Final Graduation project 2024

Automated Deployment Pipeline with Jenkins and Docker

Supervised By: Eng. Ahmed Mourad

Presented By:

1. Ahmed Khamis Abdelmoniem
2. Ahmed Elsayed Ghanem
3. Nourhan Ashour
4. Mazen Mohamed

Table of Contents

Chapter 1 - Introduction:

1.Introduction:

In the rapidly evolving world of software development, the ability to continuously integrate and deploy code has become crucial for maintaining competitive advantage. DevOps practices aim to bridge the gap between development and operations, fostering collaboration and enabling automation across the software lifecycle. Continuous Integration and Continuous Deployment (CI/CD) pipelines play a pivotal role in automating the process of building, testing, and deploying applications, ensuring faster delivery with fewer errors.

This project focuses on implementing an automated CI/CD pipeline using industry-standard tools such as **Jenkins**, **Docker**, and **Ansible**, with optional **Kubernetes** integration for container orchestration. By automating these processes, development teams can focus on writing high-quality code, while the deployment, testing, and infrastructure management tasks are handled seamlessly.

The objective of this project is to design, develop, and deploy a fully automated pipeline that integrates continuous integration and deployment practices into a cloud environment. Jenkins is utilized to manage the pipeline automation, Docker is used for containerizing the application, and Ansible for configuration management. Additionally, the project explores optional Kubernetes integration for scaling and orchestrating Docker containers.

Through this project, we aim to demonstrate the importance of automation in modern software development and how DevOps practices can significantly enhance the speed, reliability, and scalability of application deployment. The hands-on experience gained from building and refining this pipeline provides a solid foundation for understanding real-world DevOps implementations.

1.1 Overview of DevOps and CI/CD:

DevOps is a set of practices aimed at integrating software development and IT operations, promoting collaboration and automating processes to deliver high-quality software more quickly. By fostering a culture of shared responsibility, DevOps streamlines the development lifecycle and reduces manual work, making software delivery faster and more reliable.

CI/CD (Continuous Integration and Continuous Deployment) is a core DevOps practice. Continuous Integration (CI) automates the testing and merging of code changes to catch issues early, while Continuous Deployment (CD) automates the release of validated code to production. Together, CI/CD ensures faster releases, improved software quality, and reduced risks by identifying and addressing problems early in the process.

1.2 Tools and Technologies Used:

1. **Jenkins**: An open-source automation server used to build, test, and deploy applications. Jenkins orchestrates the entire CI/CD process, ensuring that code changes are automatically tested and deployed.
2. **Docker**: A containerization platform that packages the application and its dependencies into lightweight containers. Docker ensures consistency across different environments, making it easier to develop, test, and deploy applications.
3. **Ansible**: An automation tool for configuration management and application deployment. Ansible simplifies cloud infrastructure management by automating tasks such as server setup and application configuration using playbooks.
4. **Git**: A version control system used to manage code repositories. Git enables seamless collaboration by allowing multiple developers to contribute to the same codebase while maintaining a history of changes.
5. **Kubernetes**: A container orchestration platform used to manage and scale Docker containers. While not required for this project, Kubernetes can be integrated to manage more complex deployments across multiple servers.

6. **Cloud Platform (AWS)**: Cloud environment is used for deploying the application, ensuring scalability and availability. This platform allow for automated deployment and scaling of the application based on user demand.

Chapter 2- Project Overview:

2.1 Automated Deployment Pipeline:

An **Automated Deployment Pipeline** is a crucial component of modern software development, designed to streamline and automate the processes of building, testing, and deploying applications. The primary goal of this pipeline is to enable rapid and reliable delivery of software by minimizing manual intervention and reducing the likelihood of errors.

The pipeline consists of several key stages:

1. **Source Code Management**: The process begins with developers committing code changes to a version control system, such as Git. These changes are automatically tracked, allowing teams to collaborate efficiently and maintain a history of modifications.
2. **Continuous Integration (CI)**: Upon committing code, the CI server (Jenkins) is triggered to build the application. This includes compiling the code and running automated tests (unit tests, integration tests) to ensure that the changes do not introduce new bugs. The CI process ensures that every code change is verified before it is merged into the main codebase.
3. **Containerization**: Once the code is successfully built and tested, it is packaged into a Docker container. This encapsulation ensures that the application runs consistently across different environments by including all necessary dependencies and configurations.
4. **Configuration Management**: Using tools like Ansible, the pipeline automates the provisioning and configuration of infrastructure. Ansible playbooks are utilized to set up the required environments, install necessary software, and deploy the Docker containers to the target servers.
5. **Continuous Deployment (CD)**: After successful testing and configuration, the application is automatically deployed to the production environment. This ensures that new features, enhancements, or fixes are delivered to users quickly and consistently. The deployment process is monitored to ensure that any issues can be quickly identified and resolved.
6. **Monitoring and Feedback**: Post-deployment, the application is continuously monitored for performance and reliability. Metrics and logs are

collected to provide feedback on the deployment, allowing for rapid identification of any potential issues.

By implementing an automated deployment pipeline, organizations can achieve faster release cycles, improved software quality, and a more efficient development process. This approach not only enhances team productivity but also allows businesses to respond swiftly to changing market demands and user feedback.

2.2 Importance of CI/CD in Modern Development:

Continuous Integration and Continuous Deployment (CI/CD) are vital in today's software development landscape, providing numerous benefits:

1. **Faster Time to Market**: CI/CD automates the integration and deployment processes, enabling quicker delivery of features and fixes in response to market demands.
2. **Improved Software Quality**: Automated testing identifies and resolves bugs early in the development cycle, leading to higher-quality software with fewer defects.
3. **Reduced Risk**: Smaller, more frequent releases lower the risk associated with deployments, making it easier to identify and address issues promptly.
4. **Enhanced Collaboration**: CI/CD fosters teamwork between development, testing, and operations, improving communication and efficiency across teams.
5. **Greater Visibility**: CI/CD pipelines offer clear insights into the development process through automated logs and reports, aiding decision-making and tracking progress.
6. **Scalability and Flexibility**: CI/CD adapts to various project sizes and requirements, making it suitable for growing organizations.
7. **Continuous Improvement**: By integrating user feedback and metrics, CI/CD encourages iterative product enhancements, boosting customer satisfaction.

