|  |  |
| --- | --- |
| **Github Repo** |  |

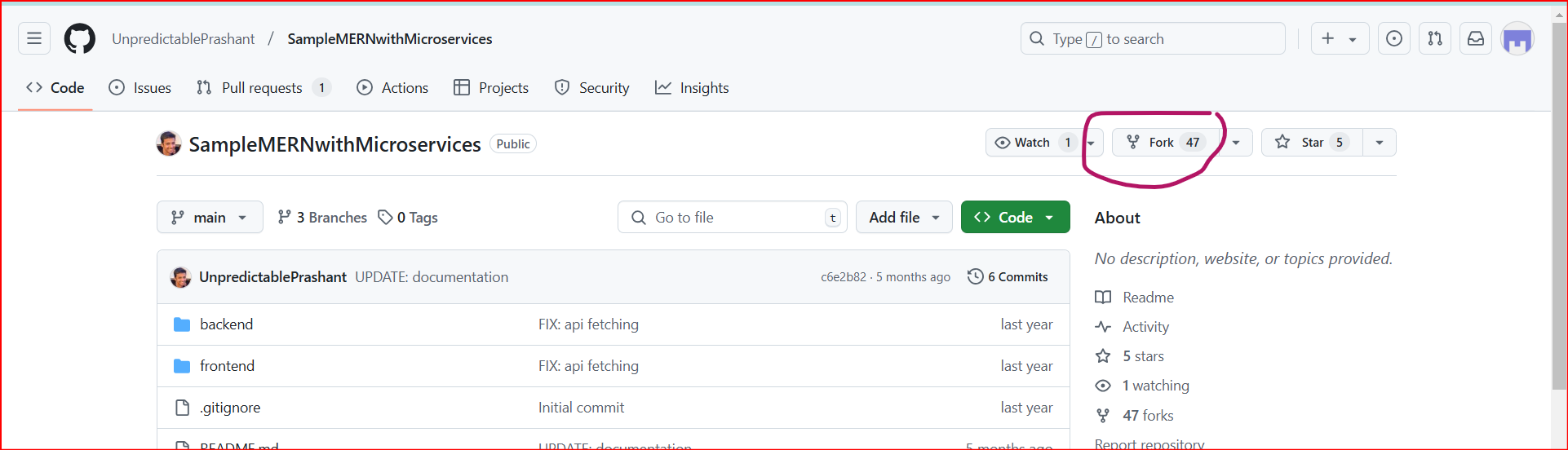
**Graded Project on Orchestration and Scsaling**

Project Link: <https://github.com/UnpredictablePrashant/SampleMERNwithMicroservices>

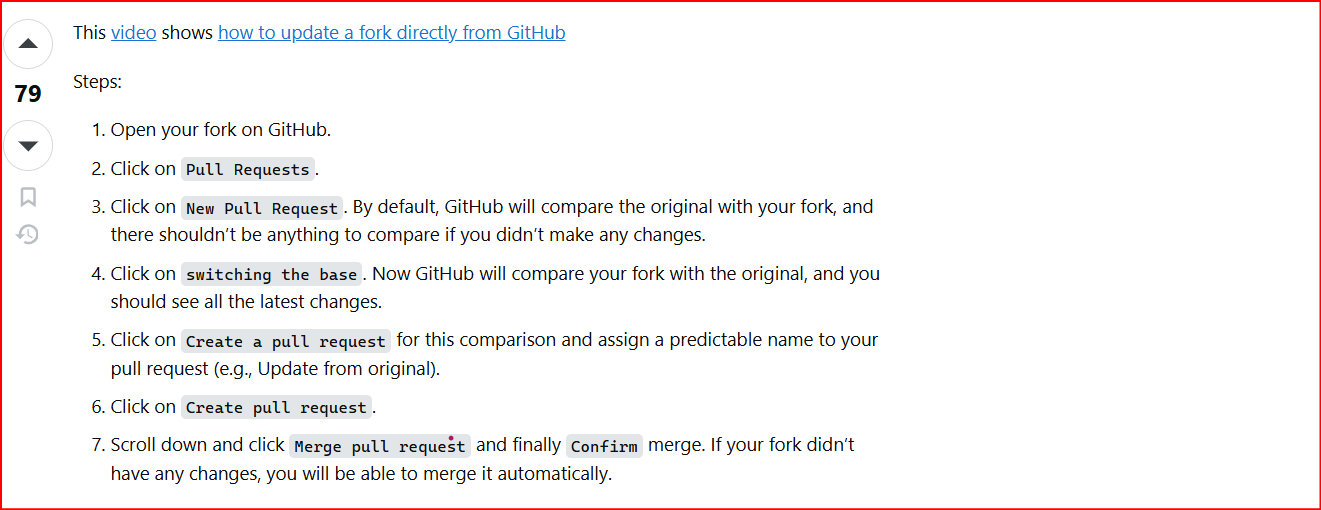
Fork this repository. For the update from the main repository, yzou can refer to this link:

<https://stackoverflow.com/questions/3903817/pull-new-updates-from-original-github-repository-into-forked-github-repository>

