Case Study Report: DevOps CI/CD Pipeline Implementation Using Git, Jenkins, Maven, Docker, Ansible, and Kubernetes

1.Introduction:

1. Overview of DevOps, CI, and CD

DevOps

DevOps is a collaborative approach that integrates software development (Dev) and IT operations (Ops). The goal is to shorten the development lifecycle and deliver high-quality software continuously. It emphasizes automation, collaboration, continuous feedback, and monitoring.

Continuous Integration (CI)

CI is the practice of regularly merging code changes into a shared repository. Each merge triggers an automated build and test process, helping identify issues early in the development cycle.

Continuous Deployment/Delivery (CD)

- Continuous Delivery ensures that the application is always in a deployable state, and deployments can happen on demand with minimal manual steps.
- Continuous Deployment goes a step further by automating the entire release process—deploying every code change that passes tests directly to production.

2. Importance of CI/CD in Modern Software Development

In today's fast-paced digital world, businesses demand faster innovation and rapid delivery of features. CI/CD addresses this need by:

- Automating repetitive tasks like testing and deployment.
- Reducing integration issues and release time.

- Increasing code quality through early bug detection.
- Enabling faster feedback loops and continuous improvement.

It's a core component of Agile and DevOps practices, critical for scalable and reliable software delivery.

3. Objective of the Case Study

The objective of this case study is to:

- Design and implement a fully automated DevOps pipeline.
- Integrate key tools for source control, build automation, configuration management, containerization, and orchestration.
- Demonstrate end-to-end automation of the software delivery lifecycle—from code commit to deployment on a Kubernetes cluster.

4. Tools and Technologies Overview

Tool	Purpose
Git/GitHub	Source code version control
Jenkins	CI/CD automation server
Maven	Build automation for Java projects
Tomcat	Web application deployment server
Docker	Containerization of applications
Ansible	Configuration management and automation
Kubernetes	Container orchestration platform

2. What Do We Cover

This case study and associated material provide a comprehensive guide to understanding and implementing a DevOps pipeline using Continuous Integration and Continuous Deployment (CI/CD). Below are the key topics covered:

Fundamentals of CI/CD in the DevOps Lifecycle

- **Continuous Integration (CI):** Learn how developers can frequently merge their code changes into a shared repository. This practice involves automated builds and tests to validate the new code and ensure it integrates smoothly with the existing codebase.
- **Continuous Deployment/Delivery (CD):** Understand the mechanisms that automatically deploy validated code changes to staging or production environments. The difference between delivery (manual trigger to deploy) and deployment (fully automated deployment) is explained to highlight automation levels.
- **DevOps Lifecycle**: Explore how CI/CD fits within the broader DevOps lifecycle, which includes planning, coding, building, testing, releasing, deploying, operating, and monitoring software applications continuously.

Required Tools and Resources to Build a CI/CD Pipeline

- **Version Control Systems:** Introduction to tools like Git and platforms like GitHub that store and manage source code.
- **Build Automation Tools:** Tools like Maven for compiling code, running tests, and packaging applications automatically.
- **CI/CD Servers**: Jenkins, GitHub Actions, or similar tools that orchestrate the pipeline by triggering builds, tests, and deployments.
- **Containerization Platforms**: Docker for creating consistent environments across development, testing, and production.
- **Configuration Management Tools**: Ansible and similar tools that automate software installation and environment configuration.
- **Container Orchestration**: Kubernetes to manage deployment, scaling, and operations of containerized applications.
- **Cloud Providers**: AWS, Azure, or Google Cloud for hosting and scaling infrastructure.

- Along with these, other supporting resources like monitoring tools and infrastructure-as-code frameworks are introduced to maintain pipeline health and scalability.

Resources to Set Up DevOps CI/CD Pipeline Using AWS Cloud Sandbox

1. AWS Free Tier Sandbox: Provides a cost-free environment to experiment with AWS services safely without impacting production.
2. Compute Resources (EC2 & EKS): Virtual servers (EC2) and managed Kubernetes clusters (EKS) to run build servers, deploy applications, and orchestrate containers.
3. Storage Services :we are using mobaxterm for enabling services
4. Identity and Access Management (IAM): Controls secure access permissions for CI/CD tools interacting with AWS resources.
5. Infrastructure Automation: Integrated with Jenkins/Ansible for automated provisioning and deployment of infrastructure.

DevOps Project Source Code

The source code for this DevOps project is sourced from the GitHub repository maintained by **yankils** (github.com/yankils/hello-world). This repository serves as a practical example for implementing a complete CI/CD pipeline and covers the following key aspects:

- **Codebase**: A sample Java-based web application with structured modules including server, webapp, and deployment configurations.
- **Build Automation**: Uses Maven (pom.xml) for managing dependencies and automating the build process.
- **Containerization**: Includes Dockerfiles to create container images for the application.
- **Deployment Automation**: Contains Ansible playbooks for automating deployment and configuration tasks.
- **Pipeline Integration**: Easily integrates with Jenkins for continuous integration and deployment workflows.

By working with this repository, learners get hands-on experience in setting up, building, testing, containerizing, and deploying an application using modern DevOps practices.

CI/CD Pipeline Using Git, Jenkins, and Maven

1. Section Introduction

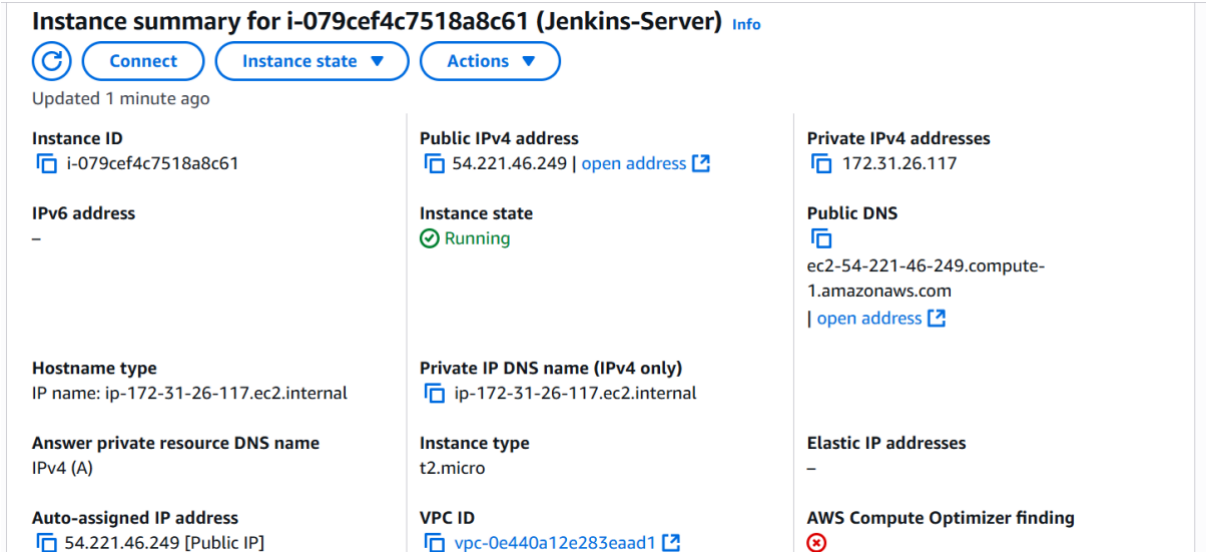
- Overview of building a Continuous Integration/Continuous Deployment pipeline using popular tools: Git for version control, Jenkins as the automation server, and Maven for building Java projects.
- Explanation of how these tools integrate to automate code building, testing, and deployment steps in a DevOps lifecycle.
- Importance of CI/CD in enabling faster and reliable software delivery.

2. Update about New Amazon Linux Version

- Introduction to the latest Amazon Linux version used as the operating system for Jenkins and build servers.
- Benefits such as improved security, performance, and compatibility with cloud environments like AWS.
- Steps to update or install Amazon Linux on EC2 instances for the DevOps pipeline.

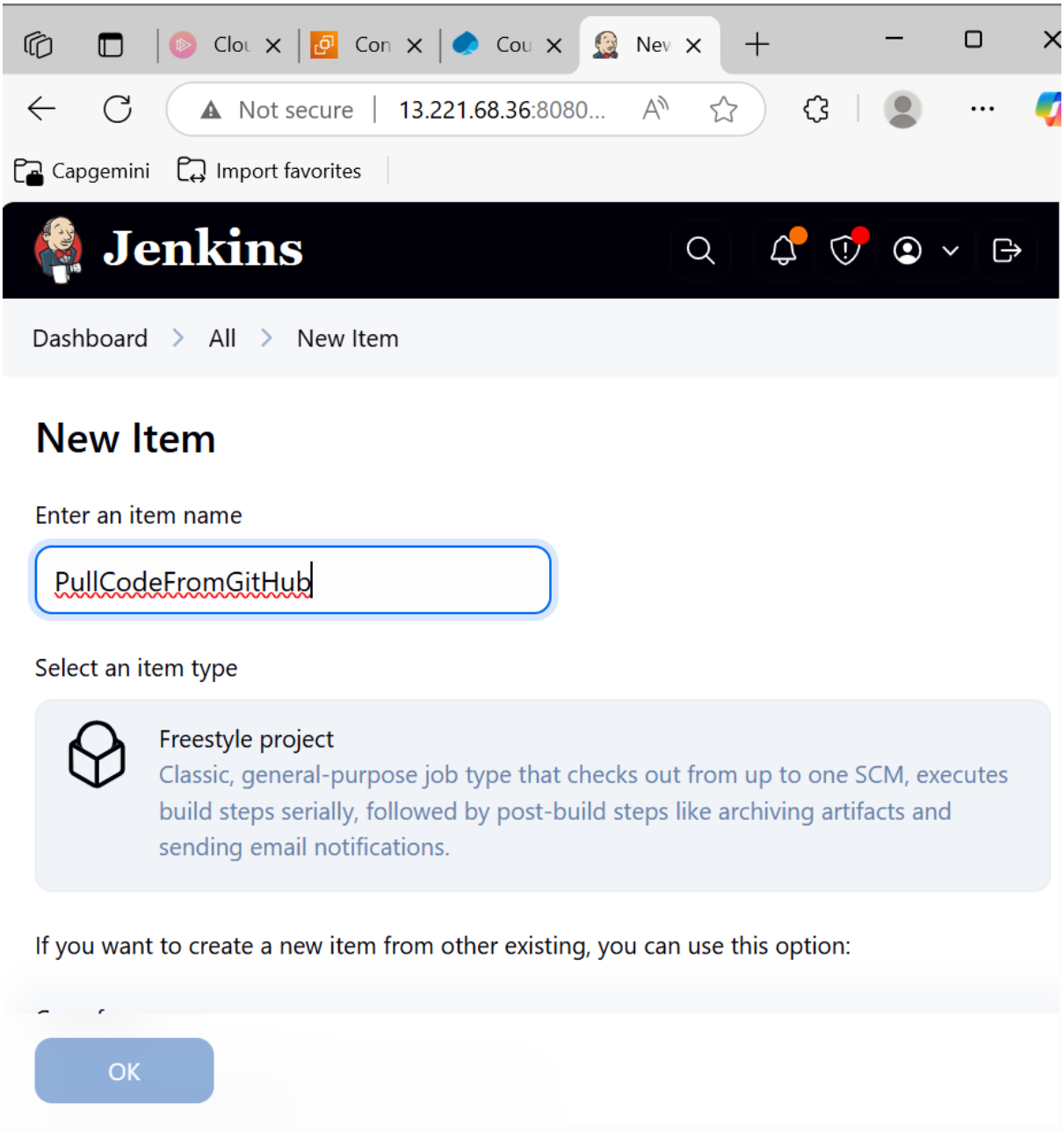
3. Setup Jenkins Server

- Installing Jenkins on the Amazon Linux server using official repositories or packages.
- Configuring Jenkins for first-time setup including admin user, plugins installation, and system settings.
- Ensuring Jenkins service is running and accessible through web UI.