2.3 Key Technologies: Jenkins, Docker, Ansible, Kubernetes:

1. **Jenkins**: An open-source automation server that orchestrates the CI/CD pipeline by automating the build, testing, and deployment processes. It integrates with version control systems to trigger builds and validate code changes.
2. **Docker**: A containerization platform that packages applications and their dependencies into portable containers. Docker ensures consistent application performance across different environments, simplifying deployment and enhancing scalability.
3. **Ansible**: An automation tool for configuration management and application deployment, using a simple, agentless architecture. Ansible automates the provisioning and configuration of infrastructure, ensuring consistent environments across servers.
4. **Kubernetes**: An open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. While optional in this project, Kubernetes enhances reliability and availability, making it ideal for managing larger deployments.

These technologies work together to streamline the software development lifecycle, promote automation, and facilitate the efficient delivery of high-quality software.

Chapter 3- Week 1: Initial Setup & Planning:

3.1 Installing Jenkins and Docker:

The initial setup of the development environment is crucial for establishing a solid foundation for the automated deployment pipeline. In this phase, we focused on installing two key components: **Jenkins** and **Docker**. These tools will serve as the backbone of our CI/CD process.

1. Installing Jenkins:

Jenkins is an open-source automation server that facilitates continuous integration and delivery. To install Jenkins, we followed these steps:

- **System Requirements**: Before installation, we ensured that our system met the necessary prerequisites, including Java (JDK 8 or higher) since Jenkins runs on the Java platform.
- **Download and Install**: We downloaded the latest stable version of Jenkins from the official Jenkins website. The installation can be performed on various operating systems, including Windows, macOS, and Linux. We opted for a Linux-based server for our setup.
- **Setup Process**: After downloading, we ran the installer and followed the on-screen instructions. During installation, Jenkins creates a default user with administrative privileges.
- **Accessing Jenkins**: Once installed, we accessed Jenkins via a web browser using the default port (8080). The initial setup wizard guided us through the configuration process, including installing recommended plugins and setting up user authentication.

2. Installing Docker:

Docker is essential for containerizing applications and ensuring consistent deployment across environments. The installation process involved the following steps:

- **System Requirements**: We verified that our system was compatible with Docker. This included ensuring a supported version of the operating system and adequate resources (CPU, RAM, and disk space).
- **Download and Install**: We installed Docker by following the official installation guide available on the Docker website. This included adding the Docker repository to our package manager, updating the package index, and installing Docker Engine.
- **Post-Installation Setup**: After installation, we added our user to the Docker group to allow running Docker commands without sudo. We verified the installation by running the command docker --version to check the installed version.

- **Running a Test Container**: To confirm that Docker was functioning correctly, we pulled and ran the "hello-world" image. This simple test confirmed that Docker could download images and run containers without issues.

  With Jenkins and Docker successfully installed, we established a robust environment for the subsequent phases of the project. These tools will facilitate the automation of the build, testing, and deployment processes, laying the groundwork for our CI/CD pipeline.

3.2 Dockerizing the Application:

Dockerizing the application involves packaging it and its dependencies into a Docker container to ensure consistent performance across environments. This process includes several key steps:

1. **Application Structure**: We organized our application with a clear directory structure, including source code, a Dockerfile, and configuration files.
2. **Creating the Dockerfile**: A Dockerfile was created with instructions such as:
   - Specifying a base image (e.g., Node.js).
   - Setting the working directory within the container.
   - Copying application files and installing dependencies.
   - Exposing necessary ports and defining the command to run the application.
3. **Building the Docker Image**: The Docker image was built using the command:

   ```
   docker build -t myapp:latest .
   ```

4. **Running the Docker Container**: We executed the container with port mapping to access the application via the host machine:

   ```
   docker run -p 3000:3000 myapp:latest
   ```

5. **Testing the Application**: After running the container, we verified the application’s functionality through a web browser.

3.3 Setting Up Ansible for Configuration Management:

Ansible is used for automating configuration management and application deployment. The setup process involved the following steps:

1. **Installation of Ansible**: We installed Ansible on the control machine, ensuring that Python and SSH access were available. Installation was verified with the command:

   ansible --version

2. **Setting Up Inventory File**: An inventory file named hosts was created to define the target servers for configuration. For example:

```
hosts
You, 1 second ago | 2 authors (You and one other)
1  [docker_server]
2  54.204.216.90  ansible_ssh_private_key_file=/home/ahmedkhamis/Final-DEPI-Project/Gh-test.pem ansible_user=ec2-user ansible_python_interpreter=/usr/bin/python3.9
```

3. **Writing Ansible Playbooks**: We created a playbook (deploy-docker.yaml) to automate tasks such as installing necessary packages, deploying application files, and starting services.

```yaml
- name: setup docker with ansible
  hosts: docker_server
  become: yes

  tasks:
    - name: install docker
      yum:
        name: docker
        update_cache: yes
        state: present

- name: start docker
  hosts: docker_server
  become: yes
  tasks:
    - name: start docker
      systemd:
        name: docker
        state: started
        enabled: yes
    - name: add ec2-user to docker group
      user:
        name: ec2-user      You, 2 weeks ago • Update Ansible playbook
        groups: docker
        append: yes

- name: install nodejs
  hosts: docker_server
  become: yes
  tasks:
    - name: install nodejs
      yum:
        name: nodejs
        update_cache: yes
        state: present
```

```yaml
- name : install docker-compose
  hosts: docker_server
  become: yes
  tasks:
    - name: install docker-compose
      get_url:
        url: https://github.com/docker/compose/releases/latest/download/docker-compose-Linux-x86_64
        dest: /usr/local/bin/docker-compose
        mode: +x

- name: copy docker-compose file
  hosts: docker_server
  tasks:
    - name: copy docker-compose file
      copy:
        src: ./mongo.yaml
        dest: /home/ec2-user/mongo.yaml

- name: copy utility app
  hosts: docker_server
  tasks:
    - name: copy /utility app
      copy:
        src: ./utility-app
        dest: /home/ec2-user/

- name: install aws-cli
  hosts: docker_server
  become: yes      You, 2 weeks ago • Update Ansible playbook
  tasks:
    - name: install aws-cli
      yum:
        name: aws-cli
        update_cache: yes
        state: present

- name: deploy docker-compose
  hosts: docker_server
  tasks:
    - name: deploy docker-compose
      become: true
      become_user: root
      command: docker-compose -f /home/ec2-user/mongo.yaml up -d
```

