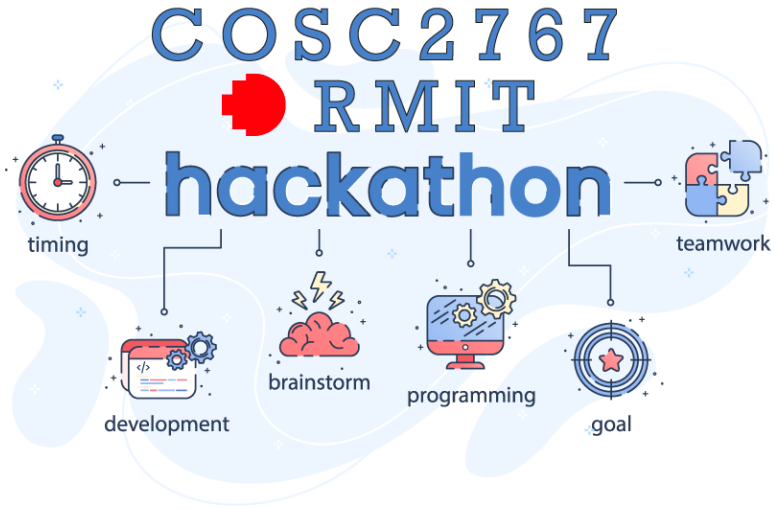
RMIT Classification: Trusted

Course: COSC2767 - System Deployment and Operations
Lecturer's Name: Mr. Tom Huynh
Submission Date: January 22nd, 2024

Assignment 3 - Group Project



Name	Student IDs
Hoang Ngoc Duan	s3817747
La Tran Hai Dang	s3836605
Huynh Van Anh	s3836320
Do Le Long An	s3963207
Pham Quang Man	s3804811

Video Demonstration Link: https://www.youtube.com/watch?v=WmzPL43t688

Table of Contents

I. The CI/CD Pipeline Solution Description:

- Brief problem statements.

As a web developer, Tom has built an RMIT E-commerce store, a website that sells different RMIT merchandising. He also decided to use LAMP Stack (Linux, Apache, MySQL/MariaDB, and PHP) as the technology behind his website. In addition, through a README.md file on his GitHub repository, he has made available the most recent commit as well as some basic instructions for configuring and launching the website on an CentOS EC2 Instance.
While Tom has multiple experiences in web development, he lacks the skill in DevOps and constructing a reliable and scalable Continuous Integration/Continuous Deployment (CI/CD) pipeline.

- The proposed solution for the pipeline.

To ensure that both the basic and expert requirements of Tom's project are met, our team – 4 Tech Team - aims to design a continuous integration/delivery (CI/CD) pipeline solution that guarantees automation, efficiency, stability, scalability, and repeatability while simultaneously automating the development and deployment process. This pipeline will help Tom as well as the web developers automate the process of building, testing, and deploying their website, making it more reliable and efficient.

- Tools and Technologies.

The pipeline will make use of a range of instruments and technologies:

- **GitHub:** a platform from which we will download, clone and host the RMIT online store's codebase. The CI/CD pipeline is started with any commits specifically made to the *PHP* (.php) file on the GitHub Repository. [1] We also use GitHub Action [2] for supporting automating our workflow.
- **Jenkins:** A free and open-source automation platform that helps programmers create, test, and release their applications. In our pipeline, Jenkins Server initiates Ansible Server upon a successful build. [3]
- **AWS CloudFormation:** A service that helps you model and set up your Amazon Web Services resources [4]. It is used to create the EC2 Instances including Apache Server, Jenkins Server, Ansible Server and Kubernetes master.
- **Ansible:** An open-source software provisioning, configuration management, and application-deployment tool enabling infrastructure as code [5]. This is the most important part of our pipeline as it allows the automation of the process of re-building and deploying the RMIT Store website.
- **Docker:** A platform that allows developers to automate the deployment, scaling, and management of applications within containers [6]. By utilizing Docker, we ensure that the image of RMIT Store is portable, lightweight and can run consistently across different environments.
- **Apache:** An open-source web server software. In our CI/CD pipeline, it is containerized on the Dev Server (RMIT Store) by running ansible playbook. Similarly, the Apache Image is run and hosted on Worker Nodes, which acts as the production server. [7]
- **Amazon RDS for MariaDB service:** An AWS service for seamless setup, operation and scale MariaDB server deployment in the cloud [8].
- **PHP:** A popular general-purpose scripting language that is especially suited to web development. It is part of the LAMP stack and is used for both client-side and server-side scripting. [9]
- **Network Load Balancer:** It directs traffic from the Worker Nodes to the Production Server so that the end users can access the website [10]. This is to ensure the availability of the application.

- **Kubernetes:** An open-source platform designed to automate deploying, scaling, and operating application containers. It manages the Worker Nodes and the Docker Apache Images within them. [11]
- **AWS CloudWatch:** A monitoring and observability service built for DevOps engineers, developers, site reliability engineers (SREs), and IT managers. [12]
- Features of this pipeline.

The pipeline provides an extensive list of processes, including:

- **Continuous Integration:** Every time a modification is made and pushed to the *PHP* file on GitHub, the pipeline automatically builds and publishes the latest version of the file.
- **Continuous Deployment:** After the code passes the automated tests for UI and Database Connection, the pipeline restarts the deployment on Kubernetes master node to update the latest RMIT Store image to the production server. As a result, the production environment is always isolated from the dev environment.
- **Scalability:** By utilizing AWS Auto Scaling and Load Balancer, the pipeline can accommodate a growing volume of work.
- **Availability:** By consistently testing the dev environment, the production server is isolated from any failed job that happened when the web developers committed to changes. As a result, the end users will always be able to see a running RMIT Store website without any issues.
- **Repeatability:** The process can be consistently repeated over time, ensuring reliability of the CI/CD pipeline.
- **Consistency:** From development on the local workstation to the production server, consistent environments are made possible by the Dockerfile and Docker Containerization.
- **Monitoring:** AWS CloudWatch makes sure that the RMIT Store hosting on EC2 Instance is running healthy. In addition, by integrating Gmail SMTP Server with Jenkins build job, we are constantly updated with the build logs and its status in our email.
- **Security:** The credentials for logging in Docker Hub as well as Jenkins Build Job are all managed by GitHub Secret.

II. RMIT Store CI/CD configuration steps from Main Requirement to Advanced requirement:

EC2 Instance Configuration (AWS CloudFormation):

As we embarked on building the CI/CD Pipeline for our project, we realized the traditional method of manually launching EC2 instances was not efficient. The repetitive task of editing user data script for each new EC2 instance became cumbersome and violated the DRY Principle in DevOps. This led me to explore AWS CloudFormation and transform our workflow by allowing us to define the AWS resources including setting up EC2 Instances, that this project required. The service took care of the rest, handling the provisioning and configuration of these AWS resources. This shift not only streamlined our project but also introduced a level of automation that was previously missing. Based on the requirement, those configurations will be customized into

an AWS CloudFormation template. Consequently, we will just proceed with ***rmit-store.yml*** for every instance configuration details. The following figures will focus on its first part:

```yaml
RMITStoreDatabaseEC2Instance:
  Type: AWS::EC2::Instance
  Properties:
    ImageId: ami-0ed9277fb7eb570c9
    InstanceType: t2.micro
    KeyName: s3963207_asm2_key
    SecurityGroupIds:
      - JenkinsSecurityGroup
    Tags:
      - Key: Name
        Value: TestRMIT-Store_DB
    Monitoring: true
    UserData:
    UserData:
      Fn::Base64: !Sub |
        #!/bin/bash
```

```yaml
AWSTemplateFormatVersion: "2010-09-09"
Description: "This is a template for launching a EC2 Instance that host a RMITStore Server"
Resources:
  JenkinsEC2Instance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: ami-0ed9277fb7eb570c9
      InstanceType: t2.micro
      KeyName: s3963207_asm2_key
      SecurityGroupIds:
        - JenkinsSecurityGroup
      Tags:
        - Key: Name
          Value: RMIT_Store_Jenkins
      Monitoring: true
      UserData:
        Fn::Base64: !Sub |
          #!/bin/bash

          echo "sudo su -" >> /home/ec2-user/.bashrc
          public_ipv4=$(curl -s http://checkip.amazonaws.com/)

          sudo su -

          echo '
          #!/bin/bash
```

```yaml
  AnsibleEC2Instance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: ami-0ed9277fb7eb570c9
      InstanceType: t2.micro
      KeyName: s3963207_asm2_key
      SecurityGroupIds:
        - AnsibleSecurityGroup
      Tags:
        - Key: Name
          Value: TestRMIT-Store_Ansible
      Monitoring: true
      UserData:
        Fn::Base64: !Sub |
          #!/bin/bash
```

```yaml
KubectlEC2Instance:
  Type: AWS::EC2::Instance
  Properties:
    ImageId: ami-0005e0cfe09cc9050
    InstanceType: t2.micro
    IamInstanceProfile: LabInstanceProfile
    KeyName: s3963207_asm2_key
    SecurityGroupIds:
      - JenkinsSecurityGroup
    Tags:
      - Key: Name
        Value: Rmit_Store_Kubectl
    Monitoring: true
    UserData:
      Fn::Base64: !Sub |
        #!/bin/bash
```

```yaml
RMITStoreEC2Instance:
  Type: AWS::EC2::Instance
  Properties:
    ImageId: ami-0ed9277fb7eb570c9
    InstanceType: t2.micro
    KeyName: s3963207_asm2_key
    SecurityGroupIds:
      - JenkinsSecurityGroup
    Tags:
      - Key: Name
        Value: TestRMIT-Store
    Monitoring: true
    UserData:
      Fn::Base64: !Sub |
        #!/bin/bash

        echo "sudo su -" >> /home/ec2-user/.bashrc
        public_ipv4=$(curl -s http://checkip.amazonaws.com/)

        sudo su -
```

