Assignment 2

[COSC2767] Systems Deployment and Operations

Team Pall

Do Tung Lam
S3963286

Nguyen Quoc An
S3938278

Truong Quang Bao Loc
S3965528

Hoang Thai Phuc
S3978081

Submission Date: January 17th, 2025
Lecturer: Dr. Tom Huynh

Contents

I. CI/CD Pipeline Solution

1. Problem statement

Dr. Tom, a web developer managing the RMIT E-commerce store, struggles to efficiently deploy updates due to his limited DevOps experience and the absence of a CI/CD pipeline.

2. Proposed Solution

To address Tom's challenges, we propose CI/CD as the core solution to automate deployments, reduce errors, and enhance reliability. Thanks to continuous integration, the code will be integrated smoothly, while continuous delivery will provide safe, staged releases with rollback capabilities. Stability would be improved because of automated testing and monitoring, gaining real-time system visibility. Structured workflow improves collaboration, scalability, and efficiency, hence making deployments seamless and predictable.

3. Tools and technologies

- **GitHub:** A platform that helps us to manage and control the version of the RMIT store [1]. This platform will be the starting point of the CI/CD, when there is any commit to change the source code, it will trigger the pipeline.
- **Jenkins:** This is an open-source automation server that allows us to implement all types of automation tasks from building, testing to deploying our products [2]. We will use this platform for complex workflows like automation testing.
- **GitHub Action:** This is a CI tool offered by GitHub and it is only used within GitHub [3]. This tool will automatically execute our workflow and also simplify some processes for Jenkins like trigger on pull request with file pattern.
- **AWS CloudFormation**: This is a service that helps us model and set up AWS resources and provides more time for applications operating in AWS rather than dealing with resource management [4]. It is used to bring up the test environment that simulates the production environment then we can shut down and eliminate that environment after testing for cost efficiency.
- **Docker:** Docker is a software platform that enables one to build, test, and deploy applications quickly. Docker packages software into standardized units called containers; these contain everything the software needs to run, such as libraries, system tools, code, and even runtime [5]. With Docker, we can ensure that RMIT store can run stably across different environments and simplify the deployment process when we just need to install docker on different environments.
- **MongoDB Atlas:** MongoDB Atlas is a fully managed cloud database that will handle all the complexities of deploying, managing, and healing your deployments on the cloud service provider of your choice: AWS, Azure, and GCP [6]. This is our main choice to deploy our database.
- **Nginx:** This is an HTTP web server, reverse proxy, content cache, load balancer, TCP/UDP proxy server, and mail proxy server [7]. In our CI/CD, Nginx acts as a reverse proxy, it provides zero-downtime deployment and ensures secured access of applications.
- **Elastic Load Balancer:** Elastic Load Balancing (ELB) automatically distributes incoming application traffic across multiple backend servers or front-end servers, Load Balancer enables the scalability, availability and reliability of the website [8].
- **Kubernetes:** Kubernetes is an open-source, portable, extensible platform for managing containerized workloads and services which facilitates both declarative configuration and automation [9]. Kubernetes allows our CI/CD to deploy automated scaling, rollbacks, load balancing, service discovery, configuration management, monitoring, and zero-downtime updates and we can also apply canary deployment with K8s.
- **Prometheus:** An open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach [10]. In our CI/CD pipeline this tool will help us to monitor the website.
- **Grafana:** This is the open-source analytics & monitoring solution for every database [11]. In our CI/CD, we use Grafana to integrate with Prometheus for visualizing time-series metrics that allow engineers to detect anomalies, optimize performance, and troubleshoot issues efficiently.

II. CI/CD Pipeline Solution

4. Problem statement

Dr. Tom, a web developer managing the RMIT E-commerce store, struggles to efficiently deploy updates due to his limited DevOps experience and the absence of a CI/CD pipeline.

5. Proposed Solution

To address Tom's challenges, we propose CI/CD as the core solution to automate deployments, reduce errors, and enhance reliability. Thanks to continuous integration, the code will be integrated smoothly, while continuous delivery will provide safe, staged releases with rollback capabilities. Stability would be improved because of automated testing and monitoring, gaining real-time system visibility. Structured workflow improves collaboration, scalability, and efficiency, hence making deployments seamless and predictable.

6. Tools and technologies

- **GitHub:** A platform that helps us to manage and control the version of the RMIT store [1]. This platform will be the starting point of the CI/CD, when there is any commit to change the source code, it will trigger the pipeline.
- **Jenkins:** This is an open-source automation server that allows us to implement all types of automation tasks from building, testing to deploying our products [2]. We will use this platform for complex workflows like automation testing.
- **GitHub Action:** This is a CI tool offered by GitHub and it is only used within GitHub [3]. This tool will automatically execute our workflow and also simplify some processes for Jenkins like trigger on pull request with file pattern.
- **AWS CloudFormation**: This is a service that helps us model and set up AWS resources and provides more time for applications operating in AWS rather than dealing with resource management [4]. It is used to bring up the test environment that simulates the production environment then we can shut down and eliminate that environment after testing for cost efficiency.
- **Docker:** Docker is a software platform that enables one to build, test, and deploy applications quickly. Docker packages software into standardized units called containers; these contain everything the software needs to run, such as libraries, system tools, code, and even runtime [5]. With Docker, we can ensure that RMIT store can run stably across different environments and simplify the deployment process when we just need to install docker on different environments.
- **MongoDB Atlas:** MongoDB Atlas is a fully managed cloud database that will handle all the complexities of deploying, managing, and healing your deployments on the cloud service provider of your choice: AWS, Azure, and GCP [6]. This is our main choice to deploy our database.
- **Nginx:** This is an HTTP web server, reverse proxy, content cache, load balancer, TCP/UDP proxy server, and mail proxy server [7]. In our CI/CD, Nginx acts as a reverse proxy, it provides zero-downtime deployment and ensures secured access of applications.
- **Elastic Load Balancer:** Elastic Load Balancing (ELB) automatically distributes incoming application traffic across multiple backend servers or front-end servers, Load Balancer enables the scalability, availability and reliability of the website [8].
- **Kubernetes:** Kubernetes is an open-source, portable, extensible platform for managing containerized workloads and services which facilitates both declarative configuration and automation [9]. Kubernetes allows our CI/CD to deploy automated scaling, rollbacks, load balancing, service discovery, configuration management, monitoring, and zero-downtime updates and we can also apply canary deployment with K8s.
- **Prometheus:** An open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach [10]. In our CI/CD pipeline this tool will help us to monitor the website.
- **Grafana:** This is the open-source analytics & monitoring solution for every database [11]. In our CI/CD, we use Grafana to integrate with Prometheus for visualizing time-series metrics that allow engineers to detect anomalies, optimize performance, and troubleshoot issues efficiently.

7. Pipeline features

- **Automated Code Integration**: Whenever a new pull request is created in GitHub, a build and test process is automatically triggered. It promotes continuous merging and validation, thereby simplifying change management and catching errors early in the development lifecycle. By integrating code in smaller batches, teams maintain a synchronized repository and reduce the likelihood of complex merge conflicts.
- **Automated Code Build and Testing**: Jenkins manages complex workflows that encompass unit tests, integration tests, and other quality checks. By automating these processes, Jenkins removes much of the human error that manual testing can introduce. This

consistent approach to validating every code change maintains the project's stability and frees developers to focus on building new features rather than troubleshooting deployment issues.

- **Automated Test Environment Management**: Using AWS CloudFormation, the pipeline dynamically provisions and tears down AWS resources for testing, effectively mirroring the production environment. This ensures consistency in infrastructure setup and eliminates the risk of configuration drift between environments. Moreover, it is cost-efficient by automatically spinning down test environments when they are no longer needed.
- **Consistent Environment and Streamlined Deployments**: Docker packages the application and all its dependencies into lightweight containers, allowing the software to run uniformly across different environments. By encapsulating everything the application needs, Docker makes it easier to ensure predictable performance and reliability in development, testing, and production stages.
- **Traffic Scalability**: Elastic Load Balancing automatically distributes traffic across multiple server instances or containers, preventing any single resource from becoming a bottleneck. As the site's traffic grows or changes, it scales resources accordingly to maintain optimal performance. This redundancy ensures the store remains available and responsive under varying loads.
- **Automated Deployment Orchestration and Management**: Kubernetes streamlines container deployments self-healing capabilities restart failed containers or replace them on different hosts if necessary. By supporting canary deployment strategy, Kubernetes reduces the risk of introducing errors into production and facilitates agile release cycles. Thanks to container orchestration and robust version control, the pipeline can quickly revert to a known stable version if a release introduces critical issues. This rollback process greatly reduces downtime and user disruption. A clear record of versions and their respective configurations ensures the team can isolate and correct problems with minimal impact on the live environment.
- **Real-time Monitoring and Alerting**: Prometheus collects real-time metrics and stores them in a time-series database, while Grafana provides dynamic dashboards that visualize these metrics. This integration enables proactive monitoring, giving the team immediate insight into any anomalies or performance overload. In addition. Grafana alerting rules which send alert notifications allow operators to respond quickly to potential issues before they escalate into outages.

III. Pipeline Component

For each completed main requirement, please provide description, configurations and screenshots as proofs.

Explanation of the chosen settings and configurations if needed.

Screenshots are important to show the pipeline can pass required use cases.

Secret and Credentials Configuration

To allow the CI/CD pipeline to work securely, sensitive credentials must be properly configured and use within the workflow.

GitHub Repository Secrets

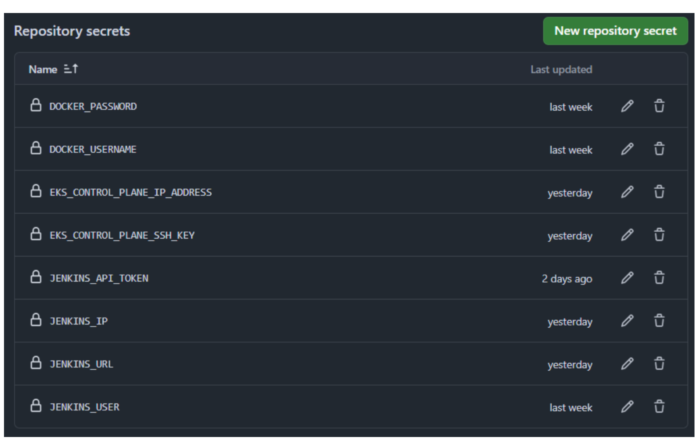


Figure 1 Repository Secrets

DOCKER_USERNAME & **DOCKER_PASSWORD**: These 2 two secrets store the credentials to login Docker registry.

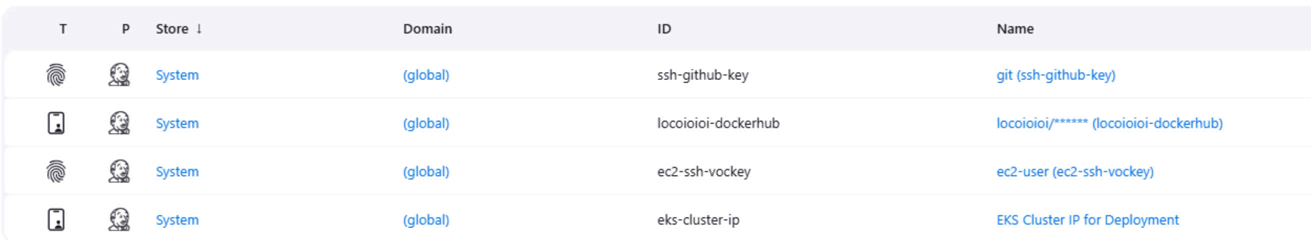
JENKINS_IP: store the IP of Jenkins server help securely trigger the Jenkins job.