***Forking the repository***

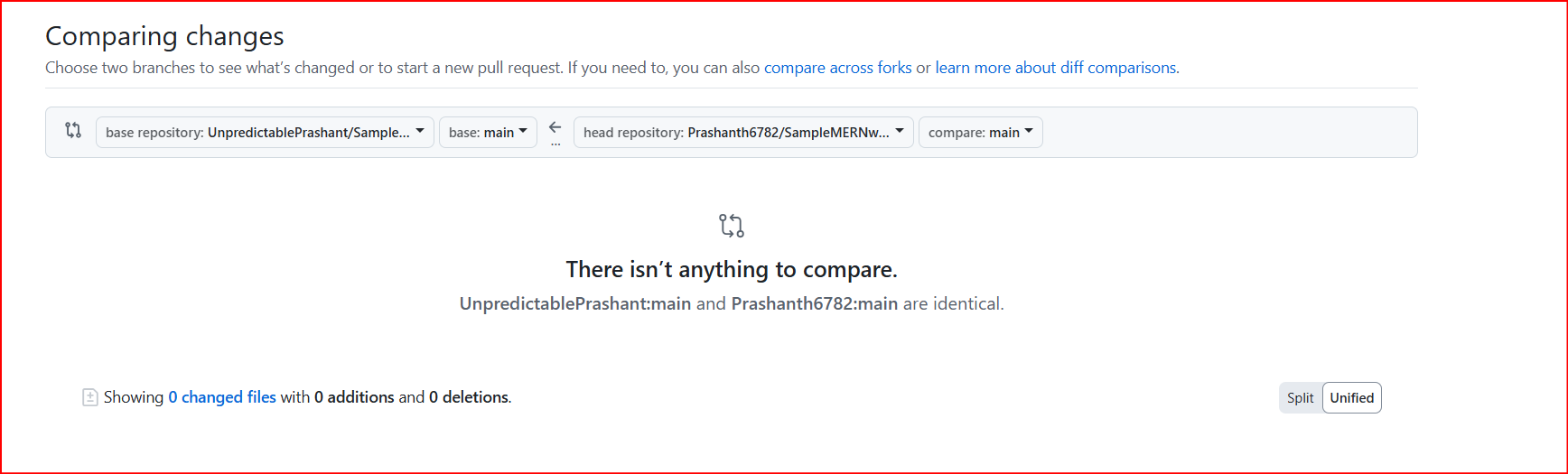
******

***For the update from the main repository***

******

***Followed the above steps***

*There are no changes*

******

***Step 1: Set Up the AWS Environment  
  
1. Set Up AWS CLI and Boto3:***

***- Install AWS CLI and configure it with AWS credentials.***

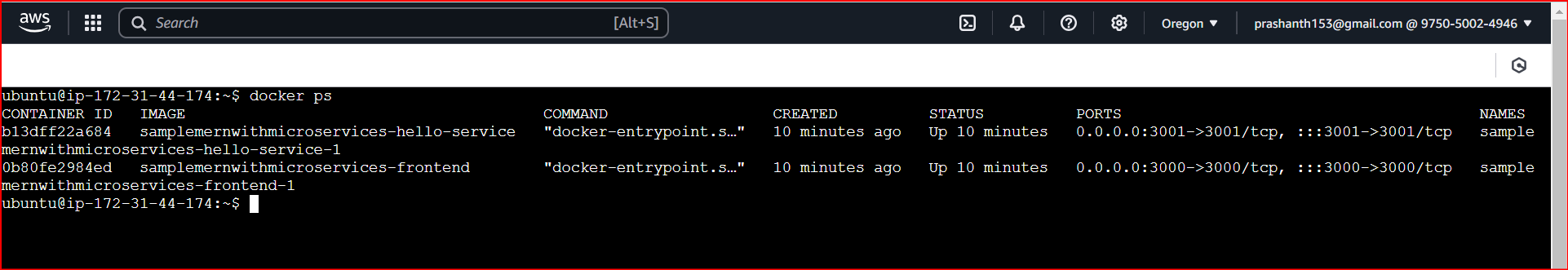
***- Install Boto3 for Python and configure it.***

*This is done using the AWS terraform script (main.tf) which has been uploaded to the GIT repository.  
Using the terraform script everything has been done and completed.****Step 2: Prepare the MERN Application***

***1. Containerize the MERN Application:***

*- Ensure the MERN application is containerized using Docker. Create a Dockerfile for each component (frontend and backend).*

*This is done using the AWS terraform script (main.tf) which has been uploaded to the GIT repository.  
Using the terraform script everything has been done and completed.*

**

***2. Push Docker Images to Amazon ECR:***

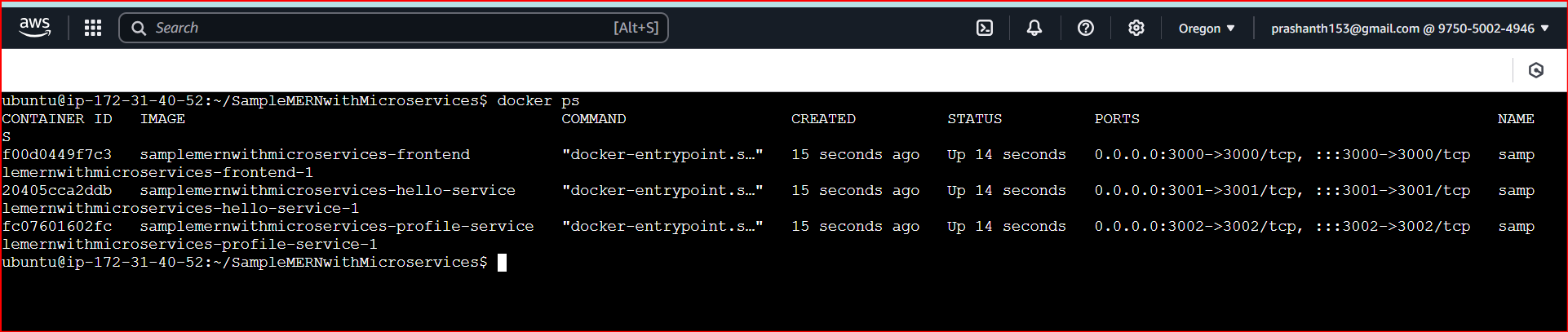
*- Build Docker images for the frontend and backend.*

*- Create an Amazon ECR repository for each image.*

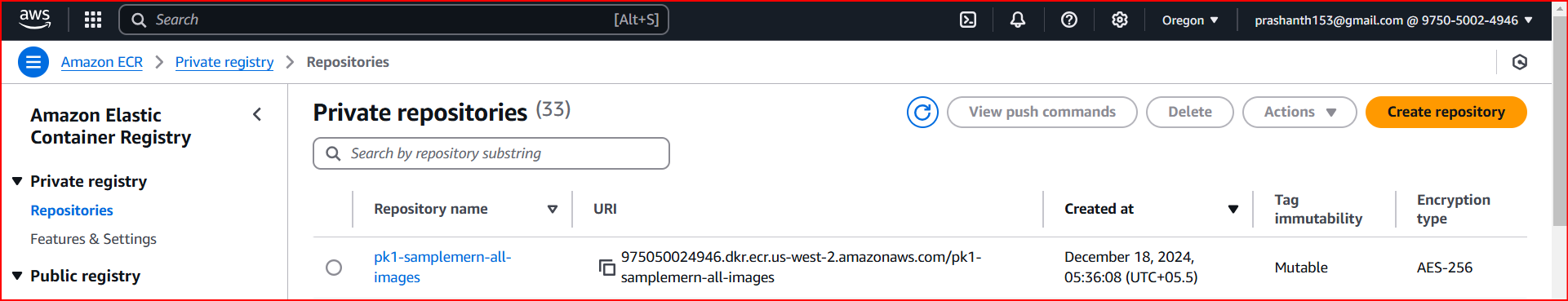
*- Push the Docker images to their respective ECR repositories.*

*All the docker containers are created using the AWS Terraform script it self as shown below and they are sent to the ECR.*

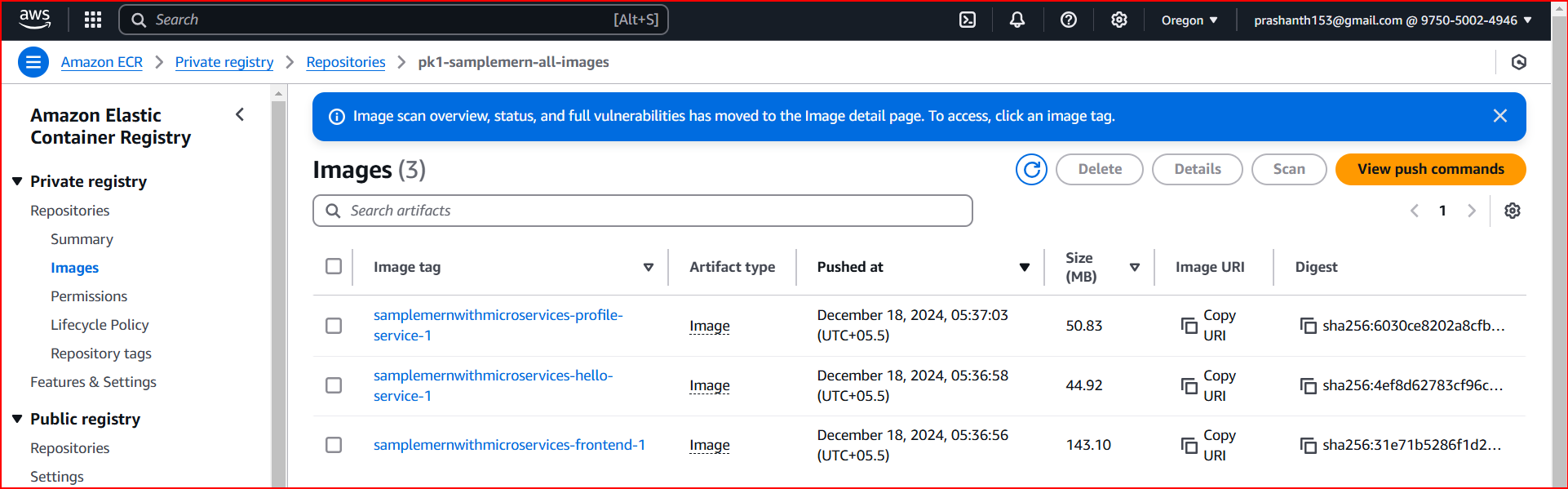
*Images of the Docker container: All the 3 containers are up and running*