Figure 1: AWS CloudFormation YAML – EC2 Configuration

From the Resources part, this is where we define the AWS resources that we want to create and configure. Each instance configuration will be specified by:

- **Type: AWS::EC2::Instance**: We specify which type of this EC2 instance.
- **Properties**: This is where we're going to set the parameters of the EC2 instance.
  - **ImageId:** This determines the Amazon Machine Image (AMI) ID employed for launching the instance.
  - **InstanceType:** t2.micro: This means that a t2.micro instance is about to be launched, which represents a single of Amazon's multipurpose instance types.
  - **KeyName: Jenkins**: This relates to an existing pair of keys which offers protected access via SSH to the instances.
  - **SecurityGroupIds:** This means the security groups associated with this EC2 instance. Here, it's linked associate with the specific name of this instance type (Ex: Jenkins Instance -> JenkinsSecurityGroup).
  - **Tags: Key and Value:** Tags are key-value pairs associated with resources. For example, the Name is set as RMIT_Store_Jenkins for Jenkins Instance or RMIT_Store_DB for the web database.
- **Monitoring: true:** This enables specified monitoring of this instance. If configured to be true, Amazon CloudWatch is going to gather and analyze raw data from the EC2 instance at periods of one minute. If specified as false, data will be gathered every five minutes. The second part focuses on establishing and associating Elastic IPs with specific EC2 instances.
- **JenkinsEIP and RMITStoreEIP:** We describes all of these are identifications for the Elastic IP (EIP) assets within this CloudFormation stack. The type indicates that these assets are EIPs.

```
JenkinsEIP:
  Type: AWS::EC2::EIP
  Properties: {}

JenkinsEIPAssociation:
  Type: AWS::EC2::EIPAssociation
  Properties:
    AllocationId: !GetAtt JenkinsEIP.AllocationId
    InstanceId: !Ref JenkinsEC2Instance

RMITStoreEIP:
  Type: AWS::EC2::EIP
  Properties: {}

RMITStoreEIPAssociation:
  Type: AWS::EC2::EIPAssociation
  Properties:
    AllocationId: !GetAtt RMITStoreEIP.AllocationId
    InstanceId: !Ref RMITStoreEC2Instance
```

Figure 2: AWS CloudFormation – EIP Configuration

- **JenkinsEIPAssociation and RMITStoreEIPAassociation:** These segments link previously established EIPs to EC2 instances. The type specifies that these are referred to as EIP associations. The Properties section specifies the AllocationId connected with the EIPs in addition to the InstanceId of the EC2 instances that they should be linked with. The !GetAtt intrinsic function recovers the value of an attribute from an existing asset, whereas the!The Ref intrinsic function recovers the numerical value of a created parameter or assets.
- **RMITStoreRecoveryAlarm:** We define the logical identification of the CloudWatch Alarm within this specific CloudFormation stack.
- **Type: AWS::CloudWatch::Alarm:** This specifies that the resource is a CloudWatch Alarm.

```
RMITStoreRecoveryAlarm:
  Type: AWS::CloudWatch::Alarm
  Properties:
    AlarmDescription:
      Trigger a recovery when instance status check fails for 15
      consecutive minutes.
    Namespace: AWS/EC2
    MetricName: StatusCheckFailed_System
    Statistic: Minimum
    Period: "60"
    EvaluationPeriods: "15"
    ComparisonOperator: GreaterThanThreshold
    Threshold: "0"
    AlarmActions: [!Sub "arn:aws:automate:${AWS::Region}:ec2:recover"]
    Dimensions:
      - Name: InstanceId
        Value: !Ref RMITStoreEC2Instance
      - Name: InstanceId
        Value: !Ref RMITStoreEC2Instance
```

Figure 3: AWS CloudWatch Alarm

- **AlarmDescription:** This converts a string that describes the alarm. In this situation, it demonstrates that the alarm will set off the recovery process when the instance status inspect fails for 15 ongoing minutes.
- **Namespace: AWS/EC2:** This defines the domain name for the metric connected with the alarm.
- **MetricName: StatusCheckFailed_System:** It represents the name of the metric connected with the alarm.
- **Statistic: Minimum:** This defines the data point to be applied to the alarm's associated metric.
- **Period: "60":** This sets out the time in seconds for which the particular statistic will be calculated.
- **EvaluationPeriods: "15":** This defines the total amount of periods over which data can be compared to the established threshold. This enables us to automate the process of setting up and configuring a CloudWatch Alarm for system status monitors on an EC2 instance, which can be more effective and error-free than doing so manually. If the system status check fails for 15 minutes in a row, the alarm will activate a recovery action.
- **ComparisonOperator: GreaterThanThreshold:** This specifies which opposition operator should be applied when comparing a particular statistic and threshold value.
- **Threshold: "0":** This defines the value to which the particular data is compared.
- **AlarmActions: - !Sub "arn:aws:automate:${AWS::Region}:ec2.recover":** This provides what steps need to be taken when this alarm shifts from any other state to ALARM. In this case, it will initiate an EC2 recovery action.
- **Dimensions: - Name: InstanceId Value:!Ref RMITStoreEC2Instance:** This defines the dimensions of the alarm's associated metric. In this case, it defines the InstanceId dimension with the value RMITStoreEC2Instance.

- Lastly, **Outputs** section which defines the values that we can import into other stacks, return in response, or view on the AWS CloudFormation console:

```
Outputs:
  ElasticIP:
    Description: Elastic IP Value
    Value: !Ref RMITStoreEIP
```

Figure 4: AWS CloudFormation – Console Output

- **ElasticIP:** This represents the logical identification of the outcome in this particular CloudFormation stack.
- **Description: Elastic IP Value:** This returns a string describing the output. It means that the outcome will be an Elastic IP value.
- **Value:** This defines the value of the output. The !Ref intrinsic function can be utilized to obtain the value of a particular parameter or asset. In this scenario, it receives the value RMITStoreIP. This implies that the outcome will be the value of RMITStoreIP.

GitHub Action configurations:

Workflow file for this run
.github/workflows/docker-image-ci.yml at a13a4cc

```yaml
name: Docker Image CI

on:
  push:
    paths:
    - '**.php'

jobs:
  push_to_registry:
    name: Push Docker image for RMIT Store to Docker Hub
    runs-on: ubuntu-latest
    steps:
      - name: Check out the repo
        uses: actions/checkout@v4

      - name: Log in to Docker Hub
        uses: docker/login-action@f4ef78c080cd8ba55a85445d5b36e214a81df20a
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Extract metadata (tags, labels) for Docker
        id: meta
        uses: docker/metadata-action@9ec57ed1fcdbf14dcef7dfbe97b2010124a938b7
        with:
          images: laansdole/s3963207-rmit-store

      - name: Build and push Docker image
        uses: docker/build-push-action@3b5e8027fcad23fda98b2e3ac259d8d67585f671
        with:
          context: .
          file: ./utilities/Dockerfile
          push: true
          tags: laansdole/s3963207-rmit-store:${{ github.sha }}, laansdole/s3963207-rmit-store:latest
```

Figure 5: GitHub Action – Docker CI

a. Developing the workflow file for Docker Image CI GitHub Action

➔ **Name:** Choosing the meaningful name for the workflow

➔ **On:** The workflow is triggered on a push event when changes are pushed to files with the **.php** extension

➔ **Jobs:** The "**push_to_registry**" job runs on an ubuntu-latest environment.

➔ **Steps:**

- Using the **actions/checkout action** to check out the repository.
- **docker/login-action** command use to log in to Docker Hub with Docker Hub **username** and **password**.
- Using **docker/metadata-action** command to extract metadata include tags and labels of the Docker image.
- **Build and push Docker image** jobs uses the **docker/build-push-action** command to build and push the new Docker image to Docker Hub.

b. Developing the workflow file for Trigger Jenkins Pipeline GitHub Action

➔ **On:** The workflow named **"Docker Image CI"** uses to triggered by listening for completed workflow runs **(types: - completed)**.

➔ **Jobs:**

- Running on the **Ubuntu** latest environment with the named "**trigger-a-jenkins-job**" job.

Workflow file for this run
.github/workflows/jenkins-trigger.yml at 84ae1e4

```yaml
name: Trigger Jenkins Pipeline

on:
  workflow_run:
    workflows: ["Docker Image CI"]
    types:
      - completed

jobs:
  trigger-a-jenkins-job:
    runs-on: ubuntu-latest
    if: ${{ github.event.workflow_run.conclusion == 'success' }}
    steps:
      - id: triggerjenkinsjob
        uses: mickeygoussetorg/trigger-jenkins-job@v1
        with:
          jenkins-server: ${{ secrets.JENKINS_HOST }} # URL of the jenkins server. ex: http://myjenkins.acme.com:8080
          jenkins-job: ${{ secrets.JENKINS_JOB }} # The name of the jenkins job to run ex: BuildAndDeployJob
          jenkins-username: ${{ secrets.JENKINS_USERNAME }} # user name for accessing jenkins ex: s3963207
          jenkins-pat: ${{ secrets.JENKINS_TOKEN }} # personal Access token for accessing Jenkins ex: TOKEN_NAME
          poll-time: 10 # how often (seconds) to poll the jenkins server for results
          timeout-value: 1200 # How long (seconds) to poll before timing out the action
          verbose: true # true/false - turns on extra logging
```

Figure 6: GitHub Action – Jenkins Trigger

- *`if: ${{ github.event.workflow_run.conclusion == 'success' }})`* is a condition using to check if the latest workflow run was successful.