JENKINS_API_TOKEN: This secret will store the API token for credentials when call the API remotely.

JENKINS_URL: This secret stores the URL that the pipeline can trigger the Jenkins job.

JENKINS_USER: Store the username for authenticating Jenkins.

Credentials



T	P	Store ↓	Domain	ID	Name
		System	(global)	ssh-github-key	git (ssh-github-key)
		System	(global)	locoioioi-dockerhub	locoioioi/****** (locoioioi-dockerhub)
		System	(global)	ec2-ssh-vockey	ec2-user (ec2-ssh-vockey)
		System	(global)	eks-cluster-ip	EKS Cluster IP for Deployment

Figure 2 Jenkins credentials

ssh-github-key: A ssh key for authenticating with GitHub repositories.

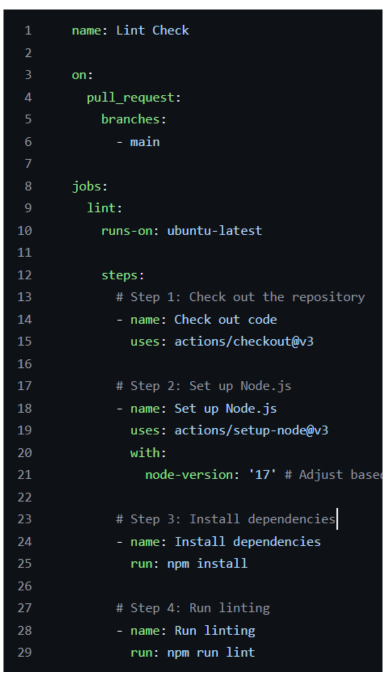
locoioioi-dockerhub: Credentials for Docker Hub associated with the account \`locoioioi\`.

ec2-ssh-vockey: SSH private key to connect with Ec2 instance.

eks-cluster-ip: Stores the IP address of the EKS control plan server for deployment purposes.

General GitHub Action Workflow

a) Workflow for linting check with GitHub Action



```yaml
name: Lint Check

on:
  pull_request:
    branches:
      - main

jobs:
  lint:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Check out the repository
      - name: Check out code
        uses: actions/checkout@v3

      # Step 2: Set up Node.js
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '17' # Adjust base

      # Step 3: Install dependencies
      - name: Install dependencies
        run: npm install

      # Step 4: Run linting
      - name: Run linting
        run: npm run lint
```

Figure 3. GitHub Lint Check Job]

Eslint

Purpose: Perform eslint check for NodeJS and ReactJS code.

Name: Lint Check

Trigger On: This job is triggered on a pull request event to the main branch.

Jobs: This job will be run on the ubuntu-latest image.

Steps:

- Check out the code using the actions/checkout action.
- Set up Node.js version 17 using the actions/setup-node action.
- Install dependencies with the npm install command.
- Run linting using the npm run lint command.

```yaml
name: reviewdog
on: [pull_request]
jobs:
  eslint:
    name: runner / eslint
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '17' # Update this to match your project

      - name: Install dependencies
        run: npm install

      - name: Run ESLint
        run: npx eslint . -f json -o eslint-report.json || true

      - name: Run reviewdog
        uses: reviewdog/action-eslint@v1
        with:
          reporter: github-pr-review
          level: error
          eslint_input: eslint-report.json
          fail_on_error: false
          filter_mode: nofilter
```

Figure 5 ReviewDog Lint

```yaml
  client-tests:
    name: Client Unit Tests
    runs-on: ubuntu-latest

    steps:
      # Step 1: Check out the code from the repository
      - name: Checkout repository
        uses: actions/checkout@v3

      # Step 2: Set up Node.js
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 20

      # Step 3: Install client dependencies
      - name: Install client dependencies
        run: |
          cd client
          npm install
          npm install --save-dev jest supertest

      # Step 4: Run client tests if "test" script exists
      - name: Run client tests
        run: |
          cd client
          if npm run | grep -q "test"; then
            echo "Running client tests..."
            npm run test
          else
            echo "No test script found. Skipping client tests."
          fi
        env:
          NODE_ENV: test
```

Figure 4 ReviewDog Lint

ReviewDog Lint

Name: Reviewdog

Purpose: Display the eslint result with ReviewDog Bot in comment and GitHub checks tab.

Trigger On: This job is triggered on a pull request event.

Jobs: This job will be run on the ubuntu-latest image.

- Check out the code using the actions/checkout action.
- Set up Node.js version 17 using the actions/setup-node action.
- Install dependencies with the npm install command.
- Run ESLint to generate a JSON report (eslint-report.json) using the npx eslint command.
- Use the reviewdog/action-eslint action to annotate ESLint errors on the pull request with the github-pr-review reporter, error level set to error, and nofilter mode enabled.

```yaml
name: Unit and Integration Test CI

on:
  pull_request:
    branches:
      - main
  workflow_dispatch:

jobs:
  server-tests:
    name: Server Unit Tests
    runs-on: ubuntu-latest

    steps:
      # Step 1: Check out the code from the repository
      - name: Checkout repository
        uses: actions/checkout@v3

      # Step 2: Set up Node.js
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 20

      # Step 3: Install server dependencies
      - name: Install server dependencies
        run: |
          cd server
          npm install
          npm install --save-dev jest supertest

      # Step 4: Run server tests if "test" script exists
      - name: Run server tests
        run: |
          cd server
          if npm run | grep -q "test"; then
            echo "Running server tests..."
            npm run test
          else
            echo "No test script found. Skipping server tests."
          fi
        env:
          NODE_ENV: test
```

Figure 7 Unit and Integration Test CI


```yaml
name: Unit and Integration Test CI

on:
  pull_request:
    branches:
      - main
  workflow_dispatch:

jobs:
  server-tests:
    name: Server Unit Tests
    runs-on: ubuntu-latest

    steps:
      # Step 1: Check out the code from the repository
      - name: Checkout repository
        uses: actions/checkout@v3

      # Step 2: Set up Node.js
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 20

      # Step 3: Install server dependencies
      - name: Install server dependencies
        run: |
          cd server
          npm install
          npm install --save-dev jest supertest

      # Step 4: Run server tests if "test" script exists
      - name: Run server tests
        run: |
          cd server
          if npm run | grep -q "test"; then
            echo "Running server tests..."
            npm run test
          else
            echo "No test script found. Skipping server tests."
          fi
        env:
          NODE_ENV: test
```


Figure 6 Unit and Integration Test CI

b) Workflow for Unit Test and Integration Test

Name: Unit and Integration Test CI

Trigger On: This job is triggered on a pull request event to the main branch and can also be triggered manually using workflow dispatch.

1. **Server Unit Tests:**

- Runs on the ubuntu-latest image.
- Steps:
  - Check out the repository using the actions/checkout action.
  - Set up Node.js version 20 using the actions/setup-node action.
  - Install server dependencies, including Jest and Supertest, using npm install commands in the server directory.
  - Run server tests if a test script exists in the server directory. The NODE_ENV is set to test.

2. **Client Unit Tests:**

- Runs on the ubuntu-latest image.
- Steps:
    - Check out the repository using the actions/checkout action.
    - Set up Node.js version 20 using the actions/setup-node action.
    - Install client dependencies, including Jest and Supertest, using npm install commands in the client directory.
    - Run client tests if a test script exists in the client directory. The NODE_ENV is set to test.

Continuous Integration

Frontend pipeline

```yaml
jobs:
  build-push-trigger:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Checkout the code from the repository
      - name: Checkout code
        uses: actions/checkout@v3

      # Step 2: Log in to Docker Hub
      - name: Log in to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      # Step 3: Build and Push Docker E2E Testing Image
      - name: Build Docker E2E Testing image
        run: |
          docker build -t ${{ secrets.DOCKER_USERNAME }}/e2e:latest -f ./client/Dockerfile.e2e ./client

      - name: Push Docker E2E Testing image
        run: |
          docker push ${{ secrets.DOCKER_USERNAME }}/e2e:latest
```

```yaml
      # Step 4: Trigger Jenkins Job (Using Secret for Jenkins IP)
      - name: Trigger Jenkins Job
        run: |
          curl -X POST "http://${{ secrets.JENKINS_IP }}:8080/job/fe-ci/buildWithParameters?branch=${{ github.head_ref }}" \
            --user ${{ secrets.JENKINS_USER }}:${{ secrets.JENKINS_API_TOKEN }} \
            --header "Content-Type: application/json"

      - name: Install jq
        run: sudo apt-get install jq

      # Step 5: Wait for Jenkins Job to Complete (Using Secret for Jenkins IP)
      - name: Wait for Jenkins Job
        id: wait_jenkins
        run: |
          JENKINS_URL="http://${{ secrets.JENKINS_IP }}:8080"
          JOB_NAME="fe-ci"
          USER="${{ secrets.JENKINS_USER }}"
          API_TOKEN="${{ secrets.JENKINS_API_TOKEN }}"
          BUILD_URL="$JENKINS_URL/job/$JOB_NAME/lastBuild/api/json"
          STATUS="IN_PROGRESS"

          while [ "$STATUS" != "SUCCESS" ] && [ "$STATUS" != "FAILURE" ]; do
            echo "Checking Jenkins job status..."
            sleep 10
            STATUS=$(curl -s --user $USER:$API_TOKEN $BUILD_URL | jq -r '.result')
            echo "Current status: $STATUS"
          done

          if [ "$STATUS" == "SUCCESS" ]; then
            echo "Jenkins job completed successfully!"
          else
            echo "Jenkins job failed. Exiting with error."
            exit 1
          fi
```

Figure 8 GitHub Action - Frontend CI for Pull Request

b) **Developing a workflow for Pull Request on Frontend with GitHub Action**

Name: Jenkins CI PR FE

Trigger On: This job is trigger on pull request event to main when the changes are pushed to folder client.

Jobs: This job will be run on the ubuntu-latest image.

Steps:

1. Login to docker hub with the DOCKERHUB_USERNAME and DOCKERHUB_PASSWORD secrets.
2. Build the e2e test image and then push it to docker hub with the tag "latest".
3. Trigger Jenkins Job named "fe-ci" with the repository secrets JENKINS IP to ensure the security of the workflow.
4. Wait for Jenkins job to finish and update the GitHub action job base on that status.

c) **Configure Jenkins job "fe-ci" that is trigger by the Jenkins PR CI**

- Ssh to the Jenkins server on AWS then login as root user.
- Run the command "service jenkins start" to start Jenkins.
- Access the Jenkins via port 8080 with the public ipv4 of the ec2 instance.
- Create a new Pipeline item with the name of "fe-ci"

Figure 10 Set up SCM for the Jenkins job

Provide the ssh URL of the repository on the Repository URL field.

