DOCTOR'S APPOINTMENT APP

ON AWS EKS WITH DOCKER AND DYNAMODB

Submitted By

Group 8

of

ENPM818R

Name	**Directory ID**	**University ID**	**Section**
Aashvi Nagendra	158aash	120187704	0101
Anvit Mirjurkar	anvit121	120140075	0101
Bhavinkumar Kanubhai Panchal	bhavin22	120278907	0101
Rincy Mariam Thomas	rmt21	120124044	0101
Saarthak Singh	saarthak	120427299	0101
Shankar Narayan Saiprasad	shankarn	121128041	AEB1
Vishal Kinnera	vkinnera	120258893	0101

Table of Contents

Figures

1. Introduction

The Doctor's Office Appointment App project is a comprehensive initiative that aims to develop a scalable and containerized web application. The project followed a structured approach to build and deploy the application using Docker and AWS cloud services, including Elastic Kubernetes Service (EKS), Elastic Container Registry (ECR), and CloudWatch. The application included a React frontend, a Node.js + Express backend, and a DynamoDB database for storing appointment data.

This report details each stage of the implementation, from building Docker images and setting up an EKS cluster to configuring CI/CD pipelines and monitoring the system's performance.

2. Architecture Overview

The project architecture was designed around microservices principles, ensuring modularity and scalability. The app consists of three main layers:

- **Frontend**: A React-based interface where users could book and view appointments.
- **Backend**: A Node.js + Express server that processed API requests and managed data operations.
- **Database**: DynamoDB was selected for its scalability and NoSQL capabilities, enabling fast and reliable data access.

2.1 Architecture Diagram

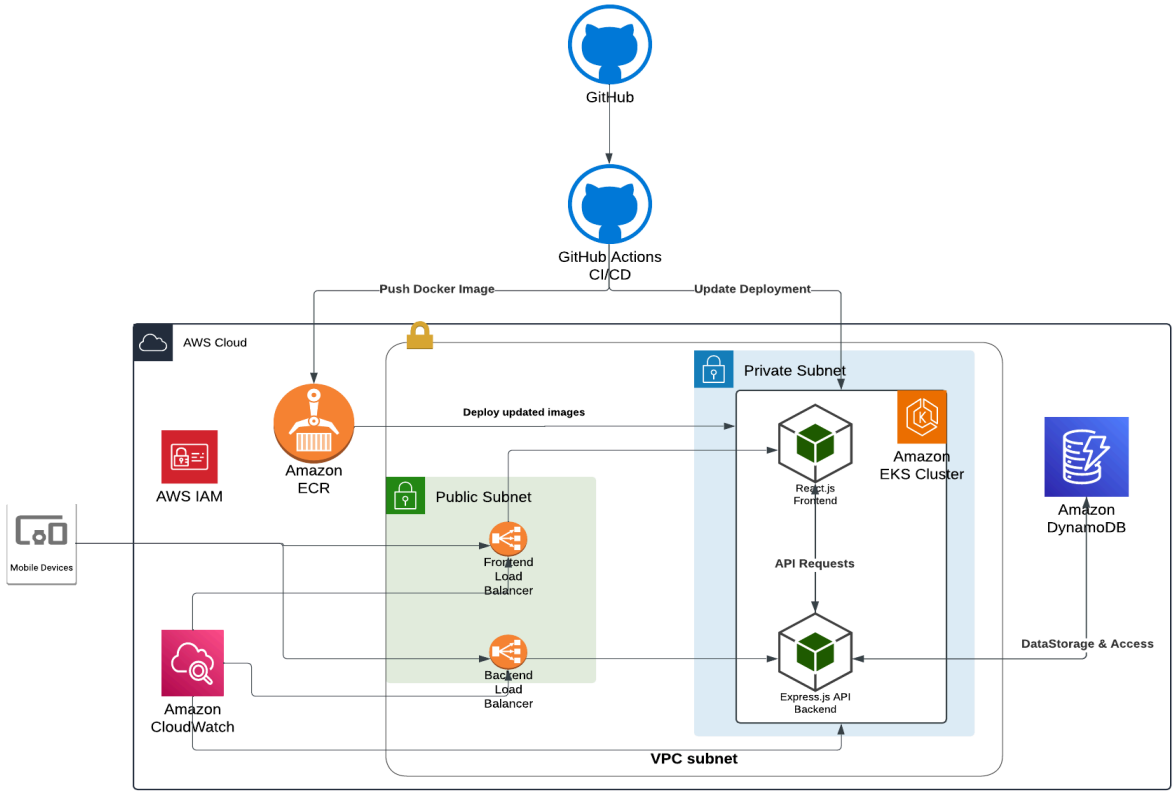

Figure 1: Architecture Diagram

Architecture Diagram: LucidChart Link

The architecture for this project leverages AWS services to create a scalable, secure, and containerized application infrastructure using Kubernetes on Amazon EKS, with AWS DynamoDB as a managed NoSQL database. Below is a breakdown of each component and its role in the system:

2.1.1 IAM (Identity and Access Management)

- **Role-Based Access**: IAM manages secure access to AWS services and resources. The EKS cluster and backend services use IAM roles to access DynamoDB, ensuring credentials aren't stored within application code or Kubernetes manifests. These roles are assigned with least-privilege policies for enhanced security.

2.1.2 VPC (Virtual Private Cloud)

- **Private and Public Subnets**: The VPC contains a private subnet for EKS and a public subnet to host the Application Load Balancer, enabling secure routing and access management between frontend and backend services.

- **Classic Load Balancer**: The architecture implements a dual Classic Load Balancer configuration within the public subnet. The primary frontend load balancer serves as the initial point of contact, efficiently distributing incoming user traffic across frontend services hosted in Amazon Elastic Kubernetes Service (EKS). For backend operations, requests are seamlessly redirected to a dedicated backend load balancer, which manages and distributes these requests across the backend services within EKS. This hierarchical load balancing approach ensures optimal fault tolerance, enables efficient traffic management, and maintains high availability across the entire application infrastructure.

2.1.3 Amazon ECR (Elastic Container Registry)

- Stores Docker images for both the frontend and backend services, which are then pulled by EKS for deployment. GitHub Actions pushes the latest versions of these images to ECR, ensuring each deployment reflects the most recent updates to the application code.

2.1.4 Amazon EKS (Elastic Kubernetes Service)

- **EKS Cluster in Private Subnet**: Hosts the containerized applications in an isolated, secure environment within a private subnet. The cluster manages both the frontend (React.js) and backend (Express.js API) services, maintaining high availability and scalability.
- **Kubernetes Deployments and Services**: Each microservice (frontend and backend) is defined using Kubernetes deployment YAML files, which specify replica counts, container images, and resource configurations. Services are exposed within the cluster, and the backend is connected to DynamoDB for data storage.

2.1.5 DynamoDB (AWS NoSQL Database Service)

- **External to the VPC**: This managed NoSQL database stores appointment data, configured with a primary key to ensure efficient data retrieval. The backend API connects directly to DynamoDB, using AWS IAM roles and policies for secure access.

2.1.6 GitHub and GitHub Actions (CI/CD Pipeline)

- **Code Repository (GitHub)**: Stores the source code for the frontend and backend applications, along with Kubernetes manifests and Docker configurations.
- **GitHub Actions**: Serves as the CI/CD pipeline, automating tasks like building Docker images, running tests, tagging images, and pushing these to Amazon ECR (Elastic Container Registry). After successful builds, the pipeline triggers deployments to Amazon EKS, ensuring continuous delivery and deployment of application updates.

2.1.7 Amazon CloudWatch (Monitoring and Logging)

- **EKS and Application Monitoring**: CloudWatch is configured to capture logs and metrics for both frontend and backend services within EKS, as well as any CI/CD

pipeline actions. This helps monitor performance, track errors, and set alerts for infrastructure health.

Key Interactions

1. **Code Changes & Deployment**: Developers push code changes to GitHub, triggering GitHub Actions to build, test, and deploy new Docker images to ECR. Kubernetes then pulls these images from ECR for deployment on EKS.
2. **Load Balancing**: The Application Load Balancer in the public subnet routes client requests to the appropriate service within the EKS cluster, ensuring efficient load distribution.
3. **Backend Database Interaction**: The backend API service securely accesses DynamoDB outside the VPC for appointment data storage and retrieval, with access controlled by IAM roles.

This setup supports a highly resilient, scalable, and secure deployment of the application, leveraging AWS-managed services to minimize operational overhead while maximizing application availability and performance.

3. Prerequisites and Setup

Required Tools:

Docker: For containerizing the backend and frontend applications.

AWS CLI: To interact with AWS services like Amazon ECR and EKS from the command line.

Kubectl: A command-line tool to interact with Kubernetes clusters, specifically for EKS in this setup.

Node.js: To manage dependencies and run the application locally before containerization.

AWS ECR (Elastic Container Registry): AWS service for storing Docker images.

Amazon EKS (Elastic Kubernetes Service): AWS service to manage Kubernetes clusters.

4. Step-by-Step Implementation

4.1 Create an EKS Cluster and Node Group

The creation of an EKS cluster involved configuring the necessary roles and permissions to ensure that the cluster could manage Kubernetes workloads effectively. The IAM roles shown in the images were set up to allow the cluster to perform administrative operations. Additionally, the cluster was initialized with appropriate configurations, such as region and node group specifications, forming the base environment for containerized application deployments.

4.1.1 Create an EKS Cluster

This step is the initial configuration for creating an Amazon EKS cluster. In this setup page, we specify basic cluster details such as the name and the IAM role.

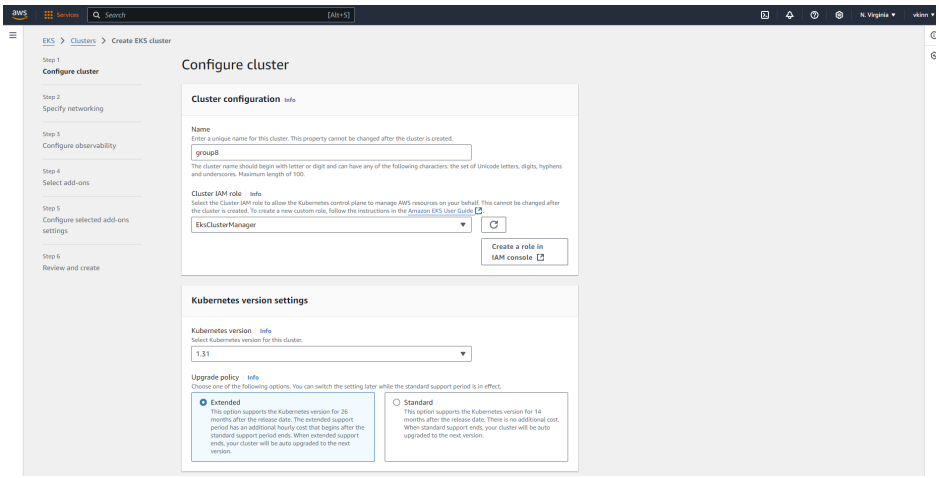

Figure 2: Configure EKS Cluster

Cluster IAM role: This IAM role setup is crucial for granting the EKS cluster permissions to manage AWS resources. This IAM role will enable the EKS cluster to interact with other AWS components. By assigning permissions early in the setup, the cluster has the authority needed to configure network and security settings.

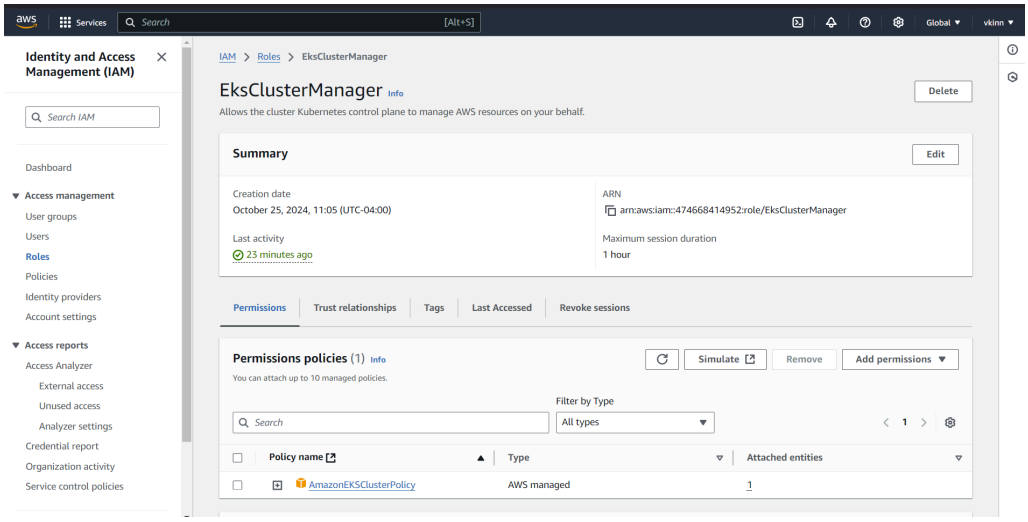

Figure 3: Cluster IAM Role

The networking configuration establishes the VPC, subnets, and security groups that will be used by the EKS cluster to manage and control traffic within the cluster. These settings will be leveraged in the deployment phase, ensuring that only specific resources can

access the cluster components, such as the backend and frontend services when they are deployed.

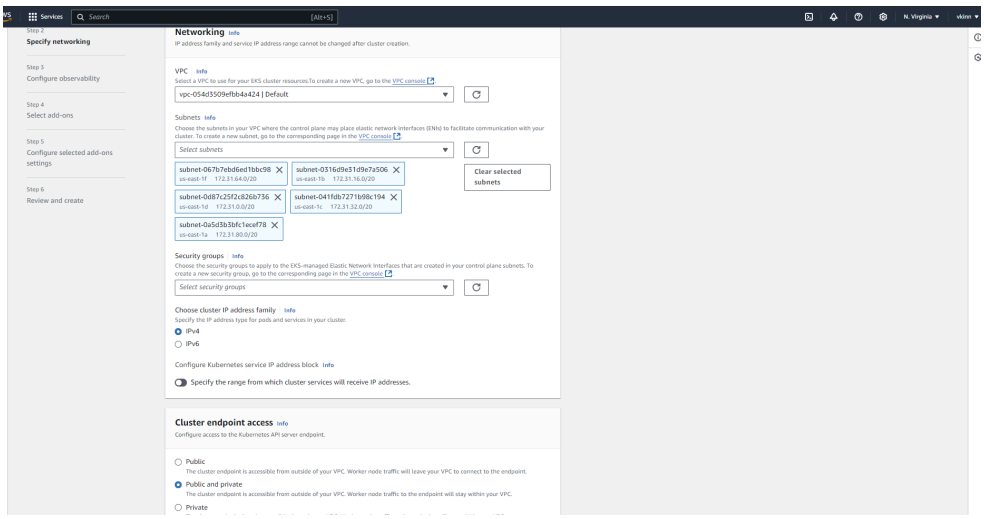

Figure 4: EKS Networking

Configuring observability allows for logging and monitoring, which is essential for tracking the health and performance of the EKS cluster. Observability will allow monitoring of cluster performance once applications are deployed.

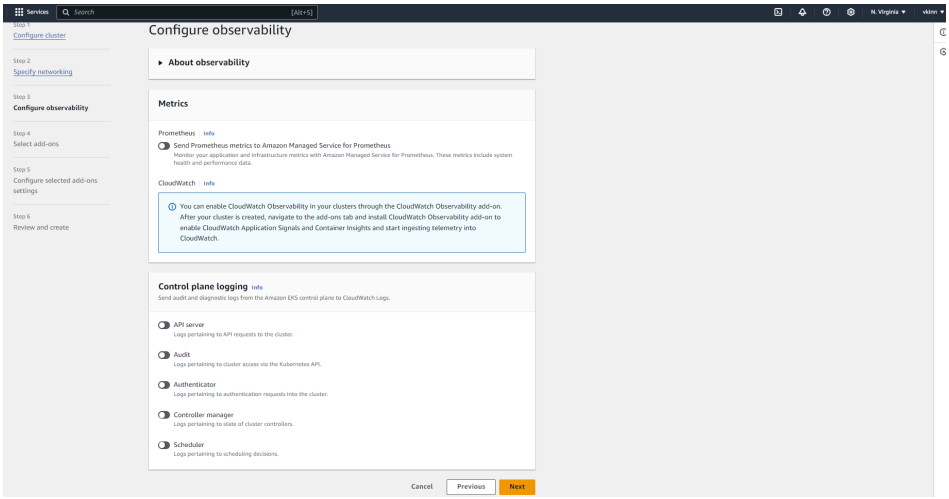

Figure 5: Configuring Observability

Add-ons are selected to enhance the functionality of the EKS cluster, particularly for managing networking and security.

