National College of Ireland

Configuration Manual

MSc Research Project
MSc Cloud Computing

Rachana Poonacha
Student ID: 22217029

School of Computing
National College of Ireland

Supervisor: Jitendra Kumar Sharma

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Rachana Kottangada Poonacha

Student ID: 22217029

Programme: MSc Cloud Computing **Year:** 2023 – 24

Module: MSCCLOUD Research Project

Lecturer: Jitendra Kumar Sharma

Submission Due Date: 16 – 09 -2024

Project Title: Integration of Security Vulnerability Tools and Kubernetes Deployment to Obtain an Enhanced CI/CD Pipeline for A Blockchain-based Decentralized Application (DApp)

Word Count: ….2625…… **Page Count:** ……18……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: K P Rachana

Date: 16 – 09 – 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Rachana Kottangada Poonacha
Student ID: 22217029

The configuration manual outlines the steps in detail that are required to setup the research environment for implementing the proposed solution, which was done in 3 phases. First phase, concentrated on the development and installation of "The Yoga Studio" Blockchain-based DApp. Second phase included the setup of Cloud environment required for the deployment of the application into AWS Elastic Kubernetes Service (AWS EKS) and pushing the Docker Images to AWS Elastic Container Registry (AWS ECR). The third and the last stage is the development of CI/CD scripts to integrate "Lint", "SonarCloud scan" and "Deploy" stages.

1 Development and Installation of "Yoga Studio" DApp

- "The Yoga Studio" DApp is a Ethereum Blockchain-based Decentralized application which contains a set of yoga courses and booking time slots for users to purchase and schedule appointments by making the payment using Ethereum.

- The application has been developed in Node.js programming language and the IDE used for the development purpose is Visual Code Studio (VSC). To setup the application, initially, the Nodejs installer is downloaded and run from Node.js website (Nodejs, 2024). The installation can be tested by running the commands "npm -v" or "node -v".

- Express framework is installed using the following command (Expressjs, 2024):
  "npm install -g express-generator"

- Create an application using the following command:
  "npx express-generator -view=ejs blockchain-latest-v2

- EJS (Embedded Java Script) and the express framework are initialized in app.js as shown in Figure 1.

```
var createError = require('http-errors');
var express = require('express');
app.set('view engine', 'ejs');
```

Figure 1: EJS and express frameworks initialized

- The application routes are configured at "routes/index.js" as shown in Figure 2.

```
router.get('/login', function(req, res, next) {
  res.render('login', {page_title: "Login"});
});
```

Figure 2: Configuration of different routes in the application

- The templating engine of EJS is used to configure reusable components using the "footer.ejs" and "header.ejs" files in "views/partials". The have included EJS syntax along with HTML content. As shown in Figure 3, these scripts are added into every single page inside "views/*.ejs" using the syntax: <%- include partials/header.ejs %> and <%- include partials/footer.ejs %>.

```
<%- include partials/header.ejs %>


<div class="intro-section">
  <div class="">
      <div class="header">
          <%- include partials/navbar.ejs %>
      </div>
  </div>
  <div class="container">
      <div class="row">
          <div class="col-md-12 col-sm-12 text-center">
              <div class="intro-caption">
                  <h1 class="intro-title">The Best Yoga Studio</h1>
```

Figure 3: "views/partials" in index.ejs

- Bootstrap has been used for the frontend design and the associated packages are download from the official website of Bootstrap (Getbootstrap, 2024). As seen in Figure 4, they contain pre-defined HTML, CSS and JavaScript class names that are copied to the node's "public/js" and "public/css" locations.

```
<!-- Bootstrap -->
<link href="css/bootstrap.min.css" rel="stylesheet">
```

Figure 4: Libraries included for Bootstrap in public/css

- As shown in Figure 5, "Web3.js" library is configured in "views/partials/footer.ejs" and this enables interaction with the underlying Ethereum network. "Ethers.js" is also a library that is included for communication with the Ethereum network and it includes a number of tool sets that can be utilized to work with Ethereum.

```
<!-- Connect Remix and perform transactions -->
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/web3/1.2.7-rc.0/web3.min.js"></script>
<script src="https://cdn.ethers.io/lib/ethers-5.6.4.umd.min.js" type="application/javascript"></script>
```

Figure 5: Libraries such as "web3:" and "ethers.js" for interaction with Ethereum network

- User login is facilitated via the MetaMask wallet which is made available by "Ethers.js" library. The function written for MetaMask Login is inside "views/login.js" as shown in Figure 6.

```
async function web3_metamask_login() {
    // Check first if the user has the MetaMask installed
    if ( web3_check_metamask() ) {
        console.log('Initate Login Process');

        // Get the Ethereum provider
        const provider = new ethers.providers.Web3Provider(window.ethereum);
        // Get Ethereum accounts
        await provider.send("eth_requestAccounts", []);
        console.log("Connected!!");
        // Get the User Ethereum address
        const address = await provider.getSigner().getAddress();
        const existingData =  localStorage.getItem("metamask-address");
```

Figure 6: Function for MetaMask login

- For the purpose of development, test ethers have been used which are provided by "Ethereum Sepolia Faucet" via "The Alchemy login" (Alchemy, 2024)