**

*Mutable repositories are created and are stored at*

**

*When I open the ECR repository I can see the below 3 Images as expected 1 for the front end and the other for the 2 Microservices*

**

***I have written a shell script which will push the contents of the repositories to ECR***

|  |
| --- |
| *#!/bin/bash*  *# Variables*  *AWS\_REGION="us-west-2"                  # AWS region*  *AWS\_ACCOUNT\_ID="975050024946"    # Replace with your AWS account ID*  *REPO\_NAME="samplemern-all-images"       # Name of the ECR repository*  *# Authenticate Docker with ECR*  *echo "Logging into ECR..."*  *aws ecr get-login-password --region $AWS\_REGION | docker login --username AWS --password-stdin $AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com*  *# Ensure the repository exists (idempotent creation)*  *echo "Ensuring repository $REPO\_NAME exists..."*  *aws ecr describe-repositories --repository-names $REPO\_NAME --region $AWS\_REGION >/dev/null 2>&1 || aws ecr create-repository --repository-name $REPO\_NAME --region $AWS\_REGION*  *# Process all running containers*  *docker ps --format '{{.ID}} {{.Image}} {{.Names}}' | while read -r CONTAINER\_ID IMAGE NAME; do*  *# Tag for ECR*  *ECR\_IMAGE="$AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/$REPO\_NAME:$NAME"*    *echo "Tagging image $IMAGE for container $NAME as $ECR\_IMAGE..."*  *docker tag $IMAGE $ECR\_IMAGE*  *echo "Pushing $ECR\_IMAGE to ECR..."*  *docker push $ECR\_IMAGE*  *echo "Successfully pushed $ECR\_IMAGE."*  *done*  *echo "All images have been pushed to the ECR repository $REPO\_NAME."* |

***Step 3: Version Control***

***1. Use AWS CodeCommit:***

*- Create a CodeCommit repository.*

*- Push the MERN application source code to the CodeCommit repository.*

*It has been agreed that there is no necessity to use Code Commit as it has been deprecated.*

***Step 4: Continuous Integration with Jenkins***

***1. Set Up Jenkins:***

*- Install Jenkins on an EC2 instance.*

*- Configure Jenkins with necessary plugins.*

***2. Create Jenkins Jobs:***

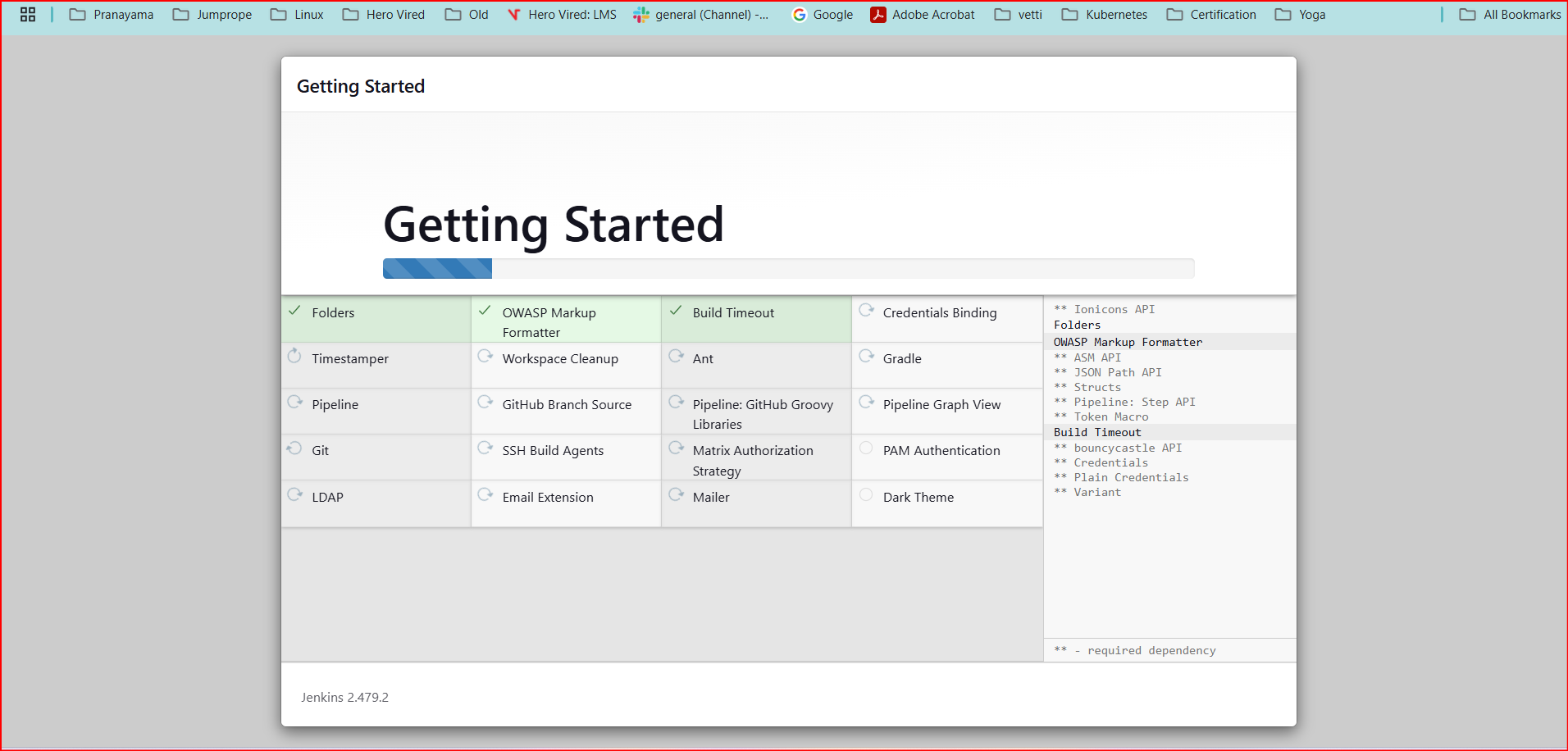
*- Create Jenkins jobs for building and pushing Docker images to ECR.*

*- Trigger the Jenkins jobs whenever there's a new commit in the CodeCommit repository.*