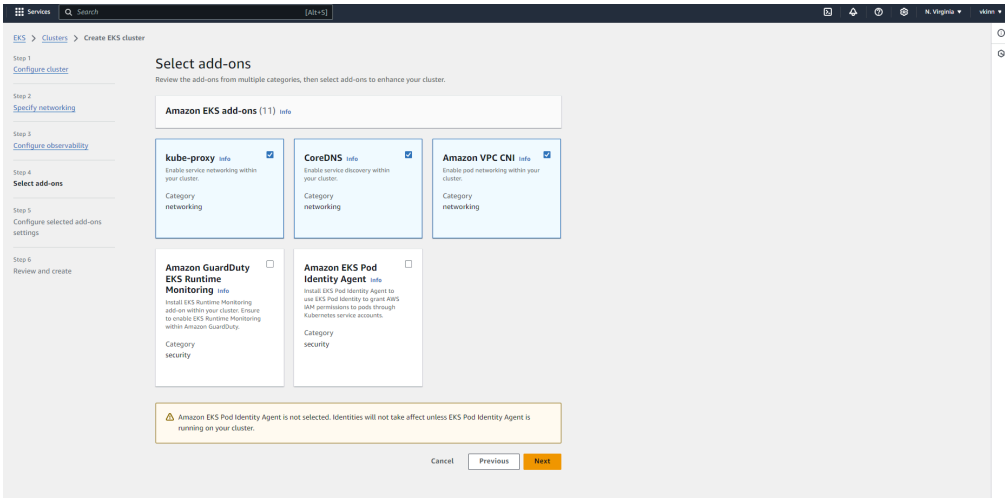

Figure 6: EKS Cluster Add-ons Selection

These add-ons are essential for enabling the Kubernetes cluster to handle networking and DNS requirements efficiently. They will play a critical role in managing network traffic and communication between services once applications are deployed on the cluster.

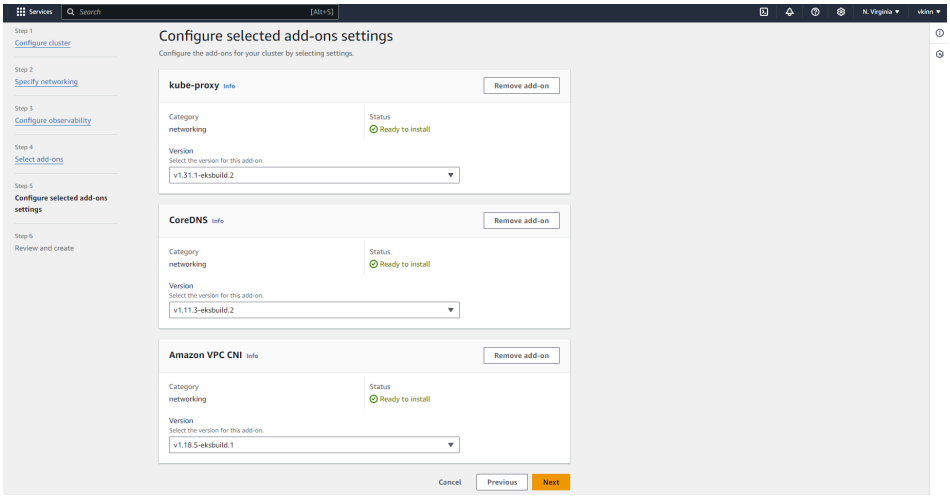

Figure 7: EKS Cluster Add-on Configuration

Creating a cluster

This overview confirms that AWS is processing the EKS cluster setup. Once the status changes to "Active," the cluster will be ready to deploy applications and utilize the IAM roles, networking, and add-ons configured in previous steps.

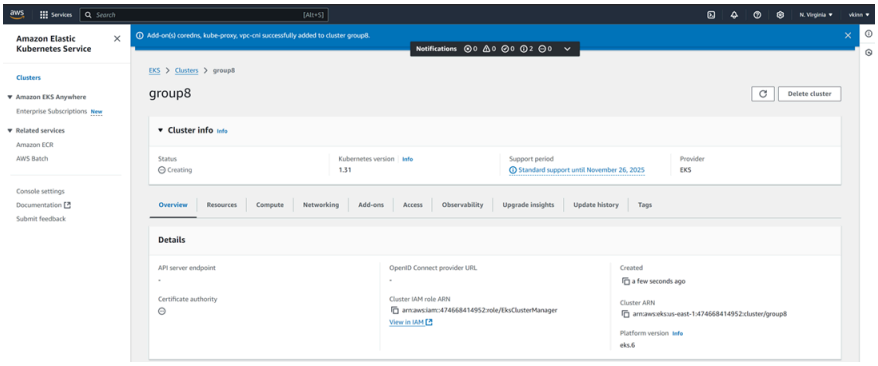

Figure 8: EKS Cluster Creation

Giving access to the cluster as cluster administration

The configuration for IAM access entry for an IAM user to access Kubernetes resources in the cluster is achieved.This configuration is critical for enabling IAM user (i.e CLIAdmin) access to the Kubernetes cluster.

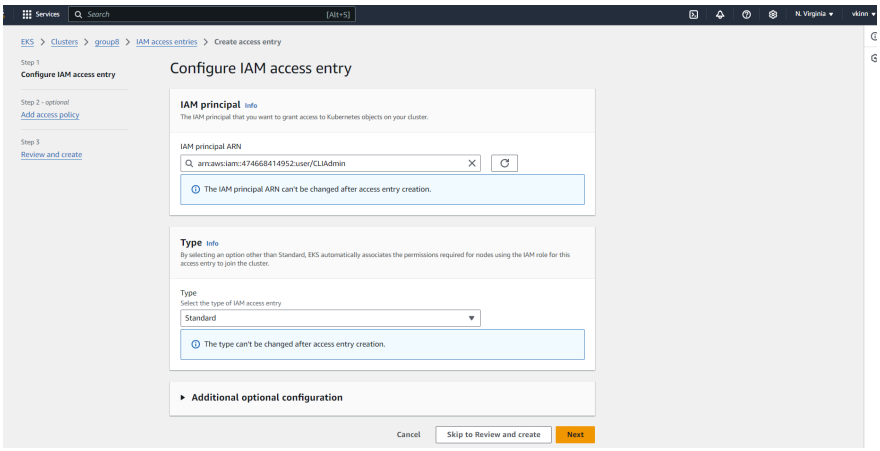

Figure 9: Configure IAM Access Entry

Giving the EKSAdminPolicy

With the EKSAdminPolicy in place, the IAM user has the authority to perform all administrative tasks within the EKS cluster, ensuring that applications can be deployed, scaled, and managed effectively.

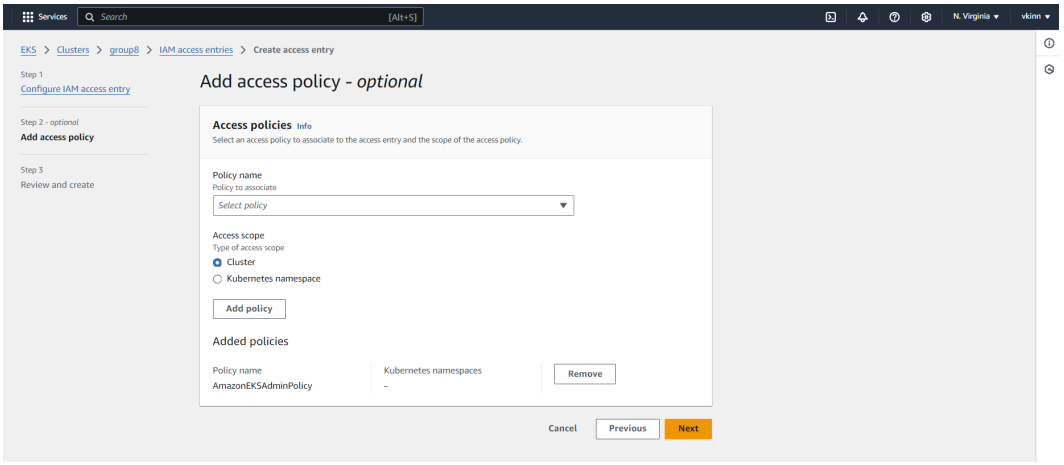

Figure 10: Configure IAM Access Scope

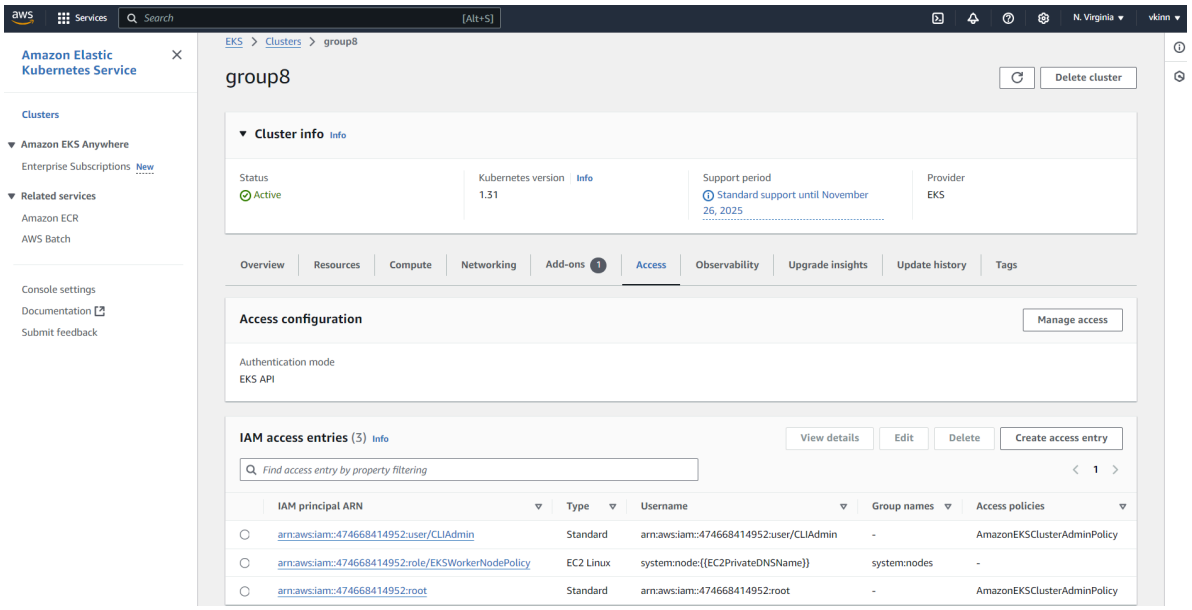

Figure 11: EKS Cluster - IAM Entries

This setup helps control access to the EKS cluster by leveraging AWS IAM to manage users, roles, and permissions.

We configure a node group in Amazon Elastic Kubernetes Service (EKS) and this is a crucial step for configuring the compute resources that will run workloads in the EKS cluster.

A node group in EKS is essentially a collection of Amazon EC2 instances that provide the underlying compute capacity for running Kubernetes workloads. These nodes (EC2

instances) join the EKS cluster and allow workloads (applications, services, etc.) to be deployed and managed by Kubernetes.

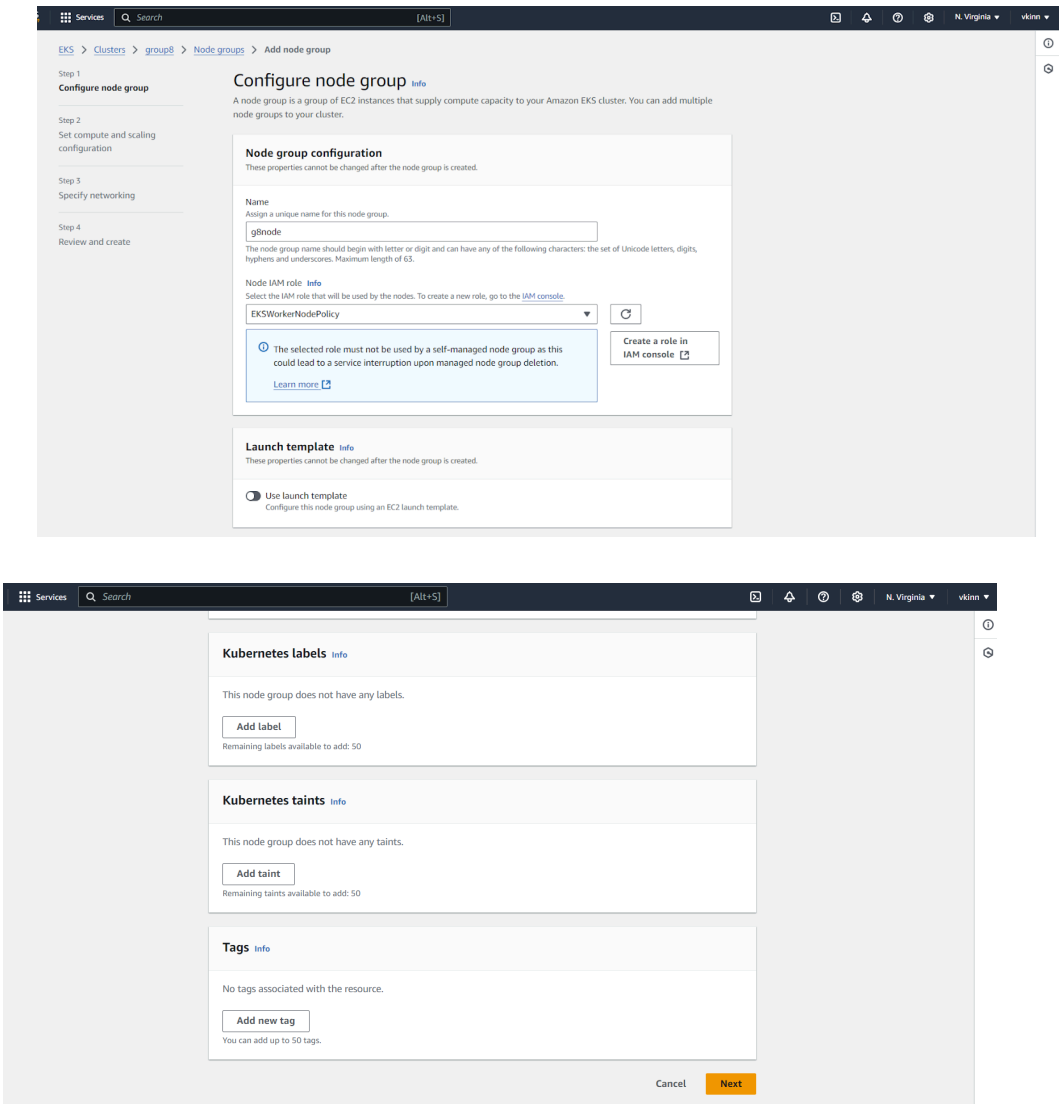

Figure 12: Configure Node Group

The IAM role - EKSWorkerNodePolicy is essential for node functionality, allowing nodes to pull container images, communicate with the Kubernetes control plane, and handle networking. It provides permissions for EC2 instances in the node group to interact with other AWS services and it ensures that nodes have the required access to perform their functions within the EKS environment.