- User will further communicate through the MetaMask Wallet for transactions related to "Payment" or "Making an appointment". In both scenarios, users will interact through MetaMask whose connections are established due to the configuration code written inside "public/js/contract.js". The same is shown in Figure 7. MetaMask can be downloaded from the MetaMask website (MetaMask, 2024) to install the plugin in your browser. For the purpose of the research, I have used MetaMask in Firefox browser.

```
const connectMetamask = async () => {
    let account;
    console.log(window.ethereum);
    if(window.ethereum !== "undefined") {
        const accounts = await ethereum.request({method: "eth_requestAccounts"});
        account = accounts[0];
        document.getElementById("userArea").innerHTML = `User Account: ${account}`;
    }
    return account
}
```

Figure 7: Function to connect to MetaMask

- The two functionalities such as "Payments" and "Book an appointment" are enabled through "Smart Contracts" written within "remix/payment.sol" and "remix/appointment.sol". The two programs are compiled within Remix IDE which then generates an AIB Code. Once compiled, they are deployed into the Ethereum

Test network (sepolia) which generates a contract address. This contract address is used to invoke the “Smart Contract” functionalities whenever the user “makes a payment” or “books an appointment” using MetaMask. The details of both the transactions are permanently lodged into the Blockchain network. Figure 8 represents the ABI code and contract address of “Payment.sol” and “Appointment.sol” smart contracts. Figure 9 represents the deployment of “appointment.sol” smart contract in REMIX IDE.

```
// payment_abi = '[{"inputs":[],"name":"deposit","outputs":[],"stateMutability":"payable","type":"function"},{"inputs":[],"name":"ge
payment_abi = '[{"inputs":[],"name": "deposit","outputs":[],"stateMutability": "payable","type": "function"},{"inputs": [],"name": "
// payment_address = '0x65CE52604D3DDC7308DB58c17E09A2c2F6b3471E'
payment_address = '0xe9b6f392d55e65e3609d447c9a517614e40e0e52'
// appointment_abi = '[{"inputs":[],"name":"getAddress","outputs":[{"internalType":"address","name":"","type":"address"}],"stateMuta
appointment_abi = '[{"inputs":[],"name": "getAddress","outputs":[{"internalType": "address","name": "","type": "address"}],"stateMut
// appointment_address = '0x8a4297836f985521f8fe42F9E947673871945F6b'
appointment_address = '0x4552a0051284c30ccb4a48a00a8cd63cf3031989'
```

Figure 8: ABI and Contract addresses for “payment.sol” and “appointment.sol”

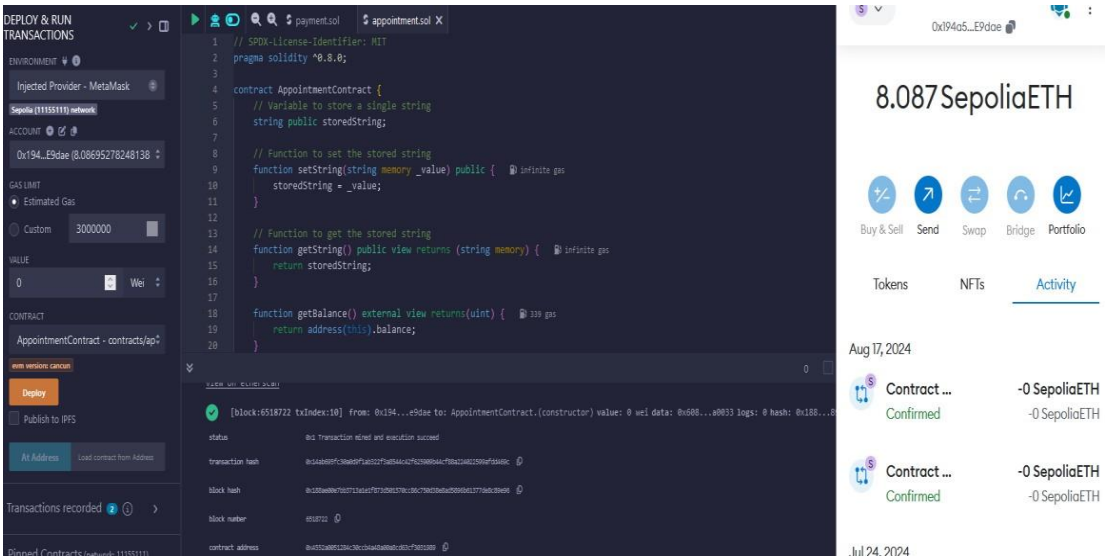

Figure 9: Deployment of Smart Contract through Remix IDE

- Every transaction conducted in Ethereum Blockchain including the deployment of Smart Contracts includes a certain price that is nominal called the “Gas” price through “Ethers”. “Gas” is the smallest unit for the measurement of the computational effort that is required to deploy “Smart Contracts” into the Ethereum Blockchain network.

2 Setup of Cloud environment: AWS EKS, AWS ECR and IAM

AWS Elastic Kubernetes Service (AWS EKS), AWS Elastic Container Registry (AWS ERC) and Identity and Access Management (IAM) are required to basically run the application, store Docker images and enable user access controls respectively.

2.1 Creating AWS EKS Cluster

- The **AWS EKS Cluster** needs to be setup to deploy the application through Kubernetes.

- The official documentation of AWS Elastic Kubernetes Service outlines the steps in detail to setup the cluster as per the project requirements (Amazon Web Services (AWS), 2024)

- Figure 10 represents the configuration details of the cluster that was used to conduct the experiment along with the Kubernetes version utilized.