4. **Running Ansible Playbooks**: The playbook was executed using:

   ansible-playbook deploy-docker.yaml

5. **Verifying Configuration**: After running the playbook, we confirmed that all tasks were successfully completed, ensuring the application environment was properly configured.

```
ahmedkhamis@DESKTOP-ME888PM:~/Final-DEPI-Project$ ansible-playbook deploy-docker.yaml

PLAY [setup docker with ansible] ***********************************************

TASK [Gathering Facts] *********************************************************
ok: [98.82.2.33]

TASK [install docker] **********************************************************
ok: [98.82.2.33]

PLAY [start docker] ************************************************************

TASK [Gathering Facts] *********************************************************
ok: [98.82.2.33]

TASK [start docker] ************************************************************
ok: [98.82.2.33]

TASK [add ec2-user to docker group] ********************************************
ok: [98.82.2.33]

PLAY [install nodejs] **********************************************************

TASK [Gathering Facts] *********************************************************
ok: [98.82.2.33]

TASK [install nodejs] **********************************************************
ok: [98.82.2.33]

PLAY [install docker-compose] **************************************************

TASK [Gathering Facts] *********************************************************
ok: [98.82.2.33]

TASK [install docker-compose] **************************************************
ok: [98.82.2.33]
```

```
TASK [install docker-compose] **************************************************
ok: [98.82.2.33]

PLAY [copy docker-compose file] ************************************************

TASK [Gathering Facts] *********************************************************
ok: [98.82.2.33]

TASK [copy docker-compose file] ************************************************
changed: [98.82.2.33]

PLAY [copy utility app] ********************************************************

TASK [Gathering Facts] *********************************************************
ok: [98.82.2.33]

TASK [copy /utility app] *******************************************************
ok: [98.82.2.33]

PLAY [install aws-cli] *********************************************************

TASK [Gathering Facts] *********************************************************
ok: [98.82.2.33]

TASK [install aws-cli] *********************************************************
ok: [98.82.2.33]

PLAY [deploy docker-compose] ***************************************************

TASK [Gathering Facts] *********************************************************
ok: [98.82.2.33]

TASK [deploy docker-compose] ***************************************************
changed: [98.82.2.33]
```

```
TASK [deploy docker-compose] ***************************************************
changed: [98.82.2.33]

PLAY RECAP *********************************************************************