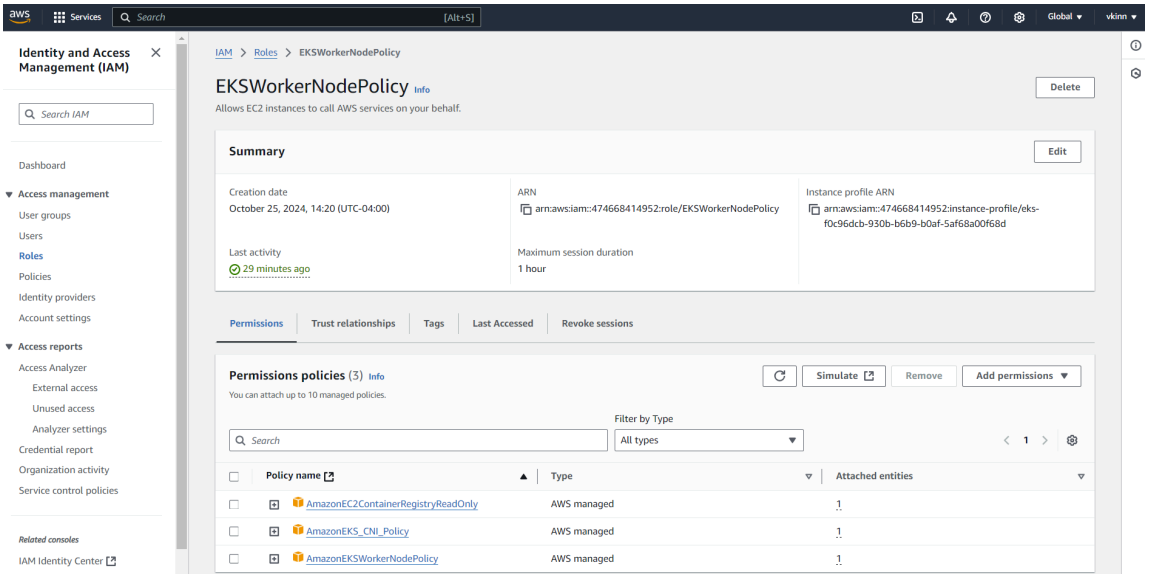

Figure 13: EKSWorkerNodePolicy

The configuration defines the compute resources and scaling policies for the node group, specifying the EC2 instance type and capacity options.

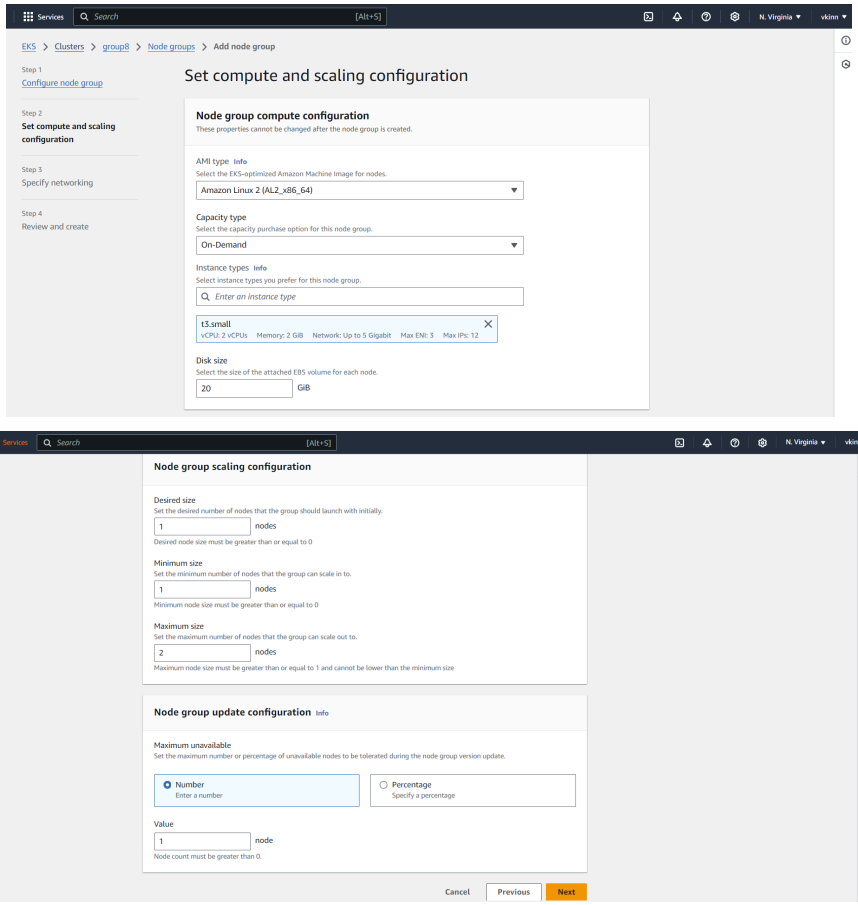

Figure 14: NodeGroup Scaling & Compute

It becomes important to link the node group to specific subnets within the VPC, ensuring that the nodes have a defined network environment.

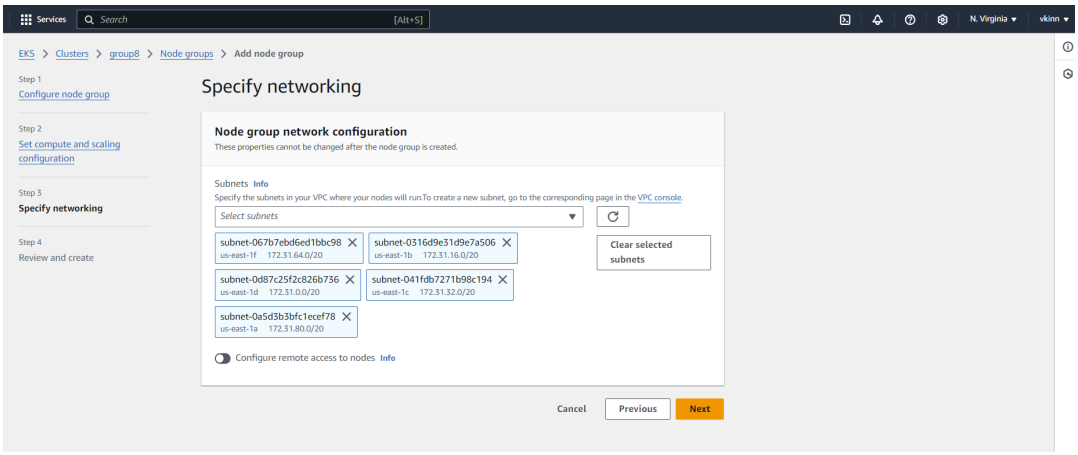

Figure 15: EKS Node Group Networking

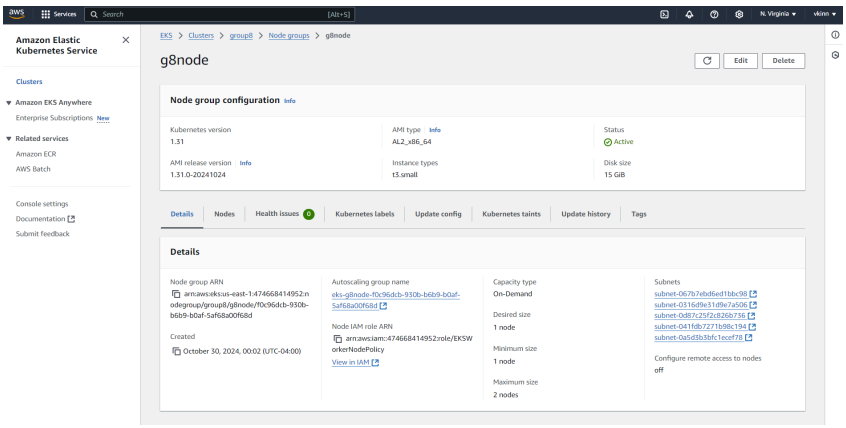

Figure 16: EKS Node Group Creation

An **EKS Node Group** provides the worker nodes to run Kubernetes applications in an EKS cluster. Node groups enable workload execution, scalability, and are managed by AWS or configured manually for greater control, essential for efficient app management.

4.1.2 Configure AWS CLI to Connect to the Cluster

“aws eks update-kubeconfig --name group8 --region us-east-1” command connects the AWS CLI to the "group8" EKS cluster in the "us-east-1" region. The output confirms that the EKS context was successfully added to the kube\config file. This allows kubectl commands to interact with the specified EKS cluster.

Figure 17: AWS CLI-Cluster Config

4.1.3 Create a Namespace in Kubernetes

Separate namespace created helps in organizing resources and managing access control, making it easier to manage multiple applications within the same Kubernetes cluster.

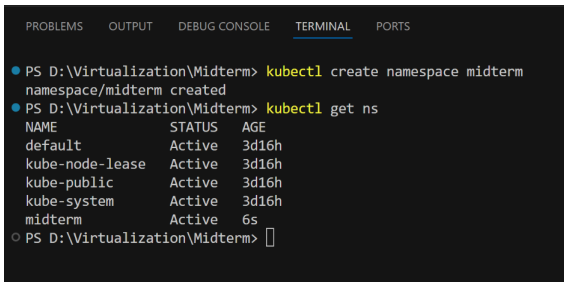

Figure 18: Kubernetes Namespace Creation

4.2 Setting up DynamoDB

4.2.1 Create a DynamoDB Table

The DynamoDB table will now be used to store appointment data. The provisioned capacity mode is beneficial for managing predictable workloads, helping control costs while providing reliable access.

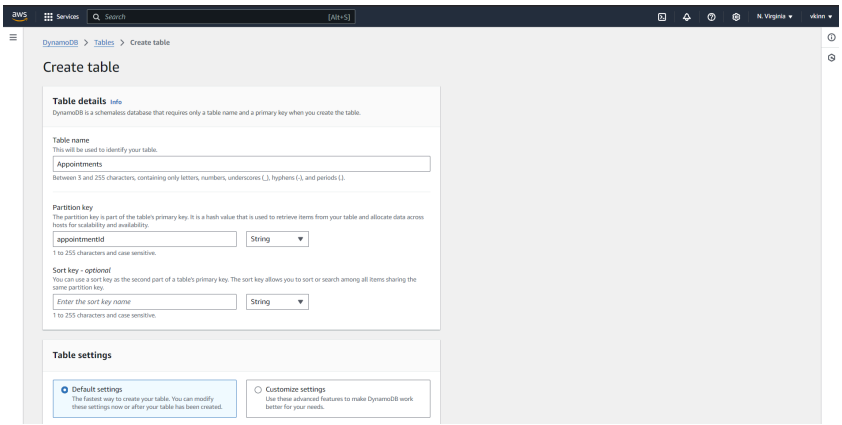

Figure 19: DynamoDB Table Creation

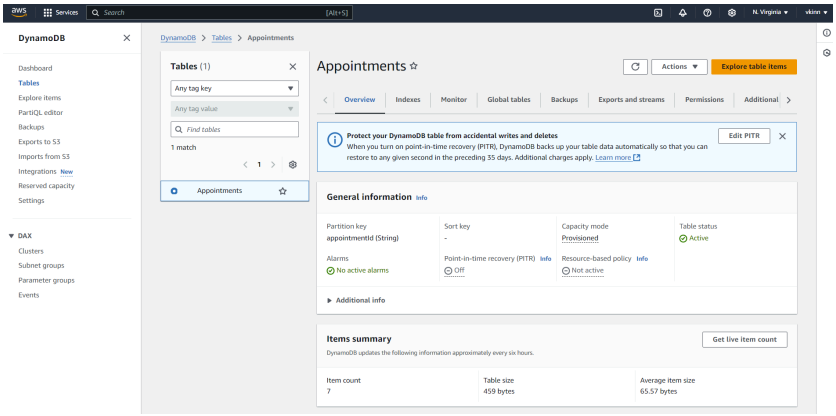

Figure 20: DynamoDB Table Overview

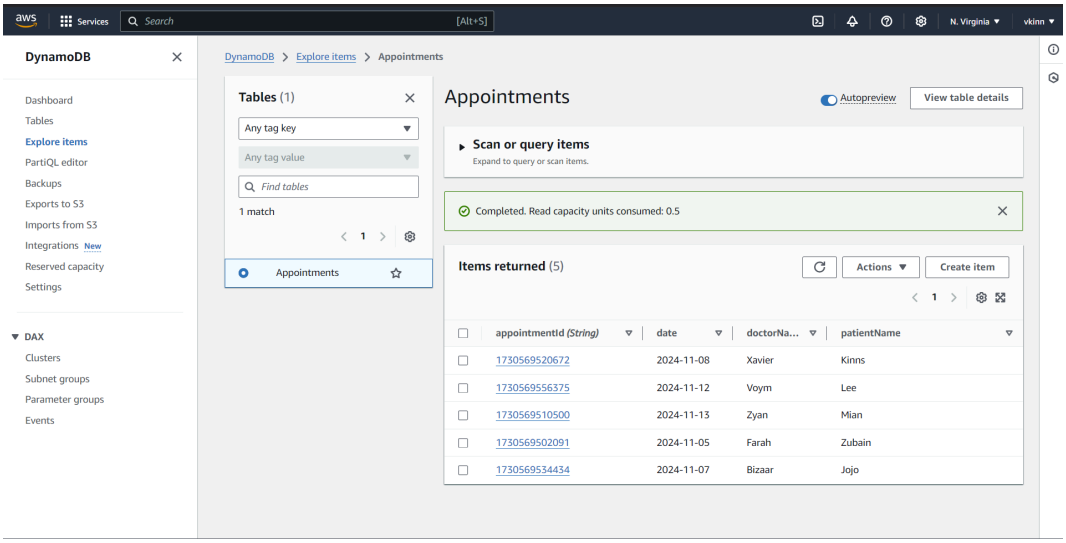

Figure 21: DynamoDB Table Items

4.2.2 Create IAM Roles and Policies

This user will have specific permissions to interact with DynamoDB, allowing database operations from the application or related services without exposing unnecessary privileges.

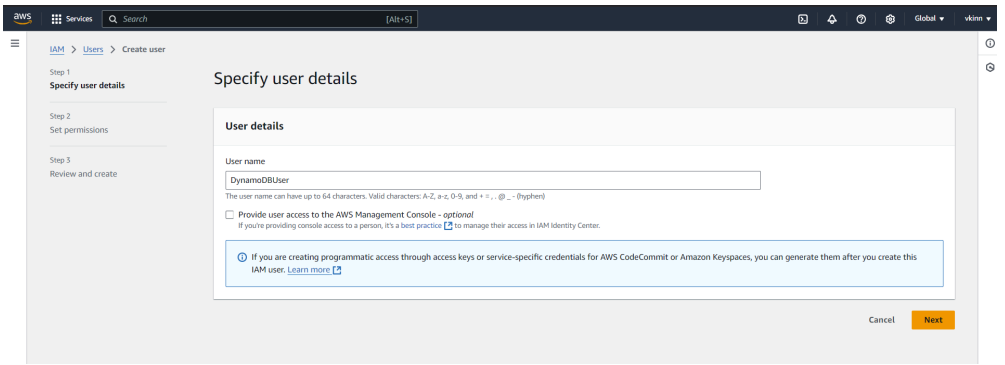

Figure 22: IAM User Creation

The permissions to the "DynamoDBUser" are set to control their access level such that the application or service using this IAM user can perform all necessary operations on the DynamoDB tables, supporting full CRUD functionality for managing appointments.