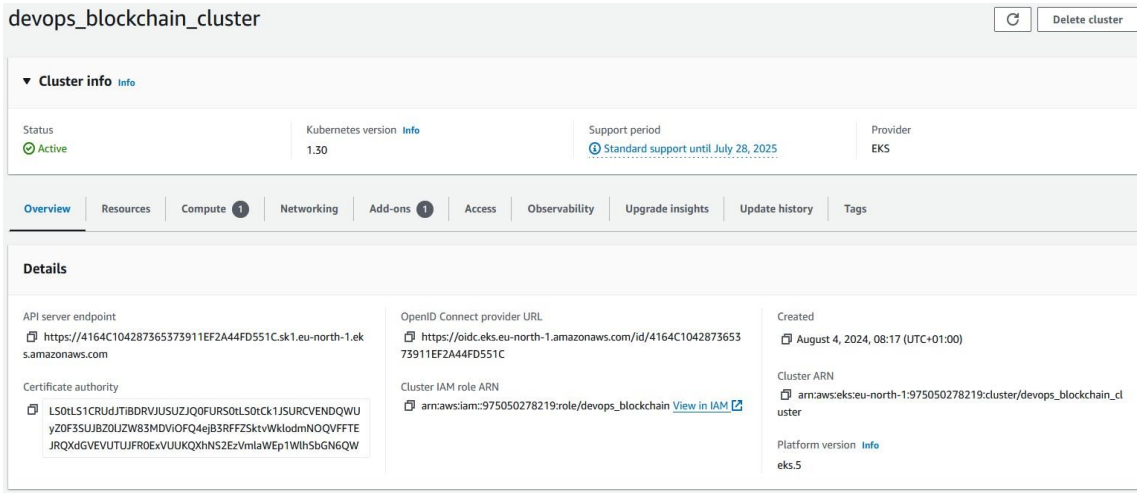

Figure 10: AWS EKS Cluster

- The workload deployed into the Kubernetes cluster include Pods, ReplicaSets and Deployments as shown in Figure 11, 12 and 13 respectively.

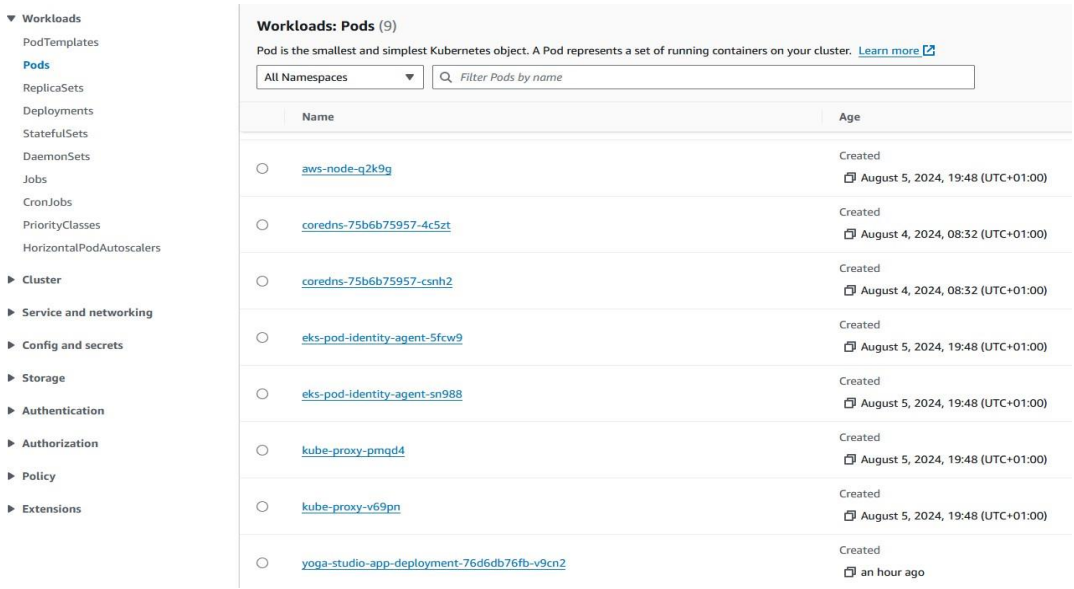

Figure 11: Pods in the EKS Cluster : “yoga-studio-app-deployment- 76d6db76fb-v9cn2”