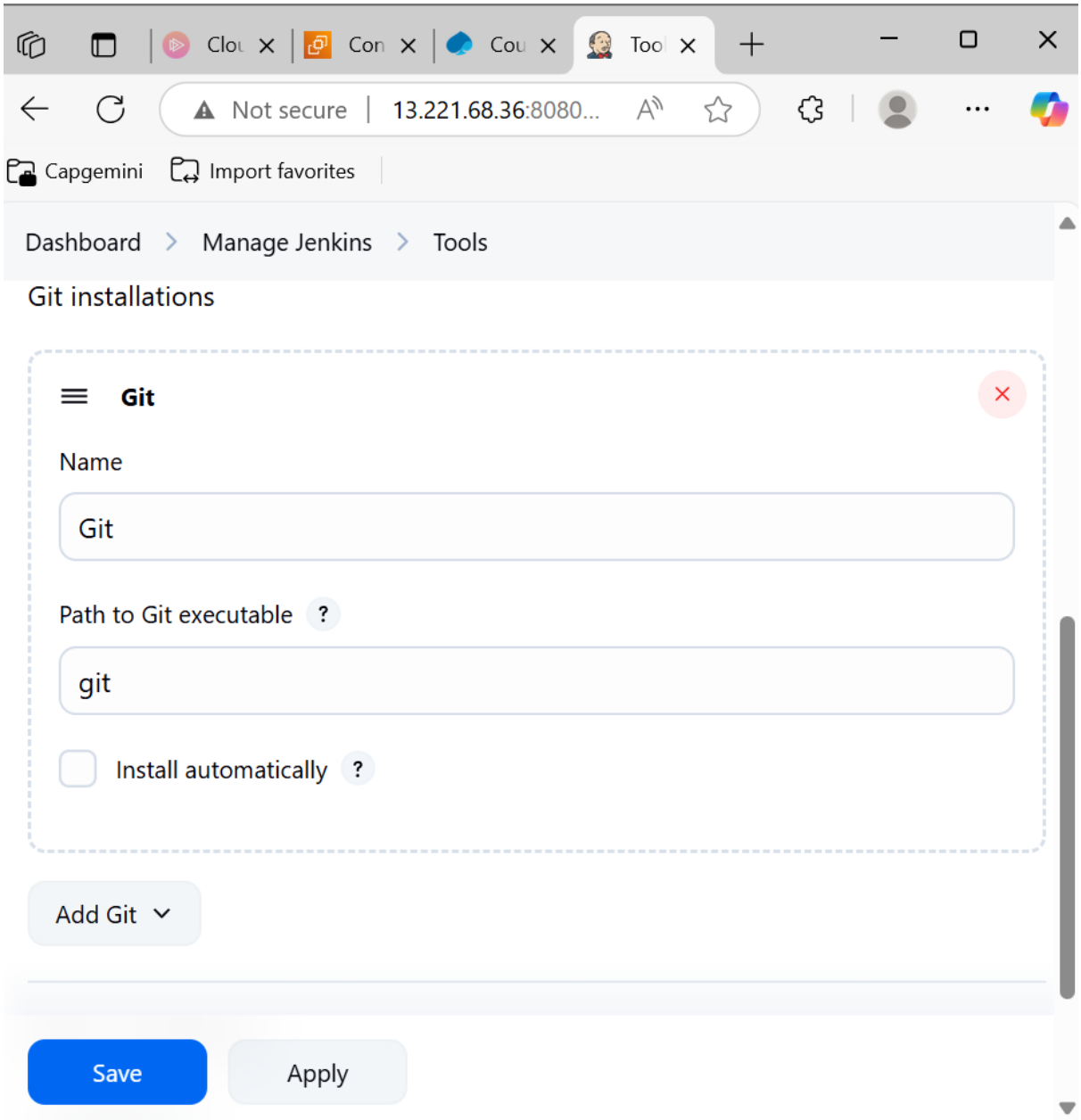
4. Run 1st Jenkins Job

- Creating a simple Jenkins freestyle job to verify Jenkins installation.
- Configuring job parameters such as source code repository URL, build triggers, and build steps.
- Executing the job and reviewing build logs to confirm successful execution.



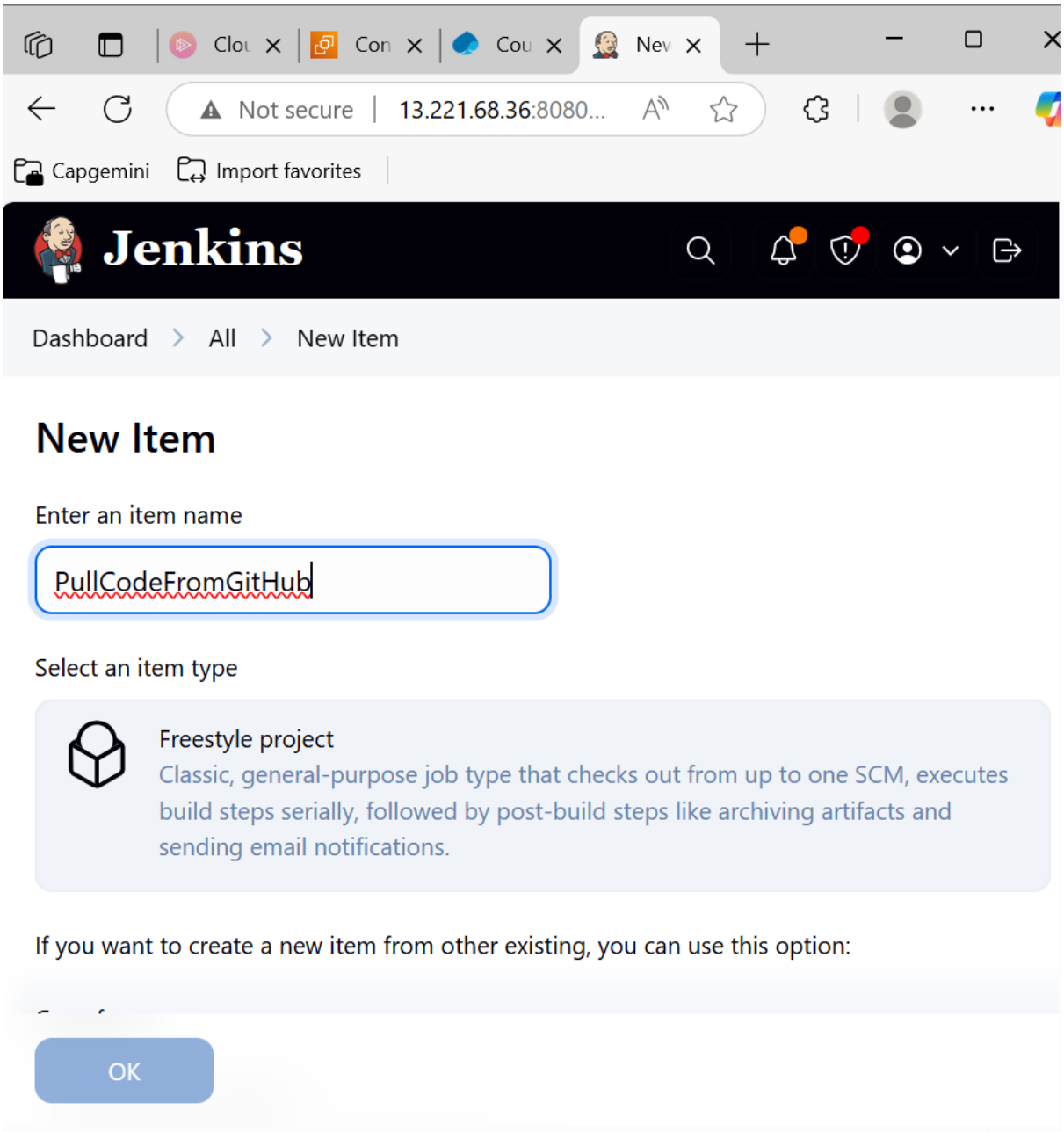
5. Integrate Git with Jenkins

- Connecting Jenkins with Git repositories (e.g., GitHub) to enable source code checkout.
- Adding Git credentials and configuring Jenkins jobs to pull code automatically on commits or at scheduled intervals.
- Setting up webhooks or polling SCM for triggering Jenkins builds on code changes.



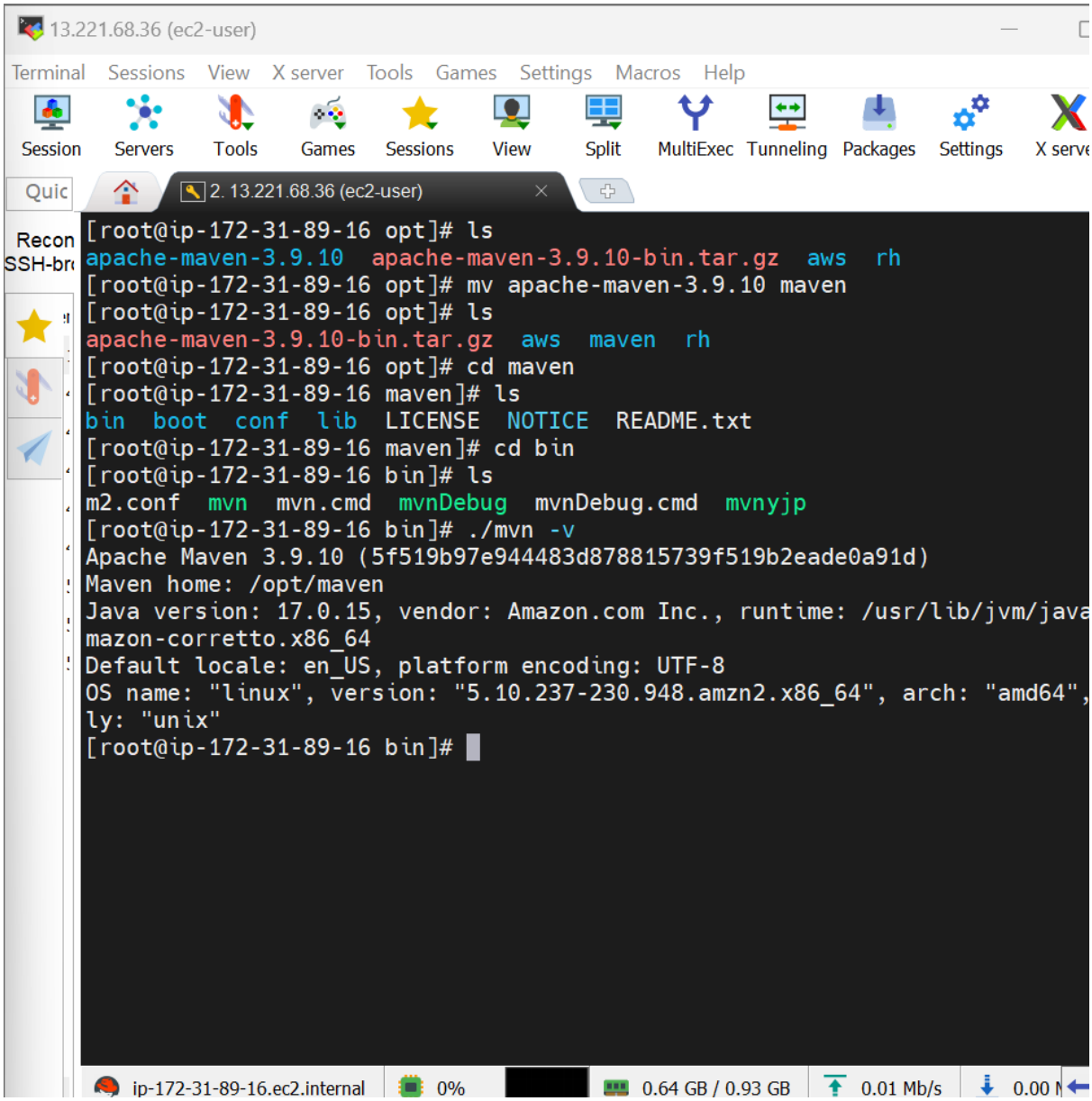
6. Run Jenkins Job to Pull Code from GitHub

- Configuring Jenkins pipeline or freestyle job to clone the latest code from GitHub.
- Ensuring correct branch and repository URLs are specified.
- Running the job to verify that code is fetched successfully into Jenkins workspace.



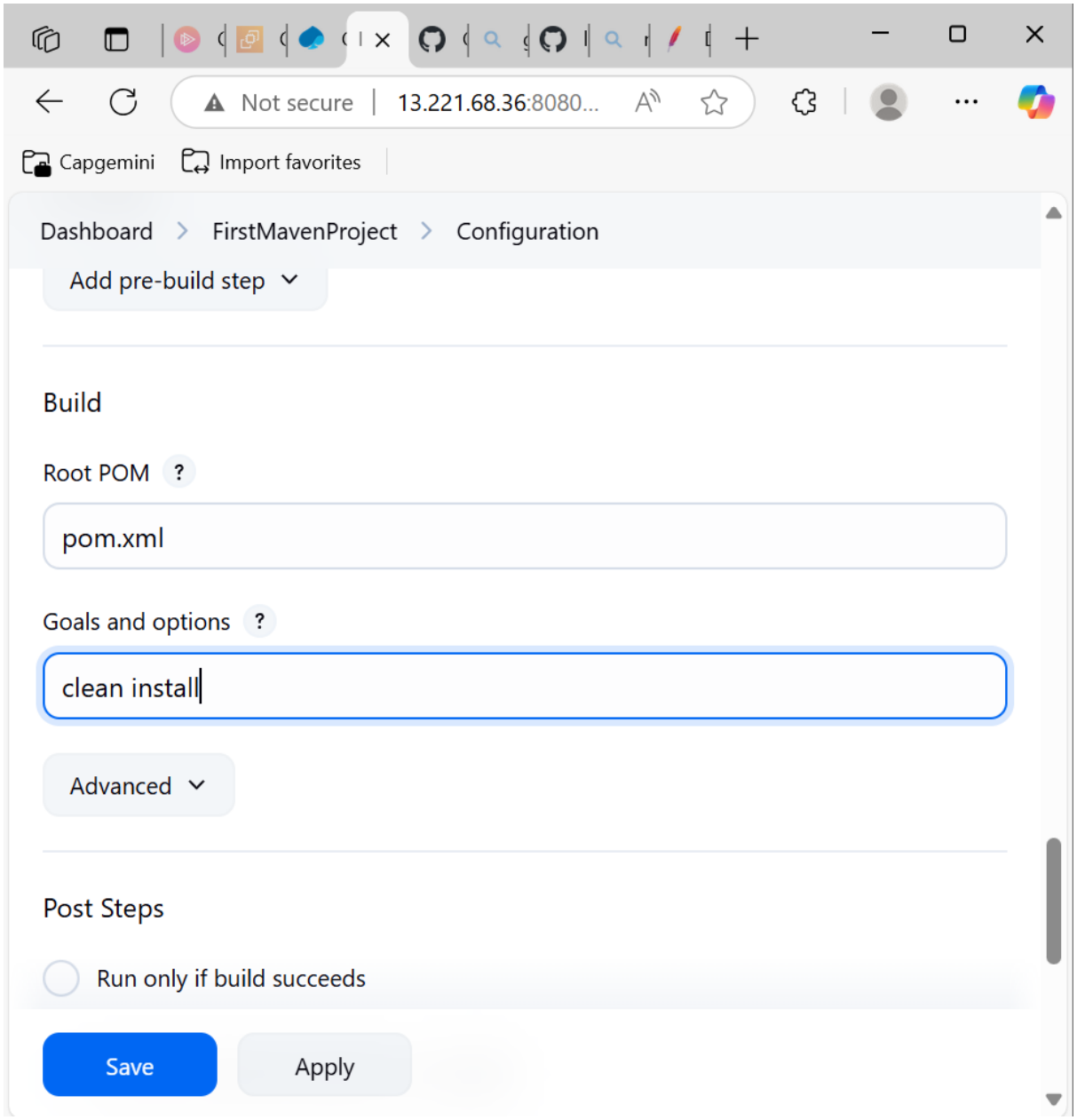
7. Integrate Maven with Jenkins

- Installing Maven on the Jenkins server or configuring Jenkins to use a pre-installed Maven version.
- Adding Maven build steps to Jenkins jobs to compile, test, and package Java applications.
- Using Maven settings.xml for dependency management and repository configuration.