98.82.2.33                 : ok=17   changed=2    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

3.4 Pipeline Planning and Documentation:

Effective planning and documentation are essential for implementing an automated deployment pipeline. This phase involved the following steps:

1. **Defining the Pipeline Structure**: We outlined the key stages of the CI/CD pipeline, including source code management, continuous integration (CI), Dockerization, configuration management with Ansible, continuous deployment (CD), and monitoring.
2. **Documenting the Pipeline Processes**: Comprehensive documentation was created, including:
    - Step-by-step instructions for setting up each pipeline component.
    - Configuration details for the Dockerfile, Ansible playbooks, and Jenkins jobs.
    - Flow diagrams illustrating the pipeline workflow.
    - Best practices for maintaining the pipeline.
3. **Collaboration and Review**: We engaged team members to review the documentation, ensuring clarity and completeness. This collaborative approach helped identify gaps and refine the plans.

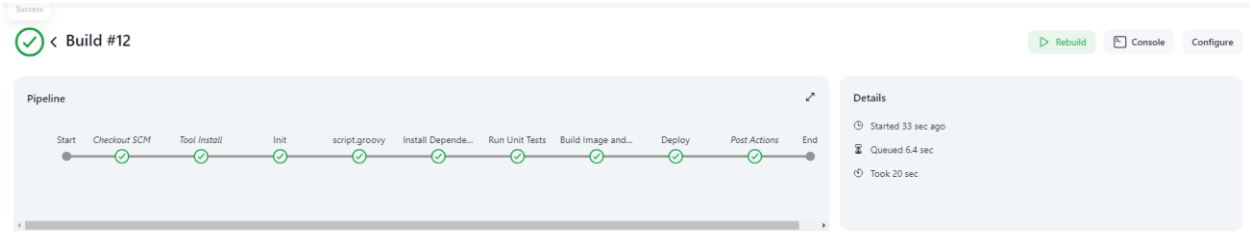
By thoroughly planning and documenting the pipeline, we established a clear roadmap for implementation and created a valuable resource for future enhancements and maintenance.

Chapter 4- Week 2: Jenkins & CI Integration:

4.1 Create Jenkins Jobs:

Creating Jenkins jobs is essential for automating the CI/CD pipeline. The process involved the following steps:

1. **Setting Up a New Jenkins Job**: We configured a pipeline Project and integrated the Git repository to trigger automatic builds when new code is committed.
2. **Build Steps**: Jenkins was set to:
    - Build the application using a build tool (e.g., npm).
    - Run automated tests to validate code.
    - Dockerize the application by building a Docker image.
3. **Post-Build Actions**: After a successful build, Jenkins:
    - Pushed the Docker image to a repository (Docker Hub).
    - Sent notifications about the build status via email or Slack.
4. **Continuous Deployment** : We configured Jenkins to automatically deploy the application using Ansible This setup automated the integration, testing, and deployment processes, forming the core of the CI/CD pipeline.

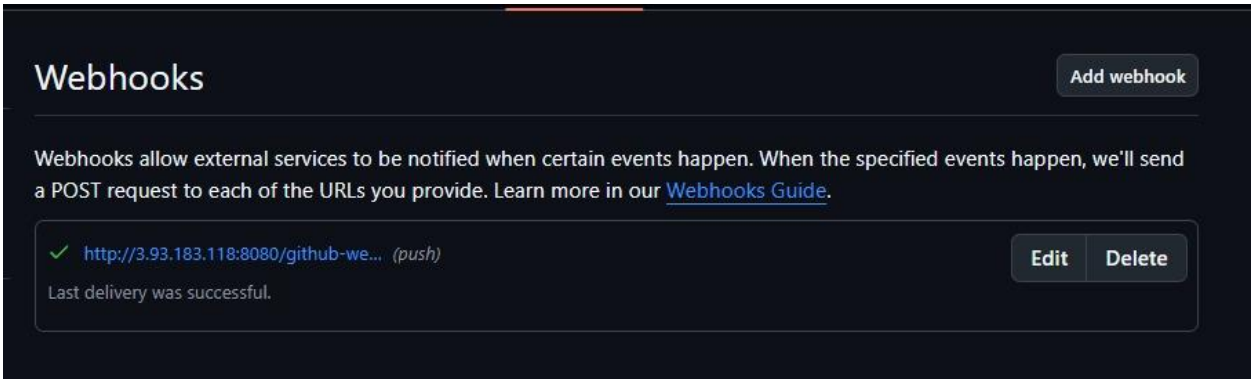


4.2 Integrate Git:

Integrating Git with Jenkins allows for automatic detection of code changes and triggers continuous integration. The process involved the following steps:

1. **Connecting Jenkins to the Git Repository**: We configured Jenkins with the Git repository URL and set up authentication using a personal access token.
2. **Branch Selection**: We specified the branch (e.g., main or test) to ensure Jenkins builds the appropriate code.
3. **Enabling Webhooks**: Webhooks were set up to notify Jenkins of changes in the repository, triggering automatic builds upon code commits.
4. **Verifying Integration**: We tested the integration by running a build, confirming that Jenkins successfully fetched the repository and triggered the build.