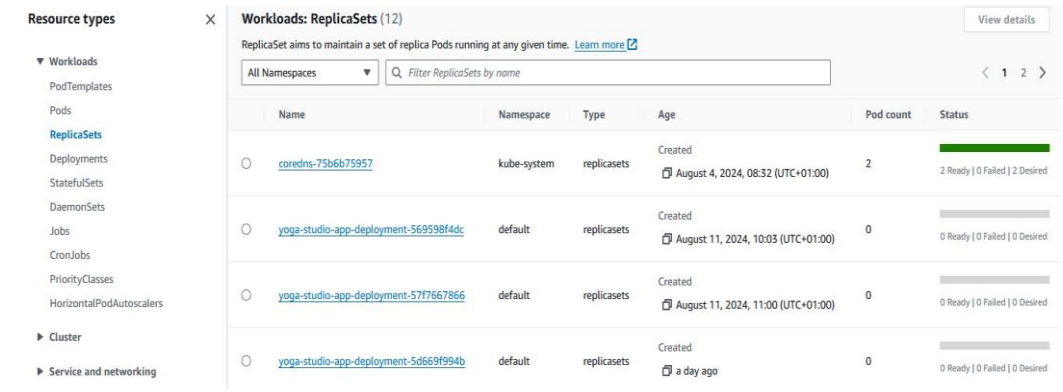

Figure 12: ReplicaSets in the EKS Cluster

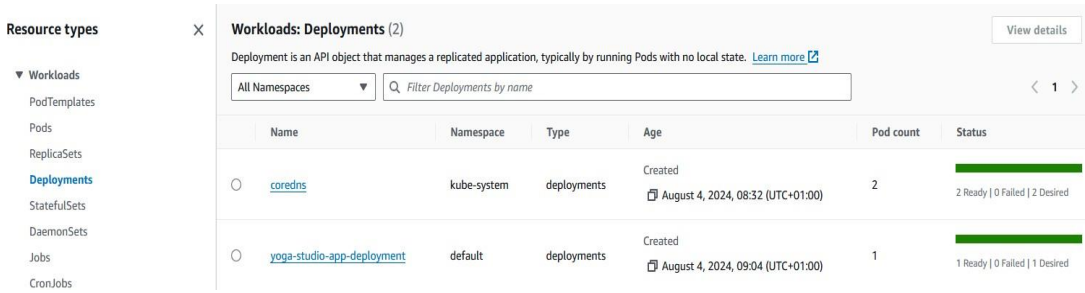

Figure 13: Deployments in the EKS Cluster

- The EKS Cluster for "The Yoga Studio" DApp deployment is configured with 2 nodes as shown in Figure 14. The application is running within these nodes.

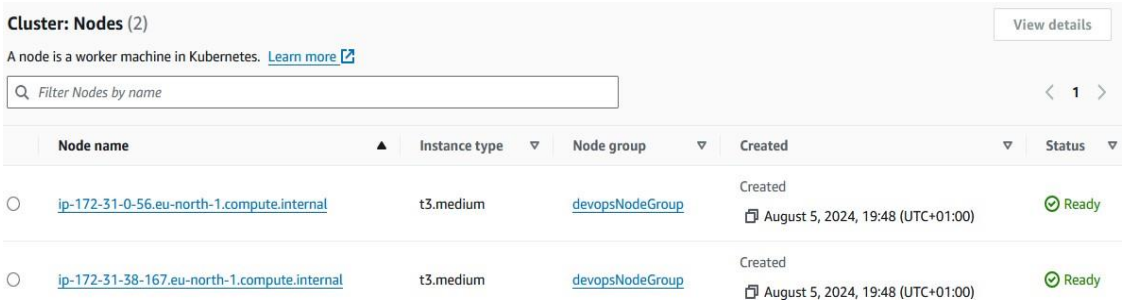

Figure 14: Nodes that form the EKS Cluster for DApp

- The application is accessible through the "Services" workload and is configured of the type "LoadBalancer" as shown in Figure 15.

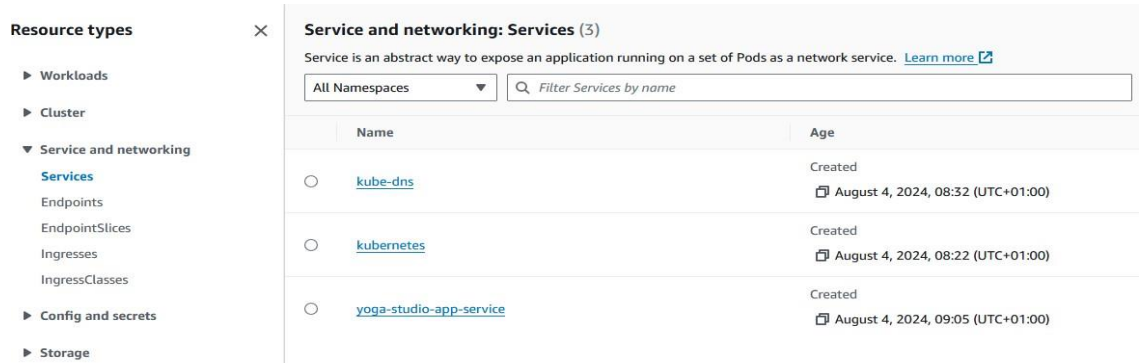

Figure 15: Services configured to access the application

- A “Classic” Load Balancer is provided by the EKS service and the configuration of the same is as shown in Figure 16. The DNS of the Load Balancer can be used to access the application: **http://a492faa991b2d4b27bd04a8eabcd6b3b-209784938.eu-north-1.elb.amazonaws.com/**