Select appropriate credentials to allow access to private repository.
Specify the correct path to the Jenkinsfile in the Script Path field.

```
pipeline {      You, last week • Loc ci (#8) …
    agent any

    environment {
        STACK_NAME = "test-env-mern-deployment-stack"    "mern": Unknown word.
        INSTANCE_IP = ""
    }

    parameters {
        string(defaultValue: 'main', description: 'Branch to build', name: 'branch')
    }

    stages {
        stage('Checkout Code') {
            steps {
                script {
                    checkout([
                        $class: 'GitSCM',
                        branches: [[name: "refs/heads/${params.branch}"]],
                        userRemoteConfigs: [[
                            credentialsId: 'ssh-github-key',
                            url: 'git@github.com:RMIT-Vietnam-Teaching/cosc2767-assignment-2-group-2024c-pall.git'    "cosc": Unk
                        ]]
                    ])
                }
            }
        }
```

Figure 11 Jenkinsfile - checkout code

- This is the Jenkinsfile that the pipeline will execute, first it will get a parameter call branch like what we configured on the pipeline.
- The first step of this pipeline is to checkout to the branch from the parameter with the appropriate credentials.

Figure 12 Jenkinsfile - checkout code

```
        stage('Trigger CloudFormation') {
            steps {
                script {
                    sh """
                    aws cloudformation deploy \
                        --stack-name ${STACK_NAME} \
                        --template-file cloudformation/create-test-server-template.yaml \
                        --capabilities CAPABILITY_NAMED_IAM
                    aws cloudformation wait stack-create-complete --stack-name ${STACK_NAME}
                    """
                }
            }
        }

        stage('Retrieve Instance IP') {
            steps {
                script {
                    INSTANCE_IP = sh(
                        script: """
                        aws cloudformation describe-stacks \
                            --stack-name ${STACK_NAME} \
                            --query "Stacks[0].Outputs[?OutputKey=='PublicIP'].OutputValue" \
                            --output text
                        """,
                        returnStdout: true
                    ).trim()
                    echo "EC2 Instance IP: ${INSTANCE_IP}"
                }
            }
        }
```

Figure 13 Jenkinsfile - Run AWS CloudFormation template

- In the next stage, the pipeline will run the AWS CloudFormation template to create a new ec2 instance for testing purposes. CloudFormation template will be described detailed in the advanced section below.
- After successfully launching the new ec2 instance, it will retrieve the INSTANCE_IP and store it within the environment variable.

Figure 14 Jenkinsfile - config the testing environment

Figure 15 Run AWS CloudFormation template

```
        stage('Install Docker & Dependencies') {
            steps {
                withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {    "vockey": Unknown word.
                    sh """
                    ssh -i \$SSH_KEY -o StrictHostKeyChecking=no ec2-user@${INSTANCE_IP} \
                    "sudo yum install -y docker git nmap-ncat && \    "nmap": Unknown word.
                     sudo systemctl start docker && \
                     sudo systemctl enable docker && \
                     sudo usermod -aG docker ec2-user && \    "usermod": Unknown word.
                     docker --version && \
                     sudo curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-compose-\$(uname -s)-\$(uname -m) -o /usr/local/bin/docker-compose && \
                     sudo chmod +x /usr/local/bin/docker-compose && \
                     docker-compose version"
                    """
                }
            }
        }

        stage('Copy Source Code to EC2') {
            steps {
                withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {    "vockey": Unknown word.
                    retry(3) {
                        sh """
                        scp -i \$SSH_KEY -o StrictHostKeyChecking=no -r . ec2-user@${INSTANCE_IP}:/home/ec2-user/project
                        """
                    }
                }
            }
        }
```

- In the test environment, first the pipeline will **install all necessary tools** to run testing like docker, and docker-compose
- After that, in the next stage, the pipeline will forward all the source code to the test environment.

Figure 16 Install dependency

```
    stage('Deploy Application Using Docker Compose') {
        steps {
            withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {    "vockey": Unknown word.
                sh """
                ssh -i \$SSH_KEY -o StrictHostKeyChecking=no ec2-user@${INSTANCE_IP} \
                "
                cd /home/ec2-user/project && \
                docker-compose up -d --build && \
                echo 'Waiting for containers to become healthy...' && \
                for i in {1..10}; do \
                    docker-compose ps | grep -q 'healthy' && break || sleep 5; \
                done || (echo 'Containers failed to become healthy' && exit 1)
                "
                """
            }
        }
    }

    stage('Run E2E Tests') {
        steps {
            withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {    "vockey": Unknown word.
                sh """
                ssh -i \$SSH_KEY -o StrictHostKeyChecking=no ec2-user@${INSTANCE_IP} \
                "
                sudo docker system prune -af && \
                docker run --name e2e-container -e TEST_URL=http://rmit-store-client-service \
                    --network project_default locoioioi/e2e && \
                docker cp e2e-container:/e2e/results /home/ec2-user/cypress-reports
                "
                """
            }
        }
    }

    stage('Retrieve Test Results') {
        steps {
            withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {    "vockey": Unknown word.
                sh """
                scp -i \$SSH_KEY -o StrictHostKeyChecking=no ec2-user@${INSTANCE_IP}:/home/ec2-user/cypress-reports/test-results*.xml ./cypress-reports/
                """
            }
        }
    }

    stage('Analyze Test Results') {
        steps {
            junit 'cypress-reports/test-results*.xml'
        }
    }
}
```

Figure 17 Jenkinsfile - run test

- Figure 8 indicates that the jenkinsfile will run the website with the docker-compose up -d –build command and then wait for it to be healthy.
- Next, the pipeline will **run end-to-end (E2E) tests** by executing a **Docker container** that runs Cypress tests against the deployed website.
- The test results will be copied from the container to the EC2 instance using docker cp.
- Afterward, the pipeline will **retrieve the test results** by securely copying them (scp) from the EC2 instance to the Jenkins workspace.
- Finally, the pipeline will **analyze the test results** using junit, allowing Jenkins to display test reports and determine the success or failure of the job.

```
post {
    success {
        echo 'Testing passed!'
    }
    failure {
        echo 'Testing failed!'
    }
    always {
        echo 'Cleaning up resources...'
        script {
            try {
                sh """
                aws cloudformation delete-stack --stack-name ${STACK_NAME}
                """
                echo 'CloudFormation stack deletion triggered.'
            } catch (Exception e) {
                echo "Failed to delete CloudFormation stack: ${e.message}"
            }
        }
        echo 'Job finished!'
    }
}
```

Figure 18 Jenkinsfile - post action

- After the pipeline is finished, it will always **clean up resources** like delete the **testing environment**.
- The pipeline will determine whether the job is **a success** or **failure** based on the stage "**Analyze test result**".

```
You, 5 days ago | 1 author (You)
describe("Product display in the shop test", () => {      You, 5 days ago • Frontend testing impl (#
  before(() => {
    cy.visit("/shop");
  });

  it("should display a list of products", () => {
    cy.get(".product-list").should("exist");

    cy.get(".product-container")
      .should("exist")
      .and("have.length.greaterThan", 0);
  });
});

describe("Product detail page test", () => {
  before(() => {
    cy.visit("/shop");
  });

  it("should navigate to the product detail page", () => {
    cy.get(".product-container").first().click();

    cy.url().should("include", "/product/");
  });

  after(() => {
    cy.go("back");
  });
});
```

```
describe("Filtering test", () => {
  before(() => {
    cy.visit("/shop");
  });

  it("Should display products high to low price", () => {
    cy.contains("Newest First").click();

    cy.contains("Price High to Low").click();

    cy.get(".product-list").should("exist");

    cy.wait(2000);
    cy.get(".product-container .price").then((priceElements) => {
      const prices = [...priceElements].map((el) =>
        parseFloat(el.innerText.replace("$", ""))
      );

      const sortedPrices = [...prices].sort((a, b) => b - a);
      expect(prices).to.deep.equal(sortedPrices);
    });
  });
});

describe("Login Page Tests", () => {
  beforeEach(() => {
```

Figure 19 E2e test

- This is the e2e test that the pipeline above will build image and run this test, and you can access this folder though client/cypress/e2e.
- It includes product display in the shop test, product detail page display test, filtering test, login test, add to cart functionality test, place order functionality test.

Backend pipeline

d) Workflow for Pull Request on Back-end with GitHub Action

Name: Jenkins CI PR BE

Trigger On: This job is triggered on a pull request event to the main branch when changes are pushed to the server folder.

Build and Push Docker Image:

- Check out the code from the pull request branch using the actions/checkout action.
- Log in to Docker Hub using the docker/login-action action with credentials from GitHub secrets.
- Extract the short commit SHA for tagging the Docker image.
- Build the Docker image for the server using the extracted commit SHA as the tag.
- Push the tagged Docker image to Docker Hub.

Trigger Jenkins Job:

- Define variables including Jenkins job name, URL, user, API token, and the Docker image tag.
- Obtain a Jenkins Crumb for secure API access.
- Trigger the Jenkins job with the Docker image tag as a parameter using an HTTP POST request.
- Check the HTTP response to confirm successful job triggering.

Wait for Jenkins Job Completion:

- Continuously check the Jenkins job status at regular intervals until it completes.
- Fetch and display Jenkins console logs upon completion.
- Exit with an error if the Jenkins job fails.

```
name: Jenkins CI PR BE

on:
  pull_request:
    branches:
      - main
    paths:
      - 'server/**'

jobs:
  build-and-push:
```

Figure 20 Jenkins CI PR BE Github Action Job

e) Trigger the Jenkins CI jobs with GitHub Action

- Repeat the steps to create a Jenkins Pipeline above, but change the string parameter to “image_tag”, and Script Path to “server/jenkins/Jenkinsfile”. Use the same credentials as the previous job for frontend CI.

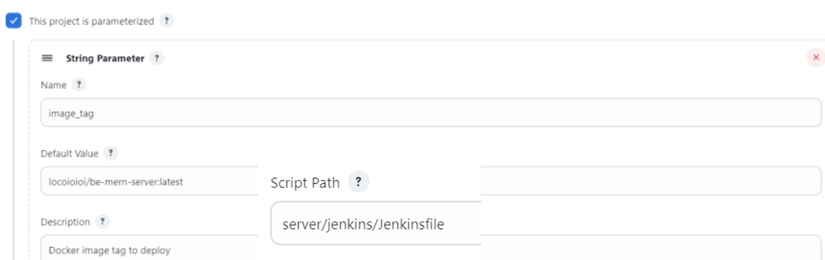


Figure 21 Jenkins CI BE Config

- This is the Jenkinsfile that the pipeline will execute, the GitHub Action jenkin-ci-be.yaml will trigger this file with input parameters equal to the extracted commit SHA.

