenkins build console.

Validate successful execution of all stages (Clone, Build Docker Image, Push Docker Image, Deploy to Kubernetes).

Expected Result: Each stage should complete without errors.

Test Case 3: Validate Docker Hub Image Push

Test Steps:

Check Docker Hub for the updated image version.

Expected Result: The image tarunshori/netflix:latest should be pushed successfully.

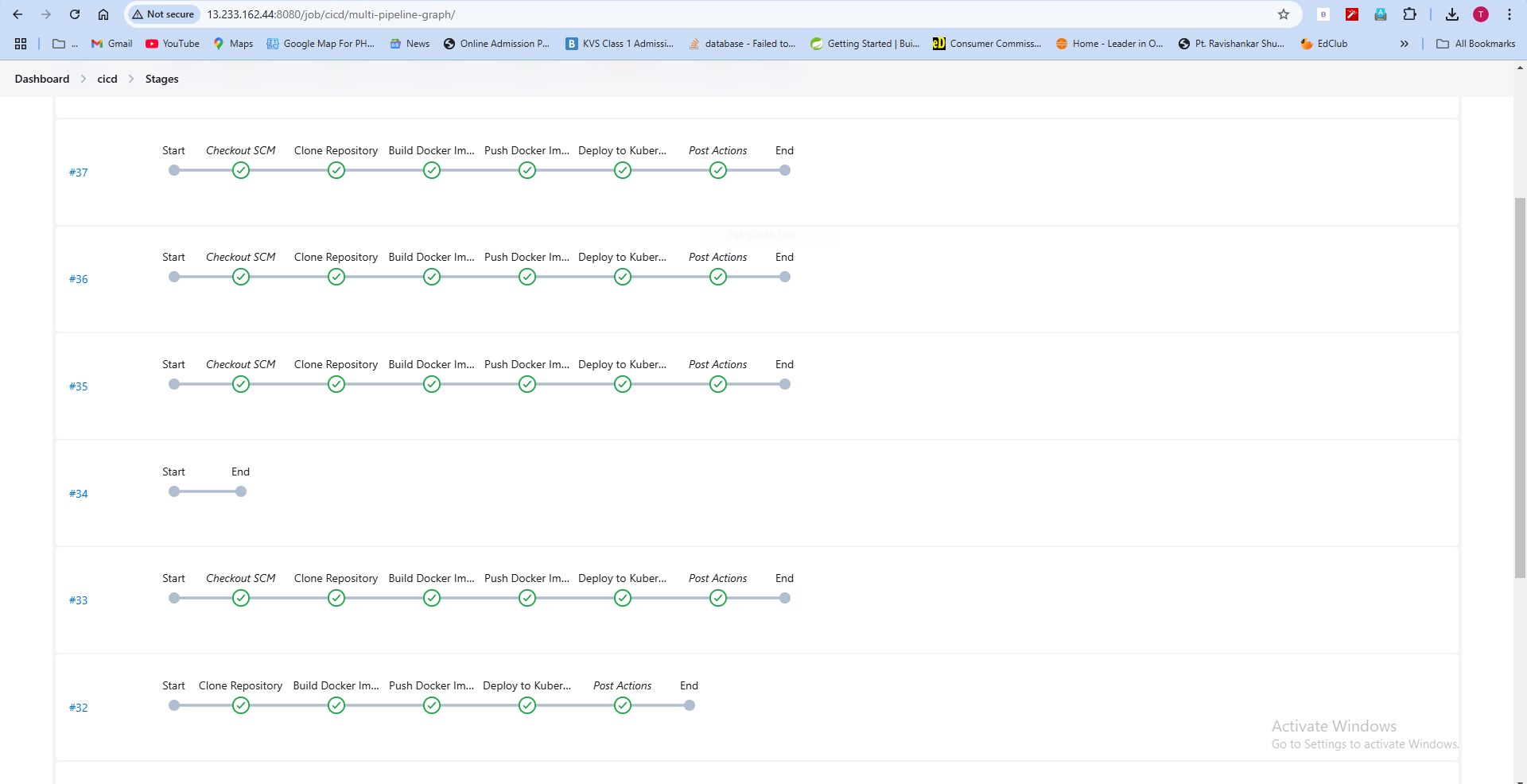
Test Case 4: Validate Application Deployment Update

Test Steps:

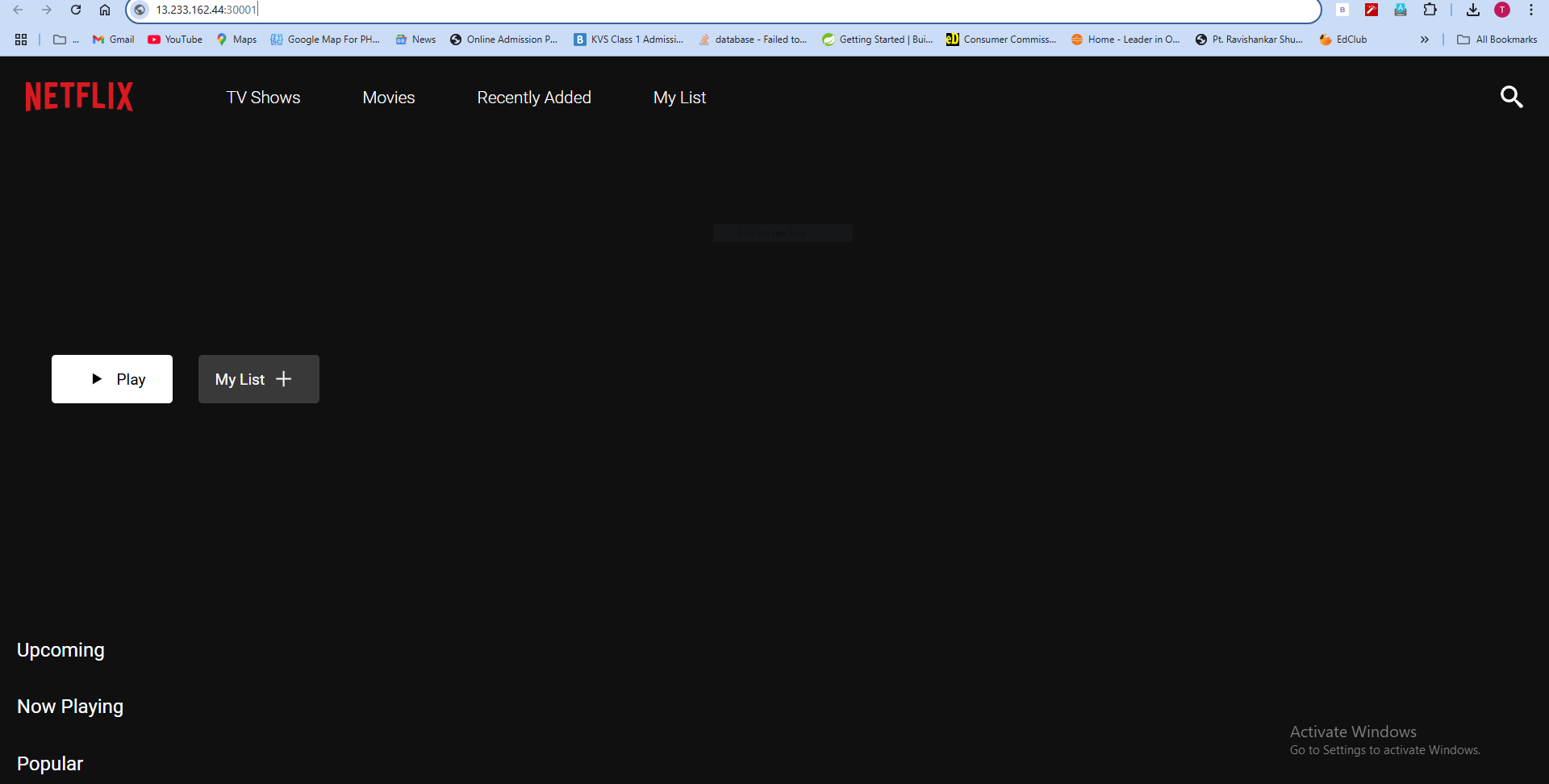
Deploy the updated Docker image using kubectl set image ....

Check if the Kubernetes deployment is updated using kubectl get pods.

Expected Result: Deployment should update successfully, and rollout status should indicate success.



Test Case 5: Check the web app is Running or accessible



### Security Best Practices:

1. **Use Encrypted Credentials**:
   * **Jenkins Credentials Plugin**: Store sensitive data like API keys, Docker credentials, or AWS keys in Jenkins' built-in credentials store. Use environment variables or bind them in pipeline steps to avoid exposing sensitive data in code or logs.
   * **Secret Management**: Consider using secret management solutions like **HashiCorp Vault** or **AWS Secrets Manager** to manage and inject secrets into Jenkins jobs securely.
2. **Role-Based Access Control (RBAC)**:
   * **Jenkins RBAC**: Use the **Role-based Authorization Strategy** plugin to assign permissions at granular levels (e.g., read, write, and execute permissions) for different Jenkins users or groups.
   * **Kubernetes RBAC**: Define fine-grained access control policies in Kubernetes by creating **Role** and **RoleBinding** or **ClusterRole** and **ClusterRoleBinding** to restrict access to Kubernetes resources.
3. **Private Registries**:
   * Ensure that any Docker images used for builds and deployments come from private, trusted registries (like **Docker Hub Private Repositories**, **AWS ECR**, or **Google Container Registry**) and that you authenticate using encrypted credentials stored in Jenkins.
   * Use **Docker Content Trust** to ensure the authenticity of images.