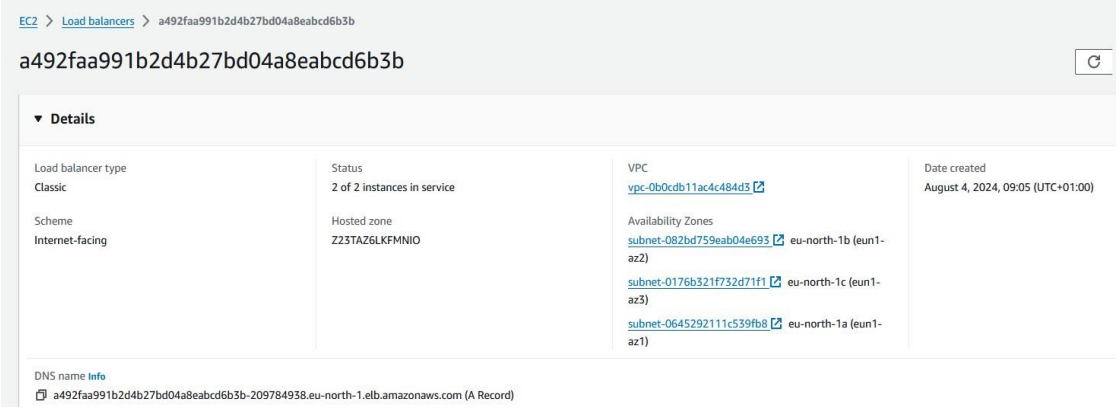

Figure 16: Load Balancer

- In Kubernetes, a “Node” is nothing but a worker machine that is used to run a containerized application which can be a physical machine or a virtual machine. A “Pod”, in Kubernetes, is defined as the smallest object of Kubernetes that runs a single instance of a process, usually encapsulates one or more containers. The “Pod” configuration is submitted to the Kubernetes API server.

- The application is deployed into the AWS EKS Cluster through Kubernetes manifests such as deployment and service yamls. A “deployment.yml” manifest file is a Kubernetes configuration file that outlines details about how the application needs to be deployed, number of replicas of the application that are required to be maintained, and how the applications should be updated. The “yoga-app-deployment.yaml” is the manifest file configured for the deployment of the application into AWS EKS cluster. The config file is as shown in Figure 17.

- A service.yaml file in Kubernetes is used to expose a set of pods as a service on the network in order to enable communication between different applications within the cluster or from outside the network. Figure 18 is the “yoga-app-service.yaml” written to run the DApp in AWS EKS.

2.2 Creating AWS Docker Container Registry

- AWS Elastic Container Registry (AWS ECR) is a managed service that hosts Docker images and other artifacts providing a platform to reliably deploy the applications (Docs AWS ECR, 2024).
- Figure 19 shows the list of docker images that were pushed to AWS ECR during the application deployment via CI/CD pipeline and stored inside “devops” repository.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: yoga-studio-app-deployment
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-security-groups: "sg-061d0cec3ed96cd48"
  labels:
    app: yoga-studio-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: yoga-studio-app
  template:
    metadata:
      labels:
        app: yoga-studio-app
    spec:
      automountServiceAccountToken: false
      containers:
      - name: yoga-studio-app-container
        image: DOCKER_IMAGE
        ports:
        - containerPort: 3000
        env:
        - name: EXAMPLE_ENV_VAR
          value: "example-value"
        resources:
          requests:
            memory: "128Mi"
          limits:
            memory: "256Mi"
```

Figure 17: yoga-app-deployment.yaml

Blockchain_latest_v2 / deployment / **yoga-app-service.yaml**

RachanaPoonachaNCI Updated deployment folder for app deployment into Kuberenetes

Code | Blame | 14 lines (14 loc) · 237 Bytes