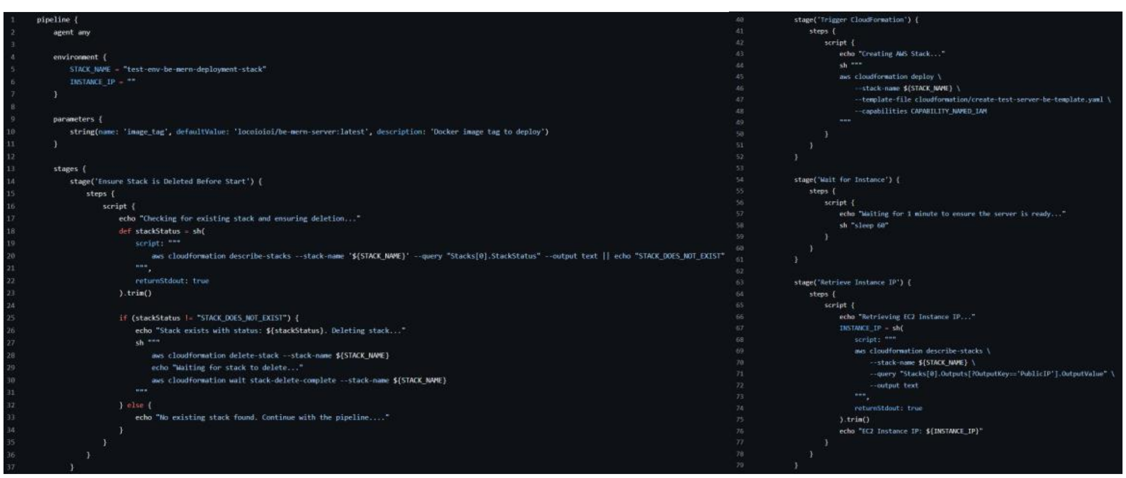


Figure 22 Jenkinsfile for CI BE

- This Jenkins job will first check if a cloudformation stack already exist or not according to the defined cloudformation template in the github repo, if yes, it will removed the stack before going on.
- If not exist, it will create the stack, which include an EC2 server, and other required resources, then retrieve the IP address.

```
stage('Install Dependencies') {
    steps {
        withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {
            sh """
            ssh -i $SSH_KEY -o StrictHostKeyChecking=no ec2-user@${INSTANCE_IP} "
            echo "Starting: Update system packages"
            sudo yum update -y
            echo "Complete: Update system packages"

            echo "Starting: Install Docker"
            sudo yum install -y docker
            sudo service docker start
            echo "Complete: Install Docker"

            echo "Starting: Install Node.js using nvm"
            # Install nvm
            curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.4/install.sh | bash

            # Load nvm in this session
            export NVM_DIR="/home/ec2-user/.nvm"
            [ -s "/home/ec2-user/nvm.sh" ] && \\. "/home/ec2-user/nvm.sh"
            [ -s "/home/ec2-user/bash_completion" ] && \\. "/home/ec2-user/bash_completion"

            # Install and use the latest LTS version of Node.js
            nvm install --lts
            nvm use --lts
            echo "Complete: Install Node.js using nvm"

            echo "Starting: Verify Node.js installation"
            node -v
            echo "Complete: Verify Node.js installation"

            echo "Starting: Verify npm installation"
            npm -v
            echo "Complete: Verify npm installation"

            echo "Install Dependencies Successfully"
            "
            """
        }
    }
}
```

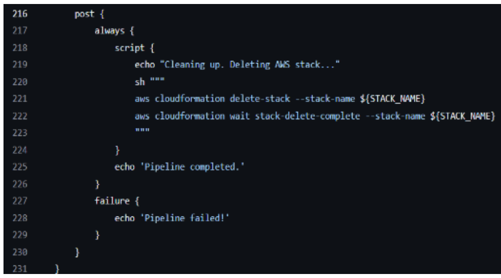
```
stage('Install MongoDB') {
    steps {
        withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {
            sh """
            ssh -i $SSH_KEY -o StrictHostKeyChecking=no ec2-user@${INSTANCE_IP} "
            # Add the MongoDB repository
            echo '[mongodb-org-6.0]' | sudo tee /etc/yum.repos.d/mongodb-org-6.0.repo
            echo 'name=MongoDB Repository' | sudo tee -a /etc/yum.repos.d/mongodb-org-6.0.repo
            echo 'baseurl=https://repo.mongodb.org/yum/amazon/2023/mongodb-org/6.0/x86_64/' | sudo tee -a /etc/yum.repos.d/mongodb-org-6.0.repo
            echo 'gpgcheck=1' | sudo tee -a /etc/yum.repos.d/mongodb-org-6.0.repo
            echo 'enabled=1' | sudo tee -a /etc/yum.repos.d/mongodb-org-6.0.repo
            echo 'gpgkey=https://www.mongodb.org/static/pgp/server-6.0.asc' | sudo tee -a /etc/yum.repos.d/mongodb-org-6.0.repo

            # Install and Start MongoDB
            sudo yum install -y mongodb-org
            sudo systemctl start mongod
            sudo systemctl enable mongod
            mongod --version
            sudo yum remove -y mongodb-mongosh
            sudo yum install -y mongodb-mongosh-shared-openssl3
            sudo yum install -y mongodb-mongosh
            echo 'Install MongoDB Successfully'
            "
            """
        }
    }
```

Figure 23 Jenkinsfile for CI BE

- It will then SSH into the EC2 and install Docker, NodeJS, MongoDB and install them.

```
stage('Deploy and Seed MongoDB') {
    steps {
        withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {
            sh """
            ssh -i $SSH_KEY -o StrictHostKeyChecking=no ec2-user@${INSTANCE_IP} "
            echo 'Pulling the Docker image: ${image_tag}'
            sudo docker pull ${image_tag}
            echo "Image pulled successfully"

            # Stop and remove any existing container named be-mern-server
            sudo docker stop be-mern-server || true
            sudo docker rm be-mern-server || true

            # Run a temporary container to seed the database
            sudo docker run --rm \
                -e NODE_ENV=test \
                -e MONGO_URI=mongodb://localhost:27017/rmit_ecommerce \
                ${image_tag} \
                npm run seed:db admin@rmit.edu.vn mypassword

            # Run the application container
            sudo docker run -d --name be-mern-server -p 3000:3000 \
                -e NODE_ENV=test \
                -e MONGO_URI=mongodb://localhost:27017/rmit_ecommerce \
                ${image_tag}

            echo "Deploy and Seed MongoDB Successfully"
            "
            """
        }
    }
}
```

```
stage('Health Check') {
    steps {
        withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {
            sh """
            ssh -i $SSH_KEY -o StrictHostKeyChecking=no ec2-user@${INSTANCE_IP} "
            echo 'Waiting for the server to be ready...'
            sleep 10
            echo 'Testing healthcheck endpoint'
            curl -v -kf http://localhost:3000/healthcheck || (echo 'Healthcheck failed' && exit 1)
            "
            """
        }
    }
}

stage('MongoDB Health Check') {
    steps {
        withCredentials([sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')]) {
            sh """
            ssh -i $SSH_KEY -o StrictHostKeyChecking=no ec2-user@${INSTANCE_IP} "
            mongosh --eval 'db.runCommand({ ping: 1 })' || (echo 'MongoDB ping failed' && exit 1)
            "
            """
        }
    }
}
}
```



```
post {
    always {
        script {
            echo "Cleaning up. Deleting AWS stack..."
            sh """
            aws cloudformation delete-stack --stack-name ${STACK_NAME}
            aws cloudformation wait stack-delete-complete --stack-name ${STACK_NAME}
            """
        }
        echo 'Pipeline completed.'
    }
    failure {
        echo 'Pipeline failed!'
    }
}
}
```

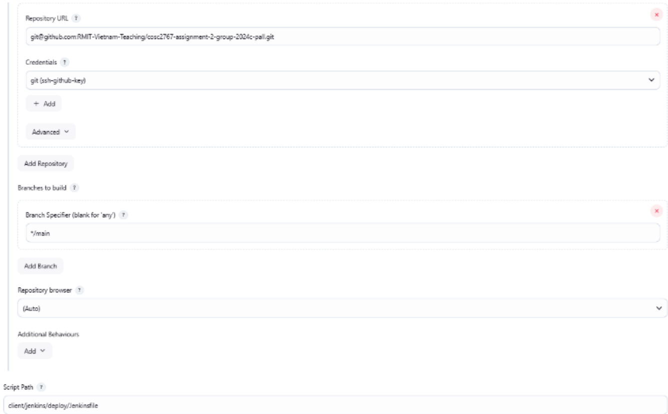
Figure 24 Jenkinsfile for CI BE

- It pull the docker image with the tag of the commit SHA, and run it. It also start the MongoDB database and seed it.
- Then it performs a healthcheck to port 3000 of the server container, and test the connection to the local MongoDB.
- If everything pass, or even if not it will cleanup the job by deleting all resources in the CloudFormation stack.

Continuous Deployment

Frontend pipeline

a) Set up Jenkins job to handle push event to main.



- Connect to Jenkins server and create new pipeline item with the name fe-deploy.
- Config the Jenkins job to trigger though webhooks of repository and read the correct jenkinsfile path.

Figure 25 Jenkins job - configuration

```
stage('Checkout Code') {
    steps {
        git branch: 'main', credentialsId: 'ssh-github-key', url: 'git@github.com:RMIT-Vietnam-Teaching/cosc2767-assignment-2-group-2024c-pall.git'
    }
}

stage('Install Dependencies') {
    steps {
        dir('client') {
            sh 'npm ci'
        }
    }
}

stage('Build React App') {
    steps {
        dir('client') {
            sh 'npm run build'
        }
    }
}
```

- This is the first step of the deployment job for Frontend, after checkout to the main branch the pipeline will install dependency and build it.

Figure 26 Jenkinsfile - build react project

```
stage('Manage Docker Image') {
    steps {
        dir('client') {
            script {
                // Login to Docker registry
                docker.withRegistry('https://index.docker.io/v1/', 'locoioioi-dockerhub') {    "dockerhub": Unknown word.
                    // Pull the existing latest image
                    sh "docker pull ${IMAGE_NAME}:${IMAGE_VERSION}"

                    // Tag the pulled image as 'stable'
                    sh "docker tag ${IMAGE_NAME}:${IMAGE_VERSION} ${IMAGE_NAME}:${STABLE_TAG}"

                    // Push the 'stable' image to Docker Hub
                    sh "docker push ${IMAGE_NAME}:${STABLE_TAG}"

                    // Build a new image tagged as 'latest'
                    def customImage = docker.build("${IMAGE_NAME}:${IMAGE_VERSION}")

                    // Push the new 'latest' image to Docker Hub
                    customImage.push()
                }
            }
        }
    }
}

stage('Deploy to Kubernetes') {
    steps {
        script {
            withCredentials([
                string(credentialsId: 'eks-cluster-ip', variable: 'EKS_IP'),
                sshUserPrivateKey(credentialsId: 'ec2-ssh-vockey', keyFileVariable: 'SSH_KEY')    "vockey": Unknown word.
            ]) {
                sh """
                ssh -i \$SSH_KEY -o StrictHostKeyChecking=no ec2-user@\$EKS_IP \
                "sudo -i sh -c 'whoami && kubectl rollout restart deployment rmit-store-client-canary'"
                """
            }
        }
    }
}
}
```

- After being built successfully, the pipeline will get the current latest tag from the registry and change it to stable tag, then it will push a new latest version of the frontend.
- In the last stage, the pipeline will ssh to the EKS control plan then exec kubectl rollout restart deployment rmit-store-client-canary to deploy.

Figure 27 Jenkinsfile - upadte image and rollout new deployment

fe_deploy

Last Successful Artifacts

Stage View

	Declarative: Checkout SCM	Declarative: Tool Install	Checkout Code	Install Dependencies	Build React App	Archive Artifacts	Test Docker	Manage Docker Image	Deploy to Kubernetes	Declarative: Post Actions
Average stage times: (Average full run time: ~1min 7s)	971ms	244ms	860ms	19s	25s	357ms	563ms	15s	1s	151ms
#25 Tháng 1 17 01:16 2 commits	1s	233ms	831ms	18s	29s	392ms	540ms	15s	1s	249ms

Figure 28 CD results

- Figure 15 indicates the pipeline is completed successfully and rollout a new version for RMIT store.