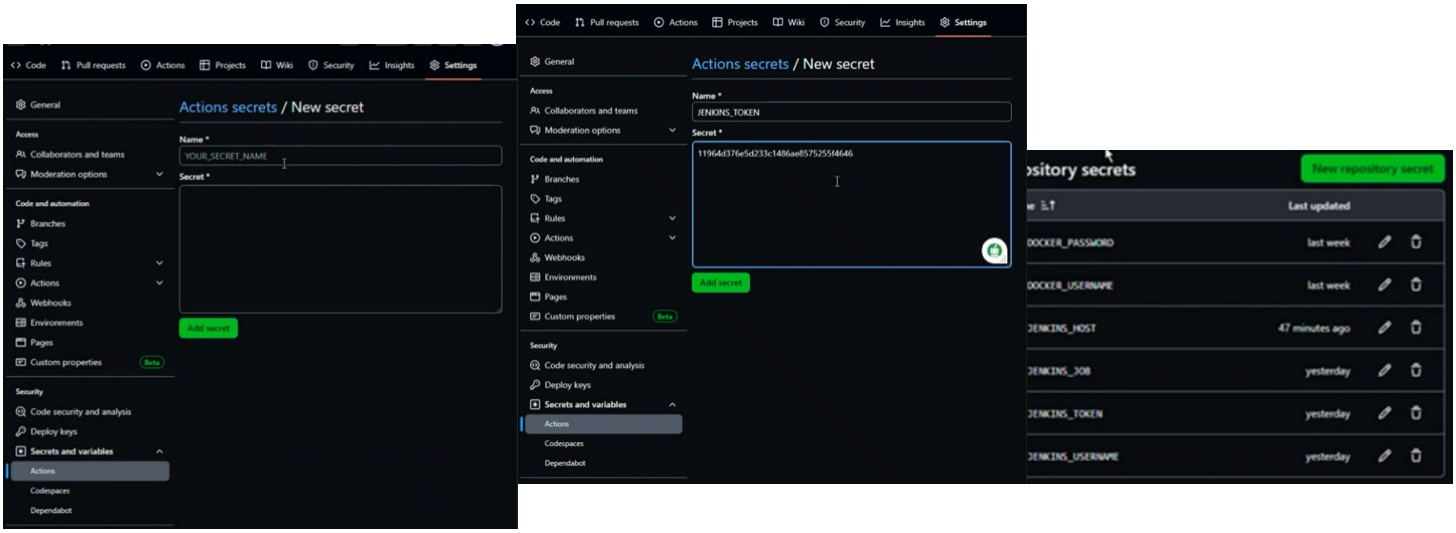
➔ **Steps:**

- The repository secrets **(JENKINS_HOST, JENKINS_JOB, JENKINS_USERNAME, and JENKINS_TOKEN)** configured in the GitHub repository to securely trigger the Jenkins job.
- The ***mickeygoussetorg/trigger-jenkins-job:v1*** is the GitHub Actions image to trigger the Jenkins job

c. The repository actions secrets configuration:

➔ In the GitHub repository, go to the **Settings** tab
➔ Choose the **Secrets and variables,** then choose **Actions**
➔ Choose the **Name** of the new secret.
➔ In the Secret, that contain the Username, Password, API Token, Private IP address that correspond with the Secrets on Workflow file for both **Trigger Jenkins Pipeline** and **Docker Image CI GitHub Action.**
➔ Click **Add secret.**

Figure 7: GitHub Action - Secrets



Ansible Playbook Explanation:

The Ansible Playbook comprises a variety of command lines that are done sequentially on a separate server, especially in this case, the Apache server integrated with Docker. The goal of sharing this Ansible playbook file is to facilitate the re-building and deployment of additional images and containers if Git detects any modifications or updates.
At the beginning of the Ansible playbook, we need to verify the host name with the private ip address by using the command:
hosts: <name of the target server>

Note: The ip address of the target server which have been added into the ***/etc/ansible/hostnames*** file
By login into the root account on the ansible server
For the **tasks** in the ansible playbook:

```
[dockerserver]
172.31.30.187
```

a. Start the Docker service.

- Using the ***service docker start*** command to start the docker service to ensure the docker is ready for deploying process.

```
- hosts: dockerserver
  tasks:
    - name: Start the docker service
      command: sudo service docker start
      ignore_errors: yes

    - name: Register all docker containers id are running
      command: sudo docker ps -aq
      register: dockerContainerIds

    - name: Register all current docker images id
      command: sudo docker images -aq
      register: dockerImagesIds

    - name: Stop all containers
      command: "sudo docker stop my-rmit-store"
      ignore_errors: yes

    - name: Remove all containers
      command: "sudo docker rm my-rmit-store"
      ignore_errors: yes
```

new commit change or update on GitHub.

Figure 8: Ansible Playbook – Stop Docker Containers

b. Register Docker Container ID

- The ***sudo docker ps -aq*** uses for listing all the running container id and store it in dockerContainerIds variable (Optional)

c. Register Docker Image ID

- The ***sudo docker images -aq*** uses for listing all the running images and store it in dockerImagesIds variable (Optional)

d. Stop and remove the container.

- Using ***docker stop <name/id of the container>*** command to stop the current running containers before removing the deprecated containers for building the new images and containers.
- Using ***docker rm<name/id of the container>*** command to remove the container that we want to replace why building the new container with the

e. Remove Docker images.

```
- name: Remove all images
  command: "sudo docker rmi --force laansdole/s3963207-rmit-store:latest"
  ignore_errors: yes
```

Figure 9: Ansible Playbook – Remove Docker images.

- ***docker rmi*** is used to remove images from the server with **--force** option to completely remove settings related to those images.
- The Docker image to be deleted is identified by the name and the tag, which provides a unique reference.

f. Run the new docker container.

```
- name: Run the apache container based on pushed docker hub image
  command: sudo docker run -d --name my-rmit-store --rm -p 80:80 laansdole/s3963207-rmit-store:latest
```

Figure 10: Ansible Playbook – Run Docker Container

- Finally, it executes a Docker container with container name using the specified Docker image and tag by ***docker run -d*** option used for disconnecting container, and **-p** for the ports **80:80** is assigned.

Dockerfile Explanation:

```
FROM centos:centos7.9.2009

# Install Apache
RUN yum -y update
RUN yum -y install httpd
RUN yum install -y mariadb-server
RUN yum install -y php php-mysql
RUN yum install -y git
```

A Dockerfile is provided in the Apache server that includes the installation steps to build a new Apache server image from **the CentOS 7.9.2009** base image.

➔ Installing packages required for deploying a LAMP stack including **Apache server, MariaDB server, PHP MySQL and Git.**

➔ Cloning the RMIT store web source code from Git repository into the Apache server root directory.

```
RUN git clone -b dev https://github.com/RMIT-DevOps-Hackathon/COSC2767-RMIT-Store.git /var/www/html/
# Install EPEL Repo
RUN rpm -Uvh https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm \
    && rpm -Uvh http://rpms.remirepo.net/enterprise/remi-release-7.rpm
```

Figure 11: Apache Dockerfile – First part

- ➔ Additional software packages such as EPEL and Remi repositories are also installed to increase usability and optimization for CentOS.
- ➔ ***sed - E -i -e*** command blocks are used to edit the Apache HTTP Server configuration file (httpd.conf) to grant override permissions and index directories.

```
RUN sed -E -i -e '/<Directory "\/var\/www\/html">/,/<\/Directory>/s/AllowOverrideNone/AllowOverriDe All/' /etc/httpd/conf/httpd.con
RUN sed -E -i -e 's/DirectoryIndex (.*)$/DirectoryIndex index.php \1/g' /etc/httpd/conf/httpd.conf
EXPOSE 80 443
# Start Apache
CMD ["/usr/sbin/httpd","-D","FOREGROUND"]
```

Figure 12: Apache Dockerfile – Second part

- ➔ Exposes external access to the web server from ports **80 and 443**.
- ➔ The container then starts the Apache web server is in the foreground.

Jenkins configurations:

- Connect to **Jenkins** server on EC2 Instance, which is created by **CloudFormation**, via SSH connection
- Login to the **root account** and start the Jenkins service
- Install the plugin (**Git, Publish Over SSH**) for supporting the CI configuration:
  - ➔ On the **Jenkins Dashboad**, go to the **Manage Jenkins** tab
  - ➔ Choose **Plugins**
  - ➔ Search and Install the two plugins, which are **Git server** and **Publish Over SSH**

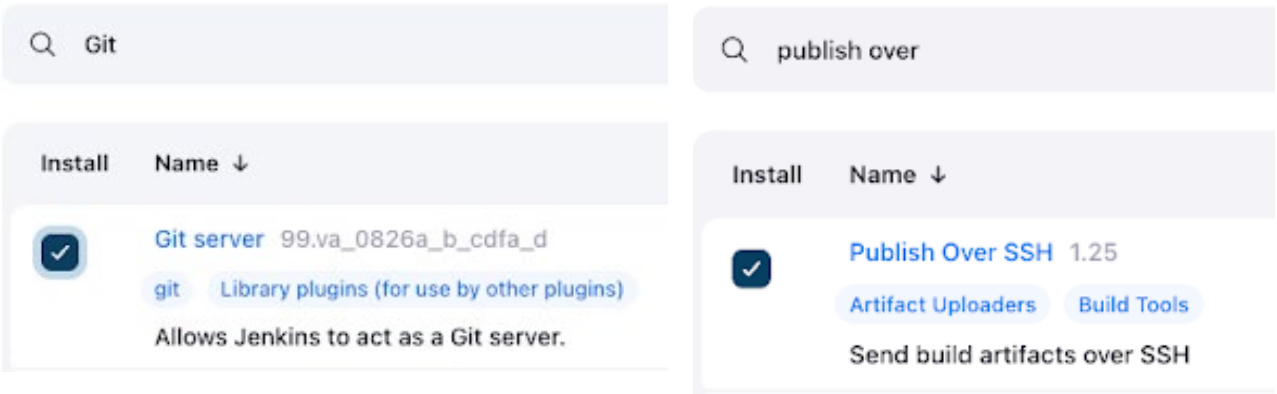


Figure 13: Jenkins Plugins

- **Configuration the Git installation:**
  - ➔ On the **Jenkins Dashboad**, go to the **Manage Jenkins**
  - ➔ Choose **Tools**
  - ➔ Go to the **Git installations** and change the name to **Git**