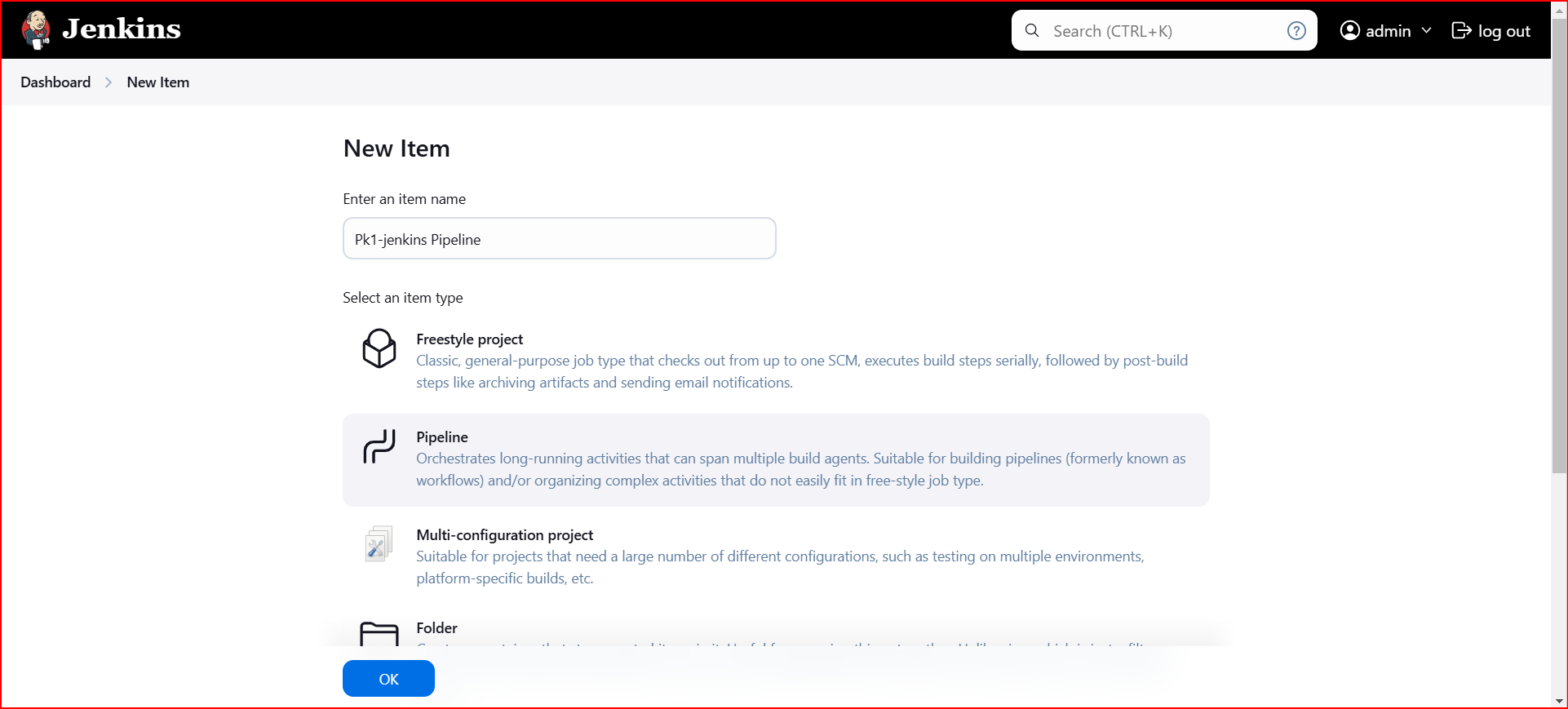
*It has been agreed that there is no necessity to use Code Commit as it has been deprecated.*

*I have created a Jenkins job which will be executed manually for now for moving the containers to ECR. The manual trigger of the Jenkins job will ensure that the shell script will be executed which will trigger the pushing of the container images to the AWS ECR.*

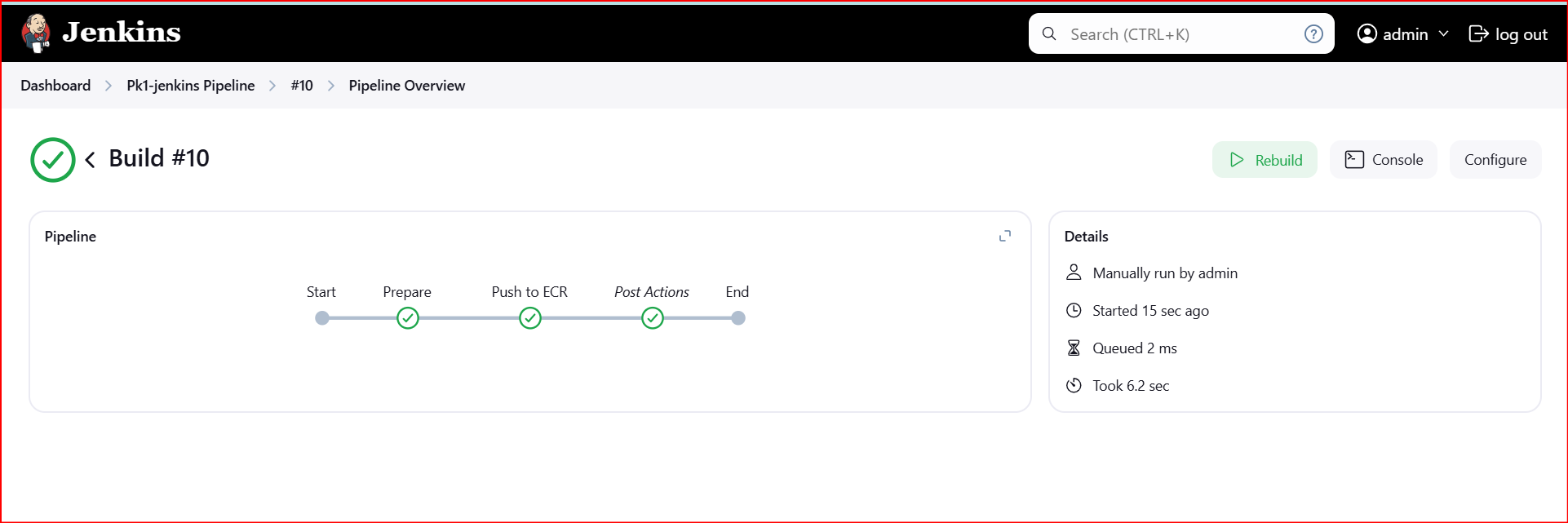
*Getting started in Jenkins within the Amazon EC2 Instance*

**

*Creating a Jenkins Job*

**

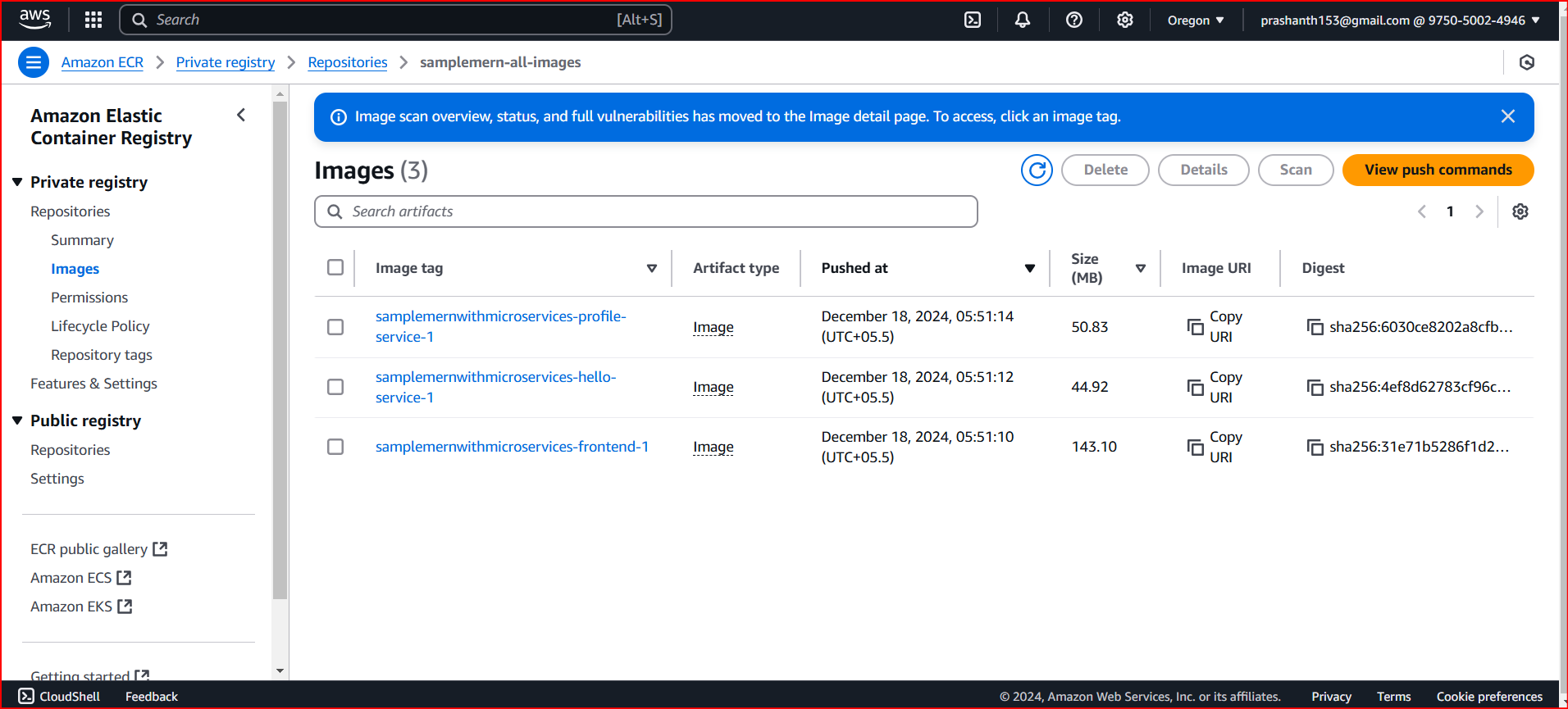
*The pipeline had run fine after sorting the permission issues*