5. **Automating the Build Process**: With Git integrated, Jenkins automatically initiated the build process for every code change, streamlining continuous integration.

This integration ensures that code changes are continuously built and tested, forming a core component of the CI/CD pipeline.

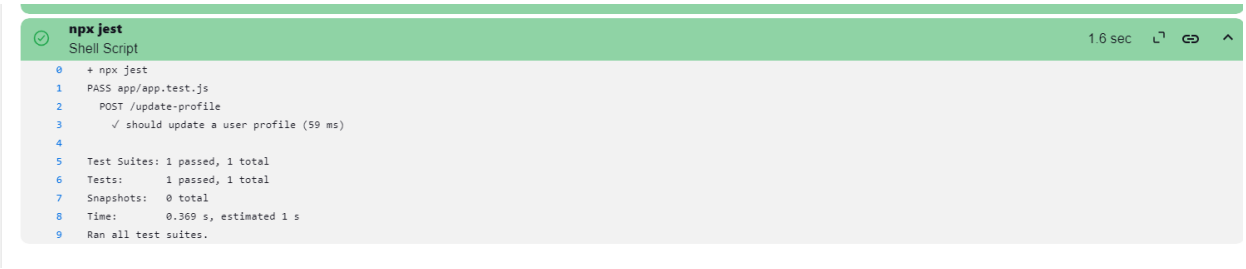


4.3 Automated Testing:

Automated testing ensures that code changes are validated before deployment. The process involved:

1. **Setting Up Tests**: We integrated unit tests into the Jenkins pipeline to automatically run with every build. These tests verify that the application functions as expected.
2. **Running Tests in Jenkins**: Jenkins was configured to run the tests after the build step. Any test failures halted the pipeline, preventing defective code from being deployed.

By automating testing, we ensured that every code change was thoroughly validated, reducing the risk of introducing bugs into production.

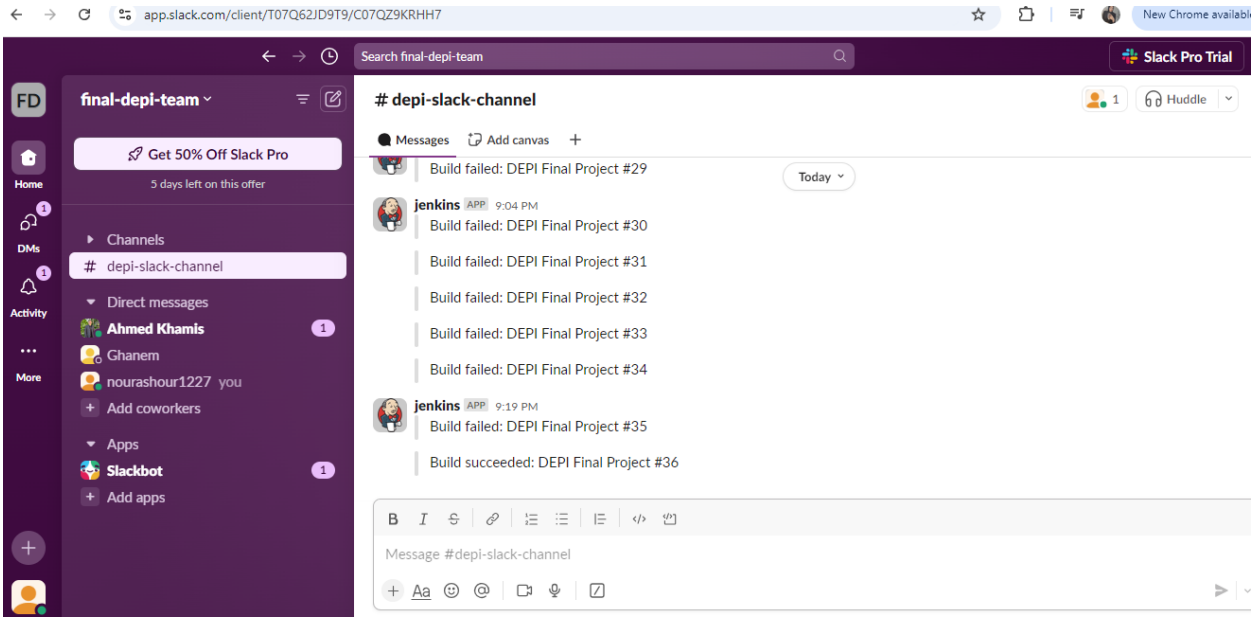


4.4 Set Up Notifications:

We configured notifications in Jenkins to keep the team informed of pipeline status:

1. **Email Notifications**: Set up to alert the team about build success or failure.
2. **Slack Integration**: Configured to send real-time build updates to a Slack channel.
3. **Custom Triggers**: Adjusted to send notifications based on specific conditions, such as failed builds.

This ensured quick responses to any pipeline issues.



Chapter 5- Docker & Deployment:

5.1 Integrate Docker Hub:

Integrating Docker Hub with Jenkins allows for automatic pushing of Docker images after successful builds. The process included:

1. **Configuring Docker Credentials**: Docker Hub credentials were added to Jenkins to authenticate and push images securely.
2. **Building and Tagging Docker Images**: Jenkins was configured to build the Docker image and tag it with latest.
3. **Pushing to Docker Hub**: After a successful build, Jenkins automatically pushed the Docker image to Docker Hub, making it available for deployment.

This integration ensures that the latest Docker images are readily available for production.

5.2 Configure Ansible for Deployment:

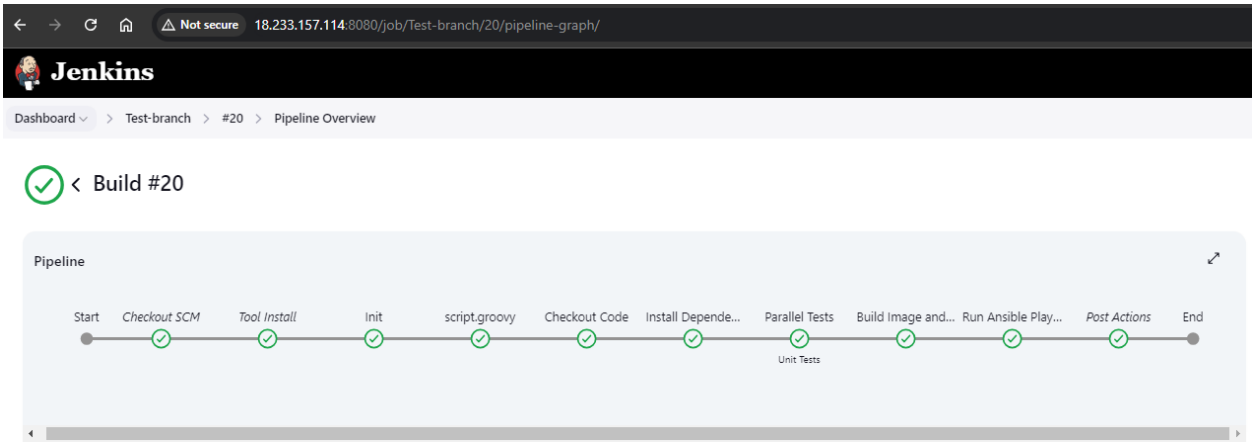
Ansible was used to automate the deployment process, ensuring consistent and efficient deployment across servers. The key steps included:

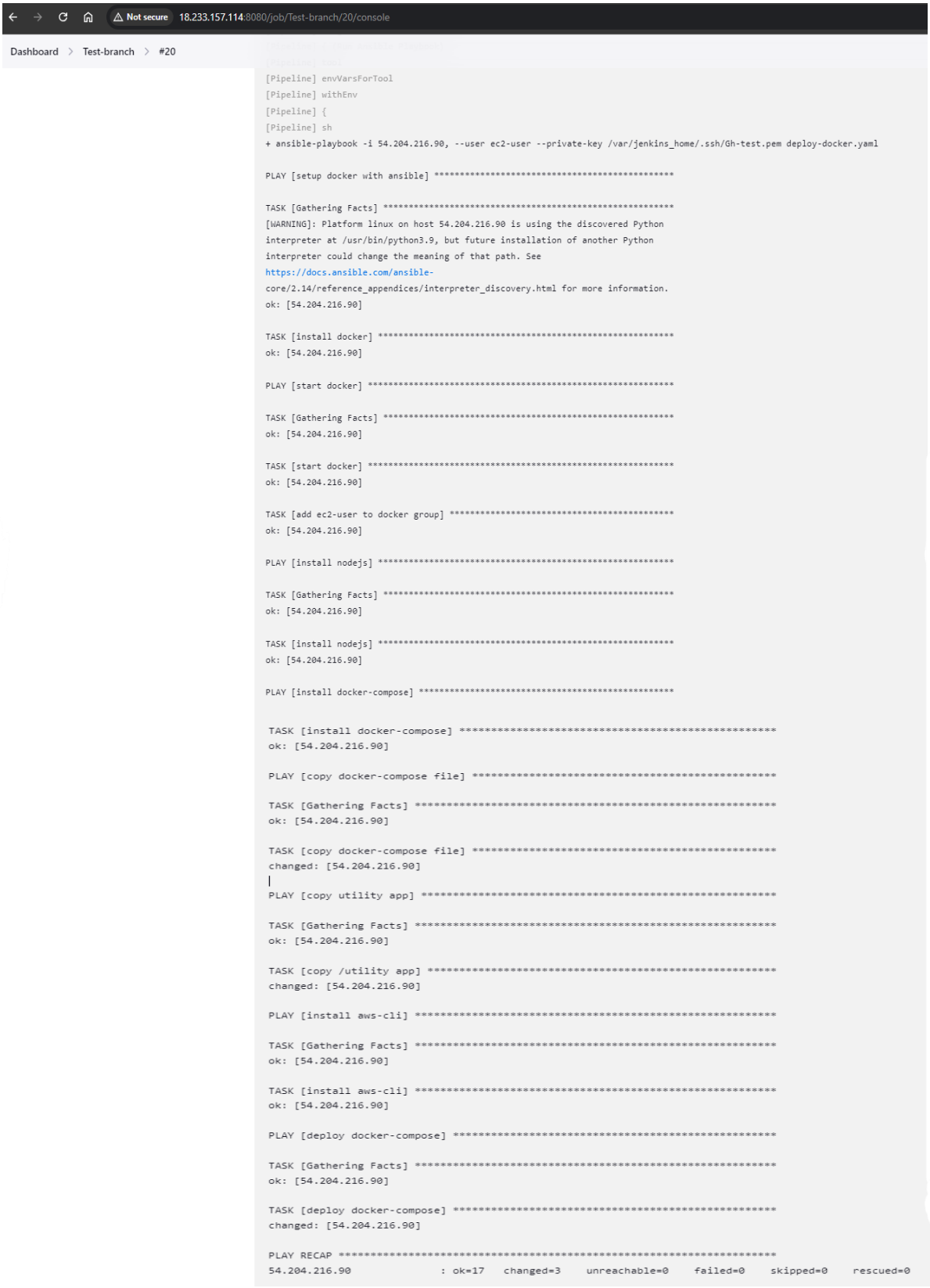
1. **Writing Playbooks**: We created Ansible playbooks to automate tasks such as installing dependencies, copying application files, and starting services on the target servers.
2. **Inventory File Setup**: An inventory file was defined to list the target servers where the application would be deployed.
3. **Jenkins Integration**: Jenkins was configured to trigger the Ansible playbooks after successful builds, ensuring the application was deployed seamlessly to the defined servers.

This setup allowed for fully automated, repeatable deployments, reducing manual intervention and ensuring consistency across environments.