Backend pipeline

b) Workflow to handle push event to main and PR closure

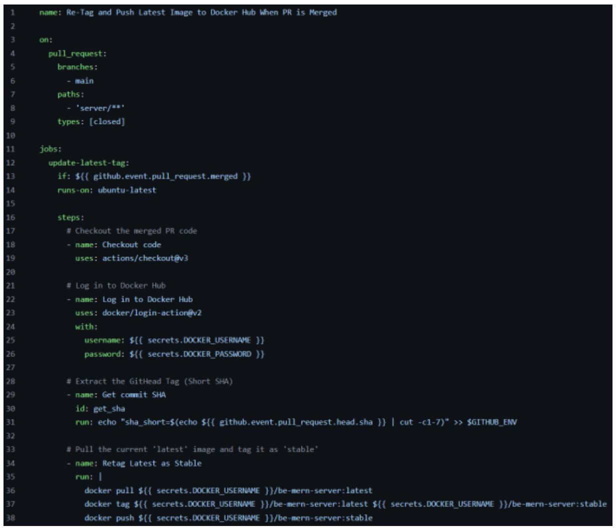
Name: Re-Tag and Push Latest Image to Docker Hub When PR is Merged

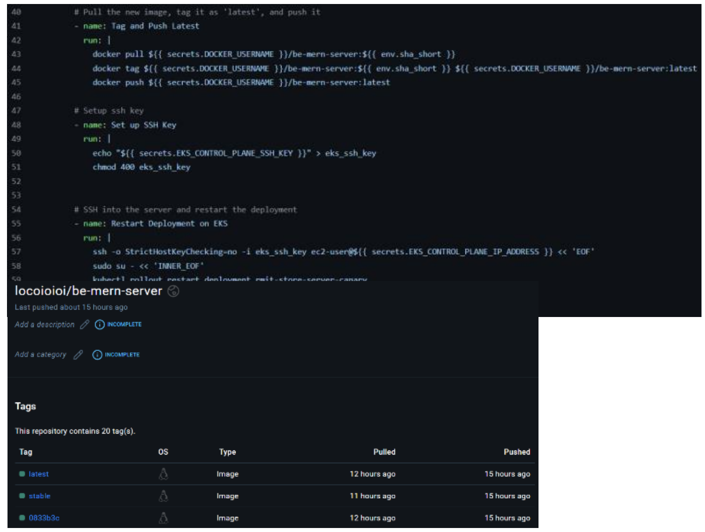
Trigger On: This job is triggered on pull request closures to the main branch that modify the server folder, only if the pull request is merged.

Update Latest Tag:

- Check out the merged pull request code using the actions/checkout action
- Log in to Docker Hub with credentials from GitHub secrets

- Extract the short commit SHA for tagging Docker images
- Pull the current latest image, tag it as stable, and push the stable tag to Docker Hub
- Pull the new image by commit SHA, tag it as latest, and push the latest tag to Docker Hub
- Set up an SSH key using the EKS control plane SSH key from GitHub secrets
- SSH into the EKS control plane and restart the deployment for the rmit-store-server-canary Kubernetes deployment





- The stable and latest image is now available in Dockerhub

Containerized Microservices

Our applications are separated into 3 parts, a frontend, backend and a database. While MongoDB is already used as a separate service hosted on the mongo cloud itself, the frontend and backend are baked into Docker images for containerization.

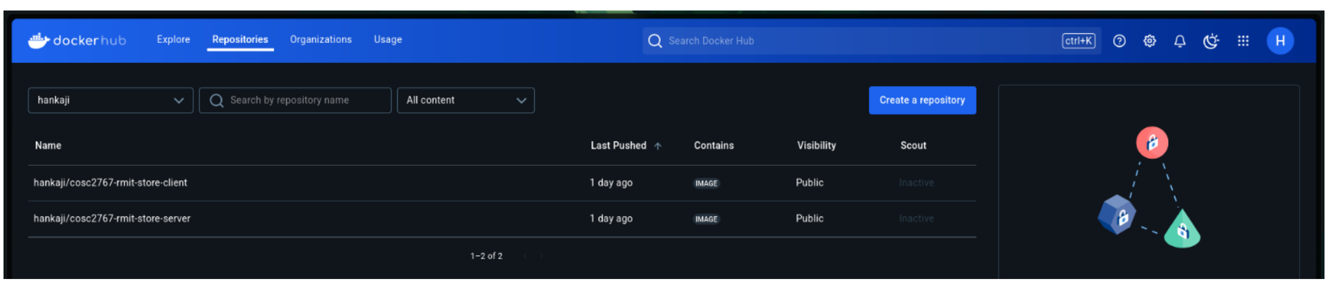


Figure 29. DockerHub Repo

These images are then pulled down by our Kubernetes deployment and deployed automatically into pods and clusters by Kubernetes. Furthermore, using Eks by Aws, we can separate our clusters into different worker group nodes, meaning that each deployment that contains a group of service clusters can run in different elastic containers, ensuring that in case of instability and any of our services goes down, it will not affect the other services and Kubernetes can easily replace it with another running services.

Figure 30. EKS Node Group

```yaml
kubernetes/application/server-deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rmit-store-server-pod
  labels:
    name: rmit-store-server-pod
    app: rmit-store-server
    version: stable
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: "/metrics"
    prometheus.io/port: "3000"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: rmit-store-server
      version: stable
  template:
    metadata:
      labels:
        app: rmit-store-server
        name: rmit-store-server-pod
        version: stable
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/path: "/metrics"
        prometheus.io/port: "3000"
    spec:
      containers:
        - name: rmit-store-server-pod
          image: locoioioi/be-mern-server:stable
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
      nodeSelector:
        eks.amazonaws.com/nodegroup: server-worker-group
```

Figure 31. Server Deployment Configuration

By using Kubernetes' deployment service, we can dynamically scale our application services at runtime without needing to rewrite configurations each time, both horizontally and vertically. Scaling is straightforward and can be achieved with a few terminal commands. For horizontal scaling, we use the kubectl scale command to adjust the number of pods allocated to a deployment, enabling even distribution of traffic across multiple pods and reducing service workloads. For vertical scaling, the kubectl resources command allows us to allocate specific system resources (e.g., CPU, memory, and disk space) to deployments or individual clusters. When application demand decreases, we can scale down vertically to conserve resources and reduce costs, avoiding unnecessary resource allocation

```
NAME                                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/rmit-store-client-canary   0/0     0            0           14h
deployment.apps/rmit-store-client-pod      1/1     1            1           14h
deployment.apps/rmit-store-server-canary   0/0     0            0           14h
deployment.apps/rmit-store-server-pod      20/20   20           20          14h
```

Figure 32. Deployment list in Kubernetes

To enhance reliability, we run our Kubernetes clusters across multiple subnets, ensuring our services are available in multiple regions. This allows consumers from different parts of the world to experience reliable connectivity to our application without encountering network lag caused by regional restrictions.

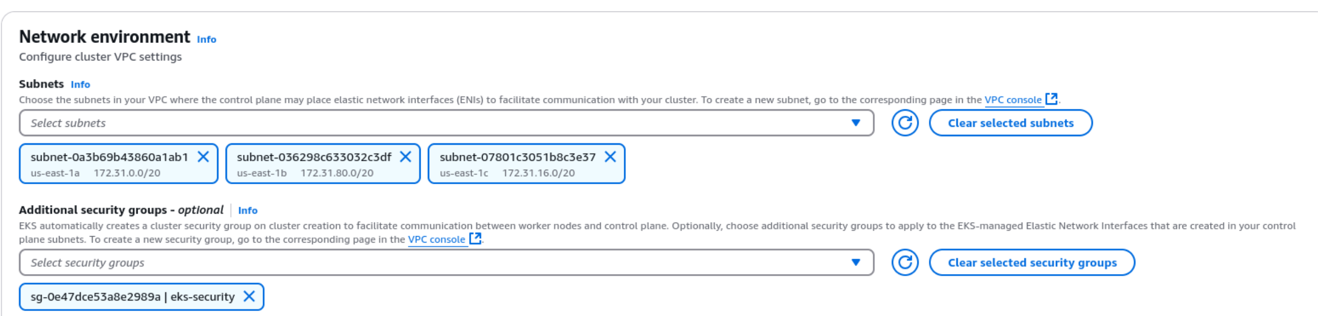


Figure 33. EKS Network Environment

Configuration management

Configurations are managed in two different ways. For our application, including the frontend and backend, we use environment variables. These variables are provided either through the Kubernetes container's environment settings or directly via a dotenv (.env) file. The configurations are applied at build time, so whenever changes are made to the application, a deployment restart is required for the new configurations to take effect.

```
server/.env
PORT=3000
# Local MongoDB URI Example
MONGO_URI_PROD=mongodb+srv://admin:123@cluster0.3vynp.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0
MONGO_URI_TEST=mongodb://localhost:27017/rmit_ecommerce #local MongoDB
appName=Cluster0
JWT_SECRET=my_secret_string
CLIENT_URL=http://rmit-store-client-server
BASE_API_URL=api
NODE_ENV=test
```

Figure 34. Dotenv configuration

```
kubernetes/application/client-deployment.yml
spec:
  containers:
    - name: rmit-store-client-pod
      image: locoioioi/cosc2767-rmit-store-client:stable
      imagePullPolicy: Always
      ports:
        - containerPort: 80
      env:
        - name: API_URL
          value: /api
  nodeSelector:
    eks.amazonaws.com/nodegroup: client-worker-group
```

```
kubernetes/grafana.yml
containers:
  - name: grafana
    env:
    - name: GF_SMTP_ENABLED
      value: "true"
    - name: GF_SMTP_HOST
      value: "smtp.gmail.com:587"
    - name: GF_SMTP_USER
      value: "dtlamdevil@gmail.com"
    - name: GF_SMTP_PASSWORD
      value: "cuty rniy libv brdy"
    - name: GF_SMTP_FROM_ADDRESS
      value: "grafana@gmail.com"
    - name: GF_SMTP_FROM_NAME
      value: "Grafana"
    - name: GF_SMTP_SKIP_VERIFY
      value: "true"
    image: grafana/grafana-enterprise
    imagePullPolicy: IfNotPresent
```

Figure 35. Kubernetes deployment env

For other services like Prometheus, we host a ConfigMap and allow the service to use its configuration by connecting to the ConfigMap through a volume mount. Same as before, whenever changes are made to the ConfigMap, an application of Configmap and deployment restart is required for the new configurations to take effect.

```
kubernetes/prometheus-config.yml
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-server-conf
  namespace: monitoring
data:
  prometheus.yml: |
    global:
      scrape_interval: 2m
      evaluation_interval: 2m
      body_size_limit: "0"
      sample_limit: 0
      label_limit: 0
      label_name_length_limit: 0
      label_value_length_limit: 0
      target_limit: 0
    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
          - targets: ['localhost:9090']
      - job_name: 'node-exporter'
        static_configs:
          - targets: ['node-exporter.default.svc.cluster.local:9100']
      - job_name: 'Store server'
        kubernetes_sd_configs:
          - role: pod
            namespaces:
              names:
                - default
        relabel_configs:
          - source_labels: [__meta_kubernetes_pod_label_version]
            target_label: version
          - source_labels: [__meta_kubernetes_pod_label_app]
            target_label: app
          - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
            action: keep
            regex: true
          - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
            action: replace
            target_label: __metrics_path__
          - source_labels: [__meta_kubernetes_pod_ip, __meta_kubernetes_pod_annotation_prometheus_io_port]
            action: replace
            target_label: __address__
            regex: (.+)
            separator: ":"
```

Figure 36. Configmap for Prometheus

```yaml
kubernetes/prometheus-deployment.yml
spec:
  serviceAccountName: prometheus
  containers:
    - name: prometheus
      image: prom/prometheus
      ports:
        - containerPort: 9090
      volumeMounts:
        - name: config-volume
          mountPath: /etc/prometheus
  volumes:
    - name: config-volume
      configMap:
        name: prometheus-server-conf
        defaultMode: 420
  nodeSelector:
    eks.amazonaws.com/nodegroup: monitoring-group
```

Figure 37. Prometheus volume mount

Orchestration

As mentioned before, we use Eks, a Kubernetes managed by Aws, to deploy our applications using containerized services, which are pushed to Docker Hub.