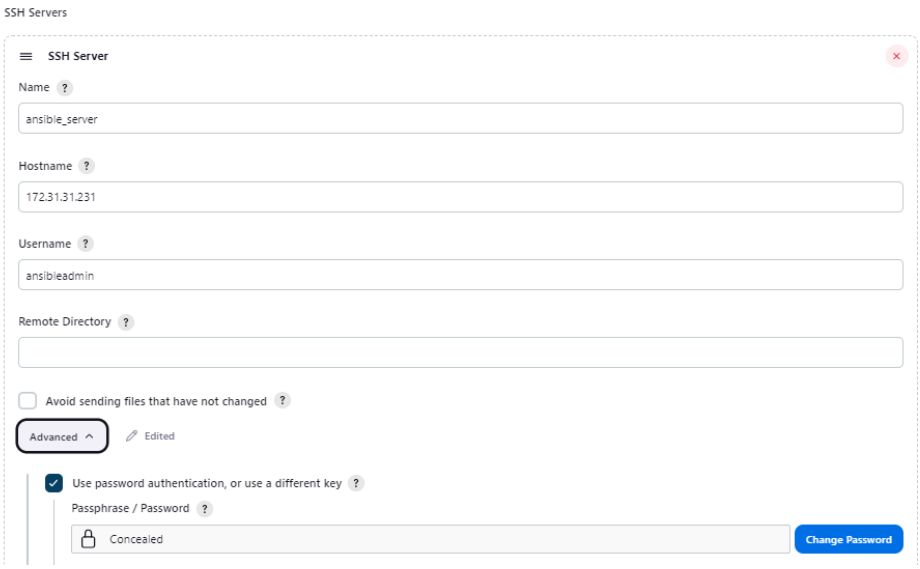


Figure 14: Jenkins SSH Server

Adding and configuration the SSH server to the Ansible target server:

- ➔ **Name:** Choosing the meaning full name for the server.
- ➔ **Hostname:** Choosing the Private IP address of the target server (Ansible server)
- ➔ **Username:** Entering the ansible username
- ➔ Choosing **Advanced**, then tick **Use pasword authentication, or use a different key**
- ➔ **Password:** Entering the password of the ansible account that we have created
- ➔ Click on **Test Connection,** if it **Success,** then **Apply and Save.**

Creating new Jenkins job:

- Creating a new Jenkins Items by choosing + **New Item** button
  - ➔ Choosing the meaningful name and description and **Freestyle project**

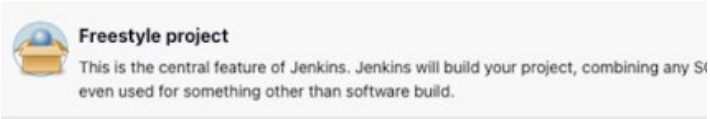


Figure 15: Jenkins Freestyle project

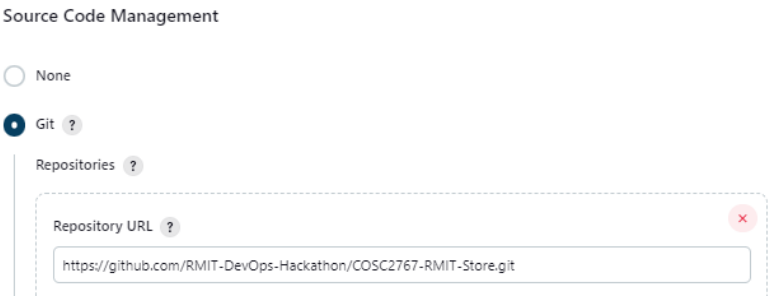
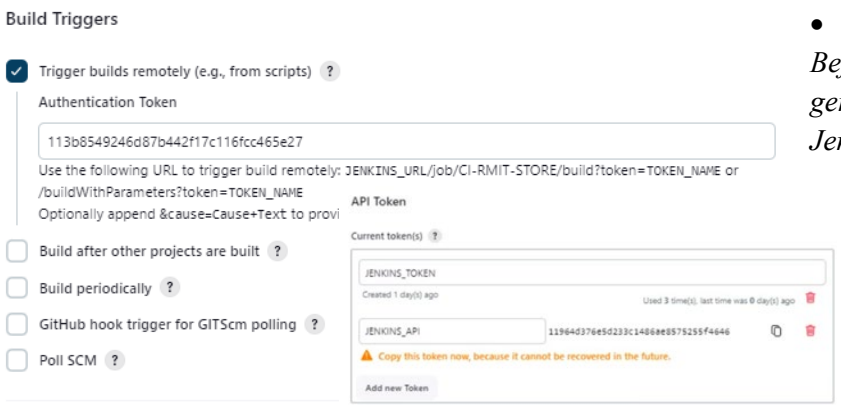


Figure 16: Jenkins job SCM

- **Configure the Source Code Management**
  - ➔ Choosing **Git** for the **Source Code Management (SCM)**
  - ➔ Using the GitHub repo URL for the **Repository URL**
  - ➔ Choosing the branch that we want to build (for our case is ***/dev** branch)



- **Configure the Build Triggers**

Before configuring the Build Triggers we need to generate the new API Token, which will be used for Jenkins and GitHub Repo Secrets

- ➔ Tick on the **Trigger builds remotely** and paste the token that we have created into the **Authentication Token**

Figure 17: Jenkins job Build Triggers

- **Configure the Build Environment**

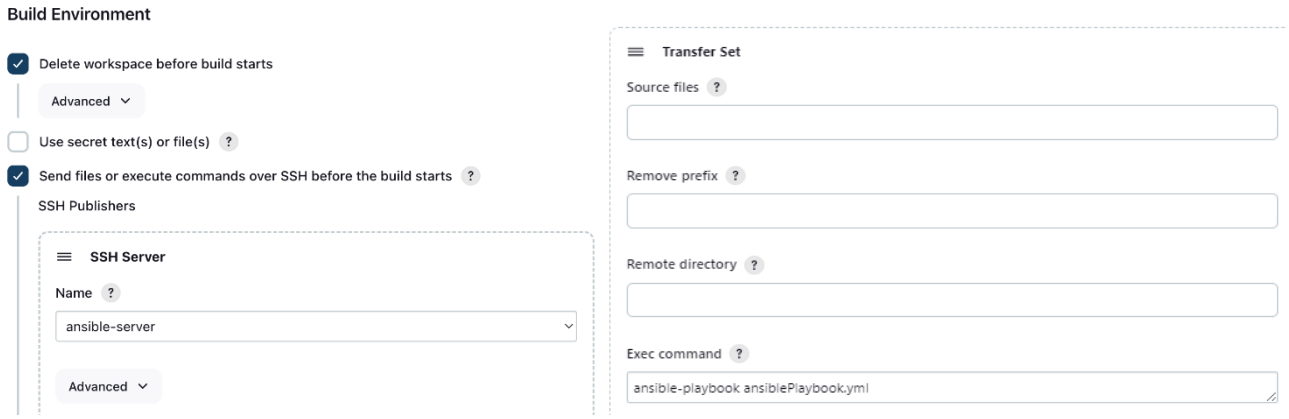


Figure 18: Jenkins job Build Environment

➔ Tick on the **Delete workspace before build starts** (To prevent the conflict between workspace)
➔ Tick on the **Send files or execute commands over SSH** before the build runs.
➔ Choosing the ansible name of the SSH server that we have created before.
➔ For the **Transfer Set,** using the ***`ansible-playbook <name of the Ansible Playbook>`*** command in **Exec command** to start running the ansible playbook on the Ansible server.

Implementing Testing Methodologies:

To ensure functionality and reliability, the provided README file outlines the steps for setting up a testing environment using Selenium, AWS, Python, Jenkins, and Git integration. Below is a step-by-step explanation of how to integrate and implement these testing methodologies in the given framework.

- Setting up Google Chrome and ChromeDriver:

Initial EC2 Installation on AWS

```
sudo yum update
sudo yum upgrade -y
```

Install Google Chrome Binary for Linux

```
sudo yum install -y liberation-fonts
sudo yum install -y https://dl.google.com/linux/direct/google-chrome-stable_current_x86_64.rpm
google-chrome --version
```

Install ChromeDriver Binary for Linux

```
sudo yum install -y unzip
wget https://chromedriver.storage.googleapis.com/98.0.4758.102/chromedriver_linux64.zip
sudo unzip chromedriver_linux64.zip
sudo mv chromedriver /usr/bin/chromedriver
chromedriver --version
```

Figure 19: Pytest Setup – Google Chrome and ChromeDriver

- **Setting Up Selenium and Python Dependencies:**

Check Python3 version

```
python3 --version
```

Check pip3 package manager

```
pip3 --version
```

Install Git on EC2 (upgrading Git to the latest version) - (optio nal)

```
git --version
```

General Moodle Selenium Script

```
pip3 install faker
pip3 install selenium
pip3 install pytest
pip3 install urllib3==1.26.8
pip3 install webdriver_manager

pytest test_conf.py
pytest test_dtb_conn.py
```

Figure 20: Pytest Setup – Selenium and Python Dependencies

- **Writing General Selenium Test Scripts:**

- Create ***test_conf.py*** and ***test_dtb_conn.py*** files with necessary Selenium test scripts.
- Use the provided Selenium fixtures and tests for testing different aspects of the web application.