### Monitoring & Logging Recommendations:

1. **Jenkins Build Monitor Plugin**:
   * **Jenkins Build Monitor Plugin**: This plugin provides a detailed view of the Jenkins build process, including the status of various jobs and their history. It allows for easy identification of build failures and helps in monitoring the pipeline health.
2. **Kubernetes Monitoring Tools**:
   * **Prometheus**: Deploy **Prometheus** on your Kubernetes cluster for monitoring the health and performance of Kubernetes resources (pods, nodes, services) and applications. Set up alerts for key metrics like resource usage, pod failures, and deployment health.
   * **Grafana**: Use **Grafana** to visualize data from Prometheus and provide interactive dashboards. Grafana offers pre-built dashboards for Kubernetes monitoring, helping you track cluster and application health effectively.
   * **Kube-state-metrics**: This is an add-on for Kubernetes that provides detailed metrics about the state of Kubernetes resources, useful for monitoring application deployments and infrastructure health.
3. **Logging**:
   * **ELK Stack** (Elasticsearch, Logstash, Kibana) or **EFK Stack** (Elasticsearch, Fluentd, Kibana): Set up centralized logging for your Kubernetes and Jenkins logs. This helps you aggregate logs for easy querying and troubleshooting.
   * **Loki** with **Grafana**: An alternative to ELK for Kubernetes logs, **Loki** integrates with **Grafana** to provide efficient log aggregation and visualization.

## **Jenkins Local Monitoring and Logging:**

### **Jenkins Build Logs**

* Jenkins provides built-in logging for each job and build, which can be accessed from the Jenkins dashboard.
* Each job has its own log output, which includes information about the execution of each step in the pipeline.
* You can access the logs by clicking on the specific build, and the logs are displayed under the "Console Output" section.

### **Jenkins System Log**

* Jenkins also has a **System Log** that logs general Jenkins activities. This can be accessed from **Manage Jenkins > System Log**.
* It provides detailed logging of Jenkins operations such as node communication, security events, and errors.

### **Jenkins Log Storage**

* To persist Jenkins logs for future reference or debugging, you can configure Jenkins to write logs to a persistent storage.
* You can use **log rotation** settings in Jenkins to manage the size and duration of the logs.

### **Jenkins Monitoring Plugins**

* **Build Monitor Plugin**: The **Build Monitor Plugin** provides a simple and visual overview of Jenkins jobs' status, which helps in monitoring build health.
* **Monitoring Plugin**: The **Monitoring Plugin** provides detailed statistics and metrics for Jenkins instances, including memory usage, CPU usage, and more. It can be installed through the Jenkins Plugin Manager.

## **2. Kubernetes Local Monitoring and Logging**

### **Kubernetes Cluster Logs**

* Kubernetes maintains logs for each pod in the cluster. You can access these logs using the kubectl logs command.

kubectl logs <pod-name> --namespace=<namespace>

This will display the logs of the specified pod. You can use options like --follow to stream logs in real time.

* **Kubectl Log Aggregation**: In a production environment, you would typically use log aggregation systems (e.g., Elasticsearch, Fluentd, and Kibana stack - **EFK** or **ELK**) for centralizing logs from all Kubernetes pods.

### **Kubernetes Metrics Server**

* **Metrics Server**: This is a lightweight, short-term storage metrics provider for Kubernetes. It collects resource usage data such as CPU and memory usage, which can be accessed through kubectl top.

kubectl top pod --namespace=<namespace>

### **Kubernetes Dashboard**

* **Kubernetes Dashboard**: You can use the Kubernetes Dashboard to get a graphical view of your Kubernetes cluster's state.
  + The dashboard shows details of pods, services, deployments, and other resources within your cluster.
  + It can be installed via Kubernetes manifests.

kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml

## **3. Local CI/CD Pipeline Monitoring**

* **Jenkins Job Monitoring**: Locally monitor your Jenkins jobs' health, performance, and status using the Jenkins Job Builder plugin or through the Jenkins dashboard.

Documentation and References:

1. Jenkins Documentation

Link: <https://www.jenkins.io/doc/>

1. Docker Documentation

Link: <https://docs.docker.com/>

1. Kubernetes Documentation

Link: <https://kubernetes.io/docs/>

1. GitHub Webhooks

Link: <https://docs.github.com/en/developers/webhooks-and-events/webhooks/creating-webhooks>

1. GitHub Repository

Link: <https://github.com/tarun-code/Netflix-Clone>

Conclusion:

By following these steps, I have successfully set up a CI/CD pipeline that automates the build and deployment of a React application. The process is triggered automatically by a Git commit, builds a Docker image, pushes it to Docker Hub, and deploys it to a Kubernetes cluster. Security has been ensured with credentials for GitHub, Docker Hub, and Kubernetes access. This solution is scalable, adaptable, and can be easily integrated with other Git repositories and Kubernetes clusters.