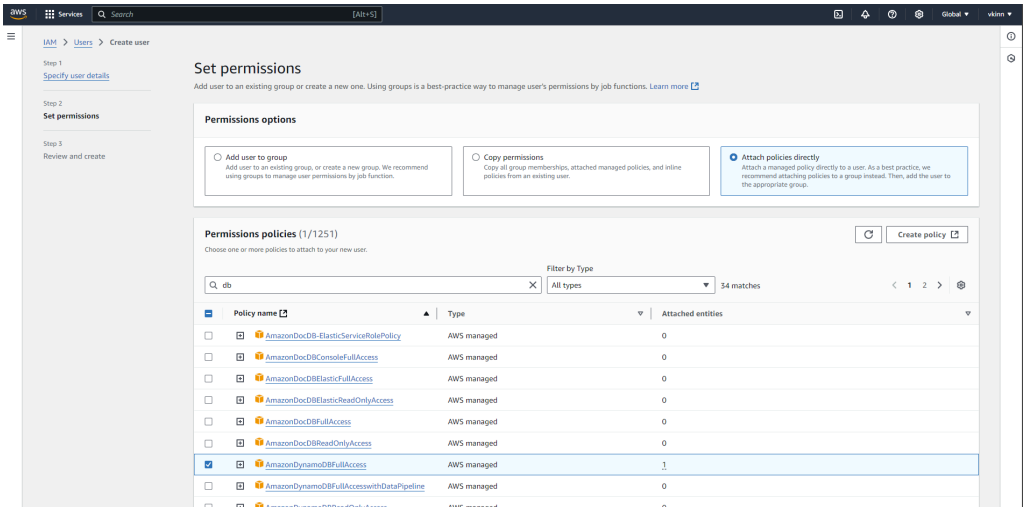

Figure 23: IAM User Permissions

This IAM user can be used to access the DynamoDB, aligning with security best practices by restricting permissions and access types based on the application's requirements.

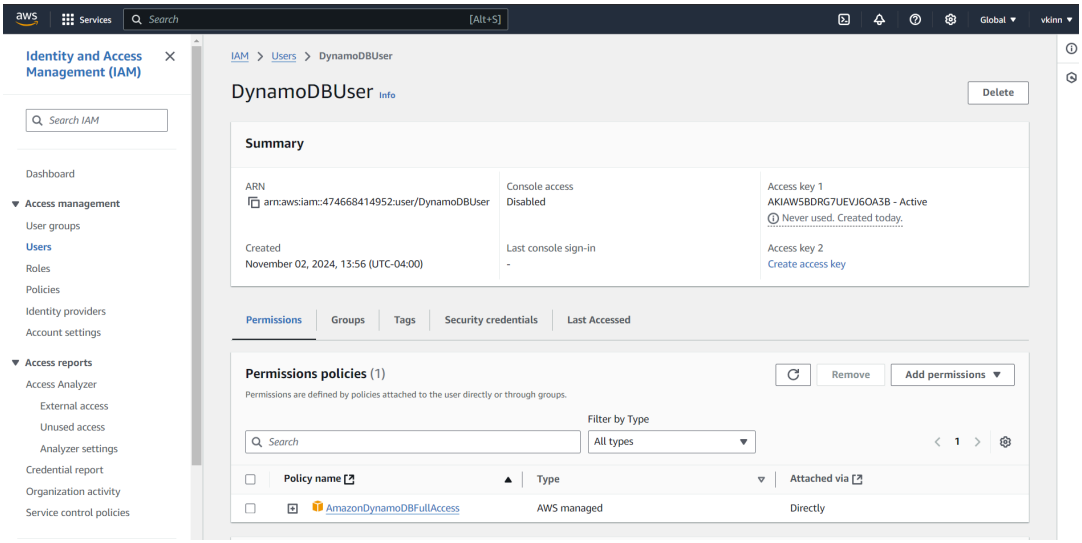

Figure 24: IAM - DynamoDB Permission Policy

4.3 Dockerizing the Application

4.3.1 Backend (Node.js + Express)

4.3.1.1 Create Dockerfile for Backend

The Dockerfile for the backend service defines the environment and dependencies required for a Node.js application with Express.

- **FROM node:14**: This command sets the base image as Node.js version 14.
- **WORKDIR /usr/src/app**: Specifies the working directory in the container.
- **COPY package.json ./**: Copies the package.json and package-lock.json files to the working directory.
- **RUN npm install**: Installs dependencies defined in package.json.
- **COPY . .**: Copies all backend files to the working directory in the container.
- **EXPOSE 5000**: Exposes port 5000 for accessing the backend service.
- **CMD ["node", "index.js"]**: Defines the command to start the application.

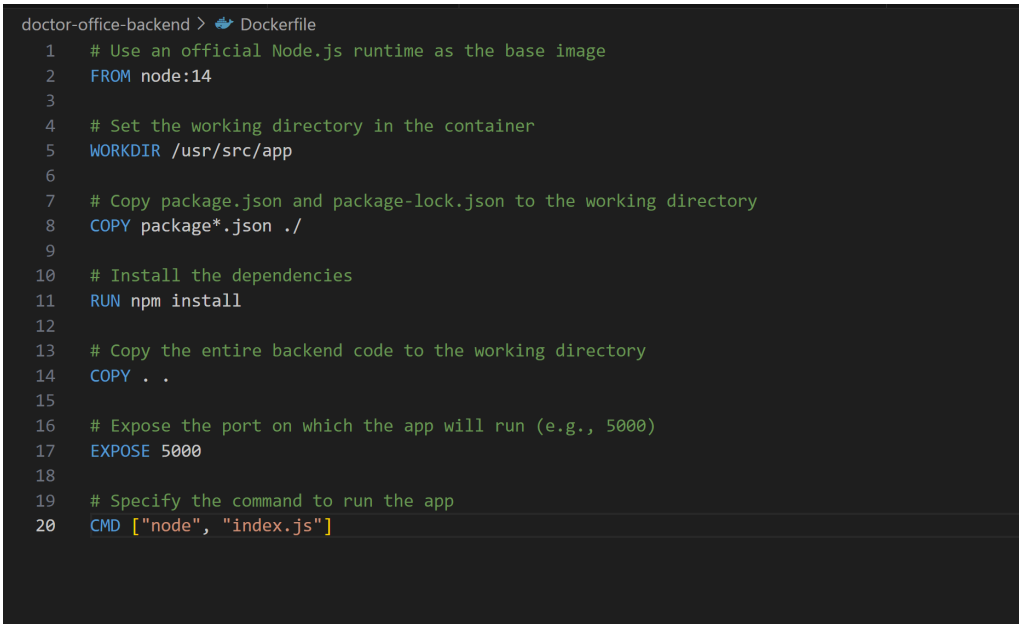

Figure 25: Dockerfile Backend

4.3.2 Frontend (React)

4.3.2.1 Create Dockerfile for Frontend

This Dockerfile is for containerizing the React frontend application. Similar to the backend, it begins with a Node.js base image, sets up the working directory, installs dependencies, and builds the React application for production. The static files are served using serve.

- **FROM node:14**: Sets the base image to Node.js version 14.
- **WORKDIR /usr/src/app**: Specifies the working directory.
- **COPY package.json ./**: Copies the dependency files.
- **RUN npm install**: Installs the frontend dependencies.
- **COPY . .**: Copies the rest of the frontend files.
- **RUN npm run build**: Builds the React application.
- **RUN npm install -g serve**: Installs serve globally to serve the static build files.
- **CMD ["serve", "-s", "build", "-l", "3000"]**: Runs the application using serve on port 3000.
- **EXPOSE 3000**: Exposes port 3000 for accessing the frontend.

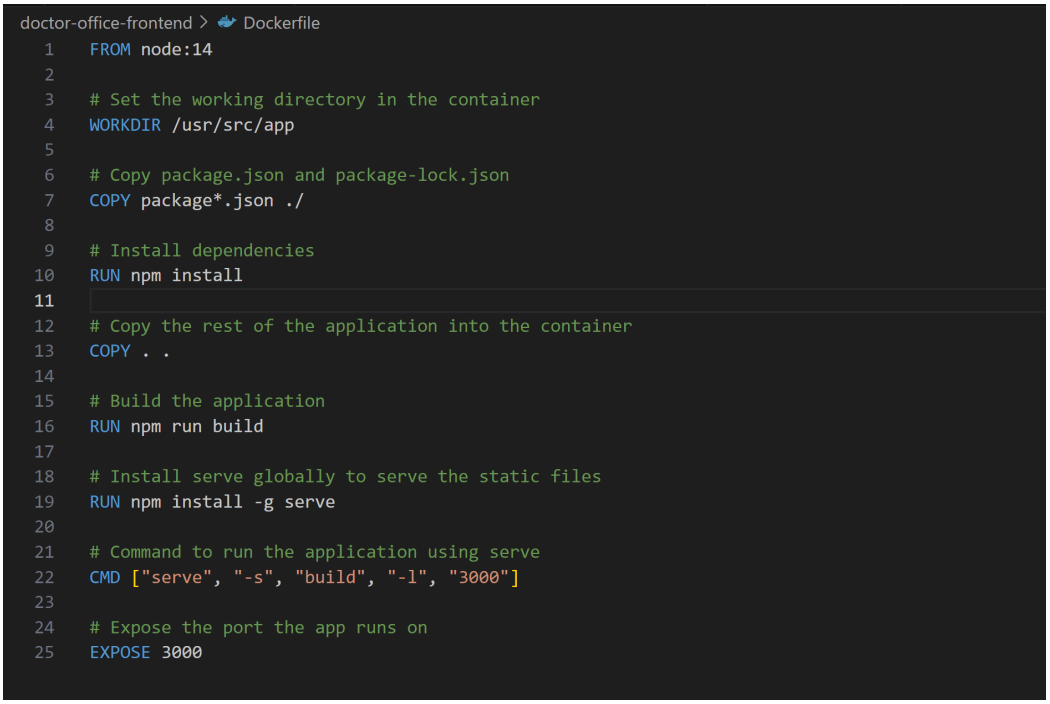

Figure 26: Dockerfile Frontend

4.3.3 Build Docker Images

The docker-compose.yaml file defines the configuration for both backend and frontend services. Each service has its own build context (from respective Dockerfiles), ports, and environment variables. It orchestrates the creation of both containers with specified configurations.

The Compose file ties both frontend and backend configurations together, enabling seamless interaction between them in a single environment setup.

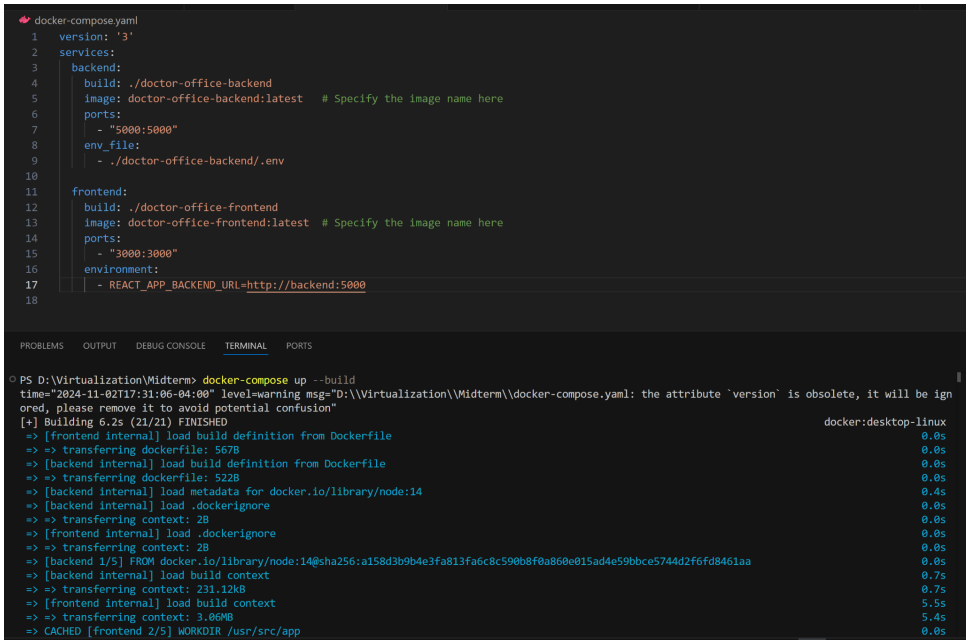

Figure 27: Docker Compose

It is verified that both Docker containers are up and running as per the Docker Compose configuration, enabling the frontend and backend to communicate effectively.

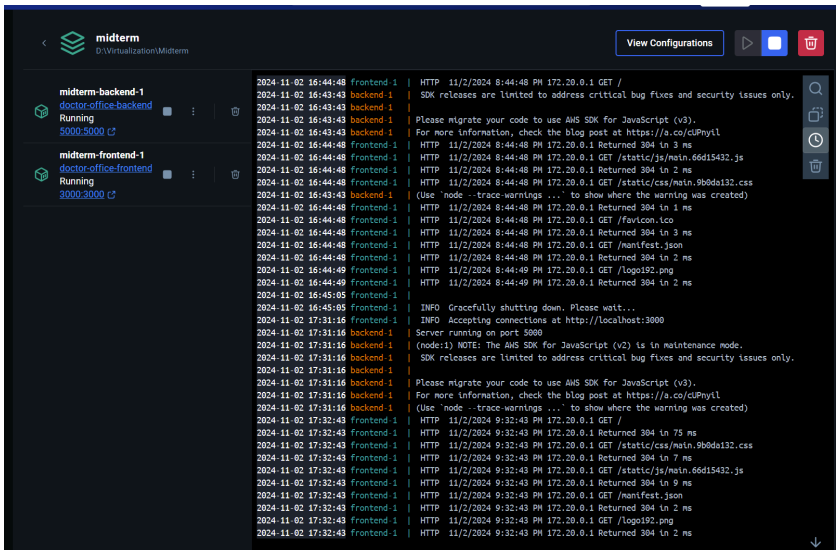

Figure 28: Docker Compose Config

4.4 Push Docker Images to AWS ECR

4.4.1 Create ECR Repositories

To manage and store the images of frontend and backend independently, we created two private registries named docker-office-frontend and docker-office-backend with the image tag set as Mutable so that the images can be overwritten with AES-256 to encrypt the images in both the repositories.

Front-end:

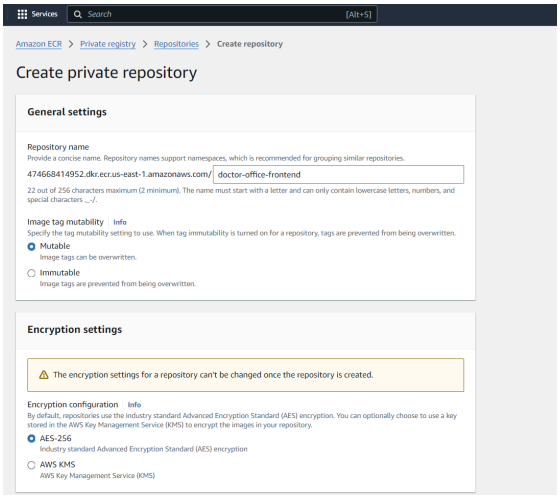

Figure 29: ECR Frontend Repository

Backend:

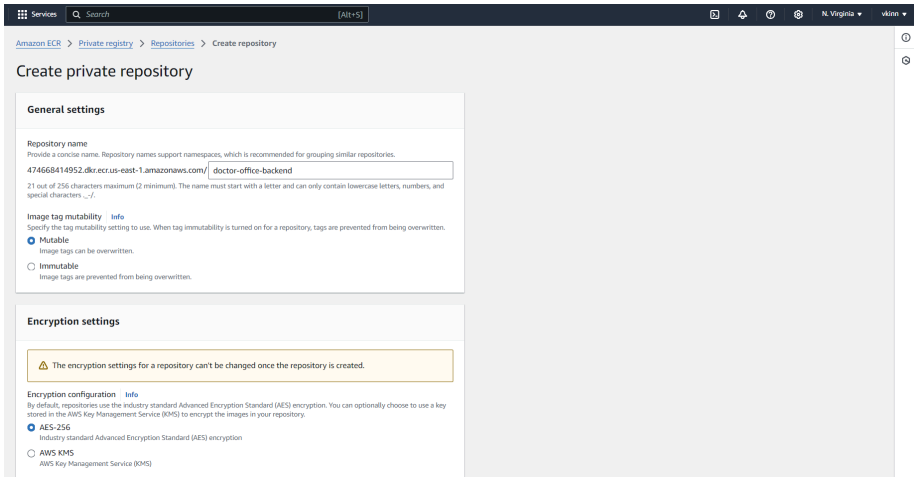

Figure 30: ECR Backend Repository

4.4.2 Tag and Push Docker Images to Amazon ECR

After the creation of the private repositories, the next step is to tag and push our local Docker images to the Amazon ECR, which is a managed container registry service, managed by AWS. Tagging the image is very useful to uniquely identify the image using its version for deployment.