Furthermore, by using Amazon EKS (Elastic Kubernetes Service), we can separate our clusters into different worker node groups. This allows each deployment, comprising a group of service clusters, to run in separate elastic containers. As a result, if instability occurs and one of our services goes down, it will not impact other services. Kubernetes can seamlessly replace the affected service with another running instance, ensuring minimal disruption.

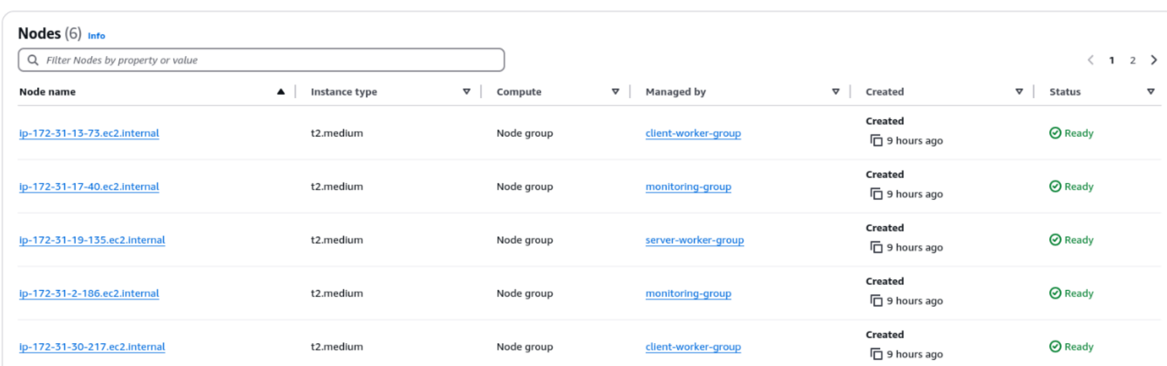
Node groups (3) Info

Node groups implement basic compute scaling through EC2 Auto Scaling groups.

Edit | Delete | Add node group

Group name	Desired size	AMI release version	Launch template	Status
client-worker-group	2	1.31.3-20250103	-	Active
monitoring-group	2	1.31.3-20250103	-	Active
server-worker-group	2	1.31.3-20250103	-	Active

Figure 38. EKS Node groups



Nodes (6) Info

Filter Nodes by property or value

Node name	Instance type	Compute	Managed by	Created	Status
ip-172-31-13-73.ec2.internal	t2.medium	Node group	client-worker-group	Created 9 hours ago	Ready
ip-172-31-17-40.ec2.internal	t2.medium	Node group	monitoring-group	Created 9 hours ago	Ready
ip-172-31-19-135.ec2.internal	t2.medium	Node group	server-worker-group	Created 9 hours ago	Ready
ip-172-31-2-186.ec2.internal	t2.medium	Node group	monitoring-group	Created 9 hours ago	Ready
ip-172-31-30-217.ec2.internal	t2.medium	Node group	client-worker-group	Created 9 hours ago	Ready

Figure 39. EKS Nodes

```yaml
kubernetes/application/server-deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rmit-store-server-pod
  labels:
    name: rmit-store-server-pod
    app: rmit-store-server
    version: stable
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: "/metrics"
    prometheus.io/port: "3000"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: rmit-store-server
      version: stable
  template:
    metadata:
      labels:
        app: rmit-store-server
        name: rmit-store-server-pod
        version: stable
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/path: "/metrics"
        prometheus.io/port: "3000"
    spec:
      containers:
        - name: rmit-store-server-pod
          image: locoioioi/be-mern-server:stable
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
      nodeSelector:
        eks.amazonaws.com/nodegroup: server-worker-group
```

Figure 40. Server deployment configuration

The deployment of each service is managed through a YAML file, which instructs Kubernetes on the actions to take. To simplify this process, Kubernetes provides a Deployment resource type, which we use for the various services in our application. This resource type offers notable features, such as the ability to dynamically allocate system resources and adjust the number of clusters running for each service. This ensures flexibility in scaling our application to meet changing demands.

By specifying the number of replicas, we can tell Kubernetes how many pods to run for a desired service. This number can also be adjusted at runtime using the kubectl scale command. Similarly, resource allocations for each deployment can be modified dynamically as needed.

However, running pods alone are not nothing if they cannot be accessed. To address this, we use the Service resource in Kubernetes to expose our application to consumers. The type of Service we use depends on the service that we want to run. For example, for our front-end service, where end users interact directly, we use the Load Balancer service type. This allows Kubernetes to automatically manage internal IP addresses and ports, exposing the application to the outside world with an external IP address, which is far more user-friendly than a random sequence of numbers.

For other types, such as the backend or Node Exporter, it is unnecessary to expose them to the outside world since only internal applications and us developers need to interact with these services. In such cases, we use the Cluster IP service type, which restricts connection to internal communication within the cluster.

We connect with these services using the specific DNS domains supplied by Kubernetes, which are formed from the service name. For example, if our backend service is called rmit-store-server-service, we can access it internally via http://rmit-store-server-service:<port> where port is what we specified in deployment and service yaml files.

```yaml
kubernetes/application/client-canary-deployment.yml
apiVersion: v1
kind: Service
metadata:
  name: rmit-store-client-service
  labels:
    name: rmit-store-client-service
    app: rmit-store-client
spec:
  ports:
    - port: 80
      targetPort: 80
  type: LoadBalancer
  selector:
    app: rmit-store-client
```

```yaml
kubernetes/application/server-deployment.yml
apiVersion: v1
kind: Service
metadata:
  name: rmit-store-server-service
  labels:
    name: rmit-store-server-service
    app: rmit-store
spec:
  ports:
    - port: 3000
      targetPort: 3000
  selector:
    app: rmit-store-server
```

Figure 41. LoadBalancer & ClusterIP Service

IV. Pipeline Advanced Component

8. Infrastructure as Code (IaC)

This is a service that helps us model and set up AWS resources and provides more time for applications operating in AWS rather than dealing with resource management [4]. It is used to bring up the test environment that simulates the production environment then we can shut down and eliminate that environment after testing for cost efficiency.

```yaml
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  TestEnvEC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      InstanceType: "t2.micro"
      KeyName: "vockey" # The key used in the Jenkinsfile credential
      ImageId: "ami-05576a079321f21f8" # Amazon Linux 2 AMI
      SecurityGroupIds:
        - !Ref TestEnvSecurityGroup
      UserData:
        Fn::Base64: !Sub |
          #!/bin/bash
          yum update -y
          yum install -y docker
          service docker start
          usermod -a -G docker ec2-user
          amazon-linux-extras enable nginx1
          yum install -y nodejs npm
          curl -fsSL https://deb.nodesource.com/setup_18.x | bash -
          yum install -y mongodb-org
          systemctl start mongod
          systemctl enable mongod

  TestEnvSecurityGroup:
    Type: "AWS::EC2::SecurityGroup"
    Properties:
      GroupDescription: "Enable SSH and HTTP access"
      SecurityGroupIngress:
        - IpProtocol: "tcp"
          FromPort: "22"
          ToPort: "22"
          CidrIp: "0.0.0.0/0"
        - IpProtocol: "tcp"
          FromPort: "80"
          ToPort: "80"
          CidrIp: "0.0.0.0/0"
        - IpProtocol: "tcp"
          FromPort: "3000"
          ToPort: "3000"
          CidrIp: "0.0.0.0/0"

Outputs:
  PublicIP:
    Description: Public IP address of the EC2 instance
    Value: !GetAtt TestEnvEC2Instance.PublicIp
```

Figure 42 AWS CloudFormation template

This template is mainly focused on set up an environment for testing purpose and it will be configured as below:

- **Type:** AWS::EC2::Instance
- **Properties:**
  - **Instance type:** it will specify the type of instance that will be used for the EC2 instance, for example, figure 21 will create t2.micro instance.
  - **KeyName:** This will specify what private key will be used for connecting to the instance.
  - **SecurityGroupIds:** Associates the instance with TestEnvSecurityGroup for security configurations.
  - **UserData:** This will specify the script will be executed after the instance is up and running.

9. Monitoring and Alerting

a. Prometheus

Prometheus is an open-source application monitoring system that collects metrics exposed by various parts of our application. To set up Prometheus, we first configure our application to expose metrics that Prometheus can scrape to monitor application health and performance. Next, we configure Prometheus to scrape these metrics by specifying the endpoints and the scrape interval in a YAML configuration file, as shown below:

```yaml
kubernetes/prometheus-config.yml
global:
  scrape_interval: 2m
  evaluation_interval: 2m
  body_size_limit: "0"
  sample_limit: 0
  label_limit: 0
  label_name_length_limit: 0
  label_value_length_limit: 0
  target_limit: 0
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter.default.svc.cluster.local:9100']
  - job_name: 'Store server'
    kubernetes_sd_configs:
      - role: pod
        namespaces:
          names:
            - default
    relabel_configs:
      - source_labels: [__meta_kubernetes_pod_label_version]
        target_label: version
      - source_labels: [__meta_kubernetes_pod_label_app]
        target_label: app
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
        action: keep
        regex: true
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
        action: replace
        target_label: __metrics_path__
      - source_labels: [__meta_kubernetes_pod_ip, __meta_kubernetes_pod_annotation_prometheus_io_port]
        action: replace
        target_label: __address__
        regex: (.+)
        separator: ":"
```

Figure 43. Prometheus config

The global scrape and evaluate interval instruct Prometheus that it needs to scrape and evaluate every 2 minutes intervals, applying to all jobs.