```yaml
1   apiVersion: v1
2   kind: Service
3   metadata:
4     name: yoga-studio-app-service
5     labels:
6       app: yoga-studio-app
7   spec:
8     selector:
9       app: yoga-studio-app
10    ports:
11      - protocol: TCP
12        port: 80
13        targetPort: 3000
14    type: LoadBalancer
```

Figure 18: yoga-app-service.yaml

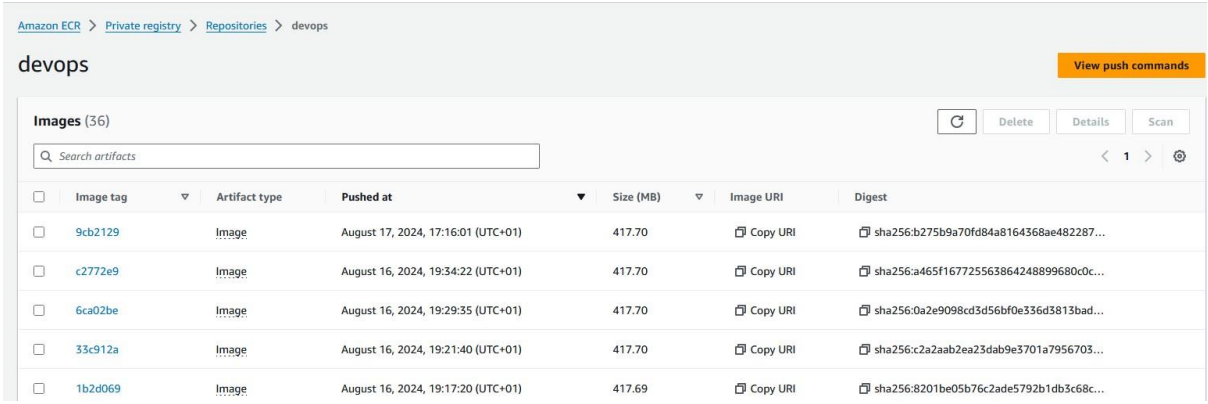

Figure 19: Docker images in “AWS ECR”

2.3 Setting up of IAM roles and policies

- The next step is to create a user and assign policies related to AWS EKS and to provide “Administrator access” to the user so that it can communicate between different services within AWS.
- The official documentation of IAM provides detailed instructions to configure IAM roles and policies as per the project specifications (AWS IAM, 2024).
- Figure 20 shows the IAM configuration done as part of this research implementation.

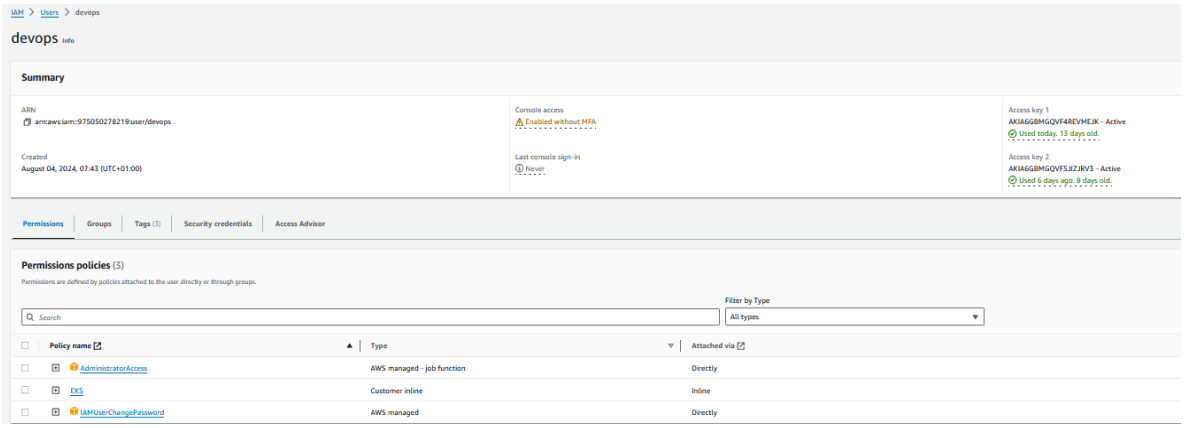

Figure 20: IAM configuration for user “devops”

3 Creation of CI/CD pipeline to integrate with “The Yoga Studio” DApp

This part of the configuration manual provides step-by-step instructions on how the CI/CD setup has been configured for implementing the solution for the research question. The CI/CD pipeline is designed to automate the process of building, security detection, code quality analysis and deployment of the application features and fixes, into the AWS EKS cluster.

3.1 Setting up of “Github actions”

- To begin with, the source code of the DApp is stored in the “Github repository”. The CI/CD pipeline is then configured in “Github actions”.
- Initially, the “Github actions” is setup by creating the “Github workflow”. The “Github workflow” is created by navigating to the root folder in your repository and then create a directory named “.github/workflows/”.
- Create a file in this directory called as “main.yml” as shown in Figure 21.
- Define the different stages of the pipeline in the workflow under “jobs” parameter.
- The workflow will be automatically triggered if there are any new changes committed into the repository.
- The environment variables pertaining to the Cloud environment are configured under “env” tag where you specify the “ECR Repository name” where the Docker images of the application build are pushed, “AWS EKS Cluster name” where the application is running, and the “AWS Region name” where the ECR repo and the EKS Cluster has been created as shown in Figure 21.

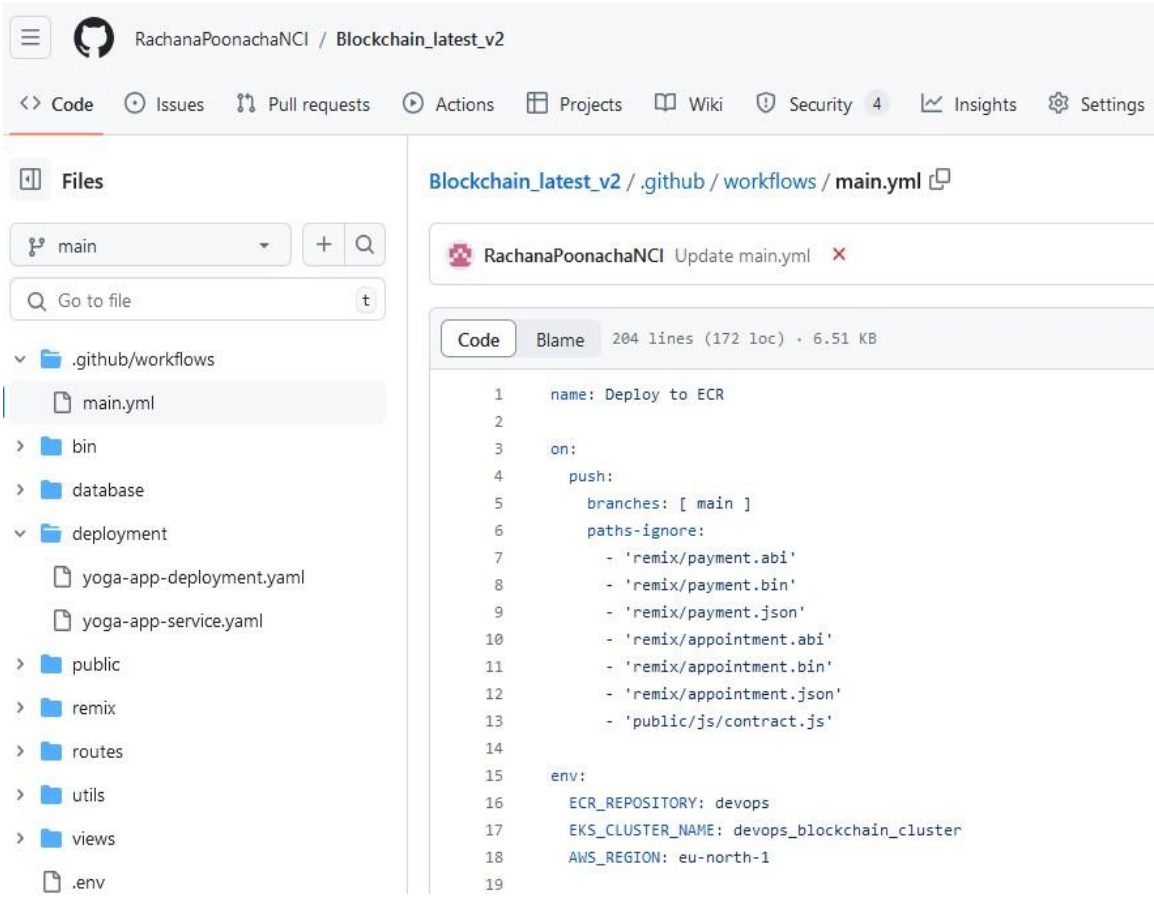

Figure 21: “Github workflow” main.yml file

3.2 Configuration of “job” for “Linting” stage using ESLint

- Figure 22 represents the configuration done in the ".github/workflows/main.yml" for the "Lint" stage of the pipeline.
- ESLint is used for detecting errors in the code which aims to improve the overall quality of the code and eventually the application.
- ESLint dependencies are installed from "package.json" file which is also present in the root folder of the repository.
- The name of the "job" in the "Lint" stage is called "lint".
- The job will run on "ubuntu-latest" OS.
- Execute "actions/checkout@v3" to fetch the entire repository to ensure that the source code is available to run the workflow.
- Define the installation of the "Node.js" runtime environment.
- Define commands to run the "ESLint" to check JavaScript files such as "utils" and "routes" in the source code.
- Run the pipeline to check for "Linting errors". If any errors detected, needs to be fixed to re-run the pipeline using the "actions" button available within the specific Github repository.