So, we tag both the doctor-office-frontend and doctor-office-backend-image as a tag named latest using the tag command. Later, we push the image to the Amazon ECR which makes it accessible to the services present in our AWS environment.

Tagging and Pushing the doctor-office-frontend to the Amazon ECR:

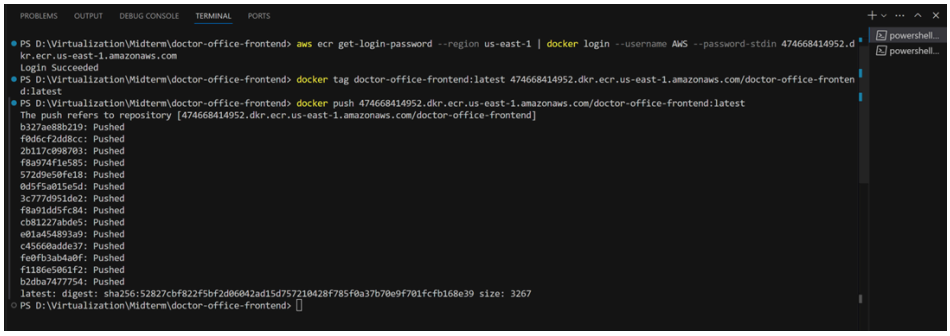

Figure 31: Tag & Push Frontend to ECR

We can verify that the image doctor-office-frontend with the tag latest is now present in the Amazon ECR.

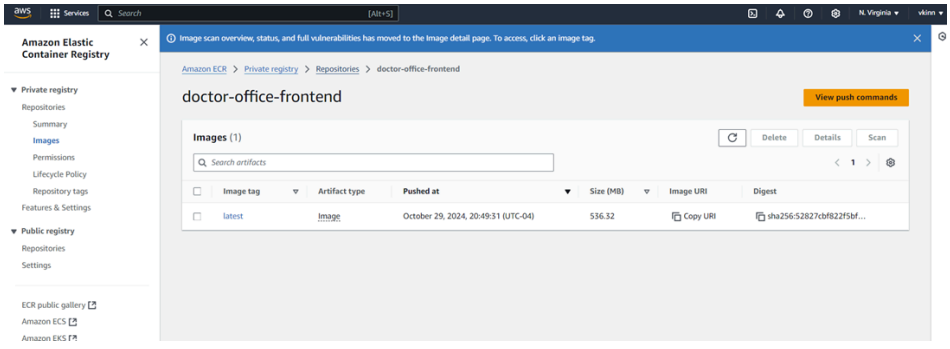

Figure 32: Frontend Image

Tagging and Pushing the doctor-office-backend to the Amazon ECR:

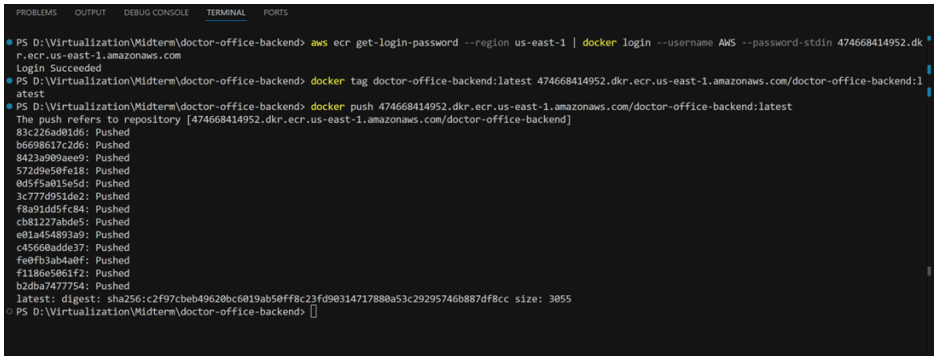

Figure 33: Tag & Push Backend to ECR

We can verify that the image doctor-office-backend with the tag latest is now present in the Amazon ECR.

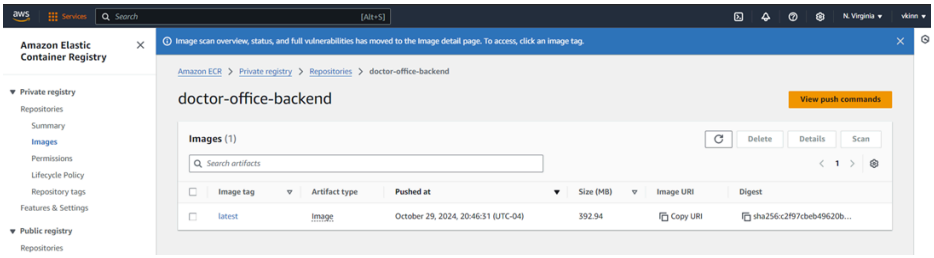

Figure 34: Backend Image

4.5 Deploy to EKS Using Kubernetes

To deploy our frontend and backend microservices that we created to the Amazon EKS (Elastic Kubernetes Service) we will be using Kubernetes deployment and service files.

4.5.1 Backend Deployment and Service Configuration

Backend - deployment.yaml file configuration

After the creation of deployment.yaml files for backend, we specify the apiVersion as apps/v1 and set the resource type as Deployment. Deployment basically manages the lifecycle of pods and helps to scale the application.

- Replicas is set to 1, which means we have one instance of the frontend application.
- We have set the CPU value of requests to 0.3 so that the container will only request 0.3% of the CPU core, and the limit is set to 0.7 so that if needed, the container can use up to 0.7% of CPU core.

- We have set the memory value of the requests to 200 Mi, which is reserved for the container to make sure it is enough for its operation, and the limit of 200 Mi is set, which is the maximum memory that the container can use.
- We have set the containerPort to 5000 to access the backend.

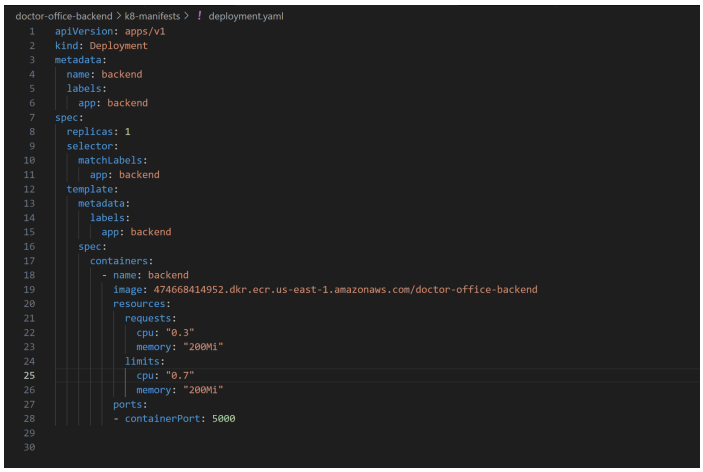

Figure 35: Backend Deployment File

Backend - service.yaml file configuration

The service.yaml file helps to make the backend application accessible to the external traffic.

- We achieve that by setting the spec type to LoadBalancer, which provides an external IP address so that the users can access our application outside the Kubernetes cluster.
- We set the protocol as TCP and the port number as 80, which the users will access via the LoadBalancer for our backend application.
- We set the targetPort value to 5000 for the frontend application.

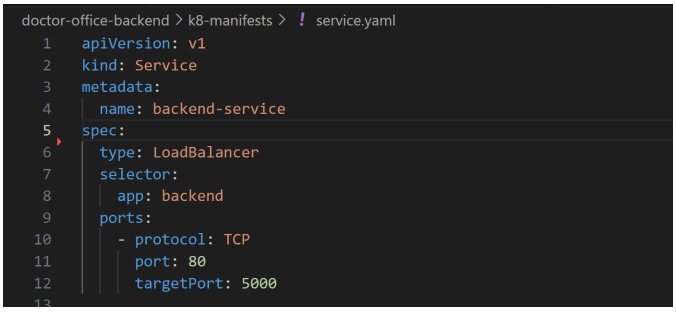

Figure 36: Backend Service File

4.5.2 Frontend Deployment and Service Configuration

Frontend - deployment.yaml file configuration

Now we create the deployment.yaml file for frontend, we specify the apiVersion as apps/v1 and set the resource type as Deployment. Deployment basically manages the lifecycle of pods and helps to scale the application.

- Replicas is set to 1, which means we have one instance of the frontend application.
- We have set the CPU value of requests to 0.3 so that the container will only request 0.3% of the CPU core, and the limit is set to 0.7 so that if needed, the container can use up to 0.7% of CPU core.
- We have set the memory value of the requests to 200 Mi, which is reserved for the container to make sure it is enough for its operation, and the limit of 200 Mi is set, which is the maximum memory that the container can use.
- We have set the containerPort to 3000 to access the frontend.

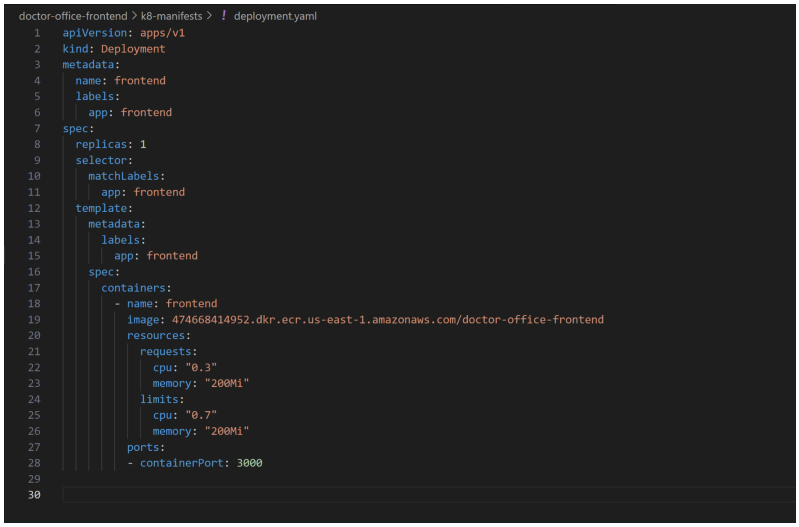

Figure 37: Frontend Deployment File

Frontend - service.yaml file configuration

The service.yaml file helps to make the frontend application accessible to the external traffic.

- We achieve that by setting the spec type to LoadBalancer, which provides an external IP address so that the users can access our application outside the Kubernetes cluster.
- We set the protocol as TCP and the port number as 80, which the users will access via the LoadBalancer for our frontend application.
- We set the targetPort value to 3000 for the frontend application.

doctor-office-frontend > k8-manifests > ! service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  type: LoadBalancer
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

Figure 38: Frontend Service File

4.5.3 Apply Kubernetes Configuration Files

Now we use the kubectl apply command to apply all the configuration files for deployment of both the frontend and backend.

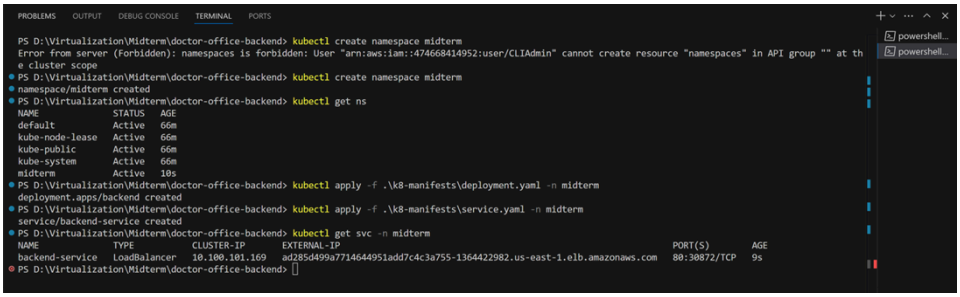

Figure 39: Update Backend using kubectl

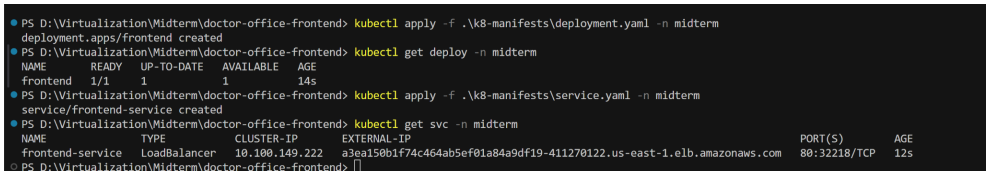

Figure 40: Update Frontend using kubectl

4.5.4 Verify Kubernetes Services

With the help of kubectl get svc -n mindterm we can notice that the frontend-service and backend-service are now running.

```
PS D:\Virtualization\Midterm\doctor-office-backend> kubectl apply -f .\k8-manifests\deployment.yaml -n midterm
deployment.apps/backend created
PS D:\Virtualization\Midterm\doctor-office-backend> kubectl get deploy -n midterm
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
backend    1/1     1            1           21s
frontend   1/1     1            1           3m3s
PS D:\Virtualization\Midterm\doctor-office-backend> kubectl apply -f .\k8-manifests\service.yaml -n midterm
service/backend-service created
PS D:\Virtualization\Midterm\doctor-office-backend> kubectl get svc -n midterm
NAME              TYPE           CLUSTER-IP       EXTERNAL-IP                                                               PORT(S)        AGE
backend-service   LoadBalancer   10.100.171.210   a2899bf786be14a8d8975e90ca17448c-741777150.us-east-1.elb.amazonaws.com   80:30814/TCP   13s
frontend-service  LoadBalancer   10.100.149.222   a3ea150b1f74c464ab5ef01a84a9df19-411270122.us-east-1.elb.amazonaws.com   80:32218/TCP   2m20s
PS D:\Virtualization\Midterm\doctor-office-backend>
```

Figure 41: Verify Service Status

We can verify that both frontend-service and backend-service are correctly deployed, running and accessible via Kubernetes service in the Amazon EKS cluster.

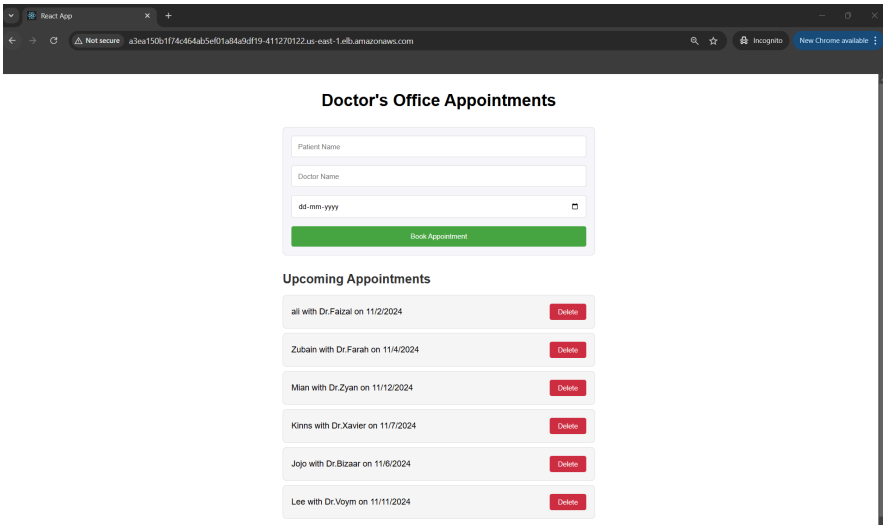

Figure 42: Appointment WebPage

5. CI/CD Pipeline

5.1 Setting up GitHub Actions

Setting up GitHub Actions involves creating a workflow YAML file in your repository's .github/workflows directory. This file defines the CI/CD pipeline, specifying actions like checking out the latest code, building Docker images, pushing them to a container registry, and deploying updates to your EKS cluster automatically on code changes.