- **Web UI Tests**

The ***test_conf.py*** is a Python script used as a configuration file for testing purposes. The given code includes Selenium test scripts that are created using the pytest framework. The main objective of this file is to provide test fixtures and test cases for doing automated testing of a web application.

Let's go through the ***test_conf.py*** code and understand its structure and purpose:

```
import pytest
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
```

Figure 21: test_conf.py import.

Here, the necessary libraries for writing Selenium tests are imported. ***pytest*** is used as the testing framework, and Selenium is used for web automation. The specific modules needed for ChromeDriver setup are also imported.

```
@pytest.fixture(scope="module")
def setup():
    url = "http://35.169.78.54"
    chrome_options = Options()

    options = [
        "--headless", "--disable-gpu", "--ignore-certificate-errors",
        "--disable-extensions", "--no-sandbox", "--disable-dev-shm-usage"
    ]
    for option in options:
        chrome_options.add_argument(option)

    driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()),
                              options=chrome_options)
    driver.set_page_load_timeout(30)
    driver.get(url)

    yield driver

    driver.quit()
```

This section defines a fixture named **setup** using the ***@pytest.fixture*** decorator. The fixture sets up a Chrome WebDriver with specific options for headless operation and other configurations. The WebDriver is created using ***ChromeDriverManager()*** to automatically download and manage the ChromeDriver binary. The fixture returns the WebDriver instance, and after the test has run (yielded), it quits the WebDriver.

Figure 22: test_conf.py setup

```python
@pytest.mark.parametrize("button_text, xpath", [
    ("Clothing", "//a[contains(text(), 'Clothing')]"),
    ("Accessories", "//a[contains(text(), 'Accessories')]"),
    ("Course", "//a[contains(text(), 'Course')]"),
    ("Stationery", "//a[contains(text(), 'Stationery')]"),
    ("Special-Collection", "//a[contains(text(), 'Special-Collection')]"),
    ("Sale", "//a[contains(text(), 'Sale')]"),
    ("Buy Now!", "//a[contains(text(), 'Buy Now!')]")
])
def test_button_is_enabled(setup, button_text, xpath):
    driver = setup

    try:
        button = driver.find_element("xpath", xpath)
        assert button.is_enabled()
        print(f"{button_text} button is enabled.")
    except Exception as e:
        print(f"Error occurred for {button_text} button: {str(e)}")
        raise

if __name__ == "__main__":
    pytest.main()
```

Figure 23: test_conf.py test enabled button

In this section, a parameterized test named ***test_button_is_enabled*** is defined using ***@pytest.mark.parametrize***. This test checks whether various buttons on the web page are enabled. The test receives parameters (***button_text*** and ***xpath***) for different buttons and attempts to find and verify the enabled state of each button using Selenium. If a button is not enabled, an error is printed, and the test fails.

The **setup** fixture is used here to provide the WebDriver instance to the test. The ***try...except*** block captures any exceptions during the test and prints an error message.

This test structure allows for testing the enabled state of multiple buttons by parameterizing the test with different button text and XPath values. It demonstrates a basic example of Selenium testing with parameterization and assertions.

- **Web Connection Tests**

The ***test_dtb_conn.py*** script is used to evaluate the functionality of a web application when encountering a database connection fault. More precisely, it verifies whether the web application correctly shows the relevant error message in case of a database connection problem. Now, let's analyze the objective and application of ***test_dtb_conn.py***:

```python
import pytest
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
from selenium.common.exceptions import NoSuchElementException
```

Figure 24: test_dtb_conn.py import

The main test function is named ***test_database_error_message***. It receives the WebDriver instance from the **setup** fixture. The script attempts to find an HTML element indicative of a database error (an

```python
def test_database_error_message(setup):
    driver = setup
    try:
        # Check if the page contains an element indicative of a database error using XPath
        error_element = driver.find_element("xpath", '//div[@class="error-content"]/h1')

        if error_element:
            # If the error element is found, there might be a database error
            error_message = driver.find_element("xpath", '//div[@class="error-content"]/p').text
            print(f"Database Error Message: {error_message}")

            # Add assertions based on the error content
            assert "Database connection error" in error_message
        else:
            # If no error element is found, assert that everything is okay
            assert True  # Add more relevant assertions if needed
    except NoSuchElementException:
        # Handle the case when the error element is not found
        print("No error element found. Possibly no database connection error.")

        # Add assertions or further handling as needed
        assert True
```

Figure 25: test_dtb_conn.py setup

<h1> tag inside a *** *** with a specific class). If the error element is found, the script assumes a database error and extracts the error message. Assertions are used to verify the content of the error message.

If no error element is found, it asserts that everything is okay, and additional assertions or handling can be added. The script catches the ***NoSuchElementException*** if the error element is not found. It prints a message and asserts that everything is okay, allowing for appropriate handling or additional assertions.

In summary, ***test_dtb_conn.py*** is designed to ensure that a web application gracefully handles and communicates database connection errors. It utilizes Selenium for web automation, ***pytest*** for test organization, and includes exception handling to cover scenarios where the expected error element is not present. This script is part of a suite of tests aimed at validating the reliability and error-handling capabilities of a web application.

Kubernetes configurations

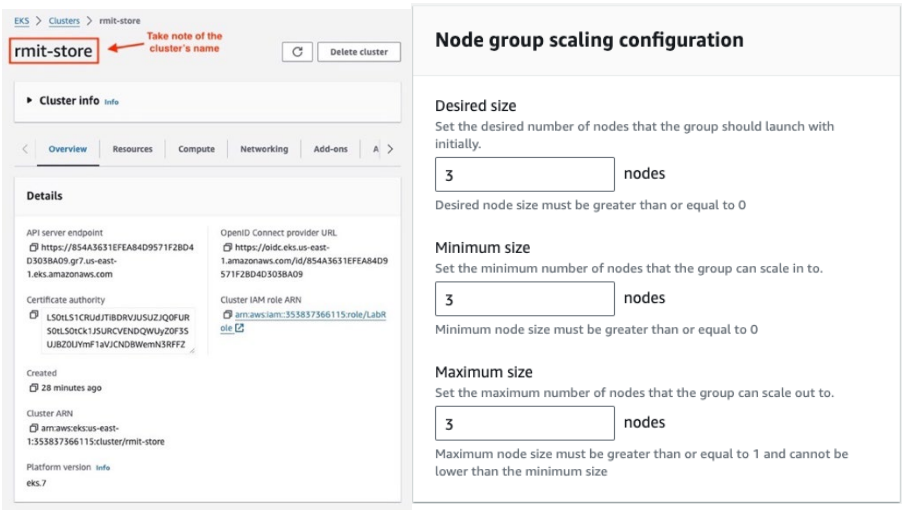
Our team picked Kubernetes as the container orchestration solution for running the production environment of our pipeline. It sits at the final stage of our pipeline: Pushing to production environment.
Our Kubernetes cluster includes:

- 3 worker nodes.
- 3 pods (hosted on the worker nodes) each running the web application container.
- 1 Kubectl server used for interacting with the cluster. (created with Cloud Formation)

Our entire cluster is created and managed using AWS's Elastic Kubernetes System. Additionally, the deployment and services created by Kubectl is saved in YAML files, which we use to initialize the cluster with one command. The following section describes a walkthrough on how we setup the cluster.

Setup resources:

- **Create an EKS cluster:**



Setup a basic EKS cluster with mostly default settings. Take note of the cluster name as we will need it later to connect the Kubectl server to the cluster.

Figure 26: Create EKS Cluster

- **Add Worker Nodes Group**

After creating the EKS cluster, proceeds to attach a Node Group to the cluster. For our demo, we added a 3 nodes Node Group, which will host Kubernetes Pods (similar to Docker containers).

Once the Node Group has been initialized, which should take a few minutes, we need to connect the worker nodes to our RDS database. This is crucial for the worker node containers to connect to the database and retrieve data.