```yaml
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3
        with:
          fetch-depth: 0

      - name: Install node
        uses: actions/setup-node@v4
        with:
          node-version: 18

      - name : Run ES lint on nodeJs files
        run: |
          npm install
          npx eslint utils/*.js
          # npx eslint routes/*.js
```

Figure 22: LINT Stage configuration in "main.yml"

3.3 Configuration of "job" for "Linting" stage using ESLint

- Initially, generate an access token in SonarCloud under the "Sonar Cloud" security section. This is required to authenticate SonarCloud to perform code analysis when Github workflow is triggered.
- GITHUB_TOKEN is configured to authenticate "Github actions" with Github.
- The access token is added as an environment variable under the "Security/secrets and variables/actions" section of the repository as shown in Figure 23.

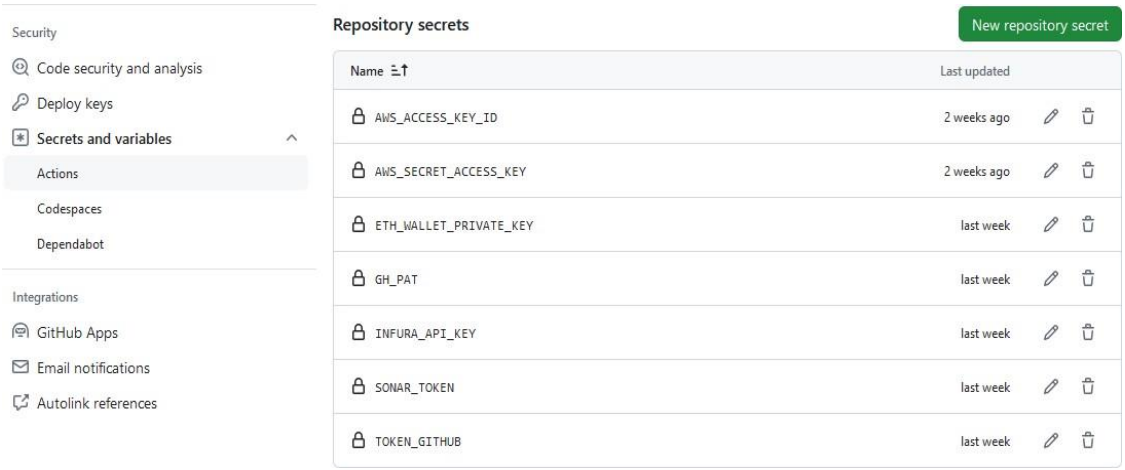

Figure 23: Repository secrets

- The "job" is configured to analyse security related issues of "HIGH" and "MEDIUM" severity. The pipeline will fail in case of such security issues are identified. The code configured for the same is as shown in Figure 24.
- The issues can be viewed in SonarCloud Web UI and a detailed report of the analysis will be present on the project dashboard.