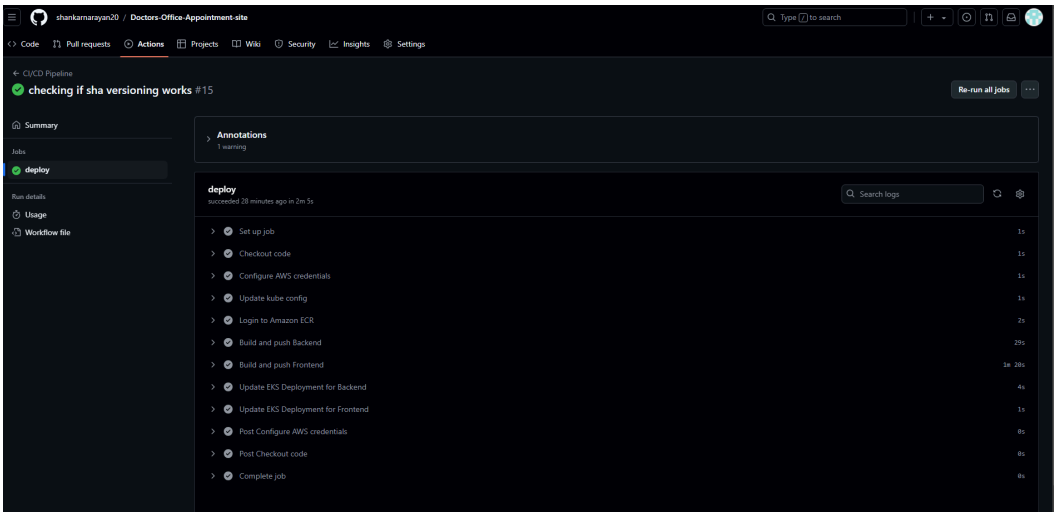

Figure 43: GitHub Actions Setup

5.2 Automating Deployments

This GitHub Actions workflow, Docker Build and Push to ECR, is set up to automatically build and push Docker images for both the backend and frontend whenever code is pushed to the main branch. Here's how it works:

- **Checkout Repository code**: It first checks out the latest code from the repository.
- **Configuration of AWS credentials:** Next step, it checks if the AWS credentials are configured using stored secrets for secure access.
- **Login into ECR**: The workflow logs in to Amazon ECR, so it has permission to push images.
- **Build and Push Backend Image**: It builds the backend Docker image from the ./doctor-office-backend folder, tags it with the current commit ID, and pushes it to the ECR repository.
- **Build and Push Frontend Image:** Finally, it does the same for the frontend image from ./doctor-office-frontend.
- **Update the image in the EKS Deployment:** The workflow updates the Kubernetes deployments in EKS with the newly pushed images, using kubectl to set the new image versions for both frontend and backend deployments, ensuring the application is running with the latest code.

With this workflow, every update to the main branch gets a fresh backend and frontend Docker image in ECR, ready for deployment.

```yaml
jobs:
  deploy:
    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Configure AWS credentials
      uses: aws-actions/configure-aws-credentials@v2
      with:
        role-to-assume: arn:aws:iam::156041403412:role/github-actions-role
        aws-region: ${{ env.AWS_REGION }}

    - name: Update kube config
      run: |
        aws eks update-kubeconfig --name ${{ env.CLUSTER_NAME }} --region ${{ env.AWS_REGION }}

    - name: Login to Amazon ECR
      run: |
        aws ecr get-login-password --region $AWS_REGION | docker login --username AWS --password-stdin $ECR_REGISTRY      You, 4 hours ago • U

    # Build and push Docker images for Backend
    - name: Build Docker image for Backend
      run: |
        docker build -t $ECR_REPOSITORY_1:latest -f doctor-office-backend/Dockerfile doctor-office-backend/
    - name: Tag and push Docker image for Backend
      run: |
        docker tag $ECR_REPOSITORY_1:latest $ECR_REGISTRY/$ECR_REPOSITORY_1:latest
        docker push $ECR_REGISTRY/$ECR_REPOSITORY_1:latest

    # Build and push Docker image for Frontend
    - name: Build Docker image for Frontend
      run: |
        docker build -t $ECR_REPOSITORY_2:latest -f doctor-office-frontend/Dockerfile doctor-office-frontend/
    - name: Tag and push Docker image for Frontend
      run: |
        docker tag $ECR_REPOSITORY_2:latest $ECR_REGISTRY/$ECR_REPOSITORY_2:latest
        docker push $ECR_REGISTRY/$ECR_REPOSITORY_2:latest
```

```yaml
    - name: Update EKS Deployment for Backend
      run: |
        kubectl set image deployment/backend backend=$ECR_REGISTRY/$ECR_REPOSITORY_1:latest
    - name: Update EKS Deployment for Frontend
      run: |
        kubectl set image deployment/frontend frontend=$ECR_REGISTRY/$ECR_REPOSITORY_2:latest
```

Figure 44: Automating Deployments

6. Monitoring and Logging

6.1 AWS CloudWatch Setup

We are setting up AWS CloudWatch for monitoring and logging in the EKS environment by configuring the EKS cluster and installing necessary add-ons like CloudWatch Observability, CloudWatch can collect and display logs and metrics from the Kubernetes nodes and pods. This setup helps maintain cluster health and aids in identifying performance issues or resource bottlenecks by providing a single pane of glass for monitoring Kubernetes workloads.

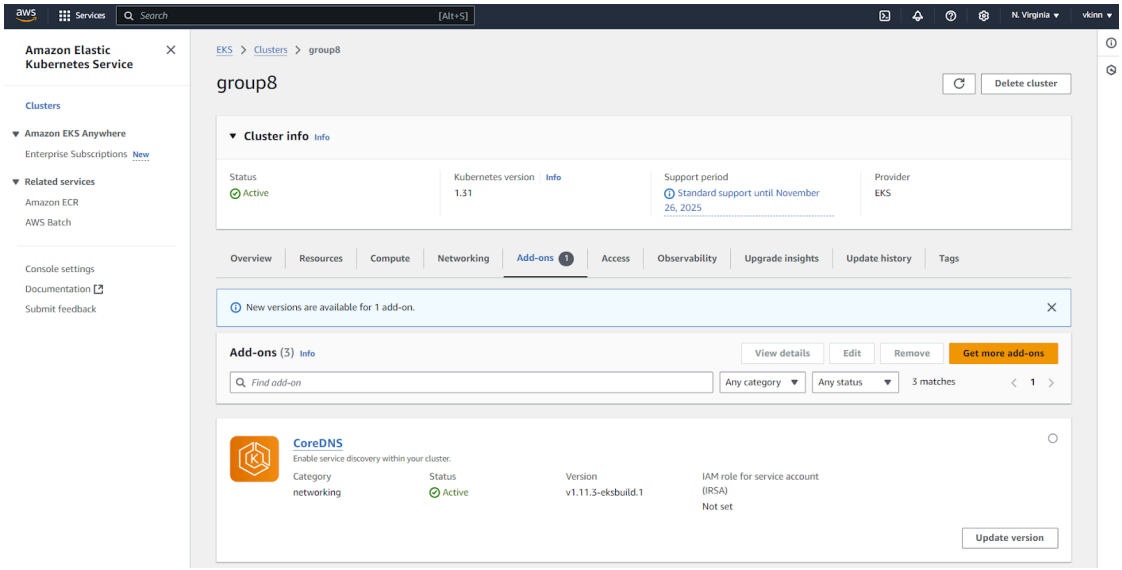

Figure 45: EKS Cluster Add-Ons

The purpose of selecting these add-ons is to extend the cluster's functionality. For instance, Amazon CloudWatch Observability is chosen here to enable monitoring and logging features directly within CloudWatch, enhancing visibility into cluster operations and performance.

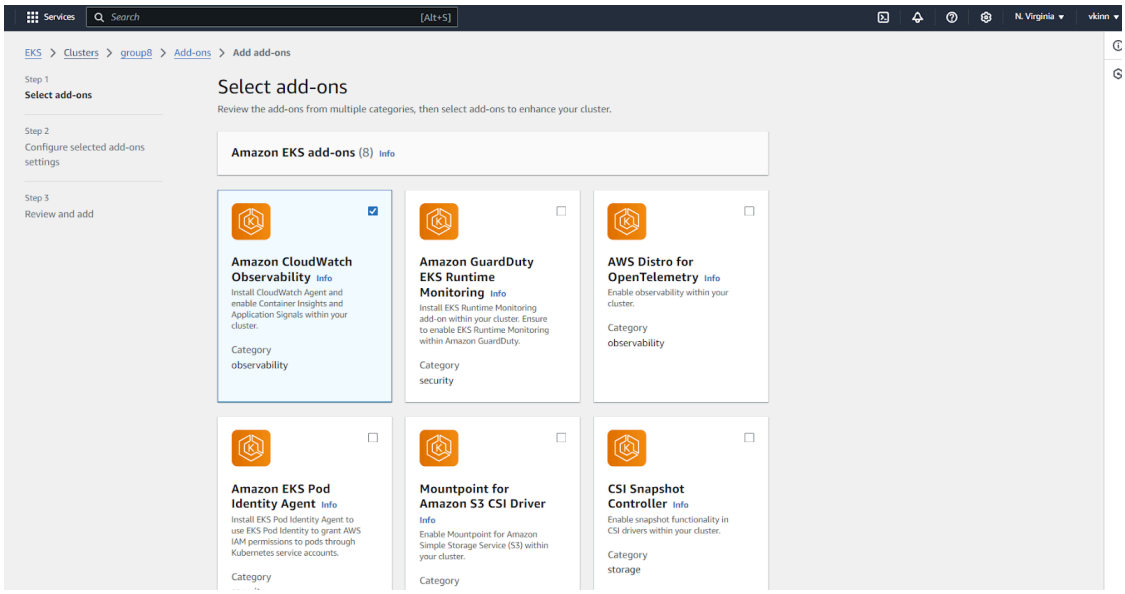

Figure 46: EKS - CloudWatch Add-On

The Amazon CloudWatch Observability add-on is selected, which enables Container Insights and Application Signals. This is part of the process of configuring CloudWatch to monitor the EKS cluster.

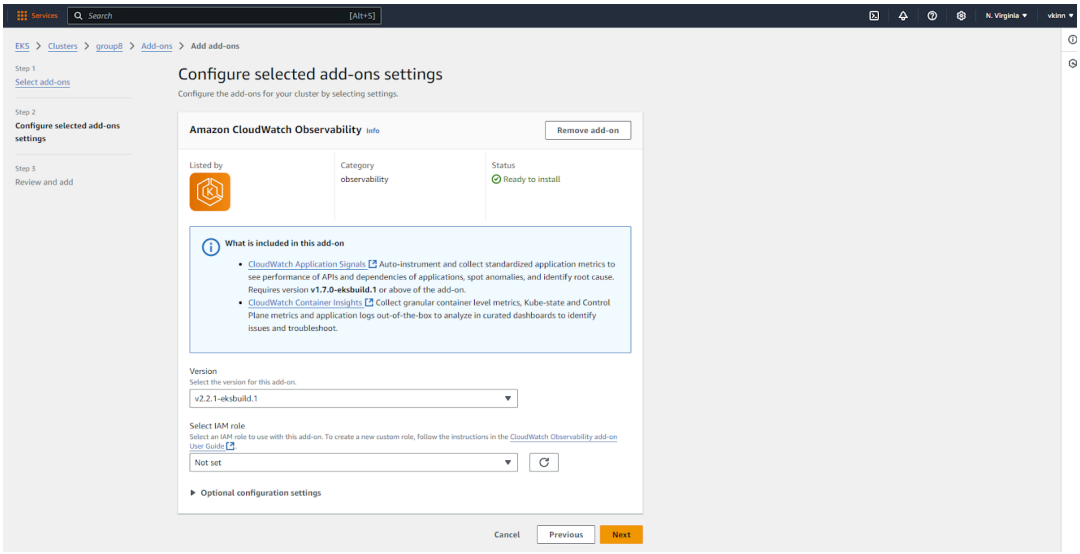

Figure 47: Configure CloudWatch

Reviewing the settings allows us to verify that CloudWatch will start collecting relevant data from the selected add-on. This step confirms that observability is ready for deployment, ensuring monitoring is correctly set up.

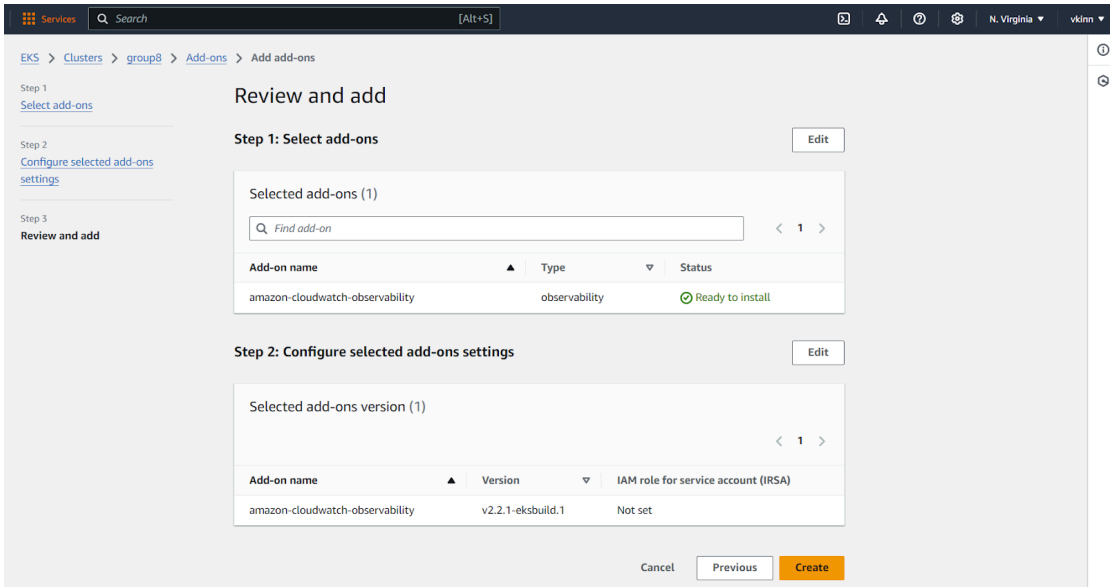

Figure 48: Review CloudWatch Config

The Application Signals dashboard is central to monitoring application-level performance metrics in real time. It will show if any services experience faults, helping you maintain application reliability by identifying problem areas within EKS workloads.

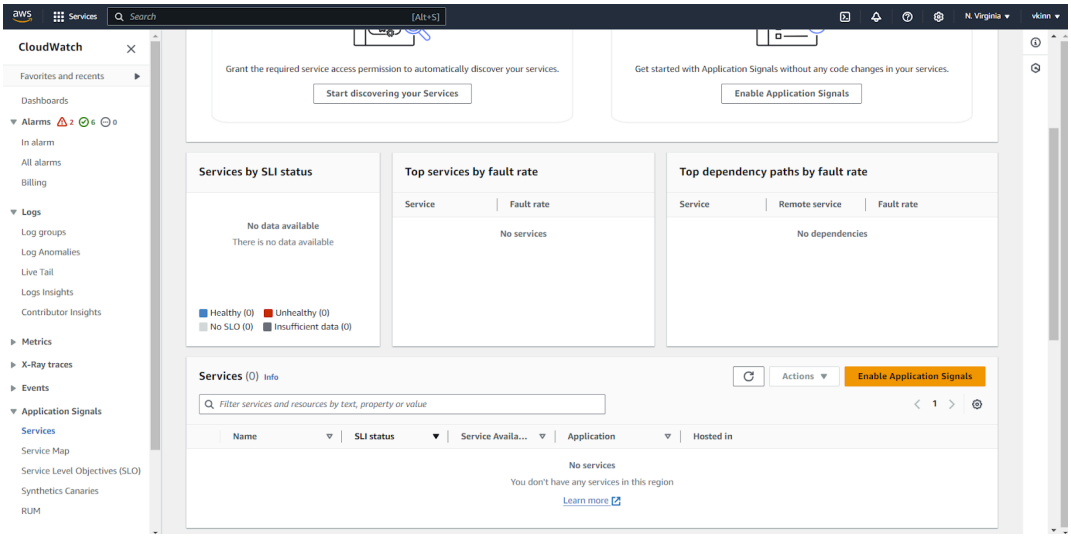

Figure 49: Enable Application Signal

This confirmation shows that the Observability add-on is successfully integrated with the EKS cluster. This step ensures that the CloudWatch Observability setup is active and ready to collect telemetry data from the EKS-hosted applications.

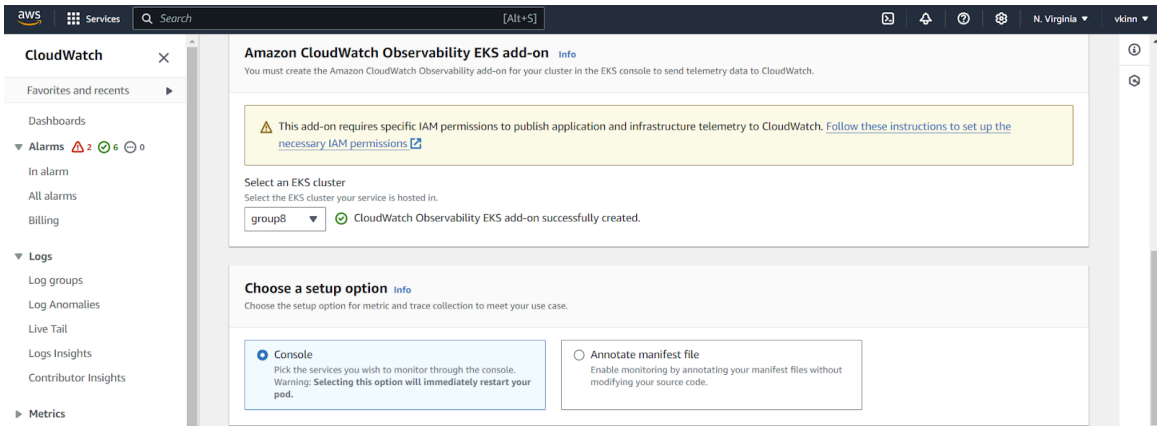

Figure 50: CloudWatch Observability EKS Add-On

Selecting namespaces and workloads allows to narrow down the monitoring focus on critical components, like the frontend and backend microservices deployed in the "midterm" namespace. This customization helps in optimizing the monitoring scope and resource usage.

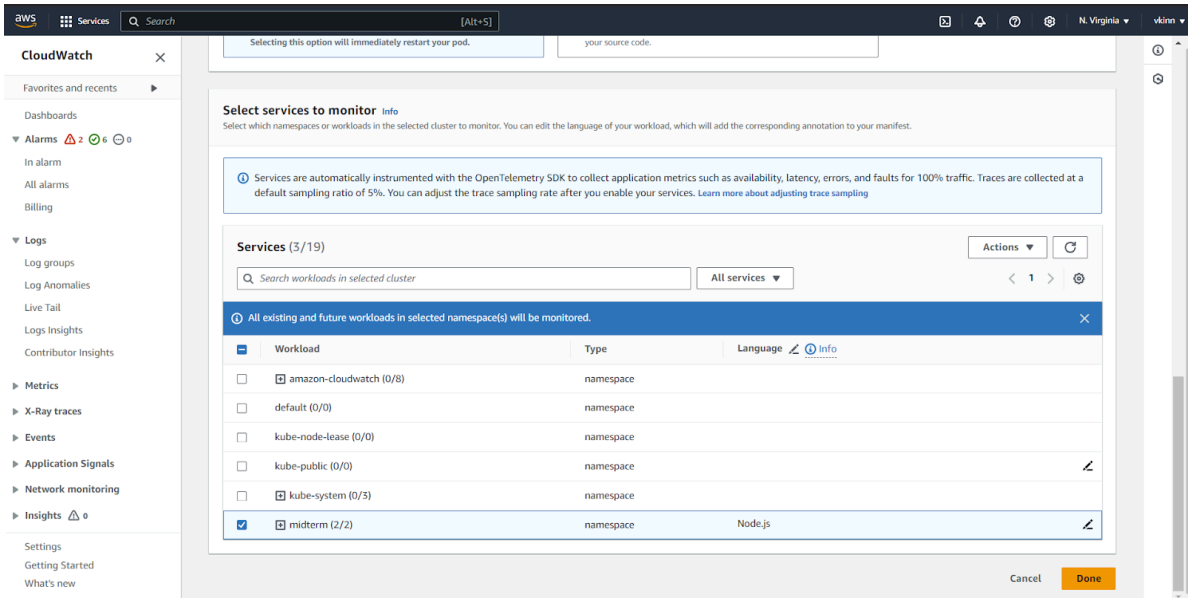

Figure 51: CloudWatch Service Selection

Container Insights provides an overview of resource usage and health status across the EKS cluster. This dashboard is crucial for understanding the overall performance and spotting resource bottlenecks or issues in the deployed services.

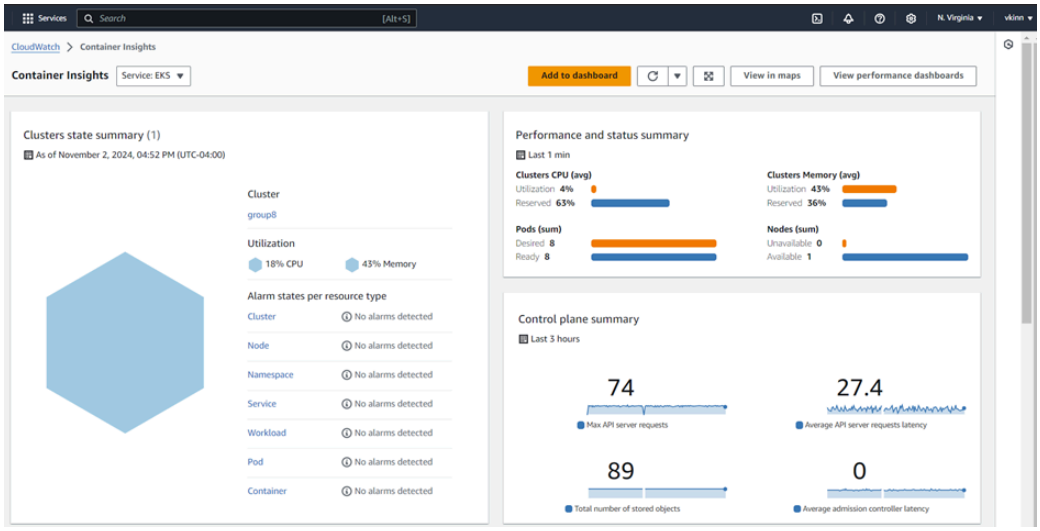

Figure 52: CloudWatch Container Insights

This gives an overview of the resource utilization and status of the group8 EKS cluster, focusing on node and pod performance metrics. It shows the overall health of nodes, including CPU and memory usage, and highlights any container restarts. Additionally, it details pod-level CPU and memory utilization, helping identify any resource-intensive pods. This view aids in monitoring cluster performance and resource allocation.

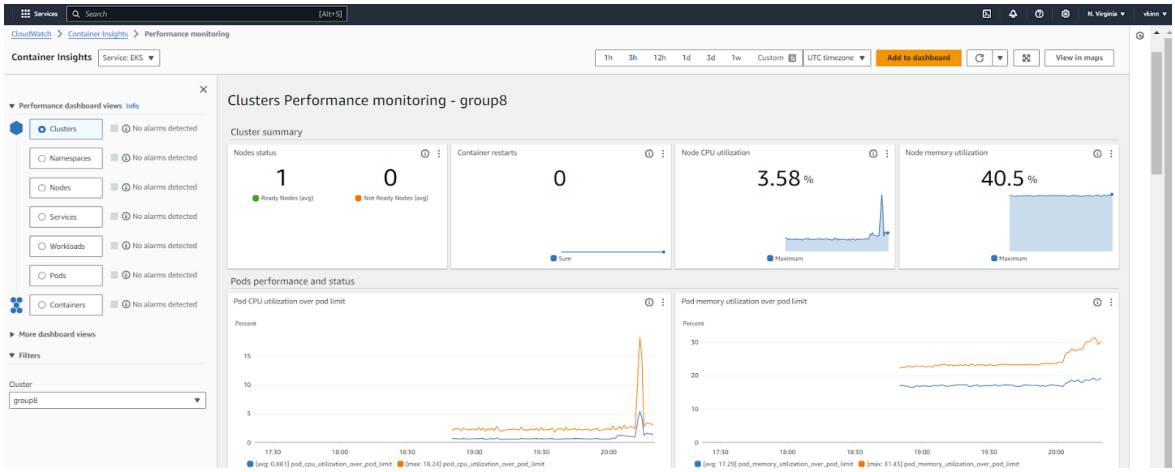

Figure 53: CloudWatch Cluster Performance Dashboard 1

This displays detailed pod-level metrics for the cluster, focusing on resource utilization and network activity. It shows pod CPU and memory usage relative to set limits, helping identify any pods close to or exceeding their allocated resources. Additionally, it tracks network traffic in terms of received and transmitted bytes, along with container restart counts. This information assists in diagnosing performance bottlenecks and monitoring resource-intensive applications within the cluster.

Figure 54: CloudWatch Cluster Performance Dashboard 2

This focuses on API server requests and performance where key metrics include API request counts, request duration, REST client requests, and ETCD request durations. These measurements help assess the responsiveness and load on the API server, as well as the performance of the

ETCD database, crucial for maintaining cluster state. Monitoring these metrics enables proactive troubleshooting of latency issues and ensures stable cluster management.

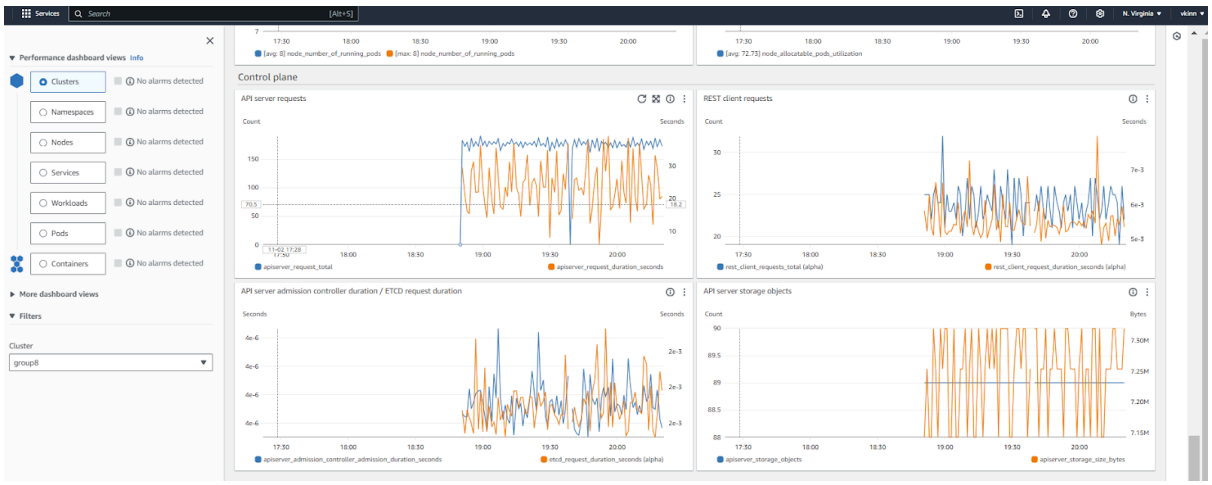

Figure 55: CloudWatch Cluster Performance Dashboard 3

It displays the number of healthy and failed nodes, the count of pods per node, CPU and memory utilization percentages, and performance metrics for each node. This helps in monitoring the resource allocation and health status of nodes, ensuring optimal performance and identifying potential issues with CPU or memory usage across the cluster nodes.

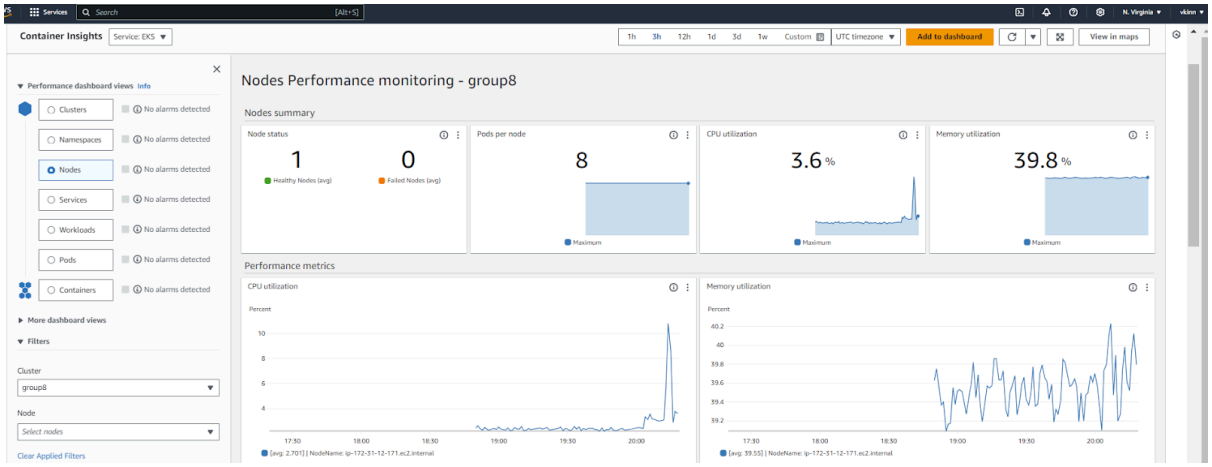

Figure 56: CloudWatch Node Performance Dashboard 1

This provides insights into Disk Utilization, Network Utilization, and Node Status Metrics for the group8 cluster in AWS CloudWatch. It monitors disk usage percentage, network bytes per second, the number of running pods, and container count across the cluster nodes.

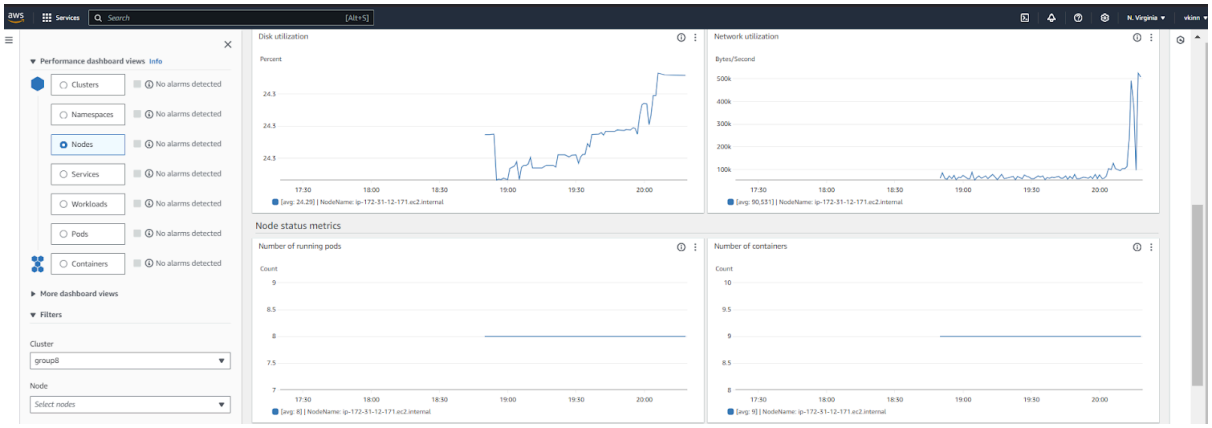

Figure 57: CloudWatch Node Performance Dashboard 2

This highlights metrics such as the number of containers (9), running containers, CPU utilization (0.97%), and memory utilization (4.59%) for pods. Detailed graphs display the pod's CPU and memory usage trends, helping monitor resource consumption and identifying any spikes or inefficiencies in pod operations. This view assists in evaluating pod performance and ensuring the cluster operates within the desired resource limits.