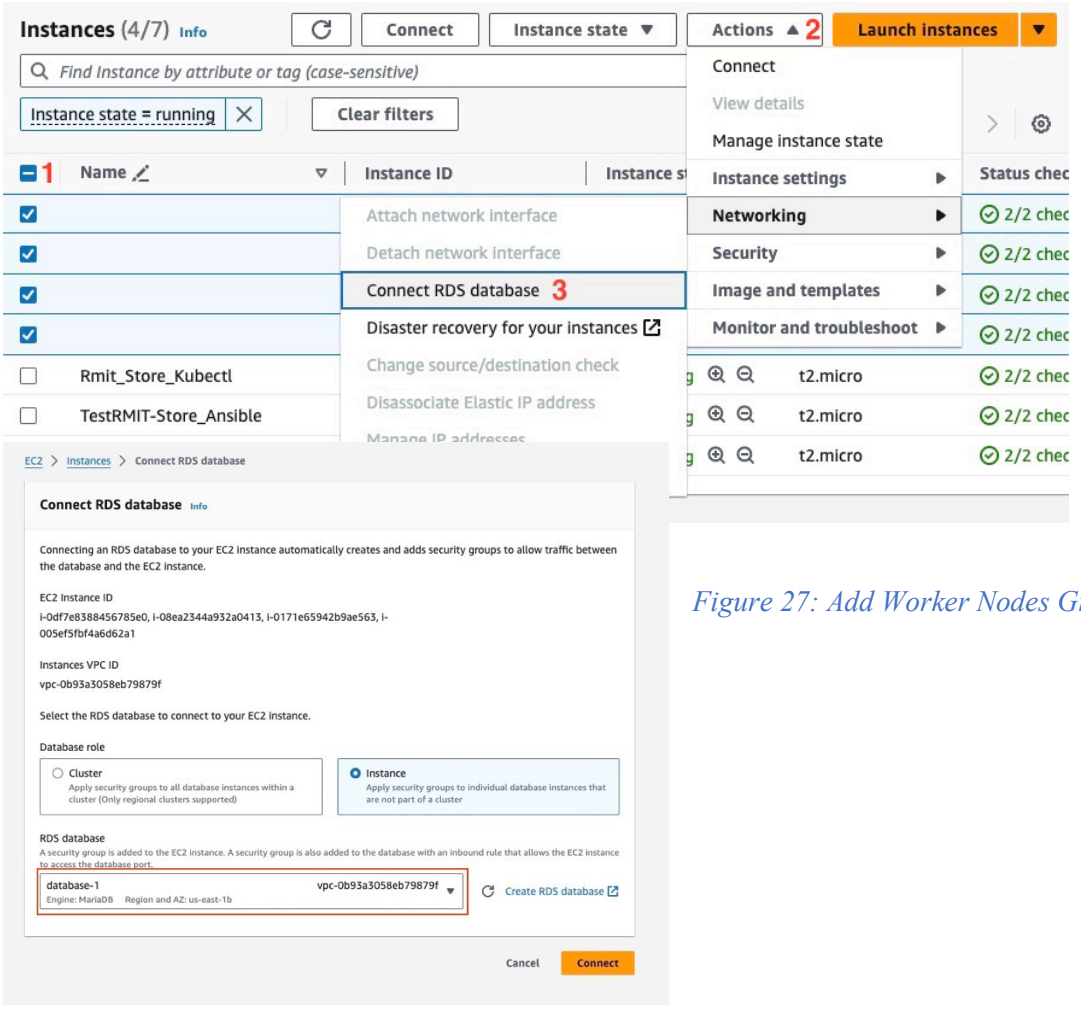


Figure 27: Add Worker Nodes Group

- **Setup Kubectl Server**
  SSH into the Kubectl server instance and run the connect-workers.sh script with the command: ***source connect-workers*** to attach this instance to the cluster's workers. The ***connect-workers.sh*** script file is created during initialization of the instance (using Cloud Formation)

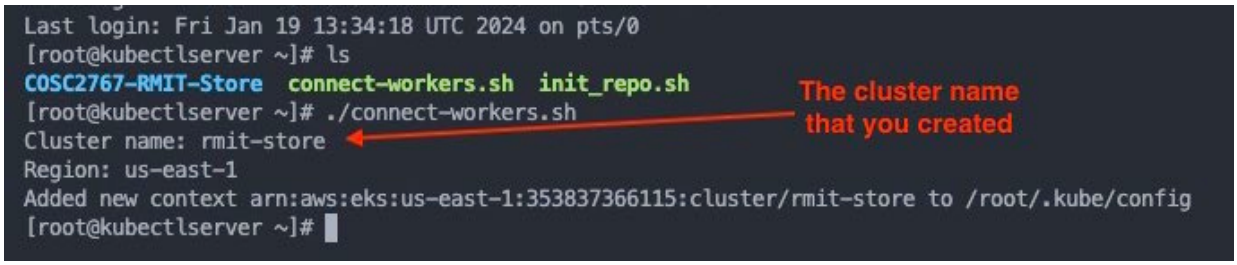


Figure 28: Setup Kubectl Server

The ***connect-workers.sh*** script prompts for EKS cluster name (the one you created on AWS) and the region name of the cluster to connect to the worker EC2 instances. The two variables are fed into command ***aws eks update-kubeconfig -name $clusterName -region $region***, which uses AWS's CLI connect Kubectl Server to the cluster's worker nodes.
Verify that the worker nodes have been connected successfully: ***kubectl get nodes***.

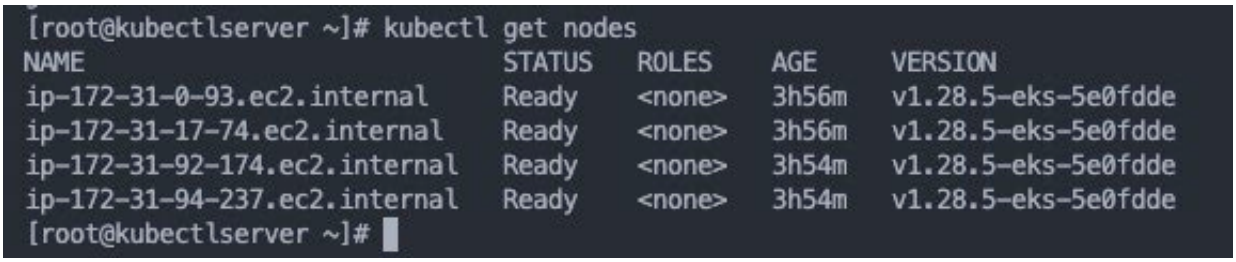


Figure 29: Verify connected Worker Nodes

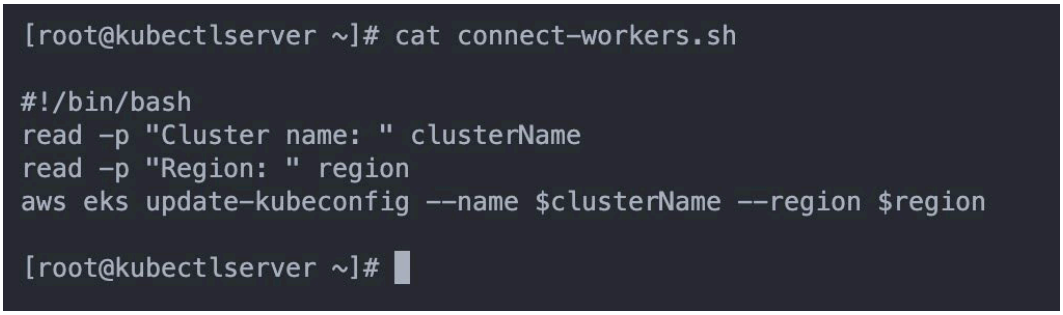


Figure 30: connect-worker.sh

- **Create & manage the cluster through Kubectl**

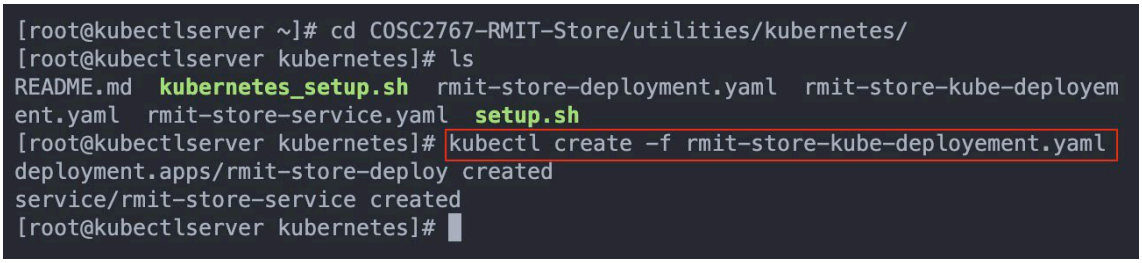


Figure 31: Create and manage cluster

Clone the GitHub repository by running the ***source init_repo.sh*** script and change directory to ***COSC2767-RMIT-Store/utilities/kubernetes***. Start the Kubernetes cluster by the following command: ***kubectl create -f rmit-store-kube-deployment.yml*** The manifest ***rmit-***

store-kube-deployment.yml file describes Kubernetes Objects needed for the cluster, including a Deployment, and a Service.

Notable Service configurations include:

- ➔ **type: LoadBalancer** - An internal load balancing service that distribute network traffic across all running pods. This is essential for any cluster running multiple pods.
- ➔ **port: 80 | targetPort: 80** – The service exposes port 80 externally while also routing that traffic to the internal pod's port 80.

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: rmit-store-service
  labels:
    name: rmit-store-service
    app: demo-rmit-store
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 80
  selector:
    name: rmit-store-pod
    app: demo-rmit-store
```

Figure 32: Service Configuration 1

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rmit-store-deploy
  labels:
    name: rmit-store-deploy
    app: demo-rmit-store
spec:
  replicas: 3
  selector:
    matchLabels:
      name: rmit-store-pod
      app: demo-rmit-store

  template:
    metadata:
      name: rmit-store-pod
      labels:
        name: rmit-store-pod
        app: demo-rmit-store
    spec:
      terminationGracePeriodSeconds: 30
      containers:
        - name: rmit-store
          image: laansdole/s3804811-rmit-store:latest
          imagePullPolicy: "Always"
          ports:
            - containerPort: 80
---
```

Figure 33: Service Configuration 2

The cluster requires approximately 5 minutes to initialize, pull latest image and run containers. After which, we use ***kubectl get all*** to get relevant information on the cluster. Additionally, a Load Balancer DNS is created, which we can use to access the running web application.