```
    }
    stage('Run Ansible Playbook') {
        steps {
            // Run the playbook and specify the private key and user
            sh '''
                ansible-playbook -i 54.204.216.90, \
                --user ec2-user \
                --private-key /var/jenkins_home/.ssh/Gh-test.pem \
                deploy-docker.yaml
            '''
        }
    }
}
```

You, 8 minutes ago • Uncommitted changes



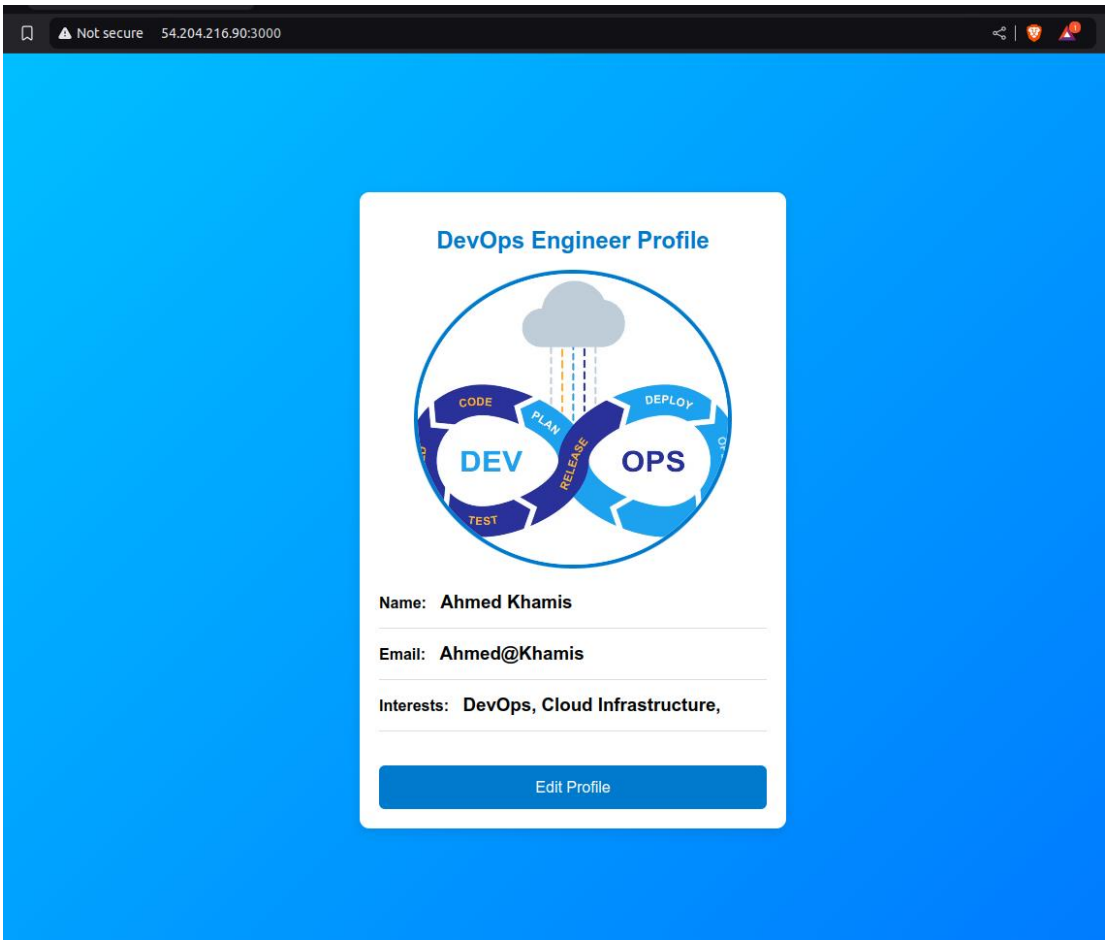


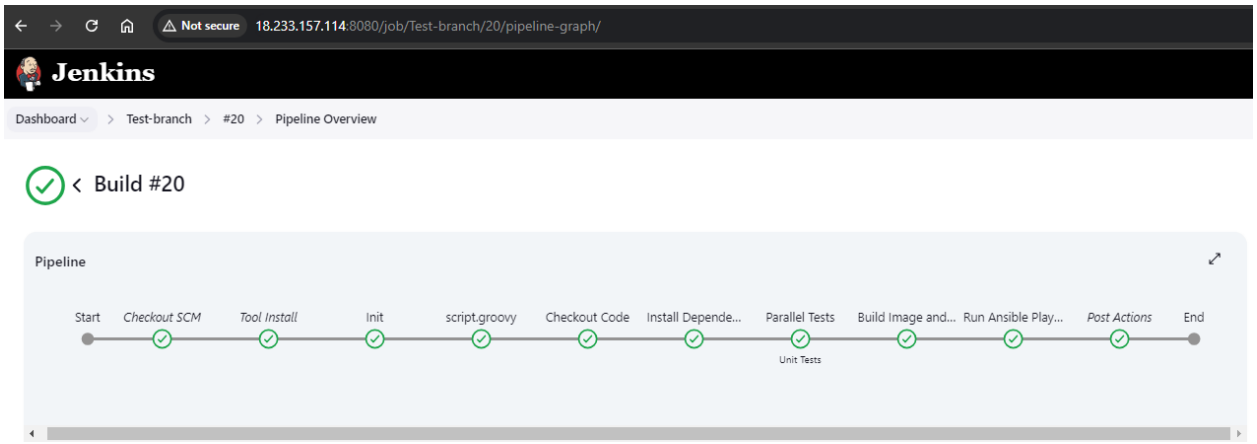
Not secure 18.233.157.114:8080/job/Test-branch/20/console

Dashboard > Test-branch > #20