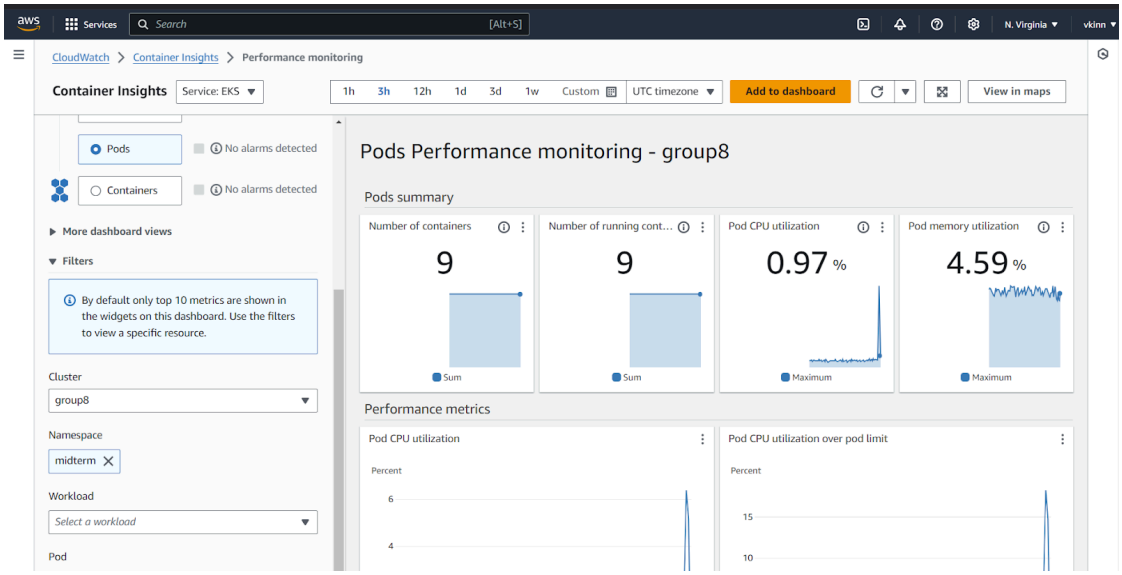

Figure 58: CloudWatch Pod Performance Dashboard 1

This provides a detailed view of Pod Performance Metrics for the group8 cluster, focusing on specific metrics within the midterm namespace. Key metrics include Pod CPU and Memory Utilization (tracking the resource consumption over time), Pod CPU and Memory Utilization Over Pod Limit (identifying potential over-usage), and Network RX/TX (monitoring received and transmitted network data). Each metric is displayed separately for backend and frontend pods, helping to assess their individual resource usage and network performance within the cluster.

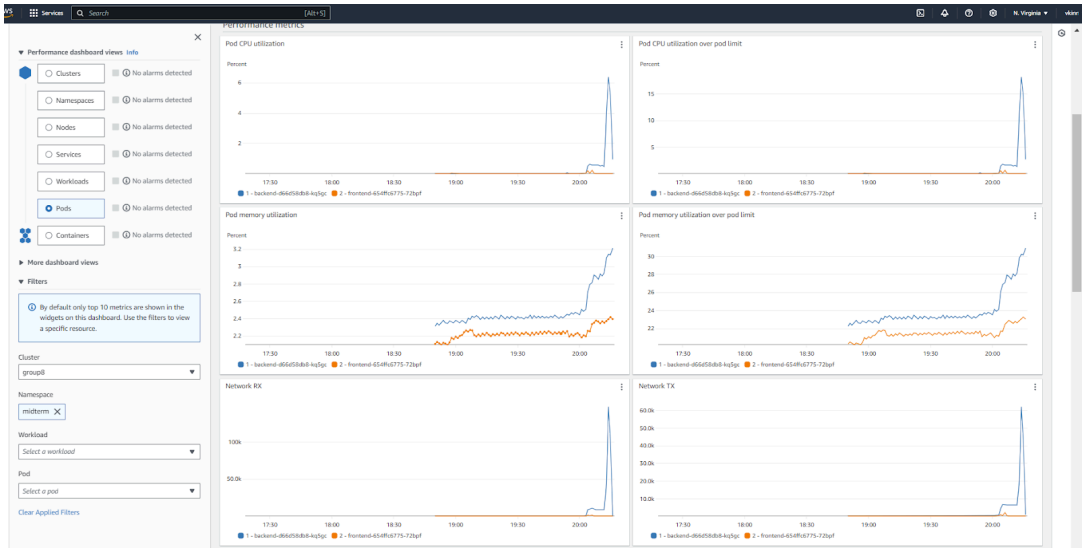

Figure 59: CloudWatch Pod Performance Dashboard 2

6.2 Setting up Alerts

This shows the Top 10 Nodes by CPU Utilization and Top 10 Nodes by Memory Utilization. Displays alerts set by AWS CloudWatch for monitoring CPU and memory utilization of my EKS cluster nodes. The high utilization threshold is set to 80% for both CPU and memory, indicating when the node usage exceeds this percentage. These alerts help track resource usage and ensure that the application does not experience performance degradation due to resource constraints. Below the graphs, the Clusters Overview Table summarizes the overall status of the cluster, showing metrics such as maximum CPU and memory usage.

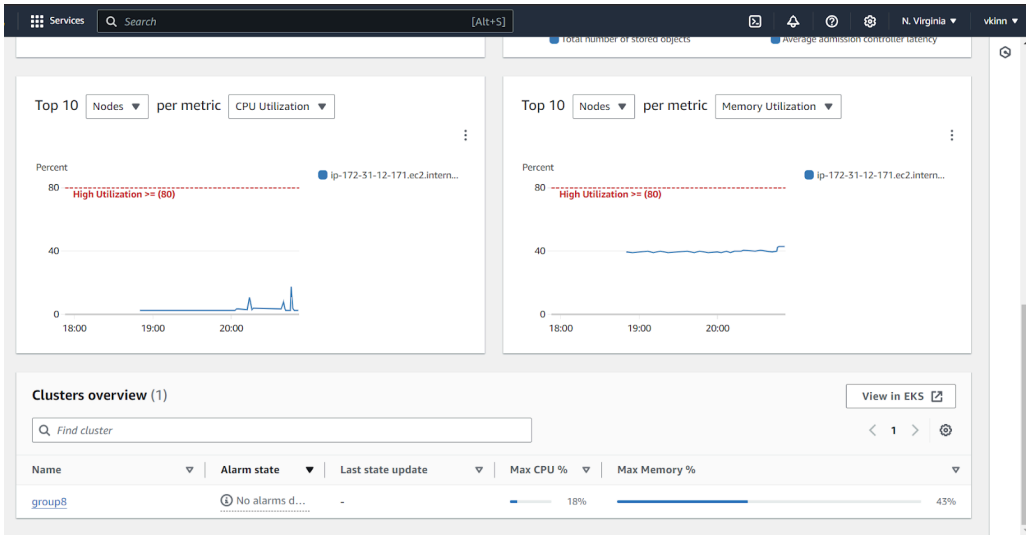

Figure 60: CloudWatch Clusters Overview

7. Testing and Scaling

7.1 Testing Microservices

We have used Postman to make HTTP requests to our backend microservice endpoint – appointments. The response body includes our entry details like appointmentId, patientName, and doctorName in the JSON format, and the status code of 200 OK confirms that the request was successful.

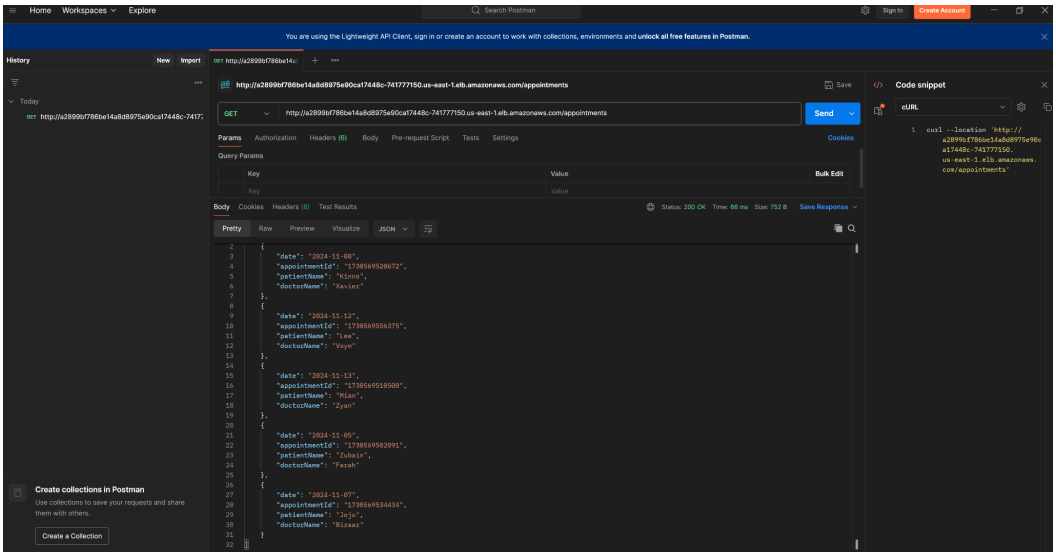

Figure 61: Postman HTTP Response

To monitor the deployed resources inside the EKS cluster, we use the command kubectl get all -n midterm.

- **Pods**: We can view all the pods along with their status. In our case, we can see that there are two pods running.
- **Services**: All the services that are associated with each of the microservices are displayed along with their type, e.g. LoadBalancer for external access. We have two services, the frontend and backend, that are running.
- **Deployment and ReplicaSets**: The deployment configurations and replicas are displayed, which ensures that the application is configured for the intended level of redundancy.

So we can confirm that all Kubernetes resources are deployed correctly, running, and are in a healthy state.

```
C:\Users\visha>kubectl get all -n midterm
NAME                                 READY   STATUS    RESTARTS   AGE
pod/backend-5b58b4554-s9p9k          1/1     Running   0          38s
pod/frontend-6cf554b48d-mk6xk        1/1     Running   0          46s

NAME                       TYPE           CLUSTER-IP       EXTERNAL-IP                                                                PORT(S)        AGE
service/backend-service    LoadBalancer   10.100.230.164   a0f108ccf3a7c4eafa7445cf7f2601c7-1257704000.us-east-1.elb.amazonaws.com   80:30654/TCP   4h20m
service/frontend-service   LoadBalancer   10.100.111.155   ad2a1a7adb2604d91bc46d8b48f29ef1-1548369576.us-east-1.elb.amazonaws.com   80:31539/TCP   4h20m

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend              1/1     1            1           7m17s
deployment.apps/frontend             1/1     1            1           7m25s

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-5b58b4554          1         1         1       38s
replicaset.apps/backend-6dc86d96f6         0         0         0       7m17s
replicaset.apps/frontend-6cf554b48d        1         1         1       46s
replicaset.apps/frontend-6f69fd57f8        0         0         0       7m25s

C:\Users\visha>
```

Figure 62: Kubernetes Resources Status

7.2 Scaling the Application

With the addition of 2 replicas for each service (backend and frontend) resulting in a total of 4 pods (2 for backend, 2 for frontend). The allocated CPU and memory resources are effectively utilized to test scalability. Each replica maintains its requested CPU and memory resources, ensuring that Kubernetes can efficiently manage the load while maintaining the desired replica count. This configuration enables better scalability, as the increased number of replicas can handle higher traffic and performance demands, ensuring the services remain responsive under increased load. The system is set up to support stable scaling and performance for each service independently while distributing the workload across multiple pods.

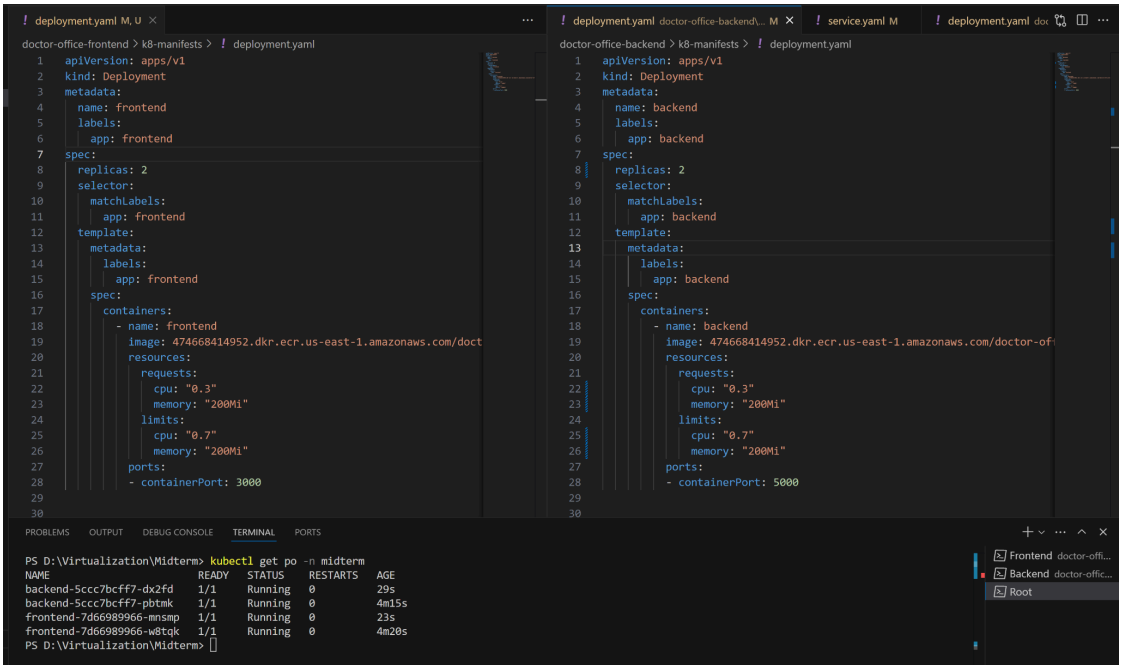

Figure 63: Scaling the Application

8. Conclusion

In this project, we successfully developed and deployed a scalable, containerized Doctor's Office Appointment application on AWS using EKS, leveraging both Kubernetes and Docker. The primary objectives—creating a robust front-end with React, a secure and functional back-end with Express.js and DynamoDB, and deploying using AWS cloud services—were achieved, showcasing the power of a microservices-based architecture.

The use of Amazon EKS for container orchestration allowed our team to automate deployments, scale the application effectively, and manage each component independently. The integration of AWS DynamoDB as the back-end data store offered low-latency data retrieval and storage, essential for managing appointment data. With the setup of GitHub Actions as a CI/CD pipeline, we were able to streamline our build, test, and deployment processes, enabling rapid iteration and reliable deployments.

Overall, this project provided valuable experience in cloud infrastructure, containerization, and microservices orchestration. Key benefits include improved scalability, high availability, enhanced security, and a strong foundation for future growth. The project not only highlights the capabilities of AWS and Kubernetes but also reinforces best practices in DevOps, automation, and modular architecture, preparing the team for similar large-scale applications in the future.

9. Appendices

- GitHub Repository - https://github.com/vi-kinn/Doctors-Office-Appointment-site
- Architecture Diagram - https://lucid.app/lucidchart/7fe4b040-c491-49b2-8cb3-dc474bd80241/edit?viewport_loc=-765%2C-209%2C3700%2C1719%2C0_0&invitationId=inv_bf13fef6-b2da-4761-ad31-930f30b41de7

10. References

- https://sysdig.com/blog/kubernetes-limits-requests/
- https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/
- https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
- https://docs.aws.amazon.com/eks/latest/userguide/sample-deployment.html