8. Build a Java Project Using Jenkins

- Creating a Jenkins pipeline or freestyle job with Maven goals such as clean install or package.
- Running unit tests during the build process and generating test reports.
- Archiving build artifacts like JAR or WAR files for deployment in subsequent pipeline stages.



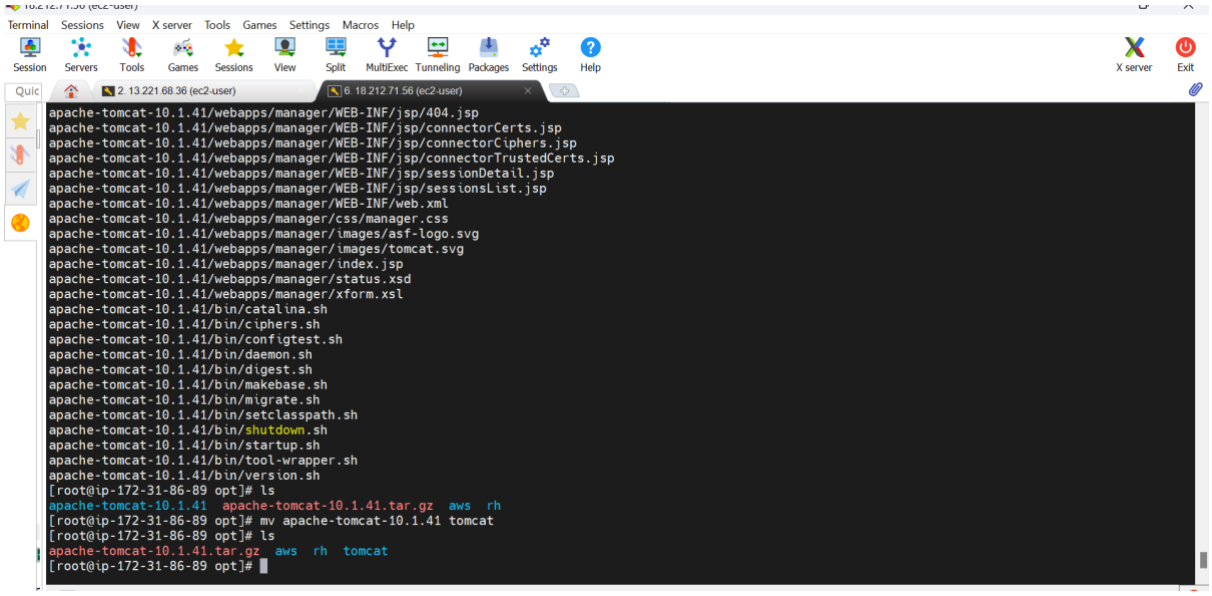
Integrating Tomcat Server in CI/CD Pipeline

1. Section Introduction

- Introduces the role of Tomcat as a web application server used to deploy Java-based applications.
- Explains how integrating Tomcat into the CI/CD pipeline enables automatic deployment of built artifacts.
- Emphasizes streamlining the delivery process from code commit to application deployment on Tomcat.

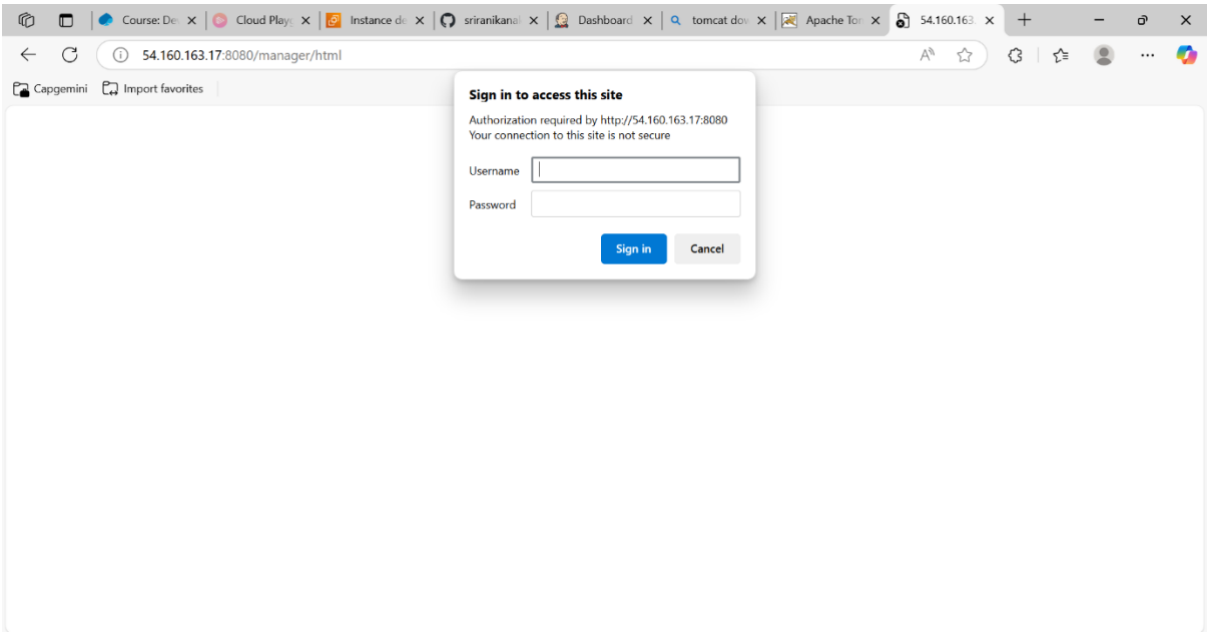
2. Setup a Tomcat Server

- Install Apache Tomcat on the target server (e.g., EC2 instance) where the application will be deployed.
- Configure necessary ports (default 8080) and ensure the server is accessible for deployment.
- Set up user roles and credentials in tomcat-users.xml to allow Jenkins or automation tools to deploy applications remotely.



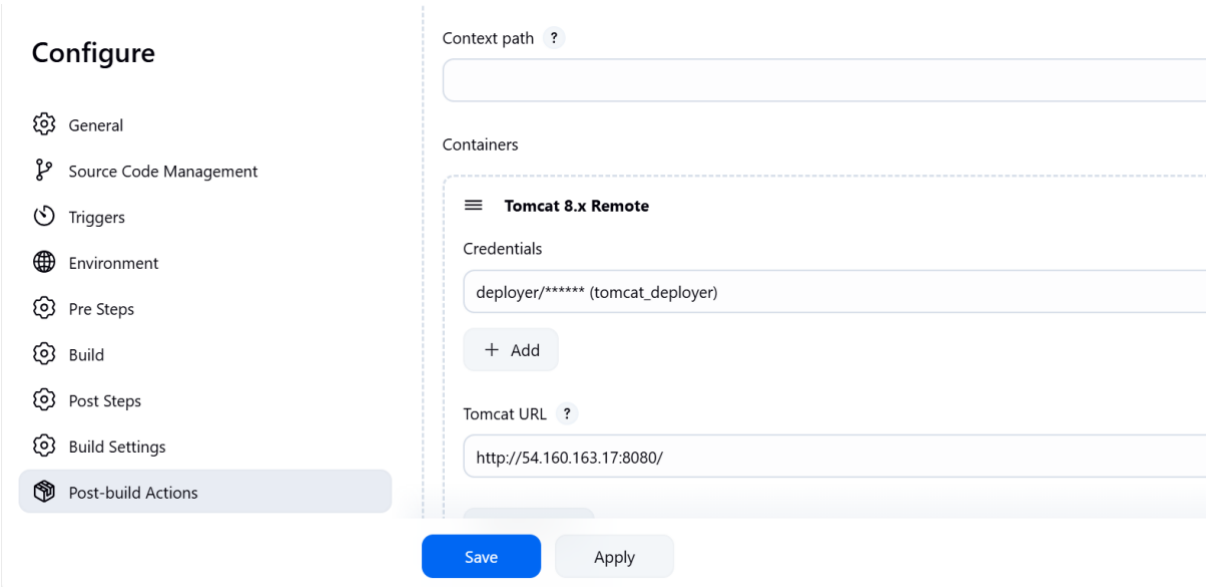
3. Integrate Tomcat with Jenkins

- Install the Jenkins “Deploy to Container” plugin to enable direct deployment to Tomcat.
- Configure Jenkins with the Tomcat server URL, user credentials, and deployment path.
- Add a deployment step in Jenkins jobs to publish build artifacts (WAR files) to the Tomcat server automatically after successful builds.



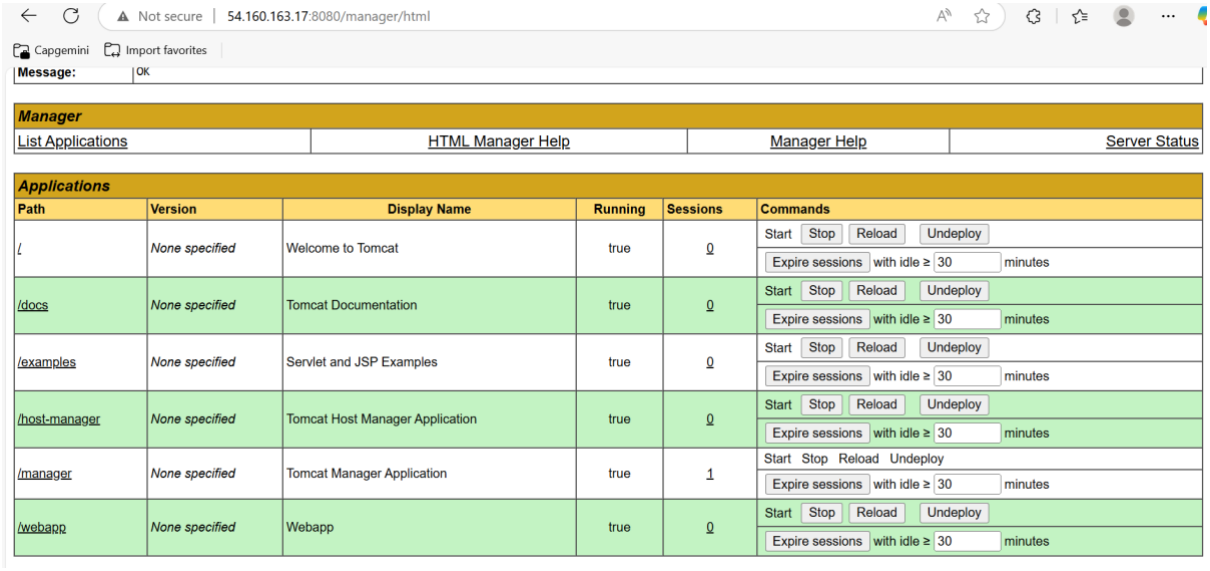
4. Deploy Artifacts on a Tomcat Server

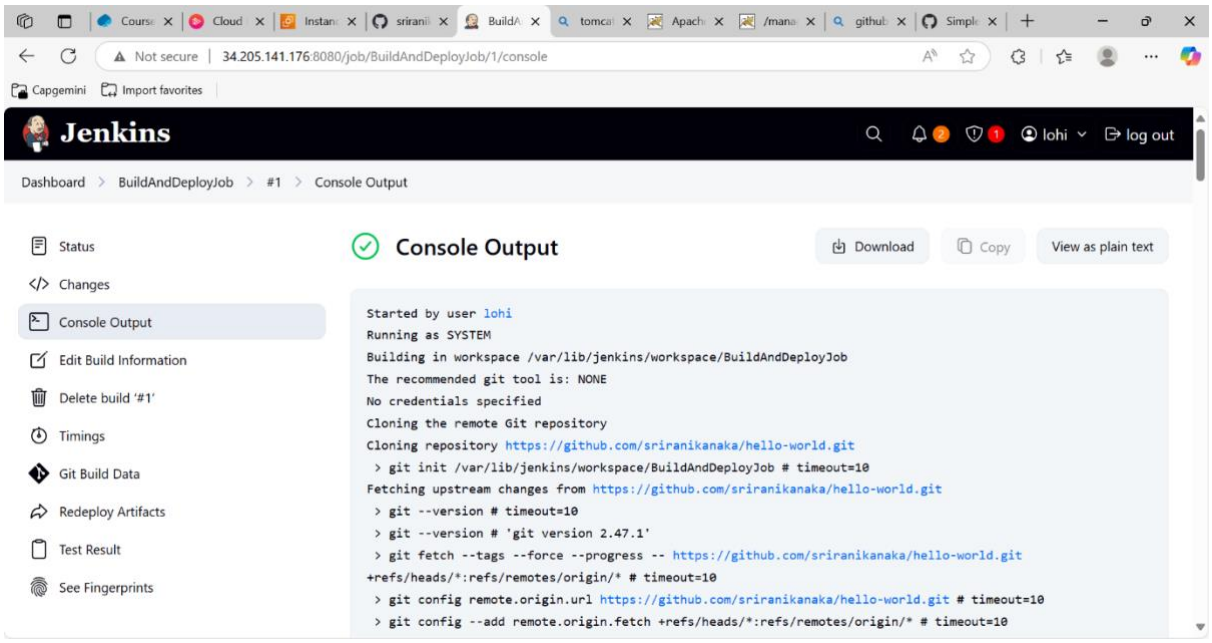
- Use Jenkins to copy or transfer the built WAR files to the Tomcat webapps directory.
- Trigger Tomcat to automatically deploy or redeploy the application upon artifact update.
- Validate deployment by accessing the application URL and checking logs for any errors.