**

*Jenkins pipeline configuration code:*

|  |
| --- |
| *pipeline {*  *agent any*  *environment {*  *AWS\_REGION = 'us-west-2'*  *AWS\_ACCOUNT\_ID = '975050024946'*  *SCRIPT\_PATH = '/home/ubuntu/SampleMERNwithMicroservices/push\_all\_to\_ecr.sh'*  *}*  *stages {*  *stage('Prepare') {*  *steps {*  *script {*  *echo 'Ensuring AWS CLI and Docker are accessible...'*  *sh 'aws --version'*  *sh 'docker --version'*  *}*  *}*  *}*  *stage('Push to ECR') {*  *steps {*  *script {*  *echo 'Running the ECR push script...'*  *sh """*  *sudo chmod +x ${SCRIPT\_PATH}*  *sudo ${SCRIPT\_PATH}*  *"""*  *}*  *}*  *}*  *}*  *post {*  *always {*  *echo 'Job completed!'*  *}*  *}*  *}* |

*Jenkins jobs have updated the ECR Elastic container registry as expected by running the shell script*

**

***Step 5: Infrastructure as Code (IaC) with Boto3***

***1. Define Infrastructure with Boto3 (Python Script):***

***- Use Boto3 to define the infrastructure (VPC, subnets, security groups).***

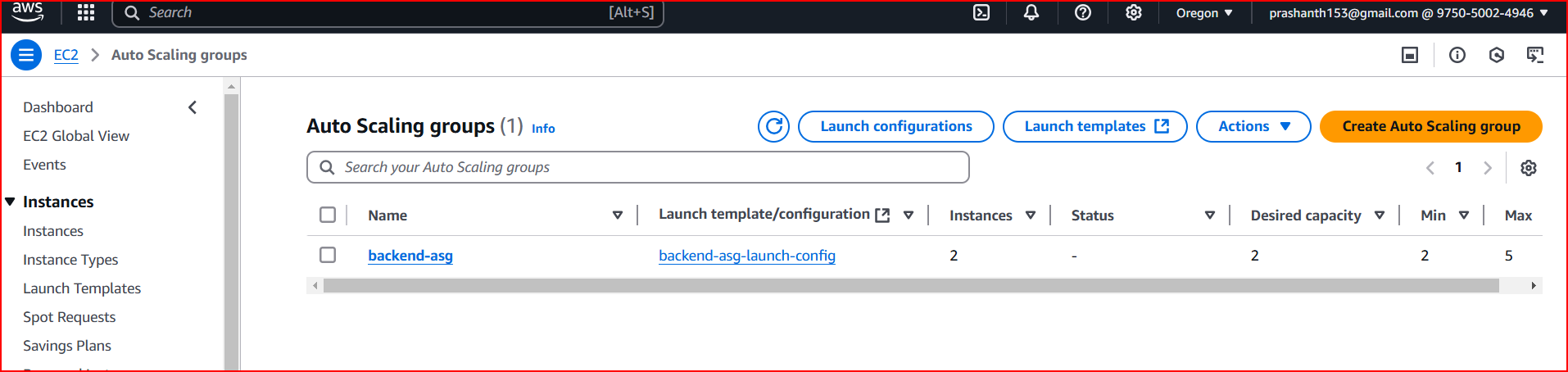
***- Define an Auto Scaling Group (ASG) for the backend.***

***- Create AWS Lambda functions if needed.***

*Boto3 code is created for the same*

|  |
| --- |
| *import boto3*  *# Parameters*  *VPC\_ID = "vpc-0321f38a7b594180d"  # Replace with your VPC ID*  *SUBNET\_IDS = [*  *"subnet-06bd72b2e4cb41d10",*  *"subnet-09bd0e0acc92d4efa",*  *]  # Replace with your subnet IDs*  *SECURITY\_GROUP\_ID = "sg-091b3da4f3bdc9d5b"  # Replace with your security group ID*  *AMI\_ID = "ami-04dd23e62ed049936"  # Replace with the correct AMI ID for your backend instances*  *INSTANCE\_TYPE = "t3.medium"  # Choose an appropriate instance type*  *USER\_DATA = """#!/bin/bash*  *sudo apt update*  *sudo apt install -y docker*  *sudo docker run -d -p 80:80 backend-image  # Replace with your backend Docker image*  *"""*  *TARGET\_GROUP\_NAME = "backend-target-group"*  *LOAD\_BALANCER\_NAME = "my-alb"*  *AUTO\_SCALING\_GROUP\_NAME = "backend-asg"*  *# Initialize Boto3 clients*  *ec2 = boto3.client("ec2")*  *elbv2 = boto3.client("elbv2")*  *autoscaling = boto3.client("autoscaling")*  *# 1. Create a Target Group for the Backend instances*  *target\_group\_response = elbv2.create\_target\_group(*  *Name=TARGET\_GROUP\_NAME,*  *Protocol="HTTP",*  *Port=80,  # or whichever port your backend uses*  *VpcId=VPC\_ID,*  *HealthCheckProtocol="HTTP",*  *HealthCheckPort="80",*  *HealthCheckPath="/health",  # Replace with your actual health check endpoint*  *Matcher={*  *"HttpCode": "200"  # Adjust this to match your backend health check response*  *},*  *)*  *target\_group\_arn = target\_group\_response["TargetGroups"][0]["TargetGroupArn"]*  *# 2. Create an Application Load Balancer (ALB)*  *load\_balancer\_response = elbv2.create\_load\_balancer(*  *Name=LOAD\_BALANCER\_NAME,*  *Subnets=SUBNET\_IDS,*  *SecurityGroups=[SECURITY\_GROUP\_ID],*  *Scheme="internet-facing",*  *Type="application",*  *IpAddressType="ipv4",*  *)*  *alb\_arn = load\_balancer\_response["LoadBalancers"][0]["LoadBalancerArn"]*  *alb\_dns\_name = load\_balancer\_response["LoadBalancers"][0]["DNSName"]*  *# Modify the Load Balancer Attributes*  *elbv2.modify\_load\_balancer\_attributes(*  *LoadBalancerArn=alb\_arn,*  *Attributes=[{"Key": "idle\_timeout.timeout\_seconds", "Value": "60"}],*  *)*  *# 3. Create an Auto Scaling Group (ASG)*  *launch\_configuration\_response = autoscaling.create\_launch\_configuration(*  *LaunchConfigurationName=f"{AUTO\_SCALING\_GROUP\_NAME}-launch-config",*  *ImageId=AMI\_ID,*  *InstanceType=INSTANCE\_TYPE,*  *SecurityGroups=[SECURITY\_GROUP\_ID],*  *UserData=USER\_DATA,*  *InstanceMonitoring={"Enabled": True},*  *)*  *asg\_response = autoscaling.create\_auto\_scaling\_group(*  *AutoScalingGroupName=AUTO\_SCALING\_GROUP\_NAME,*  *LaunchConfigurationName=f"{AUTO\_SCALING\_GROUP\_NAME}-launch-config",*  *MinSize=2,  # Minimum number of instances*  *MaxSize=5,  # Maximum number of instances*  *DesiredCapacity=2,  # Desired number of instances*  *VPCZoneIdentifier=",".join(SUBNET\_IDS),  # Replace with your subnet IDs*  *TargetGroupARNs=[target\_group\_arn],  # Associate with the target group*  *HealthCheckType="EC2",*  *HealthCheckGracePeriod=300,  # Adjust this based on your health check needs*  *Tags=[{"Key": "Name", "Value": "backend-instance", "PropagateAtLaunch": True}],*  *)*  *# 4. Register the ALB with Auto Scaling Group*  *# ALB Listener setup (HTTP Listener to forward traffic to the target group)*  *# ALB Listener setup (HTTP Listener to forward traffic to the target group)*  *listener\_response = elbv2.create\_listener(*  *LoadBalancerArn=alb\_arn,*  *Protocol="HTTP",*  *Port=80,*  *DefaultActions=[{"Type": "forward", "TargetGroupArn": target\_group\_arn}],*  *)*  *# Print the ALB DNS Name and confirmation of Auto Scaling Group creation*  *print(f"ALB DNS Name: {alb\_dns\_name}")*  *print(f"Auto Scaling Group {AUTO\_SCALING\_GROUP\_NAME} created successfully.")* |