We have two main jobs to monitor: 'Node Exporter' and 'Store Server.' For the Node Exporter, we deploy it using the Daemon Set resource provided by Kubernetes, which ensures it runs on all nodes within a namespace (default in this case). These Node Exporters monitor system resources and usage on each node, exposing the metrics data at http://<node-ip>:9100/metrics.

```yaml
kubernetes/rbac-prometheus.yml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - apiGroups: [""]
    resources:
      - pods
      - services
      - endpoints
      - nodes
      - namespaces
    verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus-binding
  namespace: monitoring
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
  - kind: ServiceAccount
    name: prometheus
    namespace: monitoring
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: monitoring
```

```yaml
kubernetes/node-exporter-daemonset.yml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    app.kubernetes.io/component: exporter
    app.kubernetes.io/name: node-exporter
  name: node-exporter
spec:
  selector:
    matchLabels:
      app.kubernetes.io/component: exporter
      app.kubernetes.io/name: node-exporter
  template:
    metadata:
      labels:
        app.kubernetes.io/component: exporter
        app.kubernetes.io/name: node-exporter
    spec:
      containers:
      - args:
        - --path.sysfs=/host/sys
        - --path.rootfs=/host/root
        - --no-collector.wifi
        - --no-collector.hwmon
        - --collector.filesystem.ignored-mount-points=^/(dev|proc|sys|var/lib/docker/.+|var/lib/kubelet/pods/.+)($|/)
        - --collector.netclass.ignored-devices=^(veth.*)$
        name: node-exporter
        image: prom/node-exporter
        ports:
          - containerPort: 9100
            protocol: TCP
        resources:
          limits:
            cpu: 250m
            memory: 180Mi
          requests:
            cpu: 102m
            memory: 180Mi
        volumeMounts:
        - mountPath: /host/sys
          mountPropagation: HostToContainer
          name: sys
          readOnly: true
        - mountPath: /host/root
          mountPropagation: HostToContainer
          name: root
          readOnly: true
      volumes:
      - hostPath:
          path: /sys
        name: sys
      - hostPath:
          path: /
        name: root
```

Figure 44. Prometheus & Node exporter configuration

However, since our Prometheus deployment and Node Exporter are in different namespaces, Prometheus does not have sufficient permissions to access pods and resources in other namespaces. To address this, we need to create RBAC (Role-Based Access Control) permissions by defining Cluster Roles and Cluster Role Bindings. Additionally, we configure a custom Service Account for Prometheus, granting it the necessary permissions to access the required pods and namespaces.

```
server/index.js
// Middleware to enable prometheus scraping
const metricsMiddleware = promBundle({
  includeMethod: true,
  includePath: true,
  includeStatusCode: true,
  includeUp: true,
  customLabels: {
    project_name: "Rmit-store",
    project_type: "Rmit-server-metrics",
  },
  promClient: {
    collectDefaultMetrics: {},
  },
});

app.use(metricsMiddleware);
```

Figure 45. Prom Client configuration

For the Store Server, we use prom-client and express-prom-bundle to expose metrics about system usage and HTTP request rates. These metrics are made available at http://<node-ip>:3000/metrics. However, instead of using a static endpoint to scrape data, we leverage Kubernetes' service discovery feature to dynamically scrape all service pods. This ensures that Prometheus can correctly gather data even when new clusters are added. Additionally, this approach automatically labels the data based on pod labels, allowing us to easily query and differentiate metrics for canary and stable deployments.

b. Grafana

Grafana is a powerful tool for visualizing data queried from a data source, such as Prometheus in our case. It allows us to import pre-built dashboards or create custom visualizations to monitor data trends over time through graphs and other visual formats. This is particularly useful for identifying application instabilities or unexpected traffic spikes, enabling us to scale the application appropriately without over-provisioning resources.

Figure 46. Grafana Dashboard

First, we demonstrate a traffic spike in our application by querying the HTTP request count from Prometheus, which shows the number of requests being made to our application. As illustrated in the figure above, the request count spikes significantly before gradually decreasing over the next few minutes. This behavior was simulated using curl to send requests to the application every 0.5 seconds for a total of 100 requests.

There is also a visualization for the canary deployment, which displays the number of services currently running. As shown in the figure above, the server service of our application starts with both the canary and stable versions running. Over time, the canary version gradually scales down to 0 instances, while the stable version scales up to 10 running instances.

Moreover, we have configured alert rules to notify us when certain thresholds are exceeded. For example, if the traffic spike surpasses a threshold of 0.1, an email alert is sent, notifying us of the high request rate in our application. This allows us to take quick action before the application becomes stressed. Similarly, an alert is triggered if RAM usage exceeds 1.5GB, informing us of high memory consumption so we can address the issue promptly.

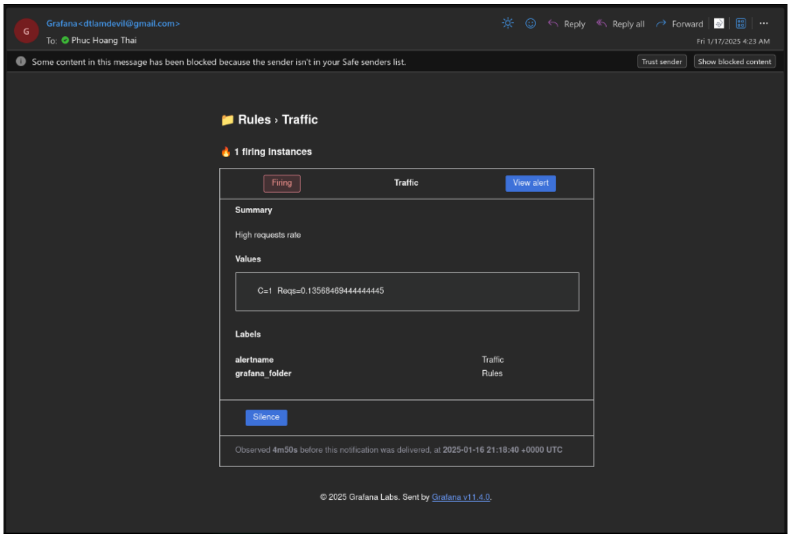


Figure 47. Grafana Email Notification

10.Canary Deployment

In our CI/CD application, we will utilize the canary strategy which will roll out the latest version of the application gradually to only a small subset of users and keep the majority of users stay on the stable deployed version. Once there are no issues with the canary releases, it will be available to broader audiences and a smaller proportion of already-deployed versions.

In the pipeline, for client and server instance, each instance will have the deployment for stable and canary version. As the configuration for the canary deployment strategies is similar among the front-end and back-end instance, only the code for the client are explained.

Stable Version Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rmit-store-client-pod
  labels:
    name: rmit-store-client-pod
    app: rmit-store-client
    version: stable
spec:
  replicas: 2
  selector:
    matchLabels:
      app: rmit-store-client
      version: stable
  template:
    metadata:
      labels:
        name: rmit-store-client-pod
        app: rmit-store-client
        version: stable
    spec:
      containers:
        - name: rmit-store-client-pod
          image: locoioioi/cosc2767-rmit-store-client:stable
          imagePullPolicy: Always
          ports:
            - containerPort: 80
          env:
            - name: API_URL
              value: /api
      nodeSelector:
        eks.amazonaws.com/nodegroup: client-worker-group
---
apiVersion: v1
kind: Service
metadata:
  name: rmit-store-client-service
  labels:
    name: rmit-store-client-service
    app: rmit-store-client
spec:
  ports:
```

Figure 48. Stable version deployment configuration file

In the stable version deployment configuration, This YAML configuration defines the deployment and service for the stable version of the rmit-store-client application, supporting canary deployment. The deployment, named 'rmit-store-client-pod', creates **2** replicas using the stable container image 'locoioloi/cosc2767-rmit-store-client:stable' from DockerHub, with the imagePullPolicy set to 'Always', ensuring the latest image is pulled during a 'kubectl rollout restart'. Pods are assigned to the 'client-worker-group' node group for efficient workload distribution when scaling vertically or horizontally and expose port '80', with an environment variable 'API_URL' set to '/api'.

The associated service, 'rmit-store-client-service', uses a 'LoadBalancer' to route traffic from the LoadBalancer IP to the pods on port '80', enabling balanced external access. This setup facilitates smooth updates, efficient traffic routing, and scalability during deployments.

Canary Version Deployment

cosc2767-assignment-2-group-2024c-pall - client-canary-deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rmit-store-client-canary
  labels:
    name: rmit-store-client-canary
    app: rmit-store-client
    version: stable
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rmit-store-client
      version: stable
  template:
    metadata:
      labels:
        name: rmit-store-client-canary
        app: rmit-store-client
        version: stable
    spec:
      containers:
        - name: rmit-store-client-canary
          image: locoioioi/cosc2767-rmit-store-client:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 80
          env:
            - name: API_URL
              value: /api
      nodeSelector:
        eks.amazonaws.com/nodegroup: client-worker-group
---
apiVersion: v1
kind: Service
metadata:
  name: rmit-store-client-service
  labels:
    name: rmit-store-client-service
    app: rmit-store-client
spec:
  ports:
    - port: 80
      targetPort: 80
  type: LoadBalancer
  selector:
    app: rmit-store-client


```

Figure 49. Canary version deployment configuration file

For the latest deployment, the canary YAML configuration defines the deployment and service for the canary version of the rmit-store-client application, supporting canary deployment. The deployment, named 'rmit-store-client-canary', creates one replica using the latest container image 'locoioloi/cosc2767-rmit-store-client:latest' from DockerHub, with the imagePullPolicy set to 'Always', ensuring the latest image is pulled during a 'kubectl rollout restart'. The pod is assigned to the 'client-worker-group' node group to optimize workload distribution and exposes port '80', with an environment variable 'API_URL' set to '/api'. The associated service, 'rmit-store-client-service', uses a 'LoadBalancer' to route traffic from the LoadBalancer IP to the pods on port '80', enabling balanced external access. This setup facilitates testing the latest version of the application while directing minimal traffic to the canary deployment, ensuring reliability during deployments.

Stable & Canary Version Traffic Distribution

As both of the stable and canary deployment: ‘rmit-store-client-pod’, ‘rmit-store-client-canary’ are both attached to the ‘rmit-store-client-service’ under the Load Balancer. When access the Load Balancer IP, the weight of distribution of 2 versions are calculate as below:

- **n1 = 2**: Number of stable pod replicas for ‘rmit-store-client-pod’
- **n2 = 1**: Number of canary pod replicas for ‘rmit-store-client-canary’

$$\boldsymbol{Traffic\ Distribution\ to\ Stable\ Version\ Deployment} = \frac{n1}{n1 + n2} \sim 66\%$$

$$\boldsymbol{Traffic\ Distribution\ to\ Canary\ Version\ Deployment} \sim 33\%$$

Based on this mechanism of Kubernetes traffic splitting, to make the canary version accessible to broader user audiences, the command ‘kubectl scale deployment [deployment_name] –replicas=[number]’ are used to either scale up the number of canary pod or scale down the stable pod replicas.