5. Automate Build and Deploy Using Poll SCM

- Enable the Poll SCM feature in Jenkins to check the source code repository periodically for changes.
- Configure polling intervals to trigger automated builds and deployments without manual intervention.
- This setup ensures continuous integration and continuous delivery, keeping the deployed app up-to-date with code changes.

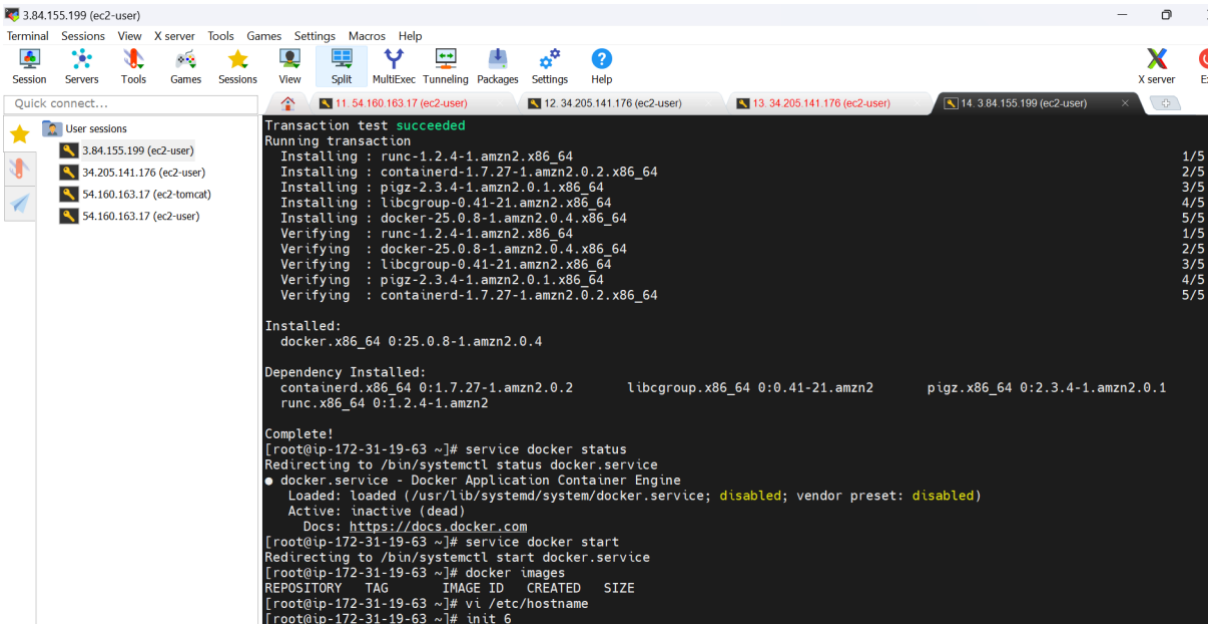




Integrating Docker in CI/CD Pipeline

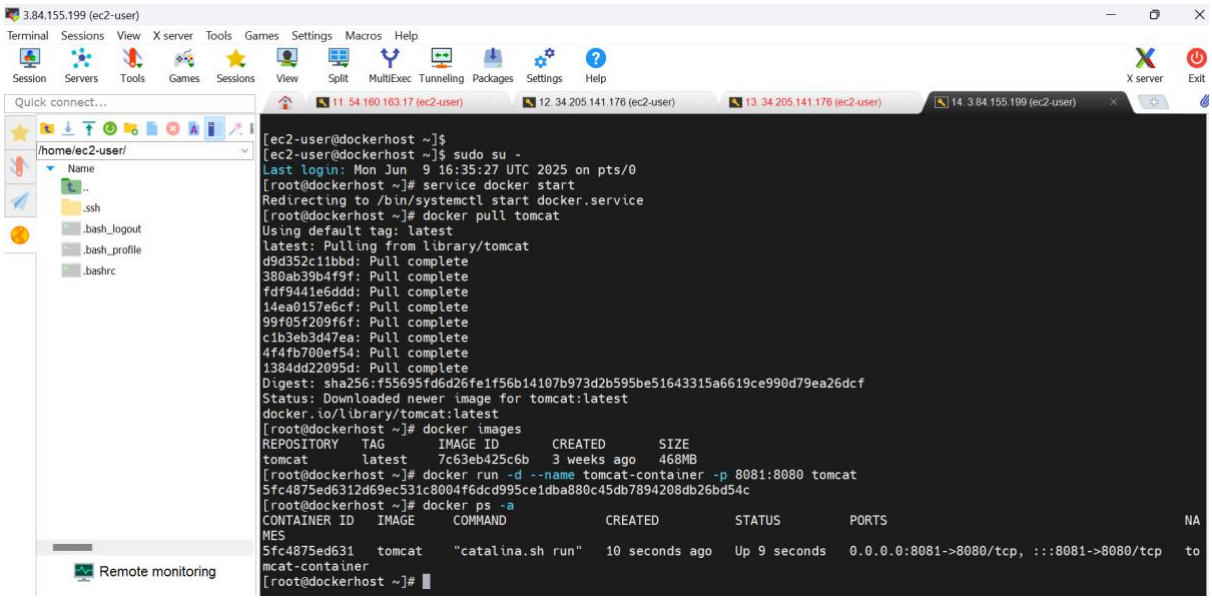
1. Setup Docker Environment

- Install Docker Engine on the build or deployment servers (e.g., Jenkins server or AWS EC2 instances).
- Verify Docker installation with basic commands like docker --version and docker run hello-world.
- Configure Docker daemon and user permissions to allow Jenkins and other tools to interact with Docker.



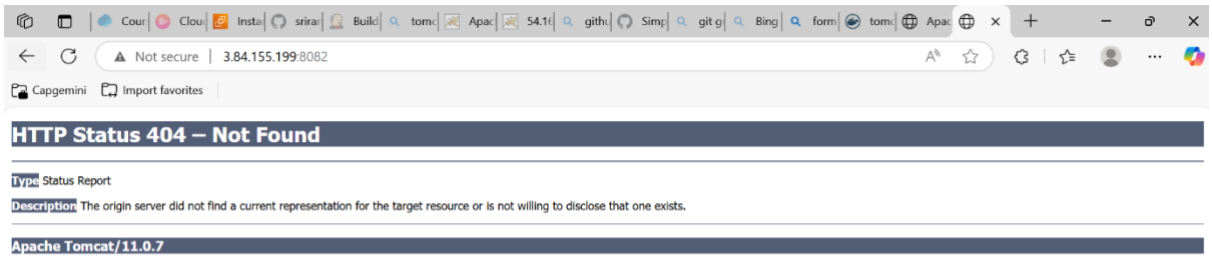
2. Create a Tomcat Container

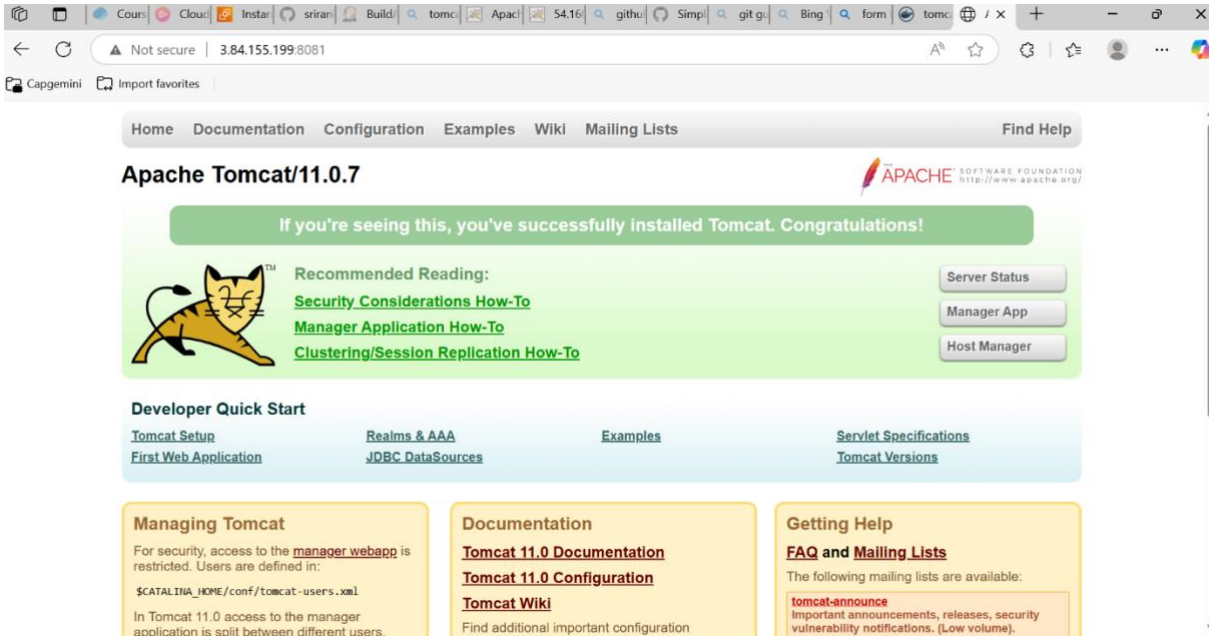
- Use the official Tomcat Docker image to spin up a containerized Tomcat server.
- Run commands such as docker run -d -p 8080:8080 tomcat to launch the container.
- Verify the container is running and accessible via the mapped port.



3. Fixing Tomcat Container Issues

- Troubleshoot common problems such as port conflicts, volume mounts, or missing dependencies.
- Adjust Dockerfile or container settings to fix deployment errors or startup failures.
- Check container logs with docker logs <container_id> to identify issues.





4. Create a First Dockerfile

- Write a basic Dockerfile that defines the environment and instructions to build a container image.
- Include steps like setting the base image (e.g., FROM tomcat), copying application artifacts, and exposing necessary ports.
- Build the Docker image locally using docker build -t mytomcat:latest .

```
[root@dockerhost ~]# vi Dockerfile
[root@dockerhost ~]# docker build -t mytomcat .
[+] Building 51.6s (8/8) FINISHED                                                                                    docker:default
 => [internal] load build definition from Dockerfile                                                                           0.0s
 => => transferring dockerfile: 585B                                                                                           0.0s
 => [internal] load metadata for docker.io/library/almalinux:8                                                                 0.3s
 => [internal] load .dockerignore                                                                                              0.0s
 => => transferring context: 2B                                                                                                0.0s
 => [1/4] FROM docker.io/library/almalinux:8@sha256:86b0e1745d34c8577ae658aa0c8d57fb2cbdb21ee7ca9c19ab309c7cc510ac51           4.6s
 => => resolve docker.io/library/almalinux:8@sha256:86b0e1745d34c8577ae658aa0c8d57fb2cbdb21ee7ca9c19ab309c7cc510ac51           0.0s
 => => sha256:86b0e1745d34c8577ae658aa0c8d57fb2cbdb21ee7ca9c19ab309c7cc510ac51 4.68kB / 4.68kB                                 0.0s
 => => sha256:4f63eb966695df3c993deeacec7c73d87728e2ea66d3b48fed4b40cb547fa7c2 1.03kB / 1.03kB                                 0.0s
 => => sha256:9450b760565e5ef8056332c7a954299d2db2b88fc7ba907be932c183df054f99 579B / 579B                                     0.0s
 => => sha256:19877a9af8e32b204f592bba0490bd55d67c33e7500c0cb9da49111339f14400 71.75MB / 71.75MB                               1.0s
 => => extracting sha256:19877a9af8e32b204f592bba0490bd55d67c33e7500c0cb9da49111339f14400                                      3.5s
 => [2/4] RUN dnf install -y java-11-openjdk curl &&     dnf clean all                                                        34.6s
 => [3/4] WORKDIR /opt/tomcat                                                                                                  0.1s
 => [4/4] RUN curl -O https://dlcdn.apache.org/tomcat/tomcat-10/v10.1.41/bin/apache-tomcat-10.1.41.tar.gz &&     tar  1.1s
 => exporting to image                                                                                                        10.8s
 => => exporting layers                                                                                                       10.8s
 => => writing image sha256:005eb8b9e034a178d039ba65adefdc72a1b96e7efb1fb3cc1e5fe8e021b45a39                                   0.0s
 => => naming to docker.io/library/mytomcat                                                                                    0.0s
[root@dockerhost ~]#
```

5. Create a Customized Dockerfile for Tomcat

- Enhance the Dockerfile to include custom configurations, environment variables, or scripts needed for the application.
- Automate deployment steps like copying WAR files into the Tomcat webapps directory within the image.
- Optimize the Docker image size and startup time for faster deployments.