*When I run the Boto3 code I can see the Load Balancer, ASG, created*

**

***Step 6: Deploying Backend Services***

***1. Deploy Backend on EC2 with ASG:***

***- Use Boto3 to deploy EC2 instances with the Dockerized backend application in the ASG.***

***Step 7: Set Up Networking***

***1. Create Load Balancer:***

***- Set up an Elastic Load Balancer (ELB) for the backend ASG.***

***2. Configure DNS:***

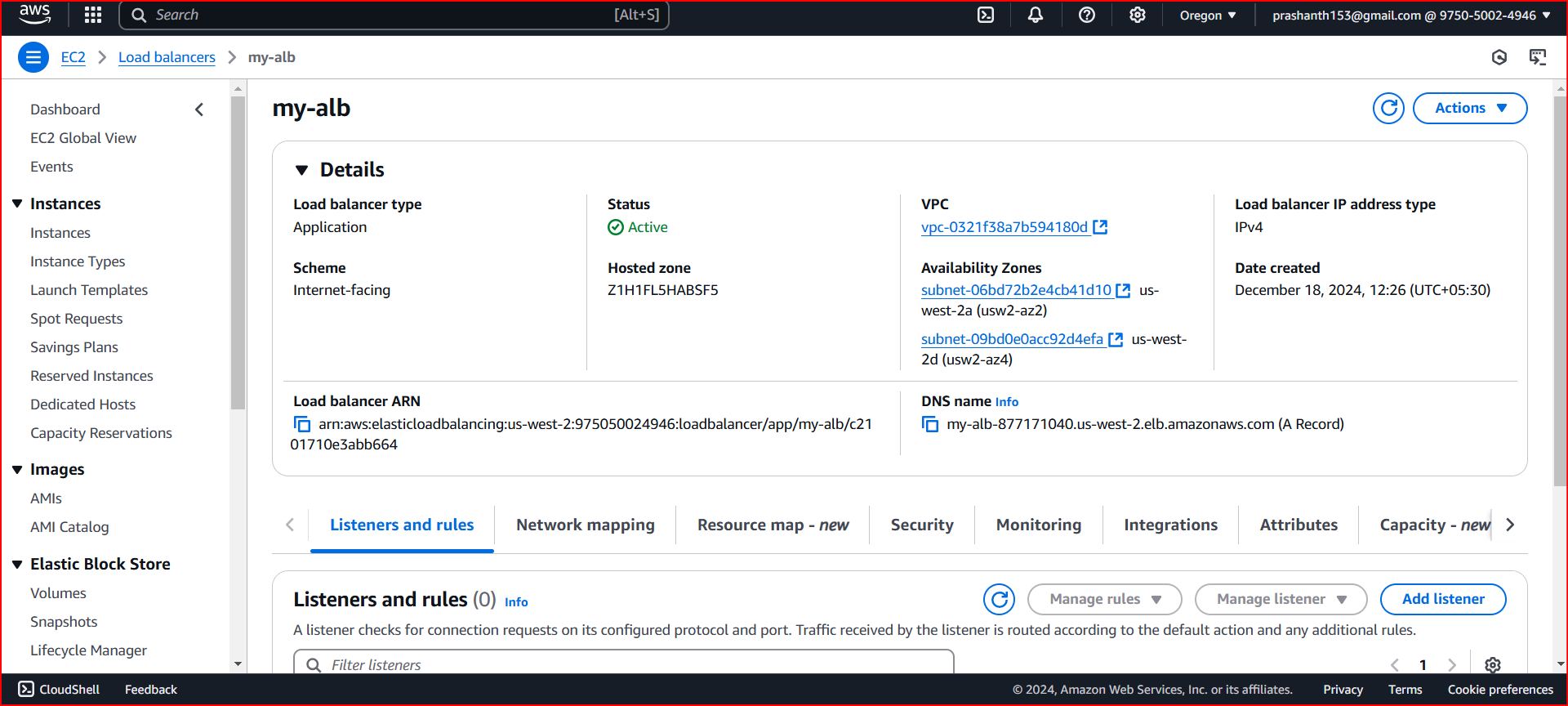
***- Set up DNS using Route 53 or any other DNS service.***

***Step 8: Deploying Frontend Services***

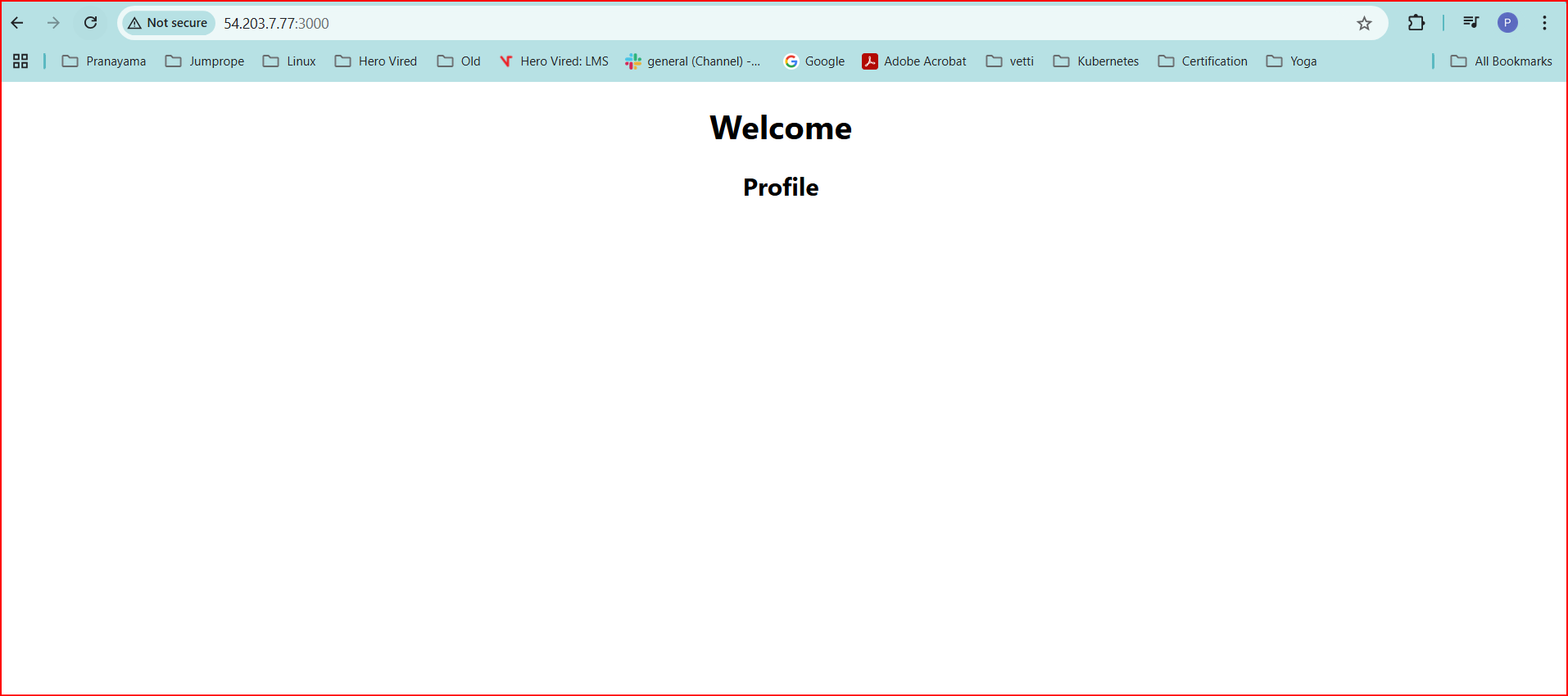
***1. Deploy Frontend on EC2:***

***- Use Boto3 to deploy EC2 instances with the Dockerized frontend application.***

*The load balancers were created and everything under the ASG the baclend containers were loaded.*

**

*The frontend was also was created with the docker container and I could see the website showing up like below when I give the ec2 module*

**

***Step 10: Kubernetes (EKS) Deployment***

***1. Create EKS Cluster:***

*- Use eksctl or other tools to create an Amazon EKS cluster.*

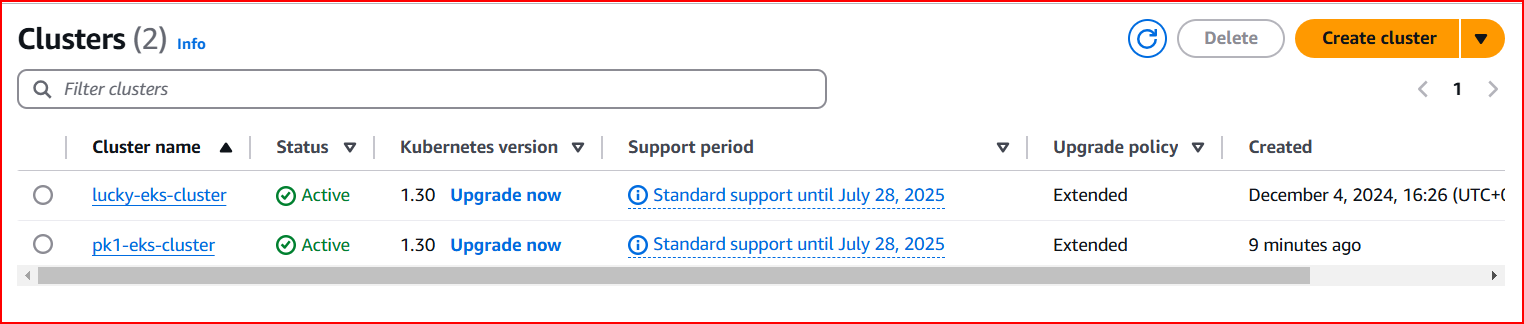
***2. Deploy Application with Helm:***

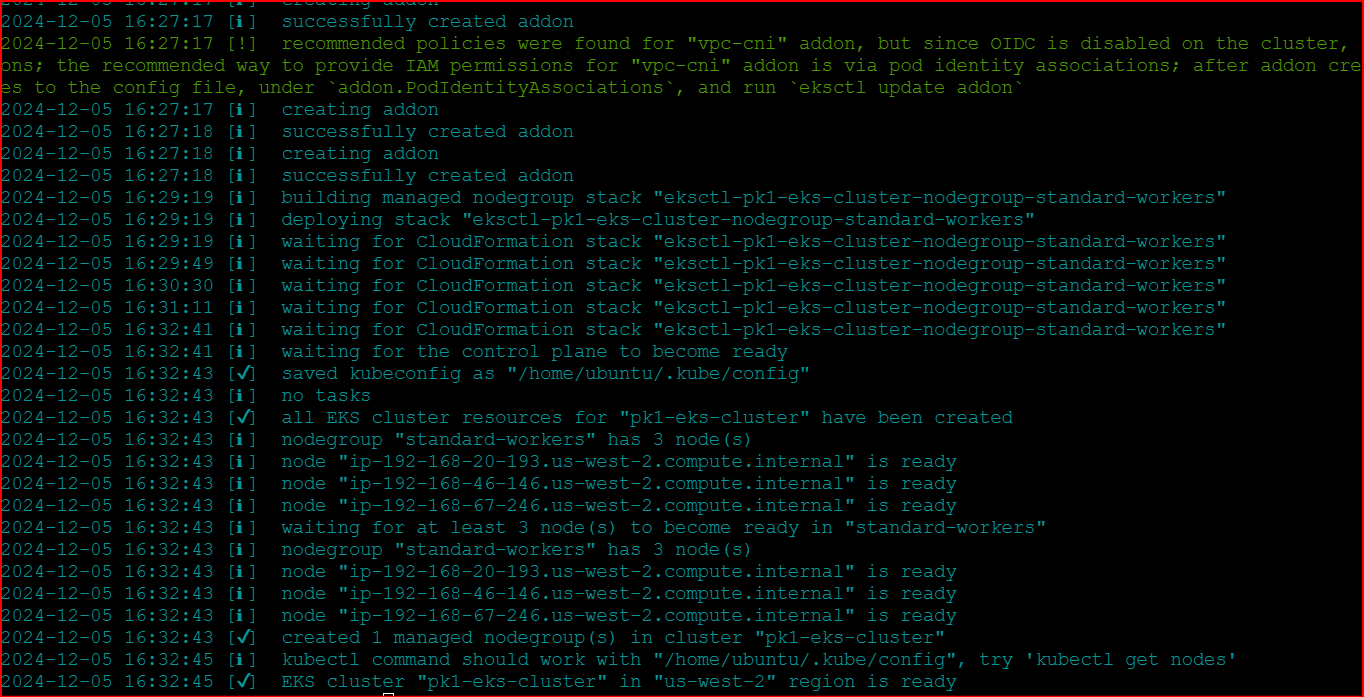
*- Use Helm to package and deploy the MERN application on EKS.*

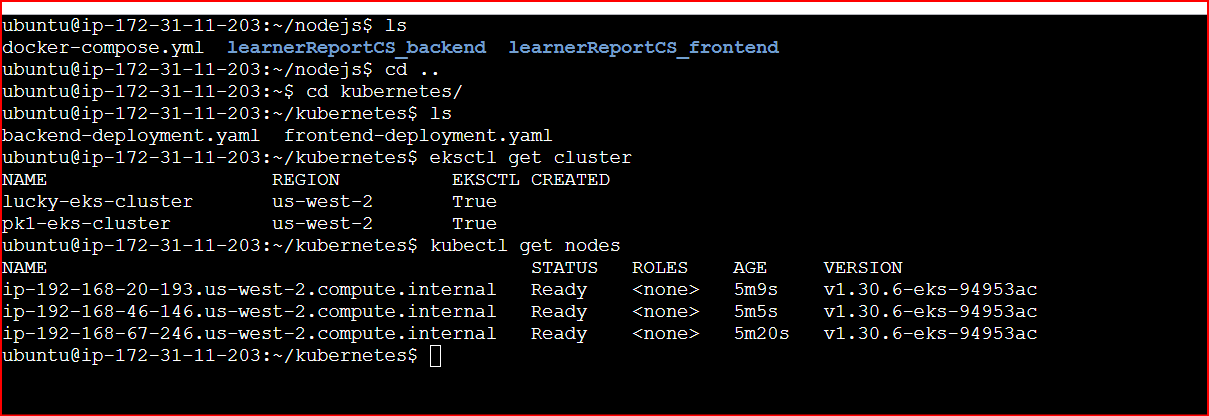
*Create the cluster using EKSCTL – 3 Nodes*

|  |
| --- |
| *eksctl create cluster \*  *--name pk1-eks-cluster \*  *--region us-west-2 \*  *--nodegroup-name standard-workers \*  *--node-type t3.medium \*  *--nodes 3 \*  *--nodes-min 3 \*  *--nodes-max 3 \*  *--managed* |

*Cluster is created with 3 Nodes and in active pk1-eks-cluster*

**

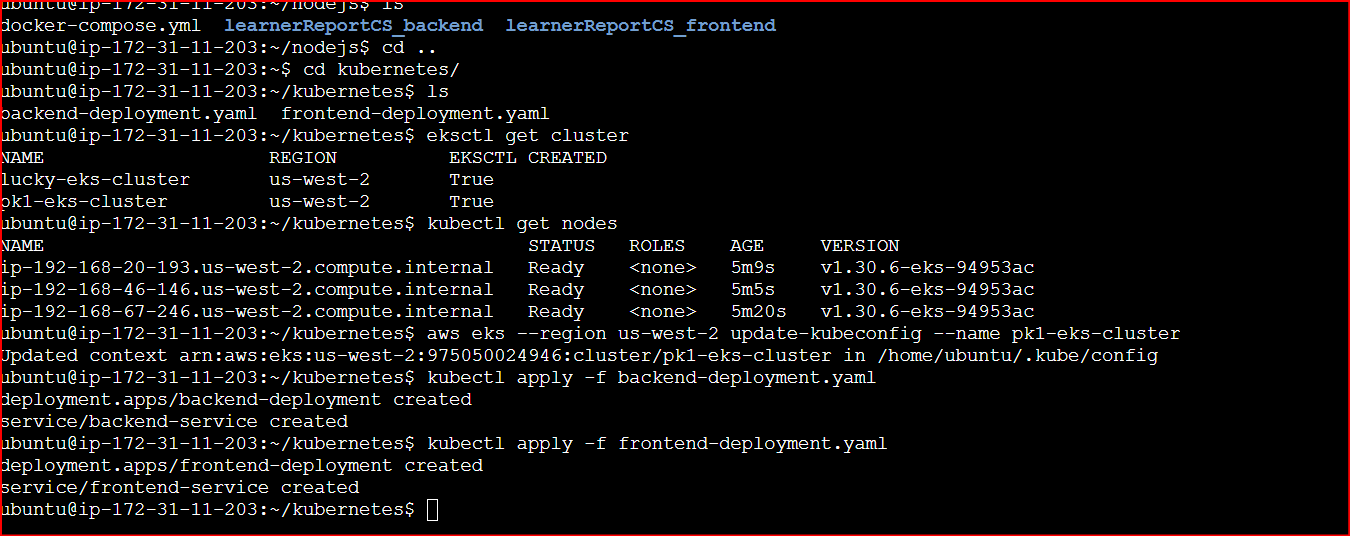
**

**

*Step4:*

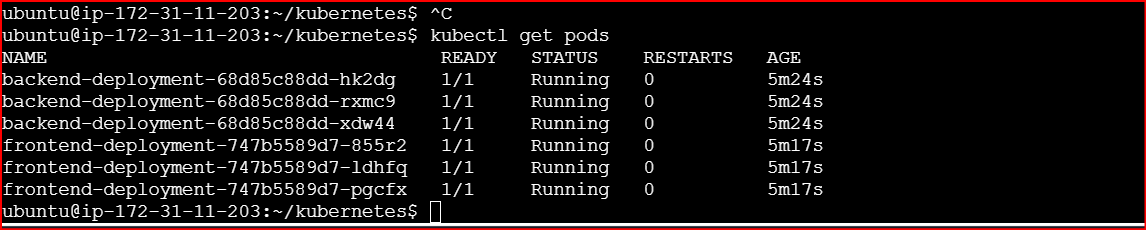
*Installing the nodejs app into the cluster and trying to run it manually. After creating the deployment files.*

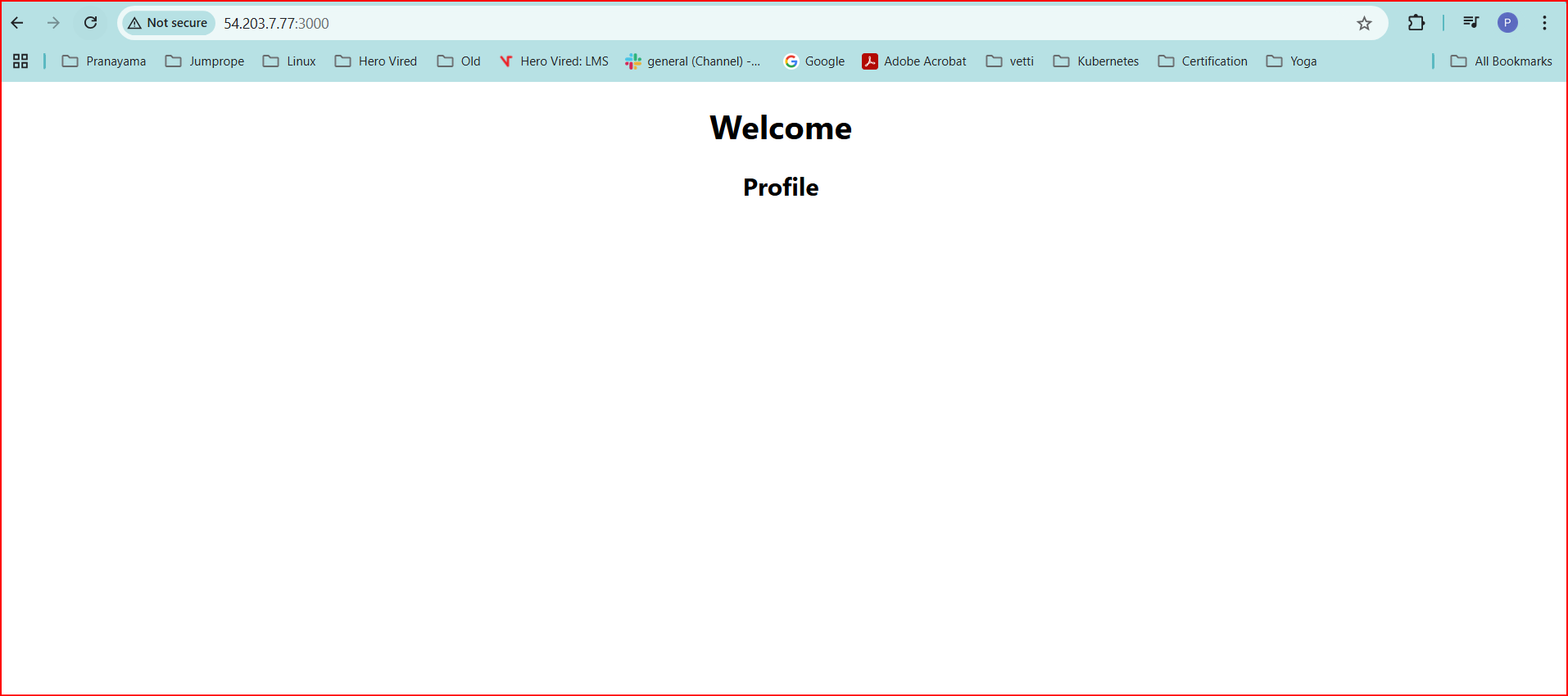
*Frontend and the Backend YAML are deployed successfully*

**

*When I give the external IP of the load balancer front end service the application is displayed as expected*

*All the 6 Pods are running as expected*

**

**

*Step5*

*Using the HELM for the package deployment:*

*mern-helm-chart/*

*├── Chart.yaml*

*├── values.yaml*

*├── templates/*

*├── frontend-deployment.yaml*

*├── backend-deployment.yaml*

*├── mongo-deployment.yaml*