V. Application Flow (Diagram)

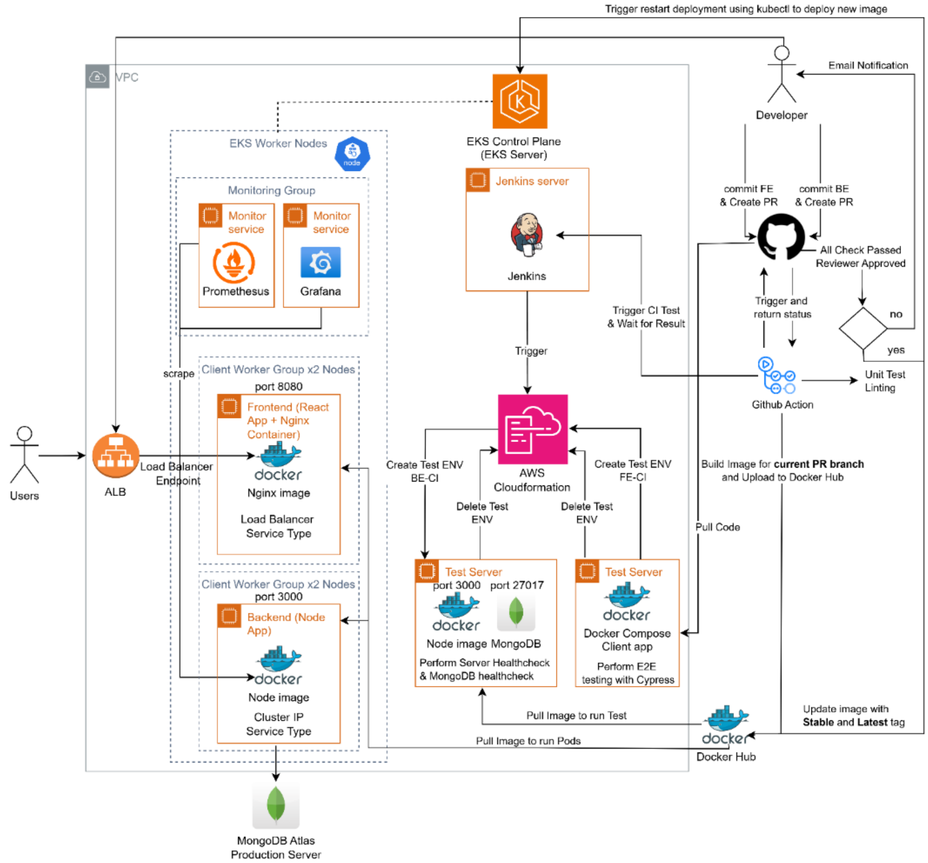


Figure 50. Application Flow Diagram

1. **GitHub Pull Request Merge to Main:** Code commit and push directly to Main branch is blocked, so the Developer will need to commit the code change to a feature branch, and create a Pull Request (PR) to merge that branch into Main branch. The PR will trigger a series of tests to ensure correctness of the code and require approval from other contributors to be able to merge.
2. **GitHub Action:** Depending on the code changes, a series of GitHub Action workflow will be triggered.
   Unit Test and Integration Test will always be triggered for any changes to both server or client code, and required to be pass, if test cases is not provided, this test will skip with success.
   Linting Check will also be triggered for any changes, lint check is not required to be pass in order to be able to merge, and serve as a recommendation for better coding styles and convention.
   a. Back-end changes (/server/**): This will trigger jenkins-ci-pr.yaml workflow, which will create an image docker and a test environment for the current PR branch.
   b. Front-end changes (/client/**): This will trigger jenkins-ci-fe.yaml workflow, which will create a test environment and conduct end-to-end testing for client.

   After a PR is merged into Main, there will also be a workflow to build and update docker image to the newest tag in Docker Hub, then trigger Kubernetes deployment on the EKS server.

3. **Jenkins:** GitHub Action will trigger the corresponding jobs be-ci or fe-ci to create the test environment, install the application and perform HealthCheck/testing.
4. **AWS CloudFormation:** Use the CloudFormation template stored in the GitHub repository to create a new EC2 instance every time a new test environment is created. The application will be installed and tested in this environment, then the instance will be terminated after the test.
5. **Docker Hub:** Docker Hub is used to store Docker Images built from the pipeline for testing and deployment to EKS.

6. **AWS Elastic Kubernetes Service (EKS):** Kubernetes cluster used to deploy the application pulled from Docker Hub, and the monitoring service. This cluster have 3 groups:
    a. Client Worker Group
    b. Server Worker Group
    c. Monitoring Group
7. **MongoDB Atlas:** The cloud-hosted MongoDB is used for production environment for the application to ensure its isolation with the test environment, and reduced management effort needed to connect to the backend running in EKS.
8. **Prometheus & Grafana:** Prometheus will scrape data and hardware usage information from the Kubernetes cluster and the application. Grafana will use that as the data source and visualize them into a dashboard.
9. **Application Load Balancer (ALB):** This is the endpoint for the user to access and interact with the application. This load balancer will ensure fair distribution and spread the load across nodes in the Kubernetes Cluster, it also serves the purpose of canary deployment type.
10. 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.

VI. Known Bugs/ Problems

- **Manual Canary Deployment Visibility**: The current process for canary deployment requires manually scaling traffic to new version pods to evaluate the updated deployment. This manual intervention increases operational overhead and slows down the deployment process. It also makes it more challenging to compare the performance and reliability of the new version against the stable one in real-time.
- **Vertical Scaling Monitor**: There is a lack of real-time metrics to monitor the computational resources utilized by containers. Without these metrics, the system is unable to identify and respond to resource bottlenecks or overutilization effectively. This limitation prevents proactive management of container performance and could lead to service degradation during high resource usage or unexpected traffic spikes.

VII. Pipeline Self-Evaluation

Completing this project provided our team with hands-on experience in building a secure, scalable, and automated DevOps pipeline while adhering to industry-standard security practices. Guided by insights from industry guest speakers, we implemented a private GitHub repository to safeguard source code, enforced approval-based pull requests to protect the main branch, and used environment variables to securely manage sensitive information. These measures ensured robust access control, reduced vulnerabilities, and adhered to best practices.

Our team believes this solution is the most appropriate because it combines automation, scalability, and reliability to address integration and deployment challenges. Automated CI/CD workflows with Jenkins ensure early error detection and consistent delivery, while Docker and Kubernetes provide portability, reliability, and efficient orchestration. AWS services like CloudFormation and Elastic Load Balancing enable scalable, consistent infrastructure management.

Automation
The pipeline is designed to automate key stages of software delivery, from code integration to testing and deployment. GitHub Actions, in conjunction with Jenkins, triggers workflows immediately upon code changes or pull requests. AWS CloudFormation provisions infrastructure automatically, while Kubernetes orchestrates container deployments, making the release process less dependent on human intervention. This extensive automation minimizes manual errors, accelerates feedback loops, and ensures consistency throughout the software lifecycle.

Efficiency
By automating repetitive tasks, the pipeline optimizes resource usage and developer time. Changes can be tested and validated shortly after they are committed, enabling faster feedback and reducing the time spent on identifying and resolving issues. Containerization with Docker further speeds up deployments by standardizing environments, and infrastructure as code via AWS CloudFormation prevents the overhead associated with manual environment setup or teardown.

Reliability

Reliability is significantly increased through automated testing and monitoring at multiple levels. Jenkins ensures each build passes standardized tests, while Kubernetes and Nginx support zero-downtime deployments and provide resilience through rolling updates. Prometheus and Grafana supply real-time system metrics and alerts, allowing teams to identify and address anomalies before they impact end-users. Rollback capabilities in Kubernetes enable a swift return to a stable version if a deployment encounters critical issues, further enhancing reliability.

Scalability

Scalability is a key focus of this setup. Elastic Load Balancing distributes incoming traffic across multiple servers or containers, mitigating performance bottlenecks. Kubernetes automatically scales containers based on metrics such as CPU and memory usage, while MongoDB Atlas accommodates growing data needs without manual database administration. These features collectively ensure that the RMIT E-commerce platform can handle increased traffic and user demands with minimal operational overhead.

Repeatability

The use of containerization and infrastructure as code promotes a highly repeatable deployment process. Docker images guarantee that each environment runs an identical stack, reducing the likelihood of environment-specific defects. With AWS CloudFormation templates, it is straightforward to replicate production-like environments for development and testing. This uniformity eliminates inconsistencies between environments and makes the entire pipeline—from code commit to deployment—easily reproducible for each new release cycle.

VIII. Conclusion

Completing this project has given our team valuable hands-on experience in designing and implementing a comprehensive DevOps pipeline using modern technologies. The project provided an in-depth understanding of the importance of automation, scalability, and reliability in delivering robust software solutions. Our pipeline automates critical processes such as code integration, build, testing, and deployment, significantly reducing manual intervention and potential errors. Through continuous integration (CI), every new pull request triggers an automated build and test process in Jenkins, ensuring early detection of errors and maintaining synchronization across the repository. Continuous delivery (CD) further enables seamless deployment to testing and production environments, improving release management flexibility.

Leveraging AWS CloudFormation for automated test environment management, the pipeline ensures consistency between testing and production environments while maintaining cost efficiency by dynamically tearing down resources when not in use. Docker simplifies the deployment process by packaging applications into lightweight containers, ensuring consistent performance across development, testing, and production. Kubernetes orchestrates these containerized applications, enabling features like self-healing, canary deployments, and rapid rollbacks, enhancing agility and minimizing downtime. Traffic scalability is achieved with Elastic Load Balancing, which dynamically adjusts to maintain optimal performance under varying traffic loads.

Additionally, the integration of Prometheus and Grafana ensures real-time monitoring and alerting, allowing the team to proactively address anomalies and performance issues before they impact users. The hands-on use of these technologies—Jenkins, Docker, AWS, Kubernetes, Prometheus, and Grafana—has provided practical DevOps knowledge and reinforced the importance of infrastructure-as-code, containerization, and monitoring in modern software development. Beyond technical skills, the project also honed our critical thinking, problem-solving, and adaptability, as we navigated challenges and developed efficient automation solutions. This project has greatly enriched our understanding of building scalable, reliable, and automated systems, preparing us for future challenges in the field of DevOps.

IX. Project Responsibility

Name	Role	Task	Contribution
Do Tung Lam	EKS Engineer, Data Engineer	Set up EKS, define nodes, pods, service, set up canary deployment strategies, set up Prometheus, Grafana for visualization	20%

Hoang Thai Phuc	Docker Engineer, Data Engineer	Set up Dockerfile, Docker compose for the application, set up Prometheus, Grafana for visualization	20%
Nguyen Quoc An	Tester, Jenkins Engineer	Creating and executing test cases, ensuring the availability of the website, Set up Jenkins Jobs	20%
Truong Quang Bao Loc	Tester, GitHub actions Engineer	Creating and executing test cases, ensuring the availability of the website, Set up GitHub actions Job	20%

X. Reference

[1] GitHub, "GitHub," *Wikipedia*, Dec. 2023. [Online]. Available: https://en.wikipedia.org/wiki/GitHub. [Accessed: 17-Jan-2025].

[2] GitHub, "GitHub Actions Documentation," *GitHub Docs*. [Online]. Available: https://docs.github.com/en/actions. [Accessed: 17-Jan-2025].

[3] Jenkins, "Jenkins User Documentation," *Jenkins.io*. [Online]. Available: https://www.jenkins.io/doc/. [Accessed: 17-Jan-2025].

[4] Amazon Web Services, "AWS CloudFormation User Guide," *Amazon AWS*, 2025. [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html. [Accessed: 17-Jan-2025].

[5] Amazon Web Services, "Docker on AWS," *Amazon AWS*, 2025. [Online]. Available: https://aws.amazon.com/docker/?nc1=h_ls. [Accessed: 17-Jan-2025].

[6] MongoDB, "MongoDB Atlas Tutorial," *MongoDB Resources*. [Online]. Available: https://www.mongodb.com/resources/products/platform/mongodb-atlas-tutorial. [Accessed: 17-Jan-2025].

[7] NGINX, "Welcome to NGINX," *NGINX.org*. [Online]. Available: https://nginx.org/en/. [Accessed: 17-Jan-2025].

[8] Amazon Web Services, "Elastic Load Balancing," *Amazon AWS*, 2025. [Online]. Available: https://aws.amazon.com/elasticloadbalancing/. [Accessed: 17-Jan-2025].

[9] Kubernetes, "Kubernetes Documentation," *Kubernetes.io*. [Online]. Available: https://kubernetes.io/. [Accessed: 17-Jan-2025].

[10] Prometheus, "Prometheus Monitoring System," *Prometheus.io*. [Online]. Available: https://prometheus.io/. [Accessed: 17-Jan-2025].

[11] Grafana Labs, "Grafana – The open observability platform," *Grafana.com*. [Online]. Available: https://grafana.com/. [Accessed: 17-Jan-2025].

XI. Appendix