Notable Deployment configurations include:

- ➔ **replicas: 3** 3 pods running the Apache web application container.
- ➔ **containerPort: 80** – expose the containers port 80 to the kubernetes network.
- ➔ **imagePullPolicy: "Always"** to force the deployment pull the latest docker image when rolling updates.

Integrate with the Pipeline

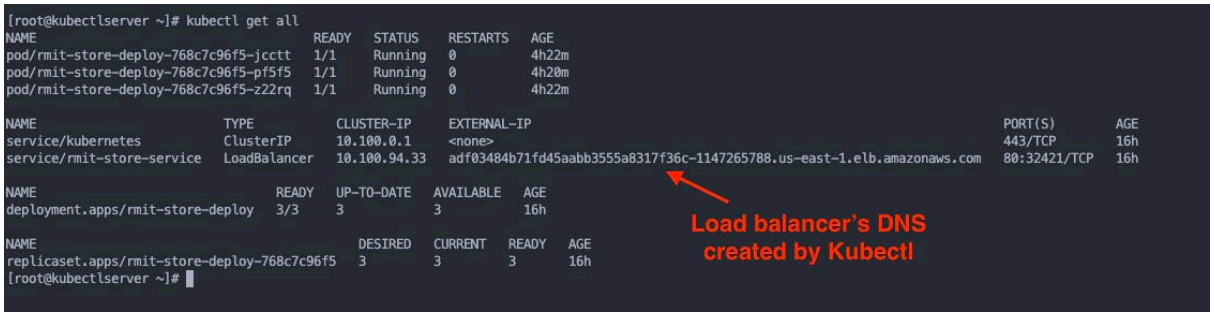


Figure 34: kubectl get all.

So far, we have successfully set up an EKS cluster on AWS with a separate Kubectl instance for managing the cluster. To connect Kubernetes to the pipeline, we need to add the Kubectl server to Ansible server's hosts by copying the public key to the Kubectl server. Afterwards, create a new playbook.

```
- hosts: kubectlserver
  tasks:
    - name: Restart deployment
      command: kubectl rollout restart deployment
      ignore_errors: yes
```

This playbook is configured as a post build step when testing and building new Docker images is successful.

Figure 35: kubectlPlaybook

The restart command will incrementally create new pods with the latest Docker images pull from Dockerhub, start the containers, and replace the old pods with the new pods. The restart command takes around 3 minutes for the new website to be deployed.

CI/CD Pipeline Outcomes:

- **Main Requirement Section**
  1. **Microservice:**
     - **Description:** The Apache server and database (MariaDB) are running on separate servers, increasing reliability and scalability.
     - **Configurations:** Separate instances are created for the Apache server and MariaDB, each with its own resources and configurations.
  2. **Continuous Integration:**
     - **Description:** The Jenkins server triggers the CI/CD pipeline automatically upon every update made to the GitHub repository.
     - **Configurations:** Jenkins is configured to watch the GitHub repository for changes and trigger the pipeline when changes are detected.
  3. **Continuous Delivery:**
     - **Description:** Updates are deployed to a Dev Server first. There is a conditional check for successful build before proceeding.
     - **Configurations:** The pipeline is configured to deploy updates to a testing server first. Upon successful testing, updates are deployed to the production server.
  4. **Cloud Integration:**
     - **Description:** AWS services like Amazon RDS and CloudFormation are being used.
     - **Configurations:** Amazon RDS is used for the database service, and AWS CloudFormation is used for infrastructure setup.

5. **Automation Tool:**
    - **Description:** Jenkins and Ansible servers are incorporated for automation of various stages of development and deployment processes.
    - **Configurations:** Jenkins is used for CI/CD, and Ansible is used for configuration management and application deployment.
6. **Containerization:**
    - **Description:** Docker is used to create Apache images ensuring consistency across multiple environments.
    - **Configurations:** Dockerfile are used to create Apache images, which are then run on the worker nodes.
7. **Testing Framework:**
    - **Description:** Pytest is used for testing in this CI/CD pipeline. It is well-suited for writing and executing unit tests.
    - **Configurations:** Pytest is configured to run a series of tests at different stages of the CI/CD pipeline. These tests can include unit tests, integration tests, system tests, and acceptance tests.

- **Advanced Requirement Section:**
    1. **Configuration Management:**
        - **Description:** Ansible is being used for efficient management of server configurations.
        - **Configurations:** Ansible playbooks are used to manage configurations and automate deployment processes.
    2. **Orchestration:**
        - **Description:** Kubernetes is being used for scaling and managing containerized applications.
        - **Configurations:** Kubernetes is configured to manage the Docker containers running on the worker nodes.
    3. **Advanced Monitoring and Alerting:**
        - **Description:** AWS CloudWatch can be integrated with the CI/CD pipeline to automate the monitoring and management of infrastructure and applications. We will set up CloudWatch alarms to trigger automated actions, such as scaling up or down resources, based on changes in metrics.
        - **Configurations:** CloudWatch is configured to collect and process raw data from Amazon EC2 into readable, near real-time metrics. These metrics can be used to monitor the performance of the CI/CD pipeline and trigger alerts when certain thresholds are crossed.

III. Application Flow (Diagram):

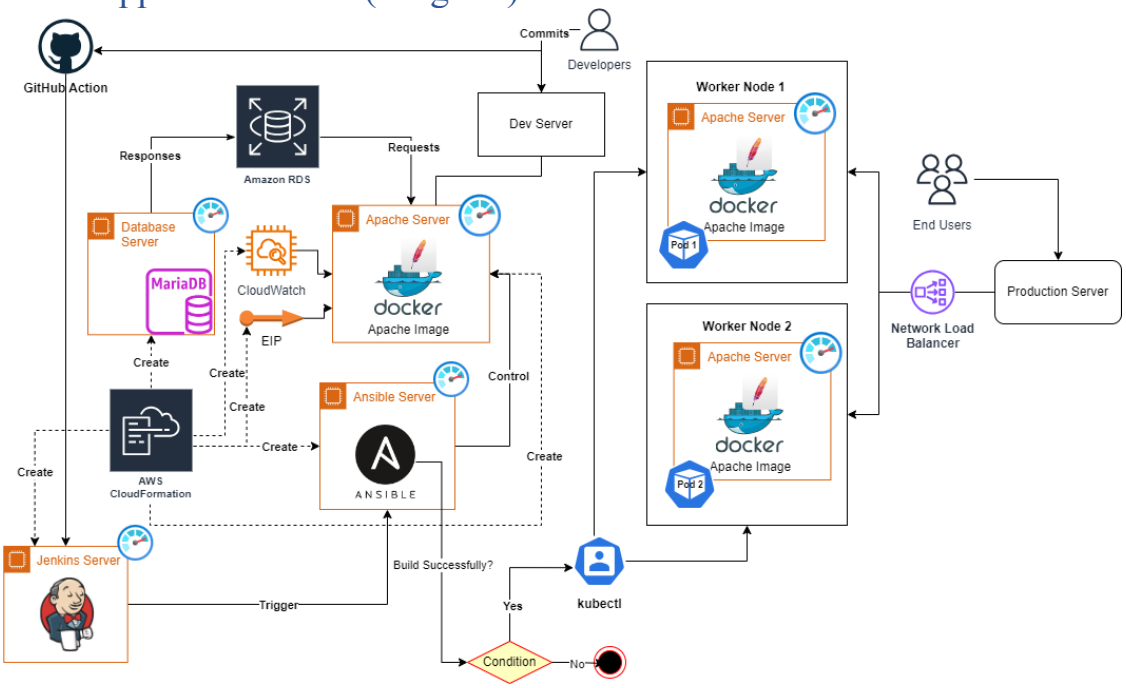


Figure 36: Application flow diagram

1. **AWS CloudFormation**: which created a Kubernetes server (master node), a Jenkins Server, an Ansible Server, a MariaDB database server, and a web server exist. Additionally, CloudFormation also creates Elastic IP (EIP) for each instance and a CloudWatch.
2. **Developers Commit Changes**: Developers commit changes to the codebase on *PHP* file. These changes triggered GitHub Action then triggered the Jenkins Server to initiate building processes in the **development environment**.
3. **Ansible Server and Docker**: In the **development environment**, the Ansible Server controls the creation of Docker Apache images. These images are used to create the web server.
4. **Build Success Check**: Several conditions were set, which are test cases built by **pytest** will check if the build process is successful. If it passed all the test cases, the process continues.
5. **Kubernetes (kubectl)**: If these images are successfully built and passed all the test cases, the created Apache images are then deployed to Worker Node 1 and Worker Node 2 using **kubectl** commands.
6. **Network Load Balancer**: These worker nodes are managed under a Network Load Balancer that distributes incoming network traffic across multiple servers. This ensures no single server bears too much demand.
7. **End Users**: In the **production environment**, end users interact with the application hosted on the Production Server, which receives processed data from the worker nodes.

IV. Self-Evaluation and Reflection of your CI/CD pipeline:

The configuration of our CI/CD pipeline solution allows the RMIT E-Commerce store website to meet the requirements, including automation, efficiency, reliability, scalability, and repeatability, which play a significant role in its success within the lifecycle of the RMIT E-Commerce store application. The below summary to convince how our CI/CD pipeline solution that we believed it met these aspects, including both the main requirement and advanced requirement:

- **Automation**

Thanks to its ability to automate, monitor, and trigger the RMIT store website build and deployment at various stages, Jenkins and GitHub Action together are used to make this entire development and deployment process completely automated without human intervention. Furthermore, with the ability to effectively manage server settings, Ansible allows for quick and easy automation of system installation and maintenance.