```
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] sh
+ ansible-playbook -i 54.204.216.90, --user ec2-user --private-key /var/jenkins_home/.ssh/Gh-test.pem deploy-docker.yaml

PLAY [setup docker with ansible] ***********************************************

TASK [Gathering Facts] *********************************************************
[WARNING]: Platform linux on host 54.204.216.90 is using the discovered Python
interpreter at /usr/bin/python3.9, but future installation of another Python
interpreter could change the meaning of that path. See
https://docs.ansible.com/ansible-
core/2.14/reference_appendices/interpreter_discovery.html for more information.
ok: [54.204.216.90]

TASK [install docker] **********************************************************
ok: [54.204.216.90]

PLAY [start docker] ************************************************************

TASK [Gathering Facts] *********************************************************
ok: [54.204.216.90]

TASK [start docker] ************************************************************
ok: [54.204.216.90]

TASK [add ec2-user to docker group] ********************************************
ok: [54.204.216.90]

PLAY [install nodejs] **********************************************************

TASK [Gathering Facts] *********************************************************
ok: [54.204.216.90]

TASK [install nodejs] **********************************************************
ok: [54.204.216.90]

PLAY [install docker-compose] **************************************************

TASK [install docker-compose] **************************************************
ok: [54.204.216.90]

PLAY [copy docker-compose file] ************************************************

TASK [Gathering Facts] *********************************************************
ok: [54.204.216.90]

TASK [copy docker-compose file] ************************************************
changed: [54.204.216.90]

PLAY [copy utility app] ********************************************************

TASK [Gathering Facts] *********************************************************
ok: [54.204.216.90]

TASK [copy /utility app] *******************************************************
changed: [54.204.216.90]

PLAY [install aws-cli] *********************************************************

TASK [Gathering Facts] *********************************************************
ok: [54.204.216.90]

TASK [install aws-cli] *********************************************************
ok: [54.204.216.90]

PLAY [deploy docker-compose] ***************************************************

TASK [Gathering Facts] *********************************************************
ok: [54.204.216.90]

TASK [deploy docker-compose] ***************************************************
changed: [54.204.216.90]

PLAY RECAP *********************************************************************
54.204.216.90              : ok=17   changed=3    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