```yaml
sonarcloud:
  name: SonarCloud Scan
  runs-on: ubuntu-latest
  needs : lint

  steps:
  - name: Checkout code
    uses: actions/checkout@v3
    with:
      fetch-depth: 0  # Shallow clones should be disabled for better relevancy of analysis

  - name: SonarCloud Scan
    uses: SonarSource/sonarcloud-github-action@master
    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information, if any
      SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
    with:
      args: >
        -Dsonar.exclusions=**/deploy_contract.py,**/contract.js


  - name: Check for HIGH and MEDIUM security issues
    run: |
      response=$(curl --location 'https://sonarcloud.io/api/issues/search?projects=Blockchain_latest_v2' \
        --header 'Authorization: Bearer ${{ secrets.SONAR_API_KEY }}')

      high=$(echo "$response" | jq '[.issues[].impacts[] | select(.softwareQuality == "SECURITY" and .severity == "HIGH")] | length')
      medium=$(echo "$response" | jq '[.issues[].impacts[] | select(.softwareQuality == "SECURITY" and .severity == "MEDIUM")] | length')

      echo "HIGH: $high"
      echo "MEDIUM: $medium"

      if [ "$high" -gt 0 ] || [ "$medium" -gt 0 ]; then
      # if [ "$high" -gt 0 ]; then
        echo "Blocking deployment due to HIGH or MEDIUM security issues."
        exit 1
      fi
```

Figure 24: "SonarCloud scan" configuration in main.yml

3.4 Configuration of “job” for “Deploy to EKS” stage using Docker and Kubernetes

- AWS_ACCESS_KEY and AWS_SECRET_KEY are generated for the created “devops” user in IAM and the same is added into the “security” section of repository secrets. These are required to authenticate to AWS ECR and AWS EKS to push the docker images and deploy the application respectively.
- The “job” name for application “Build” and “Deployment” is configured under the name “Deploy to ECR” in main.yml file as shown in Figure 25.
- Initially, the job retrieves the SHA of the latest commit into the repository.
- The entire history of the code checkout is performed.
- Python environment is setup in the underlying VM in “Github actions”.
- The required dependencies to compile and deploy the “Smart Contracts” solidity code is installed.
- The job checks for any changes in the “Smart contracts” code. If any changes found, in “payment.sol” or “appointment.sol”, the code is re-compiled and deployed in REMIX IDE.
- AWS credentials are configured.
- The application “Build” is performed using docker build commands and the image is tagged appropriately and pushed to AWS ECR to keep track of all the latest and previous changes done in the code.
- Kubernetes manifests such as “yoga-app-deployment.yaml” and “deployment/yoga-app-service.yaml” files present in the “deployment” folder within the repository is executed to deploy the application into the nodes created in the AWS EKS cluster.

```
deploy:
  name: Deployment to ECR and EKS
  runs-on: ubuntu-latest
  needs: sonarcloud

  steps:
  - name: Set short git commit SHA
    id: commit
    uses: prompt/actions-commit-hash@v2

  - name: Checkout code
    uses: actions/checkout@v3
    with:
      fetch-depth: 0  # Shallow clones should be disabled for better relevancy of analysis

  - name: Set up Python
    uses: actions/setup-python@v4
    with:
      python-version: '3.x'

  - name: Install dependencies
    run: |
      sudo add-apt-repository ppa:ethereum/ethereum
      sudo apt-get update
      sudo apt-get install solc
      python -m pip install --upgrade pip
      pip install web3
      git config --global user.name "github-actions[bot]"
      git config --global user.email "github-actions[bot]@users.noreply.github.com"
```

```
- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v1
  with:
    aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
    aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    aws-region: ${{env.AWS_REGION}}

- name: Login to Amazon ECR
  id: login-ecr
  uses: aws-actions/amazon-ecr-login@v1

- name: Build, tag, and push image to Amazon ECR
  env:
    ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
    IMAGE_TAG: ${{ steps.commit.outputs.short }}
  run: |
    docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
    docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG

- name: Update kube config
  run: aws eks update-kubeconfig --name $EKS_CLUSTER_NAME --region $AWS_REGION

- name: Deploy to EKS
  env:
    ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
    IMAGE_TAG: ${{ steps.commit.outputs.short }}
  run: |
    kubectl version
    sed -i.bak "s|DOCKER_IMAGE|$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG|g" deployment/yoga-app-deployment.yaml
    kubectl apply -f deployment/yoga-app-deployment.yaml
    kubectl apply -f deployment/yoga-app-service.yaml
```

Figure 25: “Deploy to ECR” configuration in main.yml

- The pipeline is run to deploy the application using “Actions” button within the Github repository.

4 Conclusion

To conclude, this configuration manual has detailed all the required steps to setup a comprehensive CI/CD pipeline to integrate with “The Yoga Studio” DApp. This ensures a robust software development process for Blockchain applications with enhance security, availability and integrity.

References

Alchemy (2024), “*Ethereum sepolia faucet*”, Available at: https://www.alchemy.com/faucets/ethereum-sepolia (Accessed: 17 August 2024)

AWS IAM (2024), “*What is IAM*?”, Available at: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html (Accessed: 17 August 2024).

Amazon Web Services (AWS) (2024), “*Creating an Amazon EKS cluster*”, Available at: https://docs.aws.amazon.com/eks/latest/userguide/create-cluster.html (Accessed: 17 August 2024).

Docs AWS ECR (2024), "*Amazon Elastic Container Registry (ECR) Documentation*", Available at: https://docs.aws.amazon.com/ecr/ (Accessed: 17 August 2024)

Getbootstrap (2024), "*Build fast, responsive sites with Bootstrap*", Available at: https://getbootstrap.com/ (Accessed: 17 August 2024)

Expressjs (2024), "*Express*", Available at: https://expressjs.com/ (Accessed at: 17 August 2024).

MetaMask (2024), "*Install MetaMask for your browser*", Available at: https://metamask.io/download/ (Accessed: 17 August 2024)

Nodejs (2024), "*Run JavaScript everywhere*", Available at: https://nodejs.org/en/ (Accessed: 17 August 2024).