6. Integrate Docker with Jenkins

- Configure Jenkins with Docker plugins to enable Docker commands within Jenkins jobs.
- Set Jenkins agents with Docker installed or use Docker-in-Docker setups.
- Define Jenkins pipeline steps that build, tag, and push Docker images.

7. Jenkins Job to Build and Copy Artifacts onto Docker Host

- Automate the process of building the Java application with Maven.
- Copy the build artifact (WAR/JAR) to the Docker build context or directly to the Docker host.
- Trigger Docker image build and container deployment from Jenkins jobs.

8. Update Tomcat Dockerfile to Automate Deployment Process

- Modify the Dockerfile to fully automate application deployment by embedding the build artifact during image creation.
- Ensure that each new Docker image contains the latest version of the application ready for deployment.
- Automate versioning and tagging of Docker images in the pipeline.

9. Automate Build and Deployment on Docker Container

- Create Jenkins pipeline stages for continuous building, testing, and deploying Docker containers.
- Use Docker Compose or Kubernetes if needed to manage multi-container deployments.
- Automate container lifecycle management including start, stop, and rollback procedures.

10. Jenkins Job to Automate CI/CD to Deploy Application on Docker Container

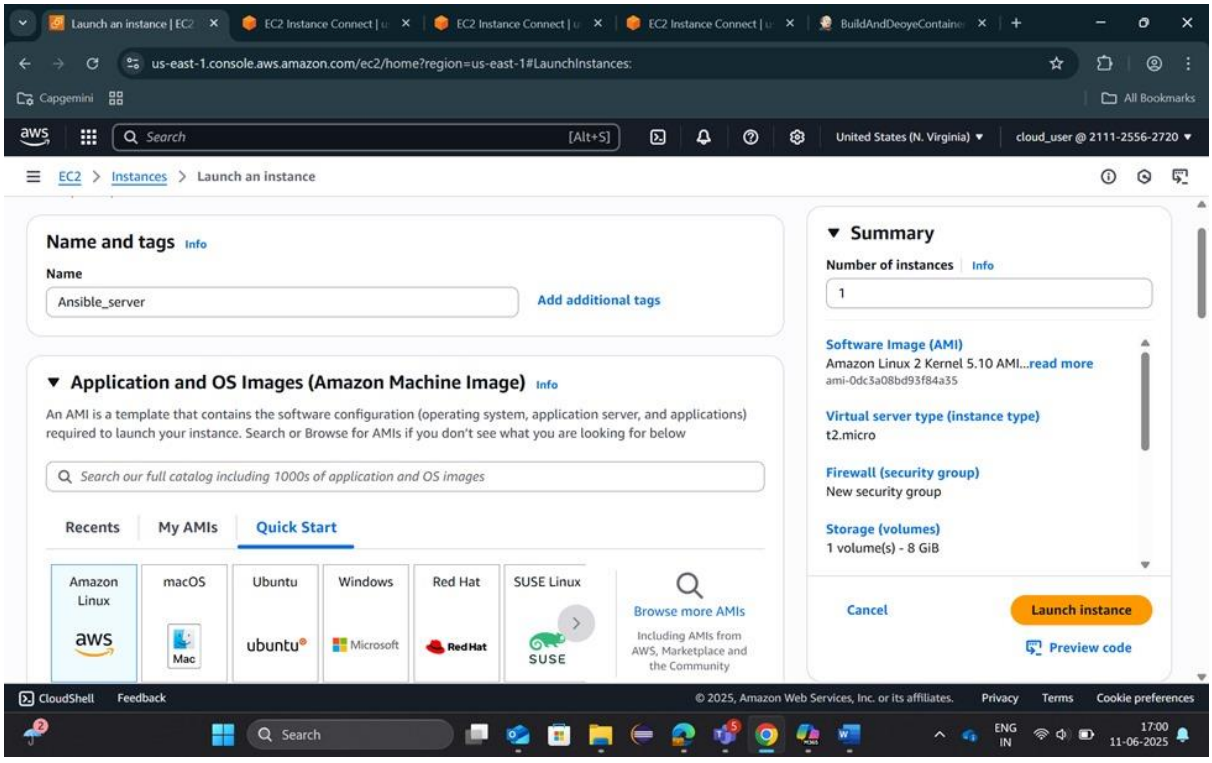
- Combine all steps into a fully automated Jenkins pipeline job that listens for code changes.
- Upon each commit, Jenkins triggers build, test, Docker image creation, and deployment to the Docker environment.
- Achieve continuous integration and continuous deployment with minimal manual intervention.

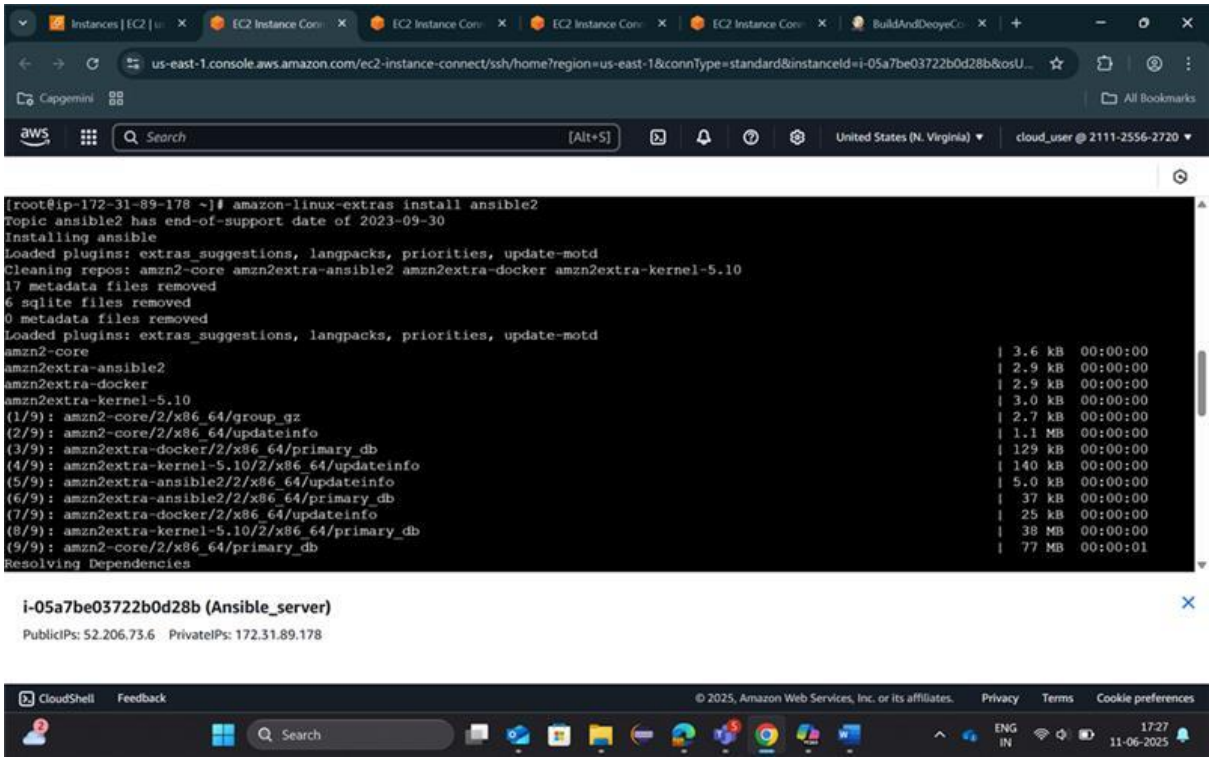
Integrating Ansible in CI/CD Pipeline

1. Section Introduction – Why Do We Need Ansible?

- Ansible is a powerful automation tool that simplifies configuration management, application deployment, and orchestration.
- It helps manage infrastructure as code with easy-to-write YAML playbooks, reducing manual intervention and errors.
- Integrating Ansible into CI/CD pipelines allows automated, repeatable, and consistent deployment processes across environments.

2. Ansible Installation

- Install Ansible on the control node (e.g., Jenkins server or a dedicated automation server).
- Verify the installation with commands like ansible --version and configure necessary dependencies such as Python and SSH.
- Set up inventory files defining the target hosts and configure SSH access for passwordless login.
- 

- 

3. Integrate Docker with Ansible

- Use Ansible Docker modules to manage Docker images, containers, and networks programmatically.
- Create playbooks that automate Docker tasks such as building images, starting containers, and cleaning up old containers.
- Enable Ansible to interact with Docker daemons on remote or local hosts.

4. Integrate Ansible with Jenkins

- Configure Jenkins to run Ansible playbooks as build or post-build steps using plugins like "Ansible plugin".
- Pass variables and parameters from Jenkins to Ansible playbooks for flexible deployments.
- Use Jenkins to trigger Ansible-driven deployments as part of the CI/CD pipeline automatically.

5. Build an Image and Create Container on Ansible

- Write Ansible playbooks that build Docker images from Dockerfiles on specified hosts.
- Automate the creation and startup of containers based on the newly built images.
- Manage container lifecycle through Ansible tasks, including start, stop, and removal.

6. Ansible Playbook to Create Image and Container

- Develop modular playbooks with clear tasks to build Docker images and deploy containers.
- Include steps to pull code, copy artifacts, and configure container environment variables.
- Use handlers and conditional tasks for efficient automation and error handling.

7. Copy Image onto Docker Hub

- Automate pushing Docker images to Docker Hub or other container registries using Ansible tasks.
- Manage authentication securely within playbooks or Jenkins credentials.
- Facilitate image distribution and version control across different environments.

8. Jenkins Job to Build an Image onto Ansible

- Configure Jenkins pipelines to invoke Ansible playbooks that build Docker images.
- Use Jenkins environment variables to control playbook behavior and image tags.
- Integrate image build step with source code changes to maintain continuous integration.

9. How to Create Container on Docker Host Using Ansible Playbook – DevOps Project

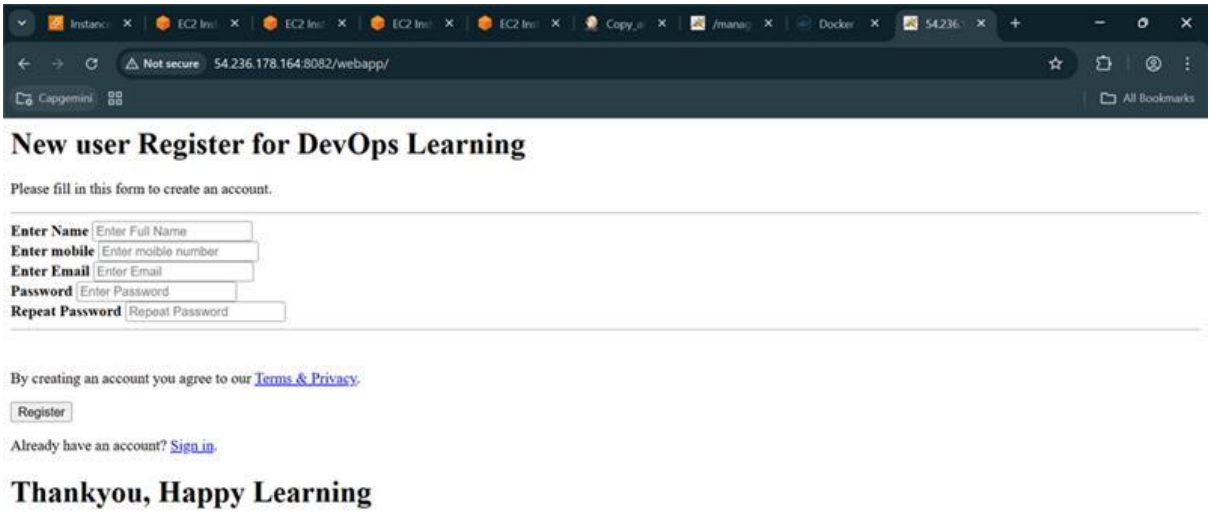
- Use Ansible playbooks to launch containers on remote Docker hosts, ensuring environment consistency.
- Configure network, volumes, and ports in playbooks to match application requirements.
- Validate container status and logs through automated post-deployment checks.

10. Continuous Deployment of Docker Container Using Ansible Playbook

- Set up Ansible playbooks to handle rolling updates, zero-downtime deployments, or blue-green deployment strategies.
- Automate rollback processes in case of failures using Ansible's idempotency features.
- Incorporate monitoring and alerting hooks within the playbooks.

11. Jenkins CI/CD to Deploy on Container Using Ansible

- Combine Jenkins and Ansible for a fully automated pipeline where code commits trigger builds and deployments.
- Use Jenkins to orchestrate the entire process from building code, creating Docker images, and deploying containers via Ansible.
- Ensure pipeline reliability and auditability with logs from both Jenkins and Ansible executions.



- 

Kubernetes on AWS

1. Section Introduction – Why Kubernetes

- Kubernetes is an open-source container orchestration platform that automates deployment, scaling, and management of containerized applications.
- It enables high availability, scalability, and efficient resource utilization in cloud environments.
- Using Kubernetes on AWS leverages cloud infrastructure benefits along with Kubernetes' powerful orchestration capabilities.

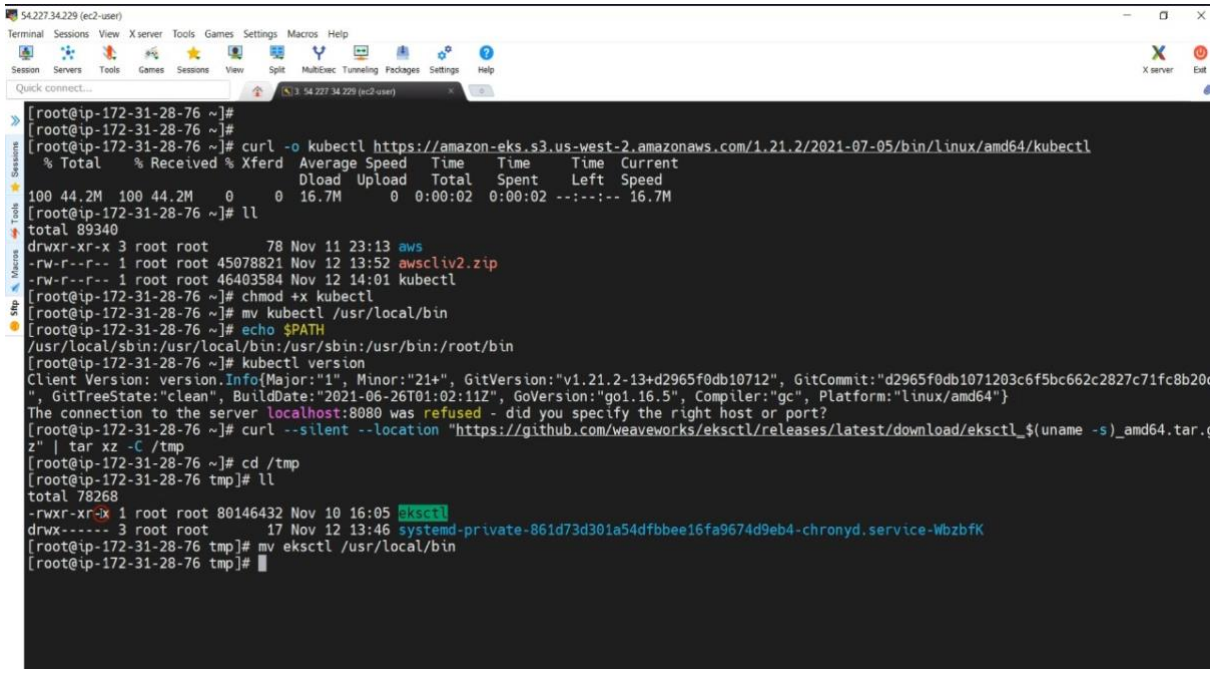
2. Kubernetes Installation Methods

- Various ways to install Kubernetes include using kubeadm, minikube (local), kops, or managed services like Amazon EKS.
- For production and scalable environments on AWS, EKS (Elastic Kubernetes Service) is preferred due to managed control plane and integrations.

- The choice depends on requirements like control, ease of management, and scale.

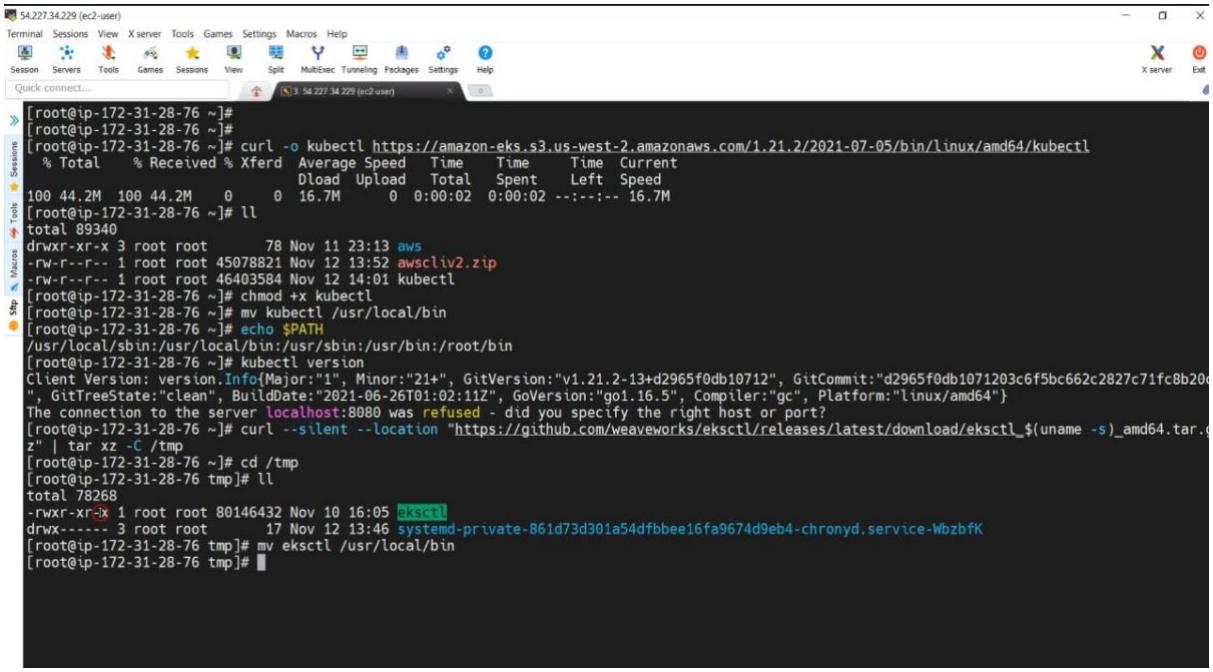
3. EKS Installation Procedure

- Use AWS Management Console or CLI tools to create an EKS cluster with required VPC, subnets, and IAM roles.
- Configure worker nodes (EC2 instances) to join the cluster.
- Set up kubectl access by updating kubeconfig with cluster details for management.

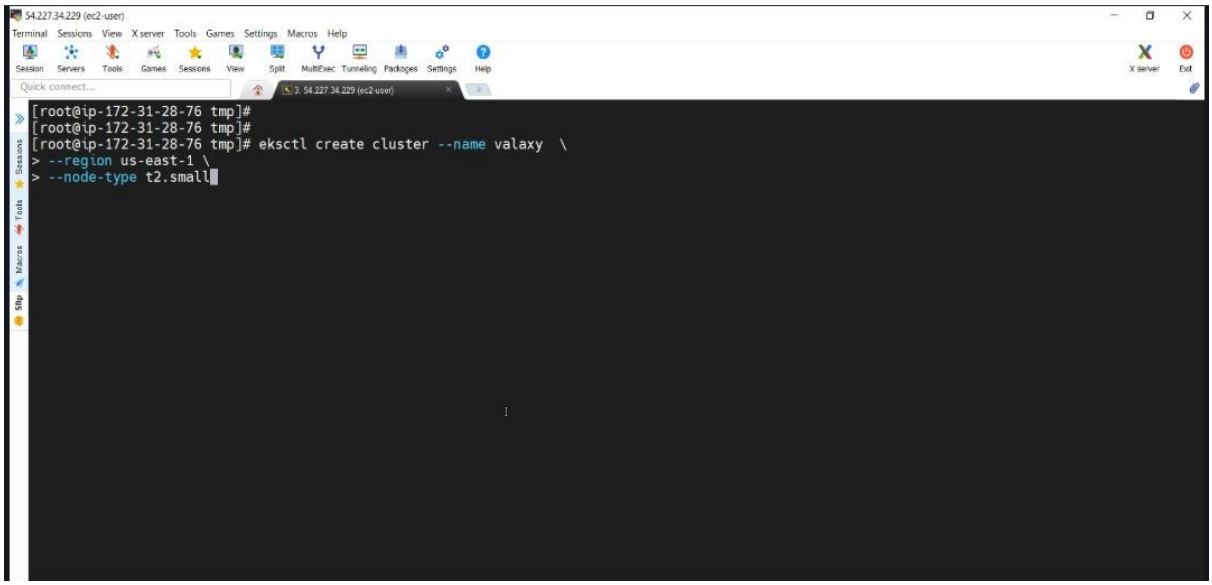


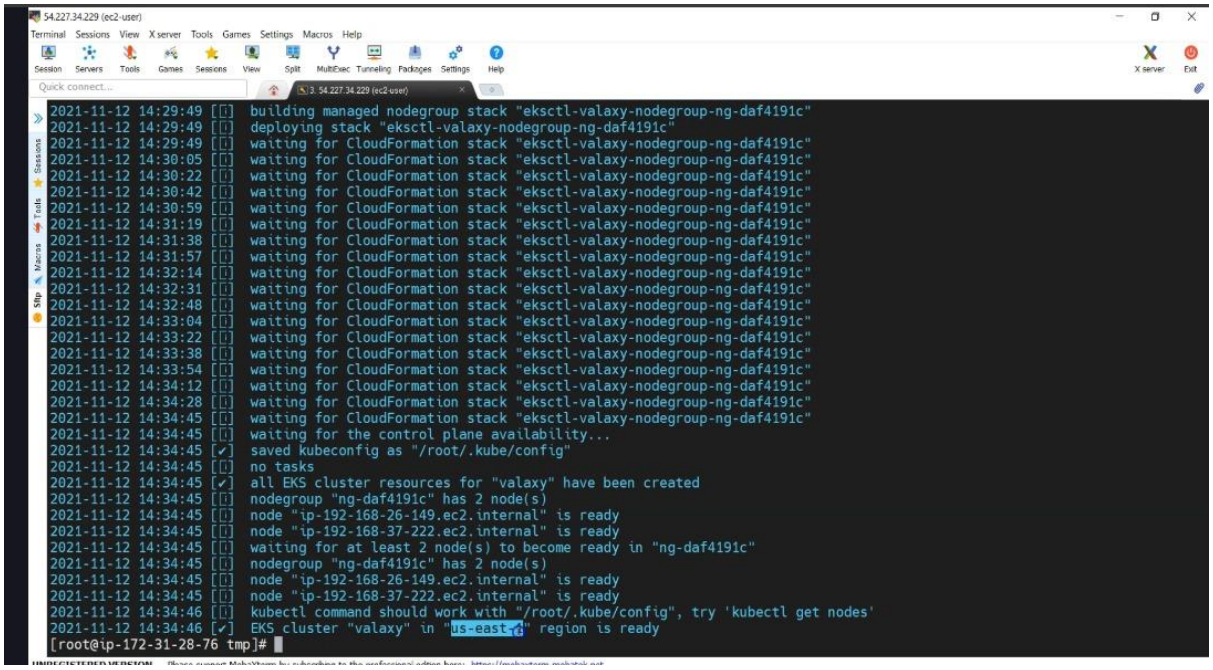
4. Setup Bootstrap Server for eksctl

- eksctl is a CLI tool simplifying EKS cluster creation and management.
- Bootstrap server is a local machine or EC2 instance configured with eksctl, AWS CLI, and kubectl.
- It is used to run eksctl commands to create and manage Kubernetes clusters.

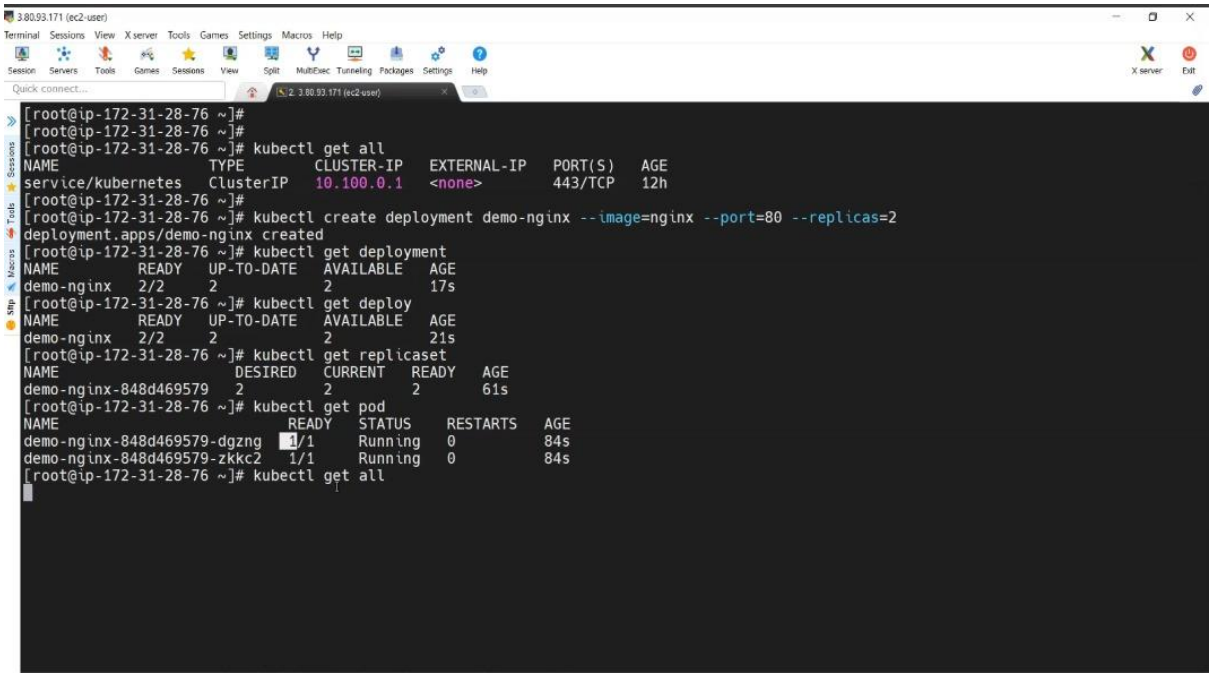


5. Setup Kubernetes Using eksctl

- Use eksctl commands to create clusters, node groups, and configure networking easily.
- eksctl automates underlying AWS resource provisioning required for Kubernetes.
- Validate cluster creation by checking nodes and pods status.
- 

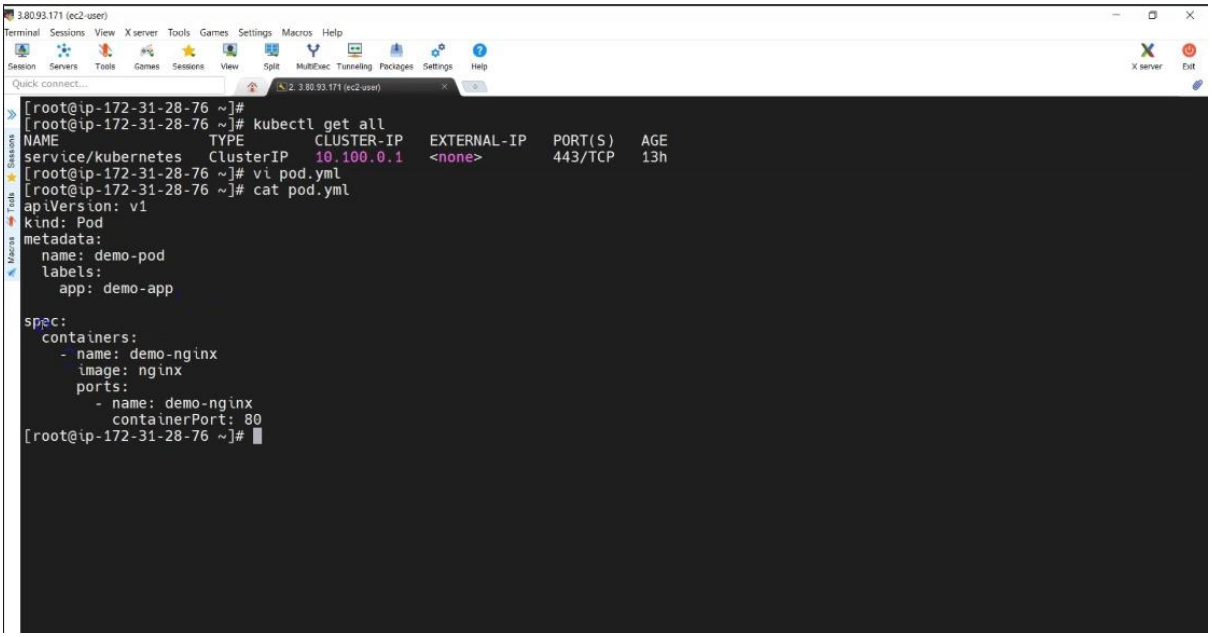


6. Run Kubernetes Basic Commands

- Use kubectl CLI to interact with the cluster: kubectl get nodes, kubectl get pods, kubectl describe, etc.
- Practice deploying sample applications and managing cluster resources.
- Monitor cluster health and troubleshoot issues using kubectl.
- 

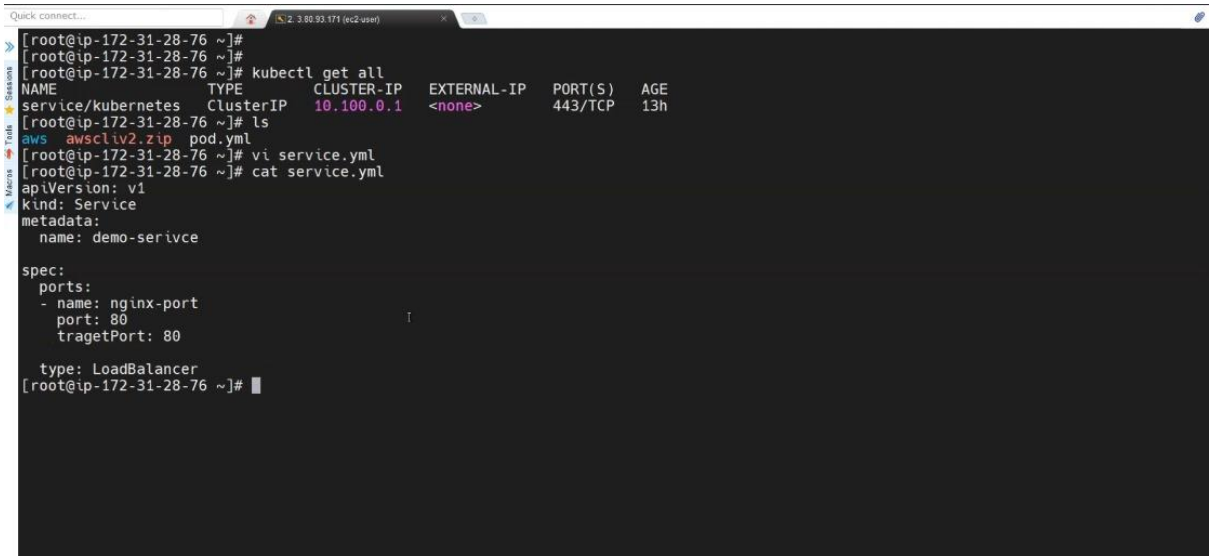
7. Create 1st Manifest File

- Write a Kubernetes manifest YAML file defining resources like Pods, Deployments, or ConfigMaps.
- Example includes defining a Pod with container image and resource limits.
- Apply the manifest with kubectl apply -f <filename.yaml>.



8. Create a Service Manifest File

- Define a Kubernetes Service to expose Pods internally or externally.
- Configure service types such as ClusterIP, NodePort, or LoadBalancer depending on use case.
- Connect the service to Pods via selectors for networking.

- 

9. Using Labels and Selectors

- Labels are key-value pairs attached to Kubernetes objects for identification and grouping.
- Selectors allow services, deployments, and other resources to target specific Pods based on labels.
- This mechanism enables dynamic management and scaling of applications.

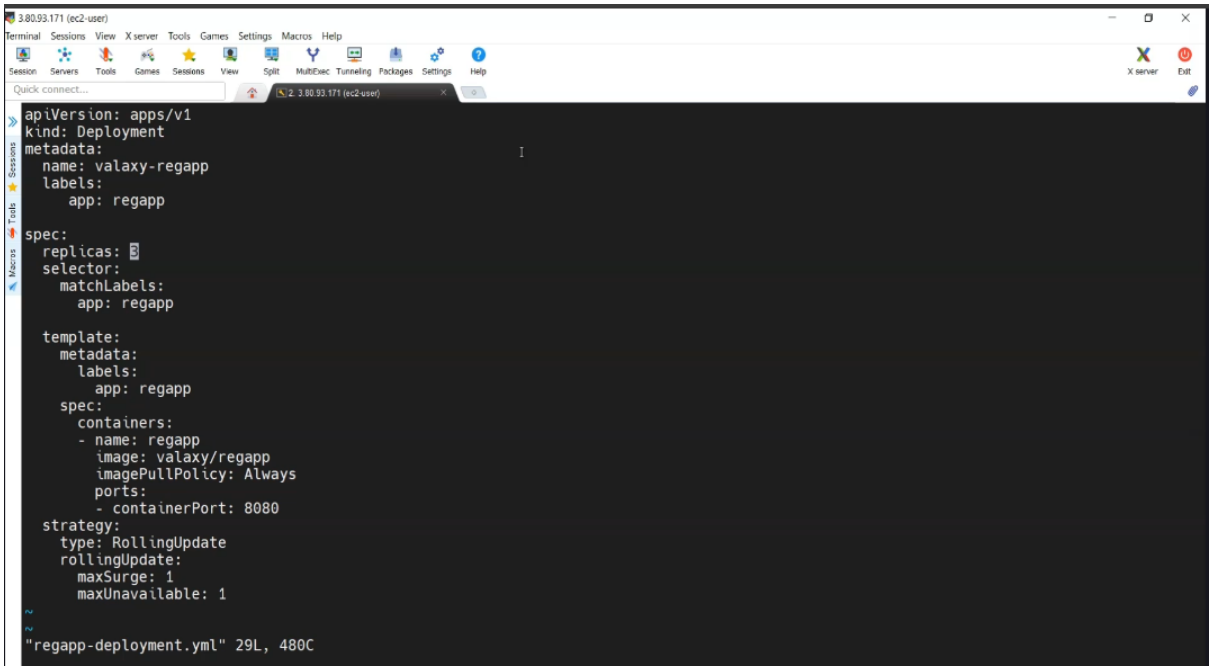
Integrating Kubernetes in CI/CD Pipelines

1. Write a Deployment File

- Create a Kubernetes deployment manifest (YAML) defining the desired state for your application pods.

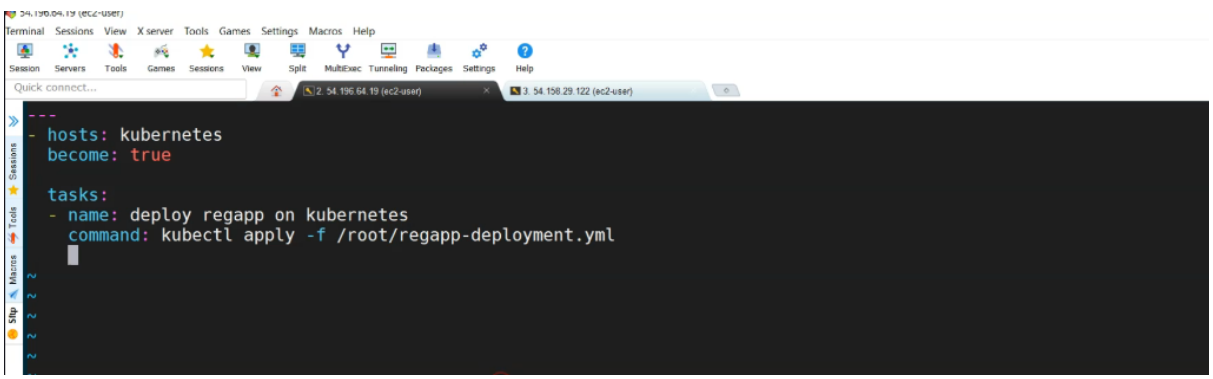
- Specify container image, replicas, update strategy, and resource requirements.

  This deployment manages pod lifecycle and scaling automatically.

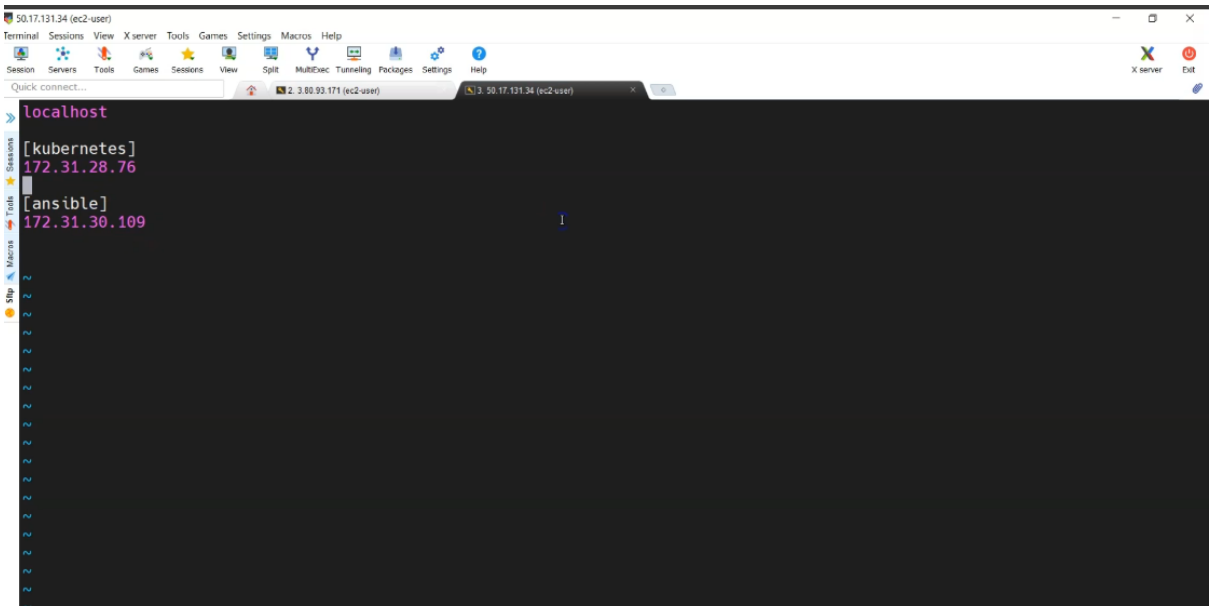
- 

2. Use Deployment and Service Files to Create and Access Pod

- Apply deployment and service manifests using kubectl apply -f commands.
- The deployment creates pods running the application, while the service exposes pods to internal or external traffic.
- Validate pod status and service accessibility through Kubernetes commands.

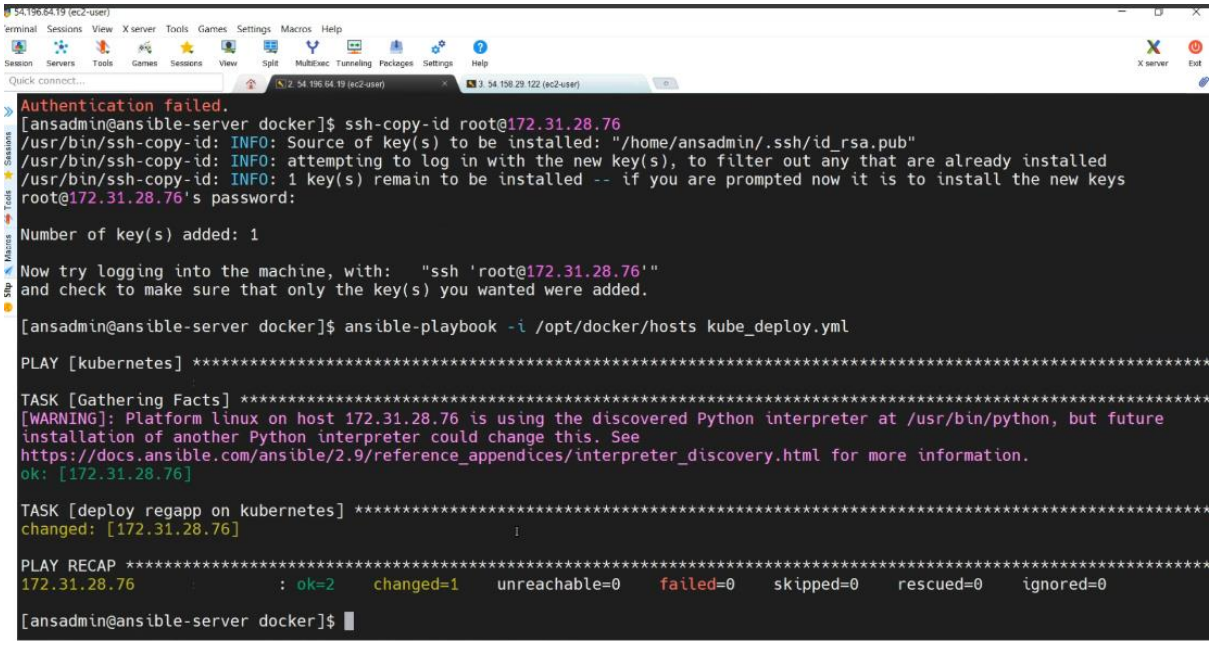
- 

3. Integrate Kubernetes Bootstrap Server with Ansible

- Use Ansible to automate management of Kubernetes bootstrap server where kubectl and cluster tools are configured.
- Playbooks can handle setup tasks such as configuration, authentication, and cluster maintenance.
- Enables automated deployment workflows integrated with infrastructure provisioning.
- 

4. Create Ansible Playbooks for Deploy and Service Files

- Develop playbooks that deploy Kubernetes manifests (deployment, service) onto the cluster.
- Playbooks run kubectl apply commands and manage rollback or updates.
- Simplifies repeatable deployments and promotes Infrastructure as Code principles.

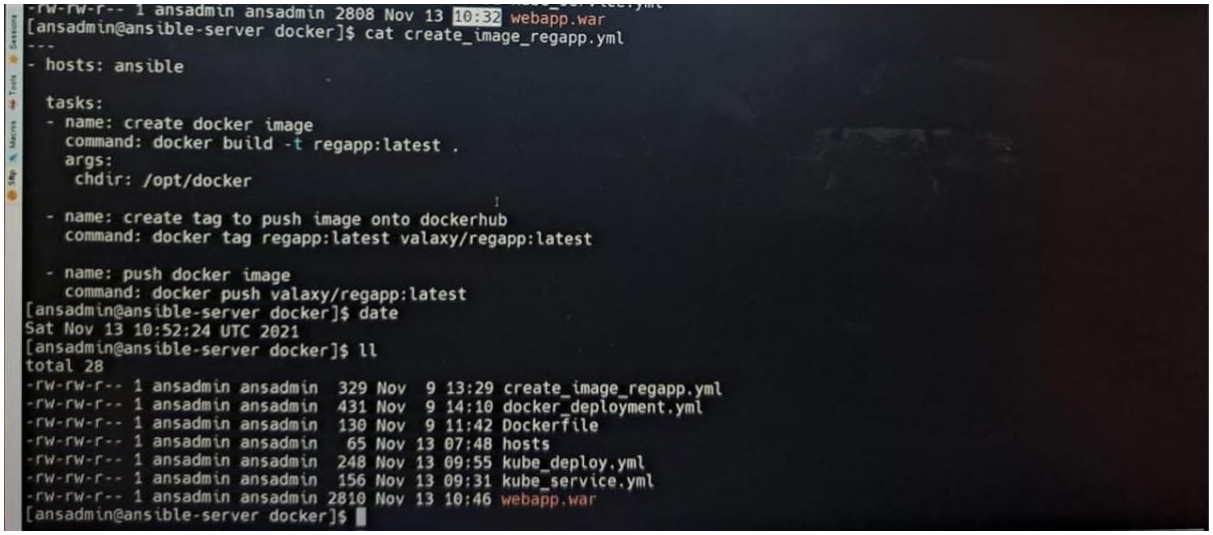
- 

5. Create Jenkins Deployment Job for Kubernetes

- Set up Jenkins pipelines that call Ansible playbooks or directly execute kubectl commands.
- Automate build, test, and deployment steps, triggering on code commits or merges.
- Monitor deployment status and report success or failure.

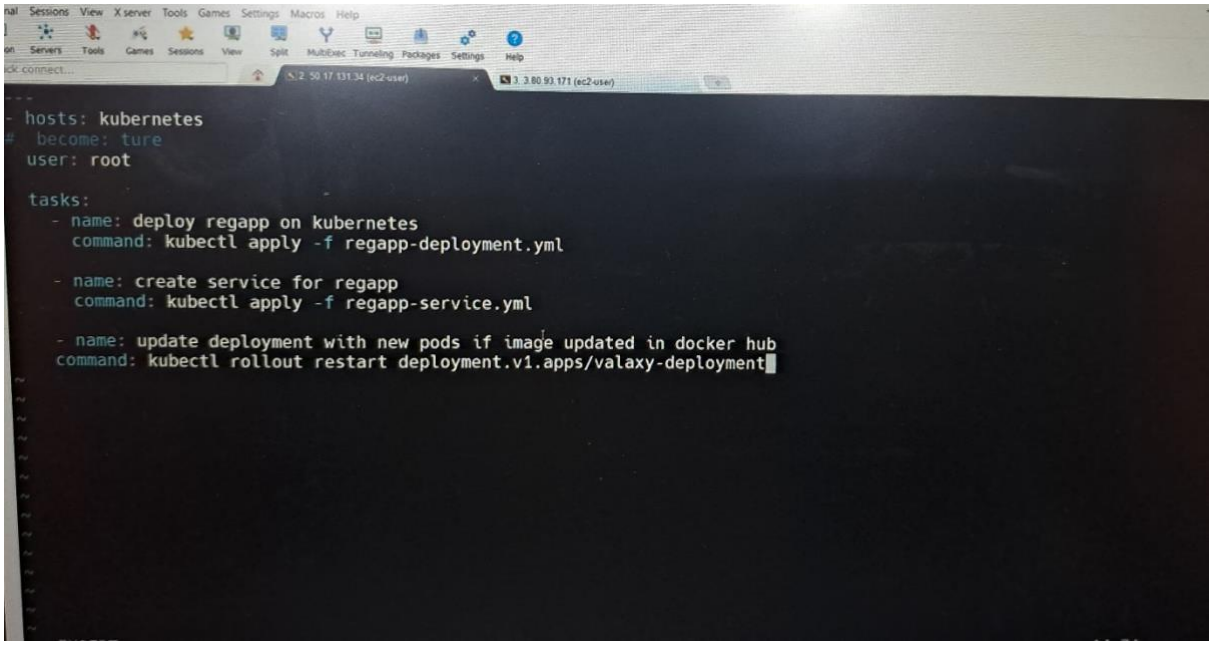
6. CI Job to Create Image for Kubernetes

- Jenkins job to build Docker images from the latest code.
- Push images to container registry accessible by Kubernetes (e.g., Docker Hub, AWS ECR).
- Ensures latest code is packaged and ready for deployment in the cluster.



7. Enable Rolling Update to Create Pod from Latest Docker Image

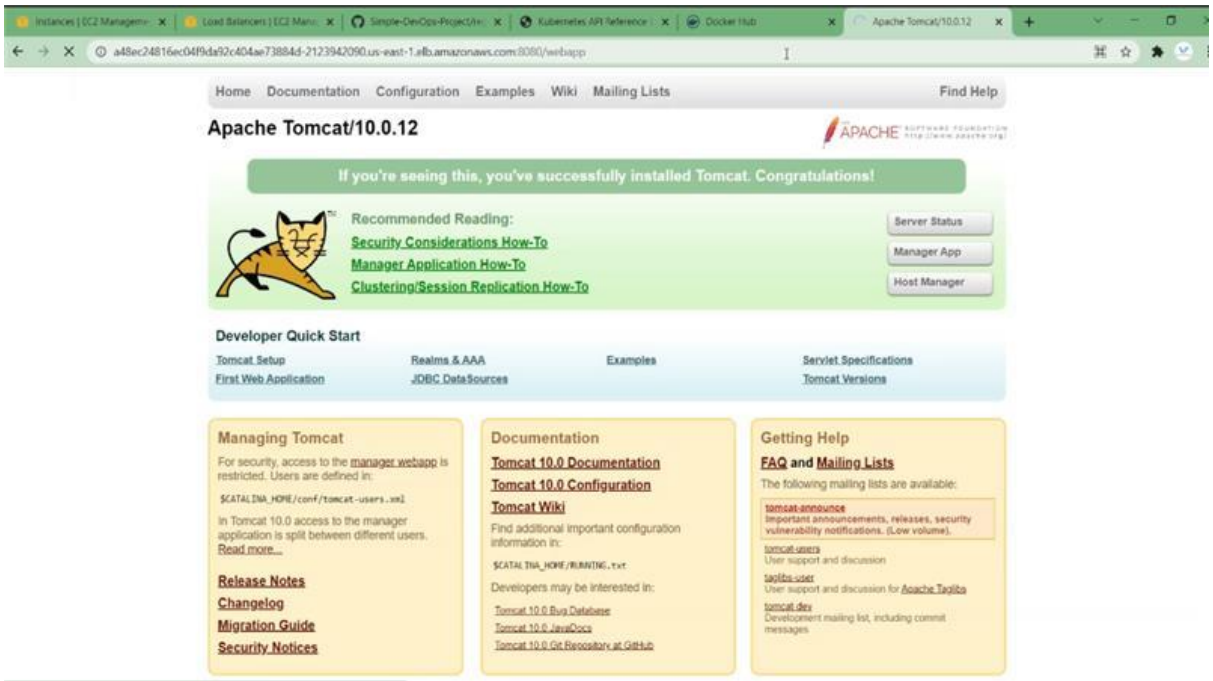
- Configure deployment strategy in Kubernetes manifest to support rolling updates.
- Allows zero downtime deployments by gradually replacing old pods with new ones running updated images.
- Monitors rollout progress and automatically rolls back on failures.



8. Complete CI and CD Job to Build and Deploy Code on Kubernetes

- Combine all pipeline stages: code build, image creation, deployment manifests application, and rollout management.

- Automate the full lifecycle from developer commit to live application update on Kubernetes.
- Enable continuous integration and delivery with fast feedback loops.



9. Clean up Kubernetes Setup

- Automate removal of unused resources like pods, services, or deployments no longer needed.
- Use Jenkins or Ansible jobs to teardown test clusters or clean namespaces.
- Helps maintain resource efficiency and security hygiene in Kubernetes environments.

```
[root@ip-172-31-25-112 /]# cd tmp/
[root@ip-172-31-25-112 tmp]# ll
total 16
-rw------- 1 root root   0 Jun 13 07:29 62291b61.eksctl.lock
-rw-r--r-- 1 root root 203 Jun 13 07:34 pod.yml
-rw-r--r-- 1 root root 481 Jun 13 07:36 regapp-deploy.yml
-rw-r--r-- 1 root root 197 Jun 13 07:36 regapp-service.yml
-rw-r--r-- 1 root root 183 Jun 13 07:34 service.yml
drwx------ 3 root root  17 Jun 13 07:08 systemd-private-72ab7dd853ea47c9be9b90aefc2d7ea1-chronyd.service-pgkTY6
[root@ip-172-31-25-112 tmp]# ls
62291b61.eksctl.lock  regapp-deploy.yml   service.yml
pod.yml               regapp-service.yml  systemd-private-72ab7dd853ea47c9be9b90aefc2d7ea1-chronyd.service-pgkTY6
[root@ip-172-31-25-112 tmp]# kubectl delete -f regapp-service.yml
service "valaxy-service" deleted
[root@ip-172-31-25-112 tmp]# kubectl delete -f regapp-deploy.yml
deployment.apps "valaxy-regapp" deleted
[root@ip-172-31-25-112 tmp]#
```

Conclusion

This case study demonstrated the successful design and implementation of a fully automated DevOps CI/CD pipeline using widely adopted tools such as Git, Jenkins, Maven, Docker, Ansible, and Kubernetes on AWS infrastructure. By integrating these technologies, the project achieved continuous integration and continuous deployment, enabling faster, more reliable software delivery with minimal manual intervention.

Setting up the pipeline involved configuring Jenkins to automate builds and tests triggered by Git commits, managing Java project builds through Maven, and deploying applications efficiently via Docker containers. Further automation was enhanced by incorporating Ansible for configuration management and Kubernetes for scalable container orchestration on AWS EKS.

The pipeline facilitated early detection of bugs, reduced deployment errors, and supported seamless rolling updates, contributing to improved software quality and accelerated release cycles. Using cloud resources and infrastructure as code principles ensured flexibility and scalability while maintaining consistency across environments.

Overall, this project highlights the practical benefits of adopting DevOps practices and CI/CD pipelines in modern software development. Future enhancements could focus on integrating advanced monitoring tools, expanding multi-cloud deployment capabilities, and implementing more sophisticated rollback mechanisms to further optimize the delivery process.