5.3 Deployment Testing:

- 

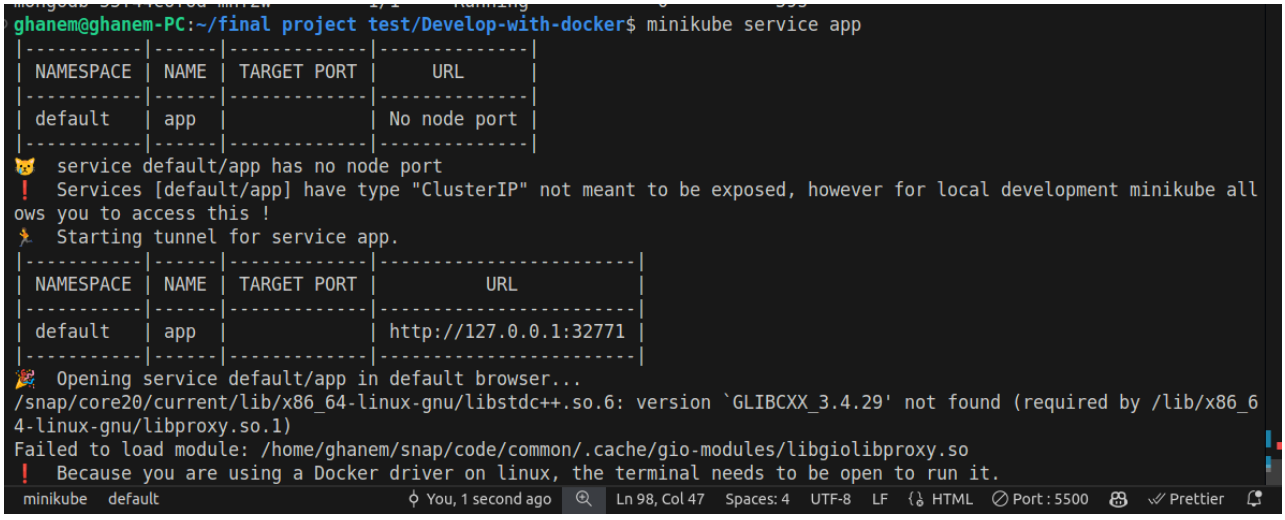


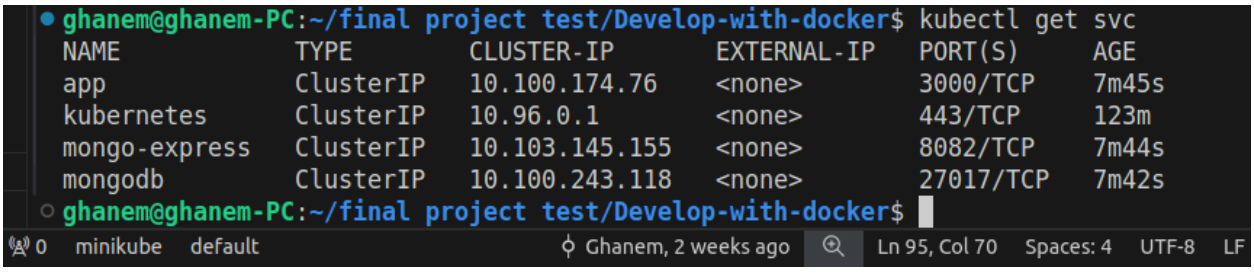
Chapter 6 - CI/CD Refinement & Final Testing:

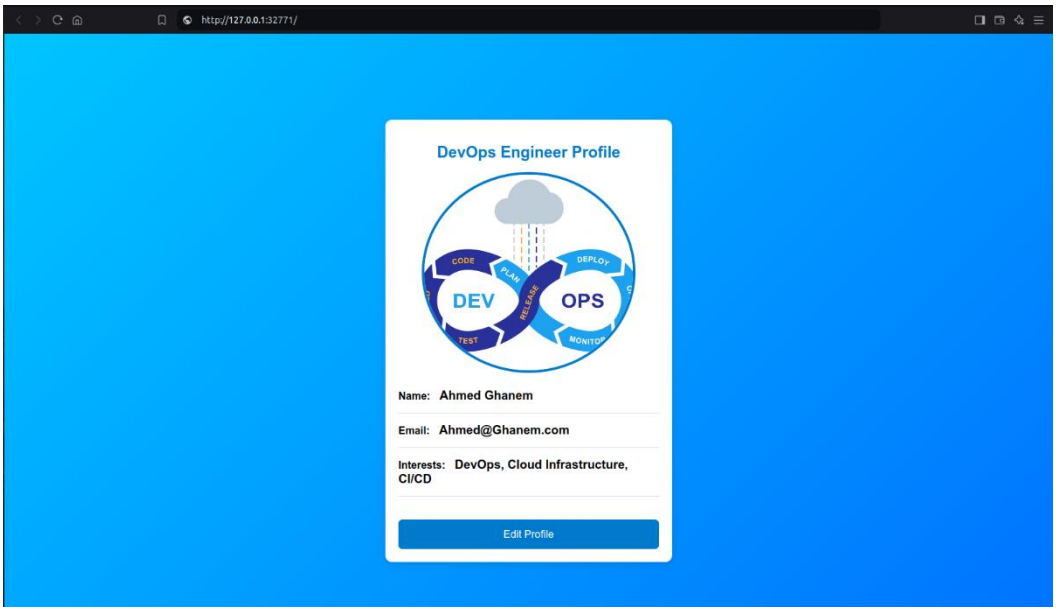
6.1 Kubernetes Integration:

Kubernetes was integrated to orchestrate and manage Docker containers in a scalable environment. The integration process included:

1. **Setting Up Kubernetes**: Kubernetes clusters were set up on minikube to manage containerized applications.
2. **Configuring Deployment Files**: Kubernetes deployment YAML files were created to define how the Docker containers should be deployed, scaled, and managed.



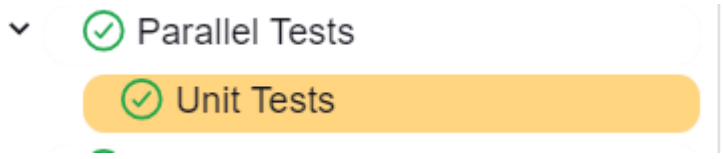


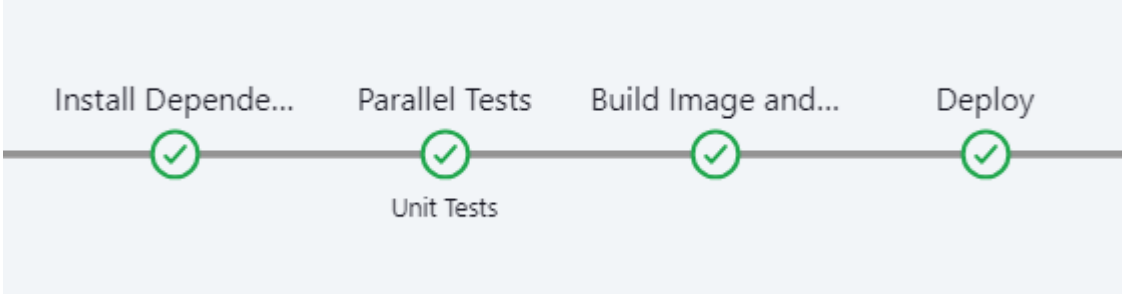


6.2 Refine the Pipeline:

Refining the pipeline focused on optimizing efficiency and enhancing performance. The process included:

1. **Adding Parallel Stages**: We introduced parallel stages in the Jenkins pipeline to run tests and builds simultaneously, significantly reducing overall execution time.





6.3 Reference:

https://github.com/AhMed-GhaNem25/Final-DEPI-Project.git