- **Efficiency**
  Our CI/CD pipeline optimizes efficiency by automating the development lifecycle for the RMIT ecommerce store using Jenkins and Docker. Every commit and deployment of code changes is completely integrated, reducing human intervention, meaning less failure from a human perspective to bring efficient deployment. Docker containerization solutions ensure consistency in configurations to suit the requirements of each situation, improving the efficiency of the workflow with each separate component server.
- **Reliability**
  The microservices architecture and separate server deployments for different independent components such as Apache, Database, Docker, Ansible, and Kubernetes are necessary to enhance reliability by minimizing dependencies and points of failure that could potentially occur. In the event of an error, editing and maintenance also become convenient, reducing downtime and providing a good user experience. Furthermore, this advanced monitoring solution provides real-time insights, ensuring early detection of problems for improved reliability.
- **Scalability**
  Orchestration technologies such as Kubernetes are also deployed by our team in the CI/CD solution for RMIT store web application to manage and scale containerized systems, which can meet the requirement to handle large workloads without downtime. Using these microservices (Docker) designs and packaging improves scalability by allowing components to operate on separate independent servers.
- **Repeatability**
  When implementing a CI/CD pipeline solution for the RMIT ecommerce store web application, our team not only configured it specifically for this scenario, but also ensured that the settings would remain in use on many other servers. Using Docker helps us ensure consistency in deployment, making the process repeatable across different environments even when they are divided into individual independent servers. Furthermore, Ansible's role in configuration management ensures that server configurations remain consistent, contributing to the integrity and repeatability of the entire system.

Overall, our continuous integration and delivery (CI/CD) process for the RMIT E-Commerce store project and DevOps in general not only meets the basic conditions through establishing important automated processes that improve operational efficiency but also gain efficiency in handling more complex needs by ensuring reliability, scalability, and repeatability. By combining modern tools and methods in the popular DevOps field today, our process dares to be confident and affirm that this is one of the solutions that brings flexibility and readiness to the challenges, bugs, and issues related to the end user experience throughout the software development and deployment lifecycle.

V. Known Bugs/Problems:

This section investigates the most typical problems that may arise during our project. Some of the possibilities discussed include failed builds caused by software issues, mistakes during test automation, and network connectivity issues. Each of these groups reflects a distinct set of obstacles that could interfere with the seamless operation of our pipeline. Understanding these potential pitfalls allows us to better predict, avoid and solve problems thereby enhancing the dependability and effectiveness of our project:

1. **Jenkins Failed builds:** For example, we could have introduced a syntax error in the bash script userdata, causing the build to fail. A further prevalent issue is problems with dependencies. In other words, if a necessary library or package that we had is not properly configured or installed, it could fail the build process.
2. **Errors in automated testing:** Tests failed due to issues with the testing environment, including a wrong setup or a shortage of assets. Furthermore, there are challenges with the tests themselves. For example, the tests themselves are bugged, or they failed in properly testing the application's actions.
3. **Ansible Deployment Failures:** When the application failed to launch properly in the production environment, the deployment failed. There are additional issues with the production environment itself, such as incorrect configuration for different hosts. Deployment failures were also caused by errors in the deployment scripts.
4. **Jenkins Configuration errors:** This often occurs when we restart the AWS Learner Lab and the IPv4 address changes. To fix this problem, AWS Elastic IP service is integrated.
5. **Virtual Environment Resource issues:** Whether any of the servers associated with the pipeline were lacking CPU, memory or disk space, this prompted failures. Suppose that our Jenkins server did not have sufficient memory and collapsed to complete buildings. Likewise, when the production server lacks enough storage capacity, the implementation will collapse completely.
6. **Network issues:** When our Jenkins server cannot connect to the code repository due to a network connection problem, it is unable to trigger builds. Moreover, the production server may be unable to access the required resources due to a network issue, resulting in an error during the deployment process.
7. **SMTP Gmail Authentication Error [13]:** This problem arose when our CI/CD pipeline required sending emails via Gmail's SMTP server. Whether the authentication details provided (such as the username and password) are incorrect, or the application attempting to send the email is not authorized to use the Gmail account, an SMTP Gmail Authentication Error occurs frequently. This may disrupt any pipeline processes that rely on email notifications.
8. **Limited Access in AWS Learner Lab [14]:** We assumed that the AWS Learner Lab was a great resource for acquiring knowledge and testing with services provided by AWS in a reliable, sandboxed environment. However, it caused us some issues because it fails to provide unlimited access to all AWS services. When we work with the pipeline, we do not have full utilization of Kubernetes, such as an administrator user or SSH access. When our pipeline requires more advanced Kubernetes operations, we may need to upgrade to a full AWS account or find other ways to complete our tasks.

VI. Project Responsibilities:

Team member	Responsibilities	Tasks	Workload
Do Le Long An	Team Leader, Jenkins Integration, GitHub Action	Overseeing the project, ensuring all tasks are assigned and completed. Managing version control. Combine and integrate the final CI/CD Pipeline	**20%**
Hoang Ngoc Duan	Technical Writing	Collecting screenshots, dividing tasks for document writing, compiling the final report	**20%**
Pham Quang Man	EKS Engineer	Managing container orchestration, ensuring scalability and reliability of the website	**20%**
Huynh Van Anh	Testing Framework	Creating and executing test cases, ensuring the availability of the website	**20%**
La Tran Hai Dang	Ansible and Jenkins Engineer	Configuring Jenkins for Dev Environment	**20%**

VII. Conclusion

Completing this Assignment 3 - Group Project has given our entire team the opportunity to learn advanced and significant knowledge about practical DevOps technology. The introduction of microservices architecture has brought proof of the significance of separating servers to enhance the stability, reliability, and scalability of the system. There has been an implementation of the concepts of continuous integration (CI), which means that the pipeline is started automatically whenever the server detects any recent updates or commits revisions on GitHub. The continuous delivery (CD) process imparted greater flexibility to release management by facilitating staged deployments to both testing and production servers.

In the scenarios of the RMIT store, deploying Amazon Web Services (AWS) is essential for cloud integration to improve speed, scalability, and infrastructure suitability. Besides, Jenkins is used as a tool to support the automation of many CI/CD process components, reducing human intervention, and maintaining consistency across servers. In the context that Assignment 3 introduced, the use of Docker and containerization has optimized processes and simplified deployment methods. Furthermore, to optimize server configuration management, Ansible was used for scalability and enhanced system consistency. By using orchestration technologies like Kubernetes, containerized systems can be scaled and maintained.

Overall, the hands-on experience gained from implementing Task 3 contributed to the team's understanding of how to build a comprehensive CI/CD pipeline using AWS, Docker, Jenkins, Ansible, and Kubernetes. In addition to imparting practical DevOps expertise, our team also develops soft skills

including critical thinking, adaptability, and automation solutions. The obstacles encountered during this project required the whole team to research and produce appropriate solutions, improving problem-solving ability. Each member's capacity was shown through the ability to adapt to many different situations and mistakes, which created continuous progress inherent in the field of DevOps in the group in general and each team member.

VIII. References:

[1] GitLab. “What is Git version control?” about.gitlab.com. Accessed Jan. 18, 2024 [Online]. Available: https://about.gitlab.com/topics/version-control/what-is-git-version-control/.

[2] GitHub Docs. “GitHub Actions documentation” docs.github.com. Accessed Jan. 20, 2024 [Online]. Available: https://docs.github.com/en/actions/.

[3] Jenkins. “Jenkins. Build great things at any scale” jenkins.io. Accessed Jan. 18, 2024 [Online]. Available: https://www.jenkins.io/.

[4] AWS Docs. “What is AWS CloudFormation?” docs.aws.amazon.com. Accessed Jan. 20, 2024 [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html/.

[5] Red Hat Ansible Automation Platform. “OVERVIEW: How Ansible Works” ansible.com. Accessed Jan. 18, 2024 [Online]. Available: https://www.ansible.com/overview/how-ansible-works/.

[6] Docker Docs. “Docker overview” docs.docker.com. Accessed Jan. 18, 2024 [Online]. Available: https://docs.docker.com/get-started/overview/.

[7] Apache HTTP Server Project. “The Number One HTTP Server On the Internet” httpd.apache.org. Accessed Jan. 18, 2024 [Online]. Available: https://httpd.apache.org/.

[8] AWS Doc. “What is Amazon Relational Database Service (Amazon RDS)?” docs.aws.amazon.com. Accessed Jan. 18, 2024 [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html.

[9] PHP. “Documentation” php.net. Accessed Jan. 18, 2024 [Online]. Available: https://www.php.net/docs.php/.

[10] AWS Docs. “What is a Network Load Balancer?” docs.aws.amazon. Accessed Jan. 21, 2024 [Online]. Available: https://docs.aws.amazon.com/elasticloadbalancing/latest/network/introduction.html/.

[11] Kubernetes. “What is Kubernetes?” kubernestes.io. Accessed Jan. 19, 2024 [Online]. Available: https://kubernetes.io/vi/docs/concepts/overview/what-is-kubernetes/.

[12] AWS Amazon Inc. “Amazon CloudWatch” aws.amazon.com. Accessed Jan. 18, 2024 [Online]. Available: https://aws.amazon.com/cloudwatch/?nc1=h_ls/.

[13] S. Alan, Login Authentication failed with Gmail SMTP (Updated), (Aug. 17, 2022). Accessed Jan. 18, 2024. [Blog]. Available: https://stackoverflow.com/questions/73383458/login-authentication-failed-with-gmail-smtp-updated.

[14] AWS Docs. “Granting access to an IAM principal to view Kubernetes resources on a cluster” docs.aws.amazon.com. Accessed Jan. 19, 2024 [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/connector-grant-access.html/.

IX. Appendices: