ENPM818R Group 3

Midterm

Microservices Application Deployment on AWS EKS

Honor Pledge:
"I pledge on my honor that I have not given or received any unauthorized assistance on this exam/assignment."

Group 3:

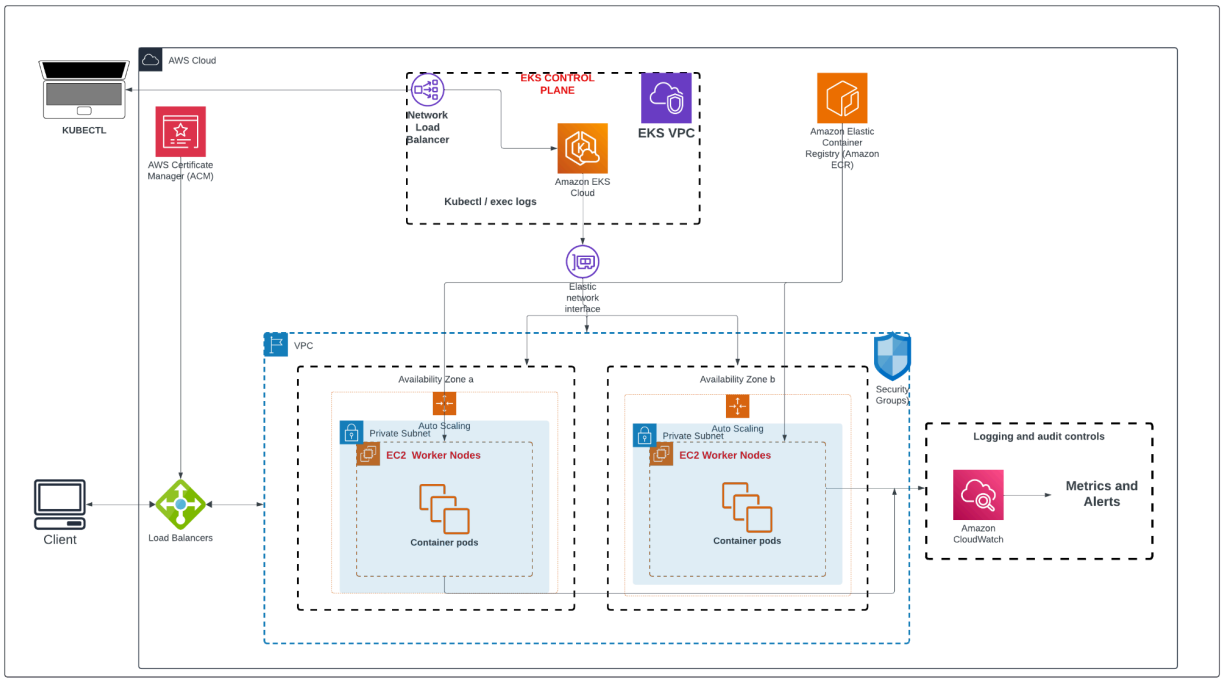
Name	Email	Section
Reuben Thomas	reuben10@umd.edu	ENPM818R 0101
Jongho Lee	jongho@umd.edu	ENPM818R 0101
Praveen Kumar Masupatri	pmasupat@umd.edu	ENPM818R 0101
Ganesh P. More	gpmore@umd.edu	ENPM818R AEB1
Rishabh Goel	rgoel22@umd.edu	ENPM818R 0101
Bhanu Teja Panguluri	bhanutp@umd.edu	ENPM818R 0101
Dhanushree Sidharamanagara Onkaramurthy	dhanuso7@umd.edu	ENPM 818R AEB1

Table of Contents

Introduction

The Doctor's Office Appointment system is designed to streamline appointment scheduling for both doctors and patients. The application features a frontend for user interactions, a backend to handle data processing, and a MongoDB database for storing appointment records. Through the frontend interface, users can schedule and review appointments, entering details such as patient and doctor names and preferred dates. The system's backend handles data transactions, allowing for appointment creation, retrieval, and storage within the database. The application was deployed on AWS with AWS EKS to orchestrate and manage the different microservices. The application once deployed will be available at the following URL: https://doctorapp-group3enpm818r.site/

Infrastructure Diagram



Overview:
The diagram illustrates how EKS is utilized and its connection to the worker nodes. It incorporates multiple AWS services, including EC2, EKS, ECR, VPC, and security groups.

EKS Control Plane:
This is among the fundamental features that EKS offers as a managed service. The control plane serves as K8s architecture's master node. This is in charge of effectively managing worker and slave nodes. They communicate with the worker node in order to -

- Schedule the pods
- Monitor the worker nodes/Pods
- Start/restart the pods
- Manage the new worker nodes joining the cluster

Worker Nodes:
These nodes are a collection of AWS-provisioned EC2 instances. Every Amazon EC2 node is set up in a single subnet. A private IP address is given to every node from a CIDR block that is assigned to the subnet.

Worker node can have one or more pods, these pods are your abstraction of a containerized application. Every worker runs these 3 key processes

- Container Runtime
- kubelet
- kube-proxy

Amazon Elastic Container Registry (ECR):

- Amazon ECR stores Docker images that are deployed to the Kubernetes pods. ECR provides a secure way to store and retrieve container images.

AWS LoadBalancer

- The Load Balancer distributes incoming traffic across multiple EC2 Worker Nodes in different Availability Zones, ensuring high availability and fault tolerance. It also works with Auto Scaling to manage traffic loads efficiently.

Monitoring and Logging:

- Amazon CloudWatch provides logging and auditing controls to monitor application performance, log errors, and ensure compliance.

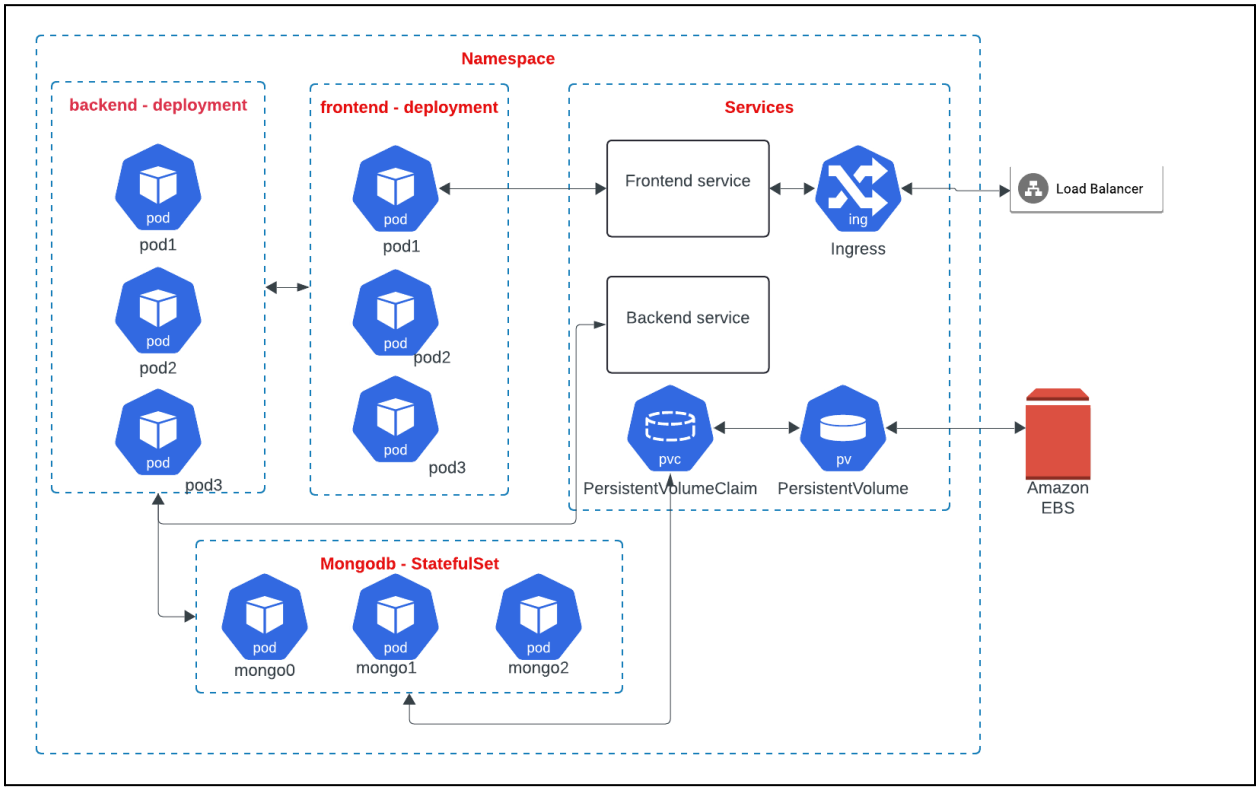
AWS ACM

- AWS Certificate Manager (ACM) is used to manage SSL/TLS certificates for the application's domain, enabling secure HTTPS communication by encrypting traffic between the client and the load balancer.

AWS Route53

- Amazon Route 53 was used for domain management and DNS routing, ensuring that traffic is directed to the correct endpoints within the AWS infrastructure, providing reliable and scalable domain name resolution for the application.

Kubernetes Implementation Diagram



Frontend Deployment:

- The frontend deployment is defined in a frontend-deployment.yaml file, specifying multiple replicas (pod1, pod2, pod3) to ensure high availability and load distribution. This deployment creates a set of identical pods that serve the application's frontend.
- A Frontend Service exposes these frontend pods, connecting them to an external Load Balancer. The Load Balancer distributes client requests across the frontend pods, enabling horizontal scaling and improving reliability by redirecting traffic in case of pod failures.

Backend Deployment:

- The backend deployment also defines a set of replica pods (pod1, pod2, pod3) to provide redundancy and scalability for the backend services.
- These backend pods are accessible within the cluster through an internal Kubernetes service, which allows the frontend pods to communicate with the backend securely without exposing the backend to external traffic.

MongoDB StatefulSet:

- MongoDB is deployed using a StatefulSet rather than a standard deployment. This ensures stable network identities and persistent storage for each MongoDB pod, critical for maintaining data consistency and state across restarts.
- The StatefulSet is configured with a Persistent Volume (PV) and a Persistent Volume Claim (PVC). The PVC connects to an Amazon Elastic Block Store (EBS) volume, providing durable and reliable storage for MongoDB data. This setup maintains data persistence even if MongoDB pods are rescheduled or restarted.

Persistent Volumes and Claims:

- Persistent Volumes (PV) and Persistent Volume Claims (PVC) are used to ensure data storage is persistent and independent of the pod lifecycle. The PVC requests storage from the EBS-backed PV, allowing MongoDB to retain data across restarts or migrations within the cluster.

Ingress:

- The Ingress controller manages external access to the Frontend Service, routing incoming traffic based on defined rules and enabling SSL termination for secure connections. It simplifies access by allowing multiple services to be exposed under a single IP address.

Deployment Prerequisites

The following prerequisites are needed for testing and deploying the application both locally and on AWS:

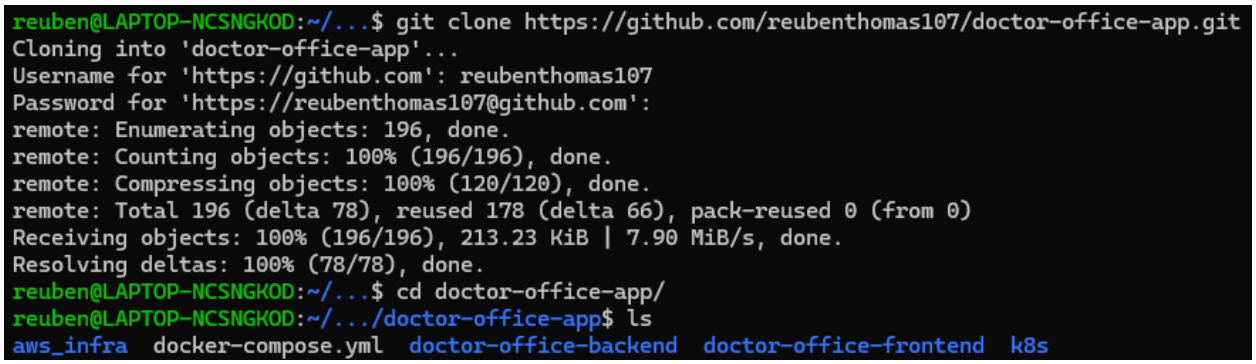
1. Git (Version control) - https://git-scm.com/downloads
2. Docker runtime
3. Kubernetes and Kubectl - https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/
4. Node - https://nodejs.org/en/download/package-manager
5. AWS CLI - https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html
6. eksctl - https://eksctl.io/installation/#for-unix

Getting Started

To deploy the Doctor's Office Appointment system, a collaborative workspace was established within a private GitHub repository, accessible at 'doctor-office-app'. This repository serves as the central hub for code organization and version control, supporting efficient code sharing, systematic reviews, and continuous integration across the development team. Each team member securely accesses the repository using a Personal Access Token (PAT), allowing for secure commit and pull operations. Utilizing GitHub's branching and pull request mechanisms, the team ensures that all contributions meet quality standards before integration. This approach enables consistent code quality and fosters an organized, collaborative development process.

Clone the repository using:

```
$ git clone https://github.com/reubenthomas107/doctor-office-app.git
```



Application codebase:

To establish the backend, a Node.js environment was configured with essential dependencies, including Express for routing and Mongoose for database connectivity. The backend provides foundational CRUD operations, ensuring efficient data handling and accessibility for users.

The frontend, created using the `create-react-app` tool, provides a responsive and intuitive interface for booking and reviewing appointments. It connects to the backend through API requests directed to `/api/appointments`, establishing reliable communication between the frontend and backend services.

Following the codebase setup, the project progresses to **Containerizing and Testing Microservice Components**, where each component is Dockerized. This step standardizes the environment, facilitates isolated testing, and prepares the application for scalable deployment on AWS EKS with Kubernetes orchestration.

Containerizing and Testing Microservice Components

Frontend Microservice

The frontend microservice is developed in React. This component provides the user interface (UI) and will serve the requests from the end-users. The web-app will be hosted locally on port 3001 and will be configured to get the appointment details from the 'backend' services when the request for '/api/appointments' is received. The frontend application code was modified to support requests to the backend URL, which would be a separate microservice.

App.js

docktor-app > doctor-office-frontend > src > App.js > App

```
1  import React, { useState, useEffect } from 'react';
2
3  function App() {
4    const [appointments, setAppointments] = useState([]);
5    const [form, setForm] = useState({ patientName: '', doctorName: '', date: '' });
6    const api_url = process.env.REACT_APP_API_URL || 'http://localhost:3000'
7
8    useEffect(() => {
9      console.log(`Here: ${api_url}`);
10     fetch(`${api_url}/api/appointments`)
11       .then(res => res.json())
12       .then(data => setAppointments(data));
13   }, []);
14
15
16   const handleSubmit = (e) => {
17     e.preventDefault();
18     fetch(`${api_url}/api/appointments`, {
19       method: 'POST',
20       headers: { 'Content-Type': 'application/json' },
21       body: JSON.stringify(form),
22     }).then(res => res.json())
23       .then(newAppointment => setAppointments([...appointments, newAppointment]));
24   };
```

Dockerfile:

Creating the Dockerfile to build the image for the frontend microservice. The file defines the build steps to execute in order to build the Docker image.

Dockerfile

doctor-office-frontend > Dockerfile > ...

```
1  FROM node:14
2
3  WORKDIR /usr/src/app
4
5  COPY package*.json ./
6
7  RUN npm install
8
9  COPY . .
10
11 RUN npm run build
12
13 EXPOSE 3001
14
15 CMD ["npx", "serve", "build", "-l", "3001"]
16
```

Building and running the ‘frontend’ Docker image:

```
$ docker build -t frontend .
```

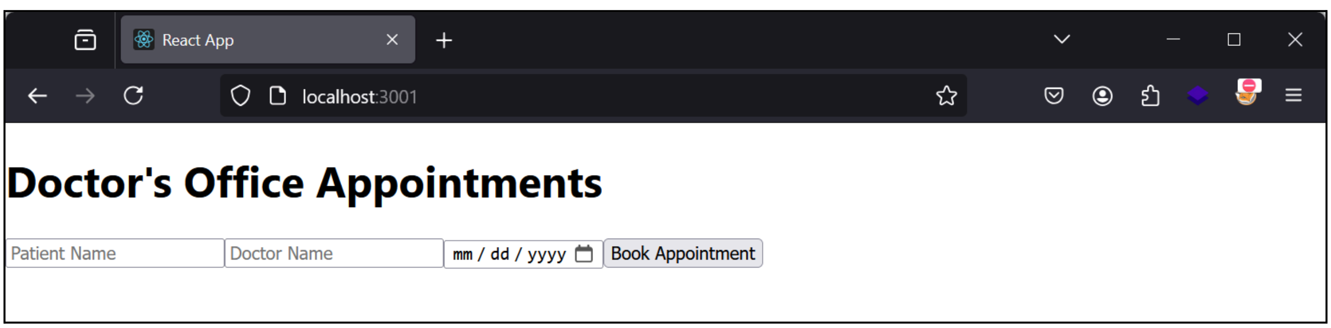
```
gpmore@DESKTOP-8BEAC60:/doctor-office-frontend $ docker build -t frontend .
[+] Building 7.9s (12/12) FINISHED                                                    docker:default
 => [internal] load build definition from Dockerfile                                            0.0s
 => => transferring dockerfile: 393B                                                            0.0s
 => [internal] load metadata for docker.io/library/node:14                                      0.8s
 => [auth] library/node:pull token for registry-1.docker.io                                     0.0s
 => [internal] load .dockerignore                                                               0.1s
 => => transferring context: 2B                                                                 0.1s
 => [1/6] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa  0.0s
 => [internal] load build context                                                               0.1s
 => => transferring context: 1.14kB                                                             0.1s
 => CACHED [2/6] WORKDIR /usr/src/app                                                           0.0s
 => CACHED [3/6] COPY package*.json ./                                                          0.0s
 => CACHED [4/6] RUN npm install                                                                0.0s
 => [5/6] COPY . .                                                                              0.0s
 => [6/6] RUN npm run build                                                                     6.7s
 => exporting to image                                                                          0.0s
 => => exporting layers                                                                         0.0s
 => => writing image sha256:2e98434a46987e010ef32ea868eb6ef9b77925f4776444ddf8d8e28177f6bba0    0.0s
 => => naming to docker.io/library/frontend                                                     0.0s
```

```
$ docker run -dit -p 3001:3001 frontend
```

```
gpmore@DESKTOP-8BEAC60:/doctor-office-frontend $ docker run -dit -p 3001:3001 frontend
986dc1ec34d850d0b27a493b468fef438a8eb263595490f218350debe343e6b4
gpmore@DESKTOP-8BEAC60:/doctor-office-frontend $ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED         STATUS         PORTS                    NAMES
986dc1ec34d8   frontend   "docker-entrypoint.s…"   2 seconds ago   Up 2 seconds   0.0.0.0:3001->3001/tcp   compassionate_matsumoto
gpmore@DESKTOP-8BEAC60:/doctor-office-frontend $
```

Frontend React application is running on Port 3001:

```
gpmore@DESKTOP-8BEAC60:/doctor-office-frontend $ curl http://localhost:3001
<!doctype html><html lang="en"><head><meta charset="utf-8"/><link rel="icon" href="/favicon.ico"/><meta name="viewport" content="width=device-width,initial
-scale=1"/><meta name="theme-color" content="#000000"/><meta name="description" content="Web site created using create-react-app"/><link rel="apple-touch-i
con" href="/logo192.png"/><link rel="manifest" href="/manifest.json"/><title>React App</title><script defer="defer" src="/static/js/main.018d26ac.js"></scr
ipt><link href="/static/css/main.e6c13ad2.css" rel="stylesheet"></head><body><noscript>You need to enable JavaScript to run this app.</noscript><div id="ro
ot"></div></body></html>gpmore@DESKTOP-8BEAC60:/doctor-office-frontend $
```



Backend Microservice

For our application's backend, we set up the NodeJS backend and install the required dependencies. Setting up the Dockerfile for the backend service:

Dockerfile

doctor-office-backend > Dockerfile > FROM

```dockerfile
FROM node:18

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "index.js"]
```

Building and running the 'backend' Docker image:

$ **docker build -t backend .**

```
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $ docker build -t backend .
[+] Building 0.6s (11/11) FINISHED                                                        docker:default
 => [internal] load build definition from Dockerfile                                                0.0s
 => => transferring dockerfile: 161B                                                                0.0s
 => [internal] load metadata for docker.io/library/node:18                                          0.5s
 => [auth] library/node:pull token for registry-1.docker.io                                         0.0s
 => [internal] load .dockerignore                                                                   0.0s
 => => transferring context: 2B                                                                     0.0s
 => [1/5] FROM docker.io/library/node:18@sha256:ddd173cd94537e155b378342056e0968e8299eb3da9dd5d412d3b7f796ac38c0   0.0s
 => [internal] load build context                                                                   0.0s
 => => transferring context: 129B                                                                   0.0s
 => CACHED [2/5] WORKDIR /usr/src/app                                                               0.0s
 => CACHED [3/5] COPY package*.json ./                                                              0.0s
 => CACHED [4/5] RUN npm install                                                                    0.0s
 => CACHED [5/5] COPY . .                                                                           0.0s
 => exporting to image                                                                              0.0s
 => => exporting layers                                                                             0.0s
 => => writing image sha256:77327bb9780122707158e66523a4dd81d3870302a4144e49be2a35c4f13bc4d9       0.0s
 => => naming to docker.io/library/backend                                                          0.0s
```

$ **docker run -dit -p 3000:3000 backend**

```
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $ docker run -dit -p 3000:3000 backend
9e3715863d985e80140f08a595afbd760eaff8bfaa52a4217e868c231433a3ab
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED         STATUS         PORTS                    NAMES
9e3715863d98   backend    "docker-entrypoint.s…"   6 seconds ago   Up 5 seconds   0.0.0.0:3000->3000/tcp   bold_proskuriakova
986dc1ec34d8   frontend   "docker-entrypoint.s…"   6 minutes ago   Up 6 minutes   0.0.0.0:3001->3001/tcp   compassionate_matsumoto
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $
```

The backend application needs a connection to the database running on port 27017 in order to run without any errors, and hence before testing this, we need to set up a MongoDB container. We will create a test-network and run the MongoDB and backend container.

Database (MongoDB) Microservice

```
$ docker run -it -d -p 27017:27017 --network test-network --name mongo mongo
```

```
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $ docker network create test-network
8bfaea3167232d7f9b1ff0acd412d2571678db98e03b106a9dbf3cf3aea850fe
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $ docker run -it -d -p 27017:27017 --network test-network --name mongo mongo
Unable to find image 'mongo:latest' locally
latest: Pulling from library/mongo
ff65ddf9395b: Pull complete
458feb307082: Pull complete
f59af5df8253: Pull complete
145c7b6ccdb9: Pull complete
35cc527541fc: Pull complete
076d157aff57: Pull complete
197a30480327: Pull complete
3736af050cc0: Pull complete
Digest: sha256:3984cf5a234e525253619060fcbff12449db0597d62a6d4e18991a18f2365c36
Status: Downloaded newer image for mongo:latest
d456c14ef4c8805f856b83a61a486c9c4fd9cd65037ae86e67b7696a2e42f0b4
```

```
$ curl http://localhost:27017
```

```
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $ curl http://localhost:27017
It looks like you are trying to access MongoDB over HTTP on the native driver port.
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $
```

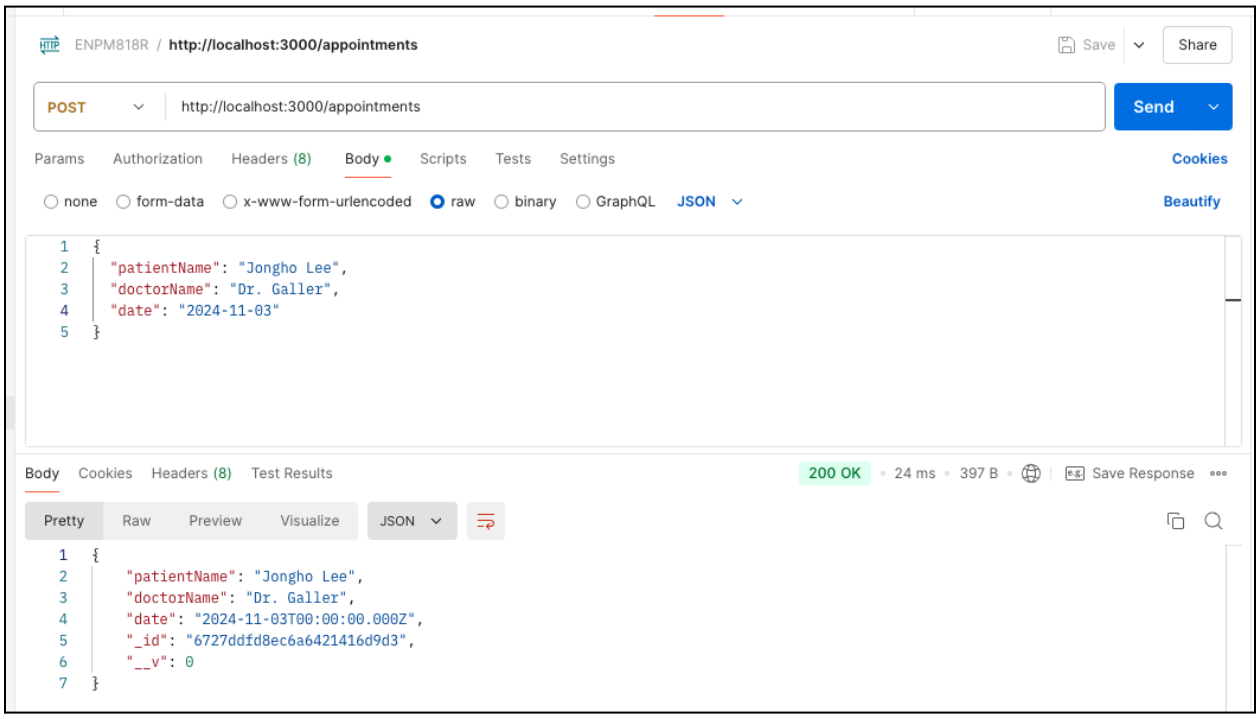
We will run our backend in the ‘test-network’.

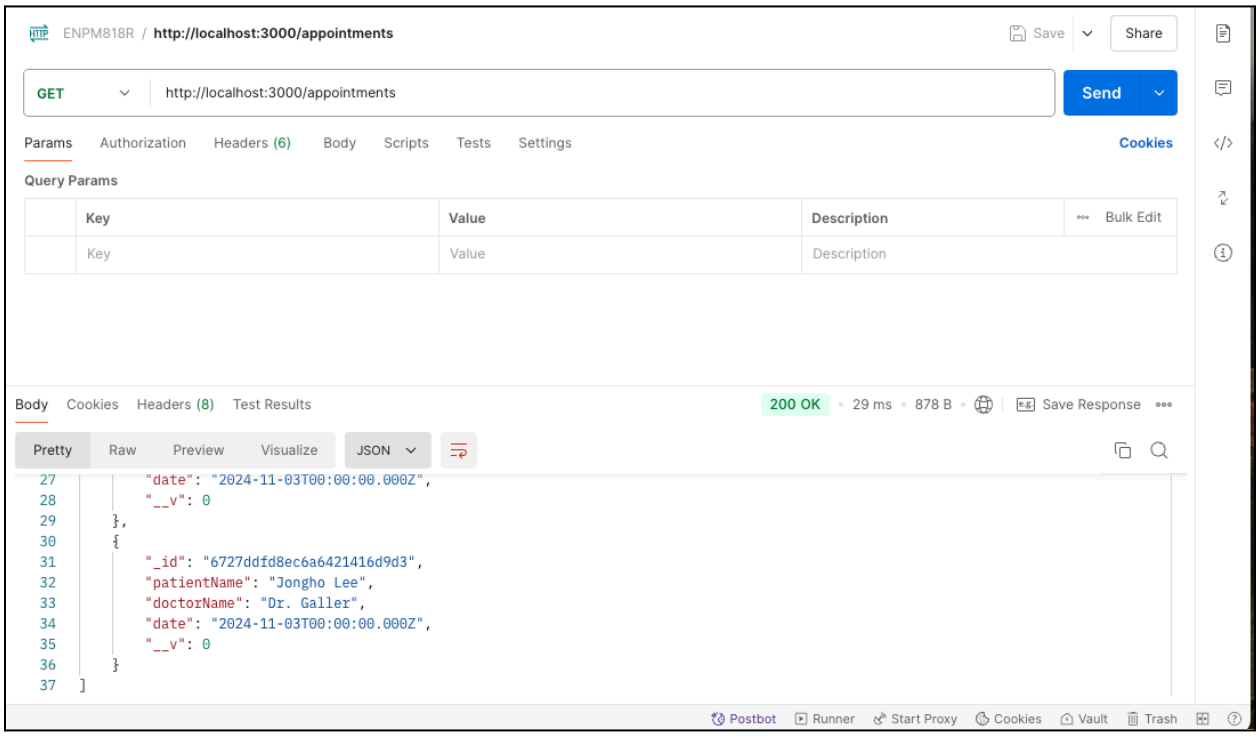
```
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $ docker run -dit -p 3000:3000 --network test-network backend
ce8ddaeb35a62b8ea3846882f4ec9619fdeb7dee81bf59730a8827d830a218e6
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS                     NAMES
ce8ddaeb35a6   backend   "docker-entrypoint.s…"   15 seconds ago   Up 15 seconds   0.0.0.0:3000->3000/tcp    vigorous_williams
d456c14ef4c8   mongo     "docker-entrypoint.s…"   2 minutes ago    Up 2 minutes    0.0.0.0:27017->27017/tcp  mongo
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $
```

Testing the backend and database service:

```
gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $ curl http://localhost:3000/appointments
[]gpmore@DESKTOP-8BEAC6O:/doctor-office-backend $
```

Test backend microservice independently using Postman:

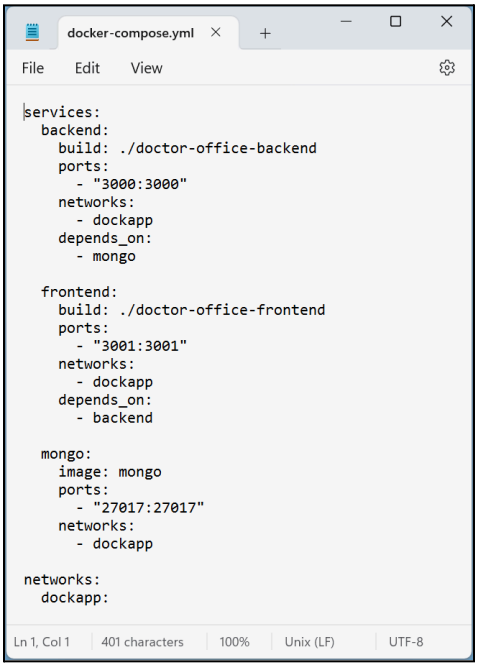




Testing the Application Locally with Docker-Compose and Kubernetes

Docker Compose

Now that the components have been individually tested, we use docker compose to orchestrate the microservices to run together. The docker-compose.yml file defines the three components of the application: frontend, backend and database.

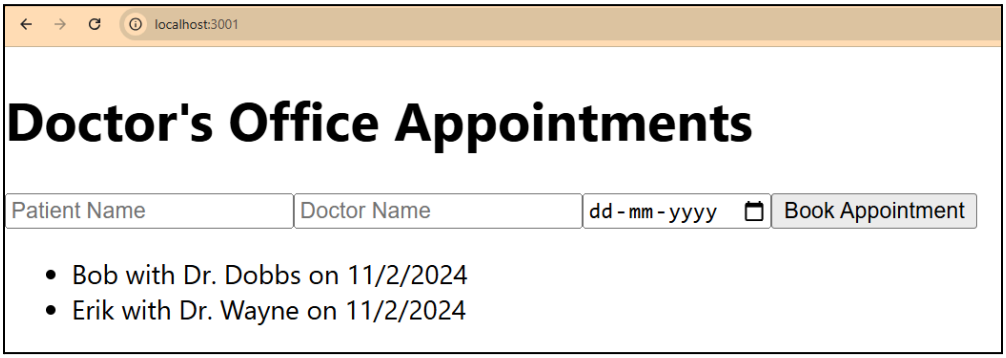


To run the application using docker-compose.yml:

```
$ docker-compose up -d
```

```
gpmore@DESKTOP-8BEAC6O:/doctor-office-app $ docker-compose up -d
[+] Running 4/4
 ✔Network doctor-office-app_dockapp        Created                                    0.1s
 ✔Container doctor-office-app-mongo-1      Started                                    0.9s
 ✔Container doctor-office-app-backend-1    Started                                    1.5s
 ✔Container doctor-office-app-frontend-1   Started                                    2.0s
gpmore@DESKTOP-8BEAC6O:/doctor-office-app $
```

```
gpmore@DESKTOP-8BEAC6O:/doctor-office-app $ docker ps -a
CONTAINER ID   IMAGE                          COMMAND                  CREATED         STATUS         PORTS                      NAMES
bffda9b242a6   doctor-office-app-frontend     "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0.0.0.0:3001->3001/tcp     doctor
-office-app-frontend-1
da7db0d5db6f   doctor-office-app-backend      "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0.0.0.0:3000->3000/tcp     doctor
-office-app-backend-1
9577cea14660   mongo                          "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0.0.0.0:27017->27017/tcp   doctor
-office-app-mongo-1
```



To ensure the application gets the URLs/variables/envs, changes were made. The frontend application Dockerfile was also updated to include an Nginx Proxy configuration that would route traffic to the backend when making the requests.

Kubernetes

To test the application locally with Kubernetes, we will use minikube.

```
$ minikube start
```

```
$ minikube status
```

```
gpmore@DESKTOP-8BEAC6O:/doctor-office-app $ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

We have already pushed frontend and backend images to Docker Hub. To deploy the pods, we need to create Kubernetes manifest files for frontend, backend and mongo.

backend-deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: gpmore/enpm818r:backend
        ports:
        - containerPort: 3000
        env:
        - name: MONGODB_URL
          value: "mongo-
service.default.svc.cluster.local:27017"
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
```

frontend-deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: gpmore/enpm818r:frontend
        ports:
        - containerPort: 3001
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3001
```

mongo-deployment.y

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: mongo
        image: mongo
        ports:
        - containerPort: 27017
---
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
  selector:
    app: mongo
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 27017
```

To deploy MongoDB service and deployment:

```
$ kubectl apply -f mongo-deployment.yaml
```

To deploy backend service and deployment:

```
$ kubectl apply -f backend-deployment.yaml
```

To deploy frontend service and deployment:

$ **kubectl apply -f frontend-deployment.yaml**

```
gpmore@DESKTOP-8BEAC6O:/local-k8s $ kubectl apply -f mongo-deployment.yaml
deployment.apps/mongo created
service/mongo-service created
gpmore@DESKTOP-8BEAC6O:/local-k8s $ kubectl apply -f backend-deployment.yaml
deployment.apps/backend created
service/backend-service created
gpmore@DESKTOP-8BEAC6O:/local-k8s $ kubectl apply -f frontend-deployment.yaml
deployment.apps/frontend created
service/frontend-service created
gpmore@DESKTOP-8BEAC6O:/local-k8s $
```

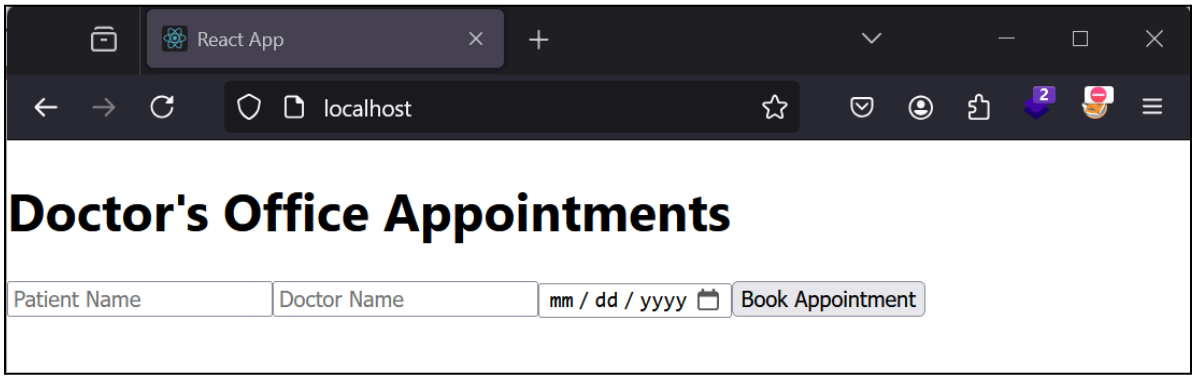
Check the deployed resources' status using:

```
gpmore@DESKTOP-8BEAC6O:/local-k8s $ kubectl get all
NAME                            READY   STATUS    RESTARTS   AGE
pod/backend-6bddb58846-fr9mz    1/1     Running   0          2m22s
pod/frontend-64cd959d68-d98bp   1/1     Running   0          2m15s
pod/mongo-98c74b57c-ngxwr       1/1     Running   0          2m37s

NAME                       TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/backend-service    ClusterIP      10.105.112.26    <none>        3000/TCP         2m22s
service/frontend-service   LoadBalancer   10.107.188.184   <pending>     80:32494/TCP     2m15s
service/kubernetes         ClusterIP      10.96.0.1        <none>        443/TCP          4h18m
service/mongo-service      ClusterIP      10.110.57.197    <none>        27017/TCP        2m37s

NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend    1/1     1            1           2m22s
deployment.apps/frontend   1/1     1            1           2m15s
deployment.apps/mongo      1/1     1            1           2m37s

NAME                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-6bddb58846    1         1         1       2m22s
replicaset.apps/frontend-64cd959d68   1         1         1       2m15s
replicaset.apps/mongo-98c74b57c       1         1         1       2m37s
gpmore@DESKTOP-8BEAC6O:/local-k8s $
```

Start the tunnel to access the load balancer:
$ **minikube tunnel**

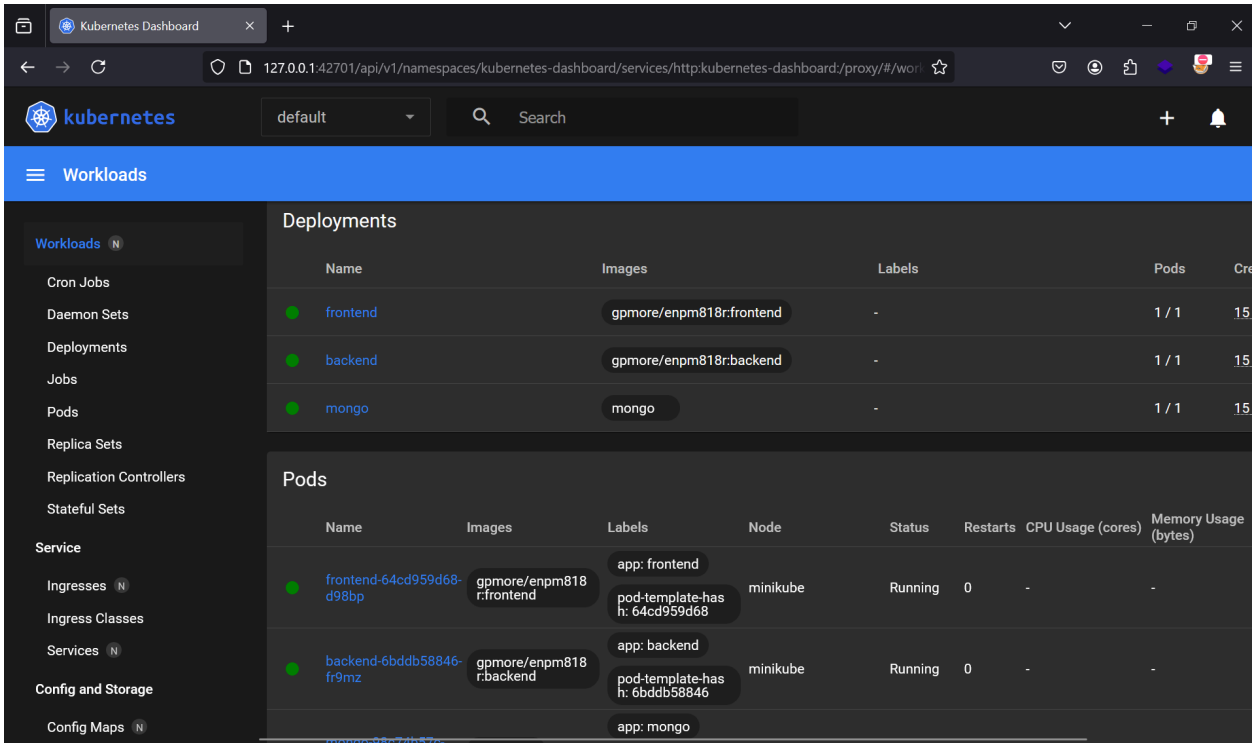
```
gpmore@DESKTOP-8BEAC6O:/local-k8s $ minikube tunnel
✅  Tunnel successfully started
```

Access the app running at http://localhost/ :



To check the kubernetes dashboard run the following command and copy paste the URL in browser:

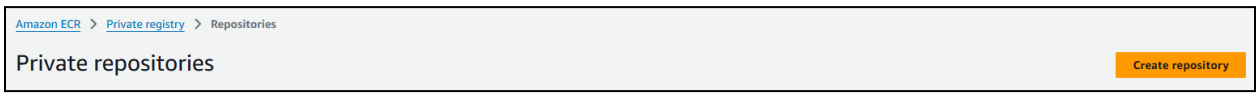
```
$ minikube dashboard --url
```



Pushing the Docker Images to AWS ECR (Elastic Container Registry)

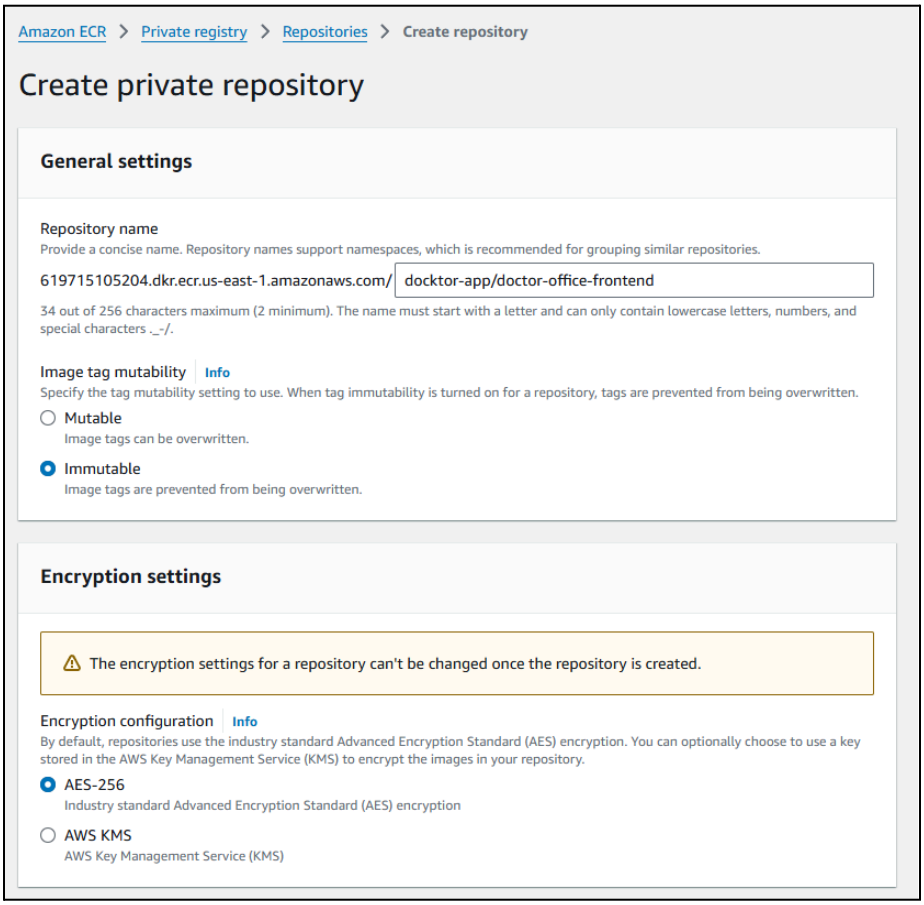
To store our application's 'frontend' and 'backend' Docker images in the private AWS ECR registry, we create a separate repository for each microservice. To create an AWS ECR repository, we perform the following steps:

In the AWS console, navigate to AWS ECR and click on '**Create Repository**'



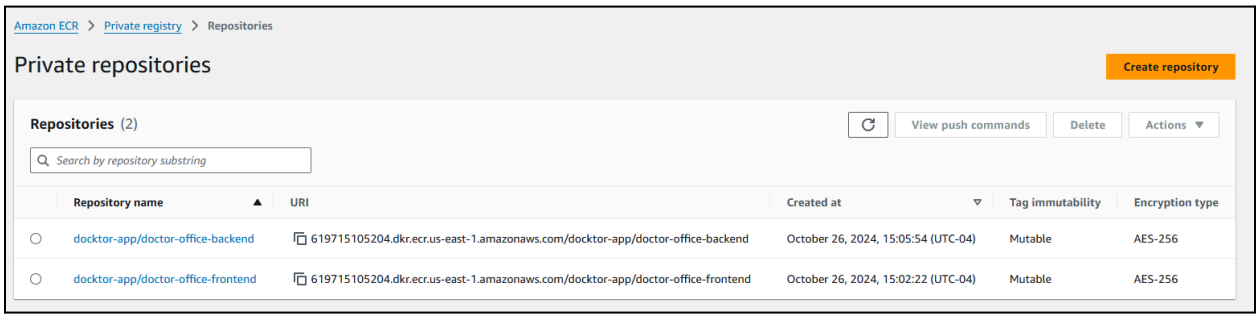
We provide a namespace for the project ('docktor-app') and a repository name for each microservice. Setting the tag mutability option to 'Immutable' prevents images from being overwritten and adds an additional layer of security. Click on 'Create' to create the private ECR repository. The two repositories 'doctor-office-frontend' and 'doctor-office-backend' will be used to store the Docker images for our 'Doctor' application.

AWS Console - ECR repository creation:



Repeat the process to create another repository for the 'backend' microservice.

Repositories in the AWS ECR private registry:



To push the Docker images to the repositories, we run the following commands for each service: Authenticate with the AWS ECR registry (Ensure AWS CLI was configured).

```
$ aws ecr get-login-password --region us-east-1 | docker login --username AWS
--password-stdin 619715105204.dkr.ecr.us-east-1.amazonaws.com
```

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/doctor-office-frontend$ aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 61971
5105204.dkr.ecr.us-east-1.amazonaws.com
Login Succeeded
```

Navigate to the microservice and build the image as follows:

```
$ docker build -t docktor-app/doctor-office-frontend .
```

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/doctor-office-frontend$ docker build -t docktor-app/doctor-office-frontend .
[+] Building 18.3s (17/17) FINISHED                                                                                                                  docker:default
 => [internal] load build definition from Dockerfile                                                                                                          0.0s
 => => transferring dockerfile: 342B                                                                                                                          0.0s
 => WARN: FromAsCasing: 'as' and 'FROM' keywords' casing do not match (line 1)                                                                                0.0s
 => [internal] load metadata for docker.io/library/nginx:latest                                                                                               0.4s
 => [internal] load metadata for docker.io/library/node:14                                                                                                    0.5s
 => [auth] library/nginx:pull token for registry-1.docker.io                                                                                                  0.0s
 => [auth] library/node:pull token for registry-1.docker.io                                                                                                   0.0s
 => [internal] load .dockerignore                                                                                                                             0.0s
 => => transferring context: 2B                                                                                                                               0.0s
 => [stage-1 1/3] FROM docker.io/library/nginx:latest@sha256:28402db69fec7c17e179ea87882667f1e054391138f77ffaf0c3eb388efc3ffb                                 0.0s
 => [builder 1/6] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa                                     0.0s
 => [internal] load build context                                                                                                                             0.0s
 => => transferring context: 603.80kB                                                                                                                         0.0s
 => CACHED [builder 2/6] WORKDIR /usr/src/app                                                                                                                 0.0s
 => CACHED [builder 3/6] COPY package*.json ./                                                                                                                0.0s
 => CACHED [builder 4/6] RUN npm install                                                                                                                      0.0s
 => [builder 5/6] COPY . .                                                                                                                                    0.1s
 => [builder 6/6] RUN npm run build                                                                                                                          15.2s
 => CACHED [stage-1 2/3] COPY ./nginx/default.conf /etc/nginx/conf.d/default.conf                                                                             0.0s
 => CACHED [stage-1 3/3] COPY --from=builder /usr/src/app/build/ /usr/share/nginx/html                                                                        0.0s
 => exporting to image                                                                                                                                        0.0s
 => => exporting layers                                                                                                                                       0.0s
 => => writing image sha256:14e69ee77af33ef1ef490098ff331edf25481e94a80eef5cb780326f76e82329                                                                  0.0s
 => => naming to docker.io/docktor-app/doctor-office-frontend                                                                                                 0.0s
```

Tag the local image with the location of the ECR registry

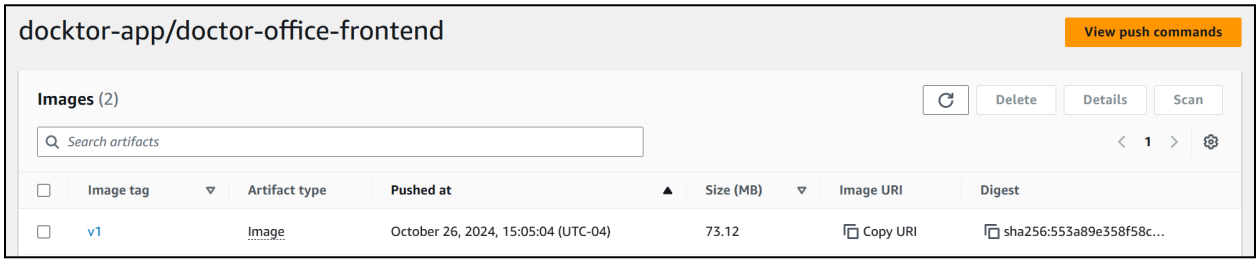
```
$ docker tag docktor-app/doctor-office-frontend:latest
619715105204.dkr.ecr.us-east-1.amazonaws.com/docktor-app/doctor-office-frontend:v1
```

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/doctor-office-frontend$ docker tag docktor-app/doctor-office-frontend:latest 619715105204.dkr.ecr.us-east-1.amazonaws.com/
docktor-app/doctor-office-frontend:v1
```

Push the image to the ECR repository

```
$ docker push
619715105204.dkr.ecr.us-east-1.amazonaws.com/docktor-app/doctor-office-frontend:v1
```

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/doctor-office-frontend$ docker push 619715105204.dkr.ecr.us-east-1.amazonaws.com/docktor-app/doctor-office-frontend:v1
The push refers to repository [619715105204.dkr.ecr.us-east-1.amazonaws.com/docktor-app/doctor-office-frontend]
4e18b5a43b27: Layer already exists
cd20adcfc1cf: Layer already exists
e4e9e9ad93c2: Layer already exists
6ac729401225: Layer already exists
8ce189049cb5: Layer already exists
296af1bd2844: Layer already exists
63d7ce983cd5: Layer already exists
b33db0c3c3a8: Layer already exists
98b5f35ea9d3: Layer already exists
v1: digest: sha256:553a89e358f58cb7c3019e064d2ba5bb8d11532abd5454812390f1e38dde6e4d size: 2195
```

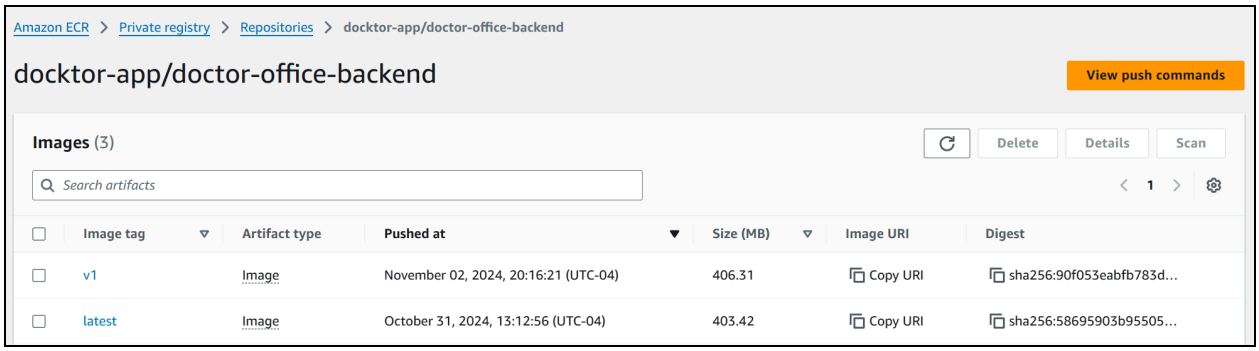


Similarly, we build and push the Docker image for the 'backend' microservice to the 'doctor-office-backend' repository.

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/doctor-office-backend$ aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 619715
105204.dkr.ecr.us-east-1.amazonaws.com
Login Succeeded
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/doctor-office-backend$ docker build -t docktor-app/doctor-office-backend .
[+] Building 1.0s (11/11) FINISHED                                                                                                  docker:default
 => [internal] load build definition from Dockerfile                                                                                          0.0s
```

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/doctor-office-backend$ docker tag docktor-app/doctor-office-backend:latest 619715105204.dkr.ecr.us-east-1.amazonaws.com/do
cktor-app/doctor-office-backend:v1
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/doctor-office-backend$ docker push 619715105204.dkr.ecr.us-east-1.amazonaws.com/docktor-app/doctor-office-backend:v1
The push refers to repository [619715105204.dkr.ecr.us-east-1.amazonaws.com/docktor-app/doctor-office-backend]
fb3130ab38c4: Layer already exists
39338e759e69: Layer already exists
3c9b8c5baaa9: Layer already exists
154f432ec09d: Layer already exists
afb48ce6bd97: Layer already exists
9a605894dac7: Layer already exists
65b08847bb33: Layer already exists
72949a6dc267: Layer already exists
d23b5e6144a7: Layer already exists
e5ee1bd83fe3: Layer already exists
43da071b5e0c: Layer already exists
ef5f5ddeb0a6: Layer already exists
v1: digest: sha256:90f053eabfb783d6e1f5b4a5eccafa8b55941c546b58cab7a6169d71d9ecd3fc size: 2841
```

The Docker images have successfully been pushed to the repository and we can confirm it by viewing it in the AWS console as follows:

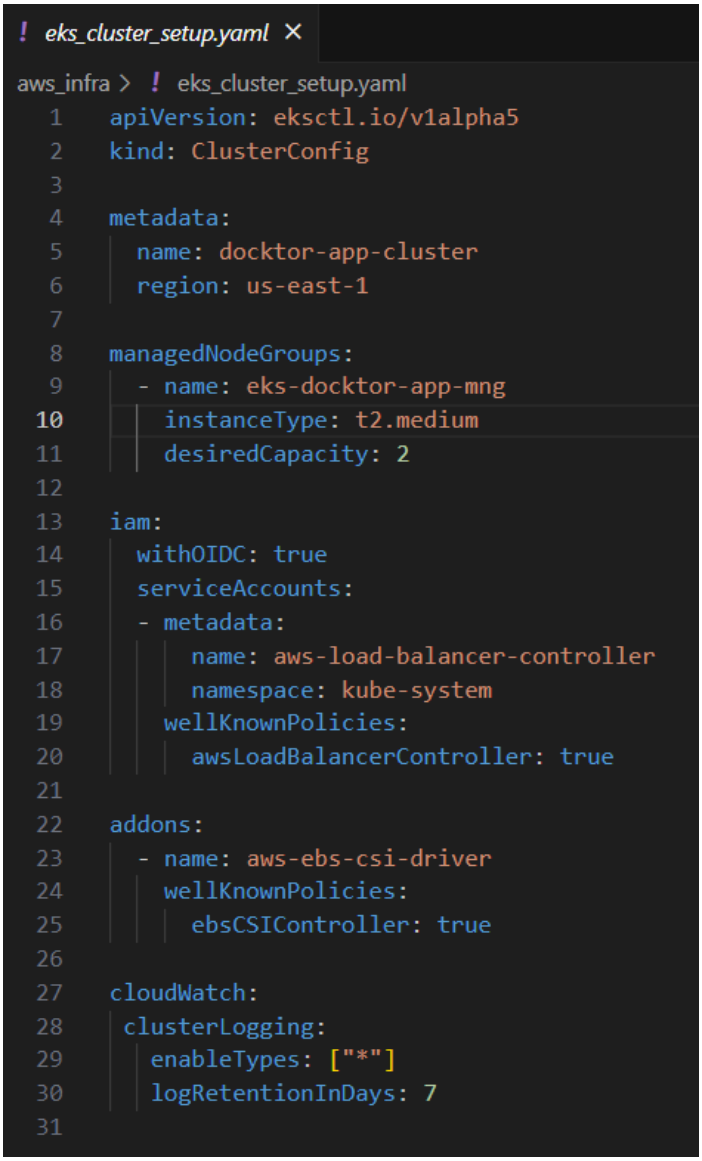


Setting up AWS EKS (Elastic Kubernetes Service)

Cluster Setup

To set up the EKS cluster on AWS, we use the 'eksctl' command-line tool. This tool helps us manage and deploy the EKS cluster on AWS by creating the Node Groups and necessary IAM roles. We create an 'eks_cluster_setup.yaml' file and define the cluster configuration. In this configuration file we define the instance type ('t2.medium'), capacity and region of our cluster nodes. The file also defines and creates the IAM service account for the Application Load Balancer (ALB), installs the EBS add-on driver and enables CloudWatch logging for the cluster.

eks_cluster_setup.yaml:



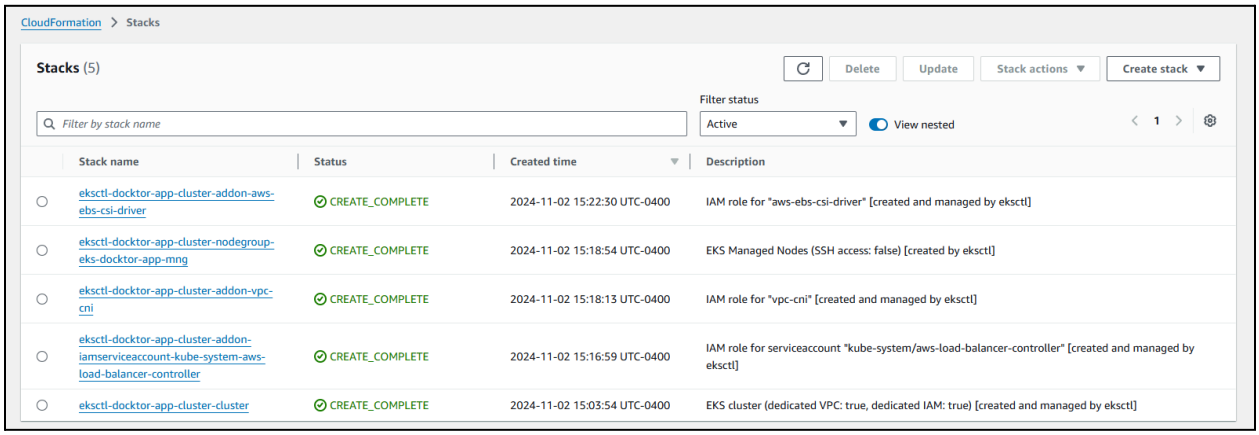
Navigate to the 'aws_infra' directory and run the following commands to deploy the configuration and to set up the EKS cluster on AWS:

```
$ eksctl create cluster -f eks_cluster_setup.yaml
```

```
reuben@LAPTOP-NCSNGKOD
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app$ cd aws_infra/
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/aws_infra$ eksctl create cluster -f eks_cluster_setup.yaml
2024-11-02 15:03:55 [ℹ]  eksctl version 0.194.0
2024-11-02 15:03:55 [ℹ]  using region us-east-1
2024-11-02 15:03:55 [ℹ]  setting availability zones to [us-east-1b us-east-1f]
2024-11-02 15:03:55 [ℹ]  subnets for us-east-1b - public:192.168.0.0/19 private:192.168.64.0/19
2024-11-02 15:03:55 [ℹ]  subnets for us-east-1f - public:192.168.32.0/19 private:192.168.96.0/19
2024-11-02 15:03:55 [ℹ]  nodegroup "eks-docktor-app-mng" will use "" [AmazonLinux2023/1.30]
2024-11-02 15:03:55 [ℹ]  using Kubernetes version 1.30
2024-11-02 15:03:55 [ℹ]  creating EKS cluster "docktor-app-cluster" in "us-east-1" region with managed nodes
2024-11-02 15:03:55 [ℹ]  1 nodegroup (eks-docktor-app-mng) was included (based on the include/exclude rules)
2024-11-02 15:03:55 [ℹ]  will create a CloudFormation stack for cluster itself and 0 nodegroup stack(s)
2024-11-02 15:03:55 [ℹ]  will create a CloudFormation stack for cluster itself and 1 managed nodegroup stack(s)
2024-11-02 15:03:55 [ℹ]  if you encounter any issues, check CloudFormation console or try 'eksctl utils describe-stacks --re
gion=us-east-1 --cluster=docktor-app-cluster'
2024-11-02 15:03:55 [ℹ]  Kubernetes API endpoint access will use default of {publicAccess=true, privateAccess=false} for clu
ster "docktor-app-cluster" in "us-east-1"
2024-11-02 15:03:55 [ℹ]  configuring CloudWatch logging for cluster "docktor-app-cluster" in "us-east-1" (enabled types: api
, audit, authenticator, controllerManager, scheduler & no types disabled)
2024-11-02 15:03:55 [ℹ]  default addons vpc-cni, kube-proxy, coredns were not specified, will install them as EKS addons
2024-11-02 15:03:55 [ℹ]
```

The ‘eksctl’ command will spin up CloudFormation templates to deploy the EKS infrastructure. This process takes around 20 minutes. ‘eksctl’ will automatically **create IAM roles** (EKS cluster, nodegroups, ALB controller) for the cluster and set up the worker node instances.

CloudFormation templates created/managed by ‘eksctl’:



Once the cluster is up and running, access the cluster using:

`$ kubectl get nodes`

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/aws_infra$ kubectl get nodes
NAME                           STATUS   ROLES    AGE     VERSION
ip-192-168-24-137.ec2.internal   Ready    <none>   9m1s    v1.30.4-eks-a737599
ip-192-168-32-161.ec2.internal   Ready    <none>   8m45s   v1.30.4-eks-a737599
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/aws_infra$
```

Note: ‘eksctl’ automatically updates the kube-config

We see the two nodes running on the ‘EC2’ instances in AWS

Installing Add-ons

Set up Application Load Balancer (ALB) Controller for EKS Cluster

The ALB controller would be used in setting up the Ingress for our application. The controller automates provisioning and configuration of the AWS application load balancer (ALB).

Setting the environment variables for the cluster region and cluster VPC:

```
$ export CLUSTER_REGION=us-east-1
```

```
$ export CLUSTER_VPC=$(aws eks describe-cluster --name docktor-app-cluster --region $CLUSTER_REGION --query "cluster.resourcesVpcConfig.vpcId" --output text)
```

Using Helm to install the add-on

```
$ helm repo add eks https://aws.github.io/eks-charts
```

```
$ helm repo update eks
```

```
$ helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
    --namespace kube-system \
    --set clusterName=docktor-app-cluster \
    --set serviceAccount.create=false \
    --set region=${CLUSTER_REGION} \
    --set vpcId=${CLUSTER_VPC} \
    --set serviceAccount.name=aws-load-balancer-controller
```

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/aws_infra$ export CLUSTER_REGION=us-east-1
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/aws_infra$ export CLUSTER_VPC=$(aws eks describe-cluster --name docktor-app-clust
er --region $CLUSTER_REGION --query "cluster.resourcesVpcConfig.vpcId" --output text)
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/aws_infra$ helm repo add eks https://aws.github.io/eks-charts
WARNING: Kubernetes configuration file is group-readable. This is insecure. Location: /home/reuben/.kube/config
WARNING: Kubernetes configuration file is world-readable. This is insecure. Location: /home/reuben/.kube/config
"eks" already exists with the same configuration, skipping
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/aws_infra$ helm repo update eks
WARNING: Kubernetes configuration file is group-readable. This is insecure. Location: /home/reuben/.kube/config
WARNING: Kubernetes configuration file is world-readable. This is insecure. Location: /home/reuben/.kube/config
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "eks" chart repository
Update Complete. ⎈Happy Helming!⎈
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/aws_infra$ helm install aws-load-balancer-controller eks/aws-load-balancer-contro
ller \
    --namespace kube-system \
    --set clusterName=docktor-app-cluster \
    --set serviceAccount.create=false \
    --set region=${CLUSTER_REGION} \
    --set vpcId=${CLUSTER_VPC} \
    --set serviceAccount.name=aws-load-balancer-controller
WARNING: Kubernetes configuration file is group-readable. This is insecure. Location: /home/reuben/.kube/config
WARNING: Kubernetes configuration file is world-readable. This is insecure. Location: /home/reuben/.kube/config
NAME: aws-load-balancer-controller
LAST DEPLOYED: Sat Nov  2 15:30:54 2024
NAMESPACE: kube-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
AWS Load Balancer controller installed!
```

Set up CloudWatch Observability Add-on for EKS Cluster

This add-on enables logging and monitoring for our workloads in the EKS cluster.

Create an IAM service account ‘cloudwatch-agent’:

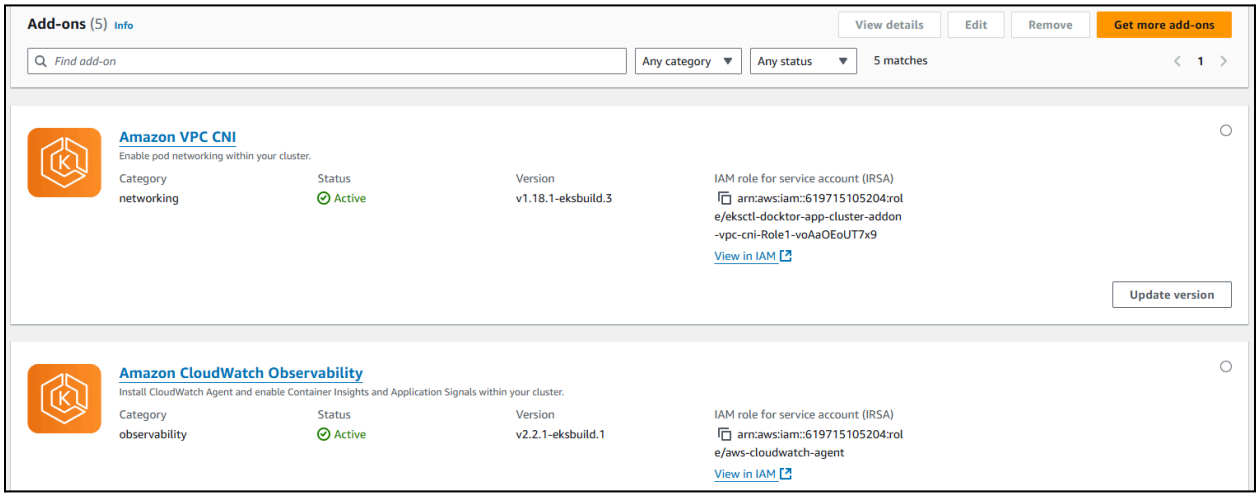
```
$ eksctl create iamserviceaccount \
  --name cloudwatch-agent \
  --namespace amazon-cloudwatch --cluster docktor-app-cluster \
  --role-name aws-cloudwatch-agent \
  --attach-policy-arn arn:aws:iam::aws:policy/CloudWatchAgentServerPolicy \
  --role-only \
  --approve
```

Install the add-on to the ‘docktor-app-cluster’:

```
$ aws eks create-addon \
--addon-name amazon-cloudwatch-observability \
--cluster-name docktor-app-cluster \
--service-account-role-arn arn:aws:iam::619715105204:role/aws-cloudwatch-agent
```

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/aws_infra$ aws eks create-addon \
--addon-name amazon-cloudwatch-observability \
--cluster-name docktor-app-cluster \
--service-account-role-arn arn:aws:iam::619715105204:role/aws-cloudwatch-agent
{
    "addon": {
        "addonName": "amazon-cloudwatch-observability",
        "clusterName": "docktor-app-cluster",
        "status": "CREATING",
        "addonVersion": "v2.2.1-eksbuild.1",
        "health": {
            "issues": []
        },
        "addonArn": "arn:aws:eks:us-east-1:619715105204:addon/docktor-app-cluster/amazon-cloudwatch-observability/2ec97730-2
bed-99dc-9e5b-010b93cd3d72",
        "createdAt": "2024-11-02T15:35:44.480000-04:00",
        "modifiedAt": "2024-11-02T15:35:44.496000-04:00",
        "serviceAccountRoleArn": "arn:aws:iam::619715105204:role/aws-cloudwatch-agent",
        "tags": {}
    }
}
```

AWS EKS Add-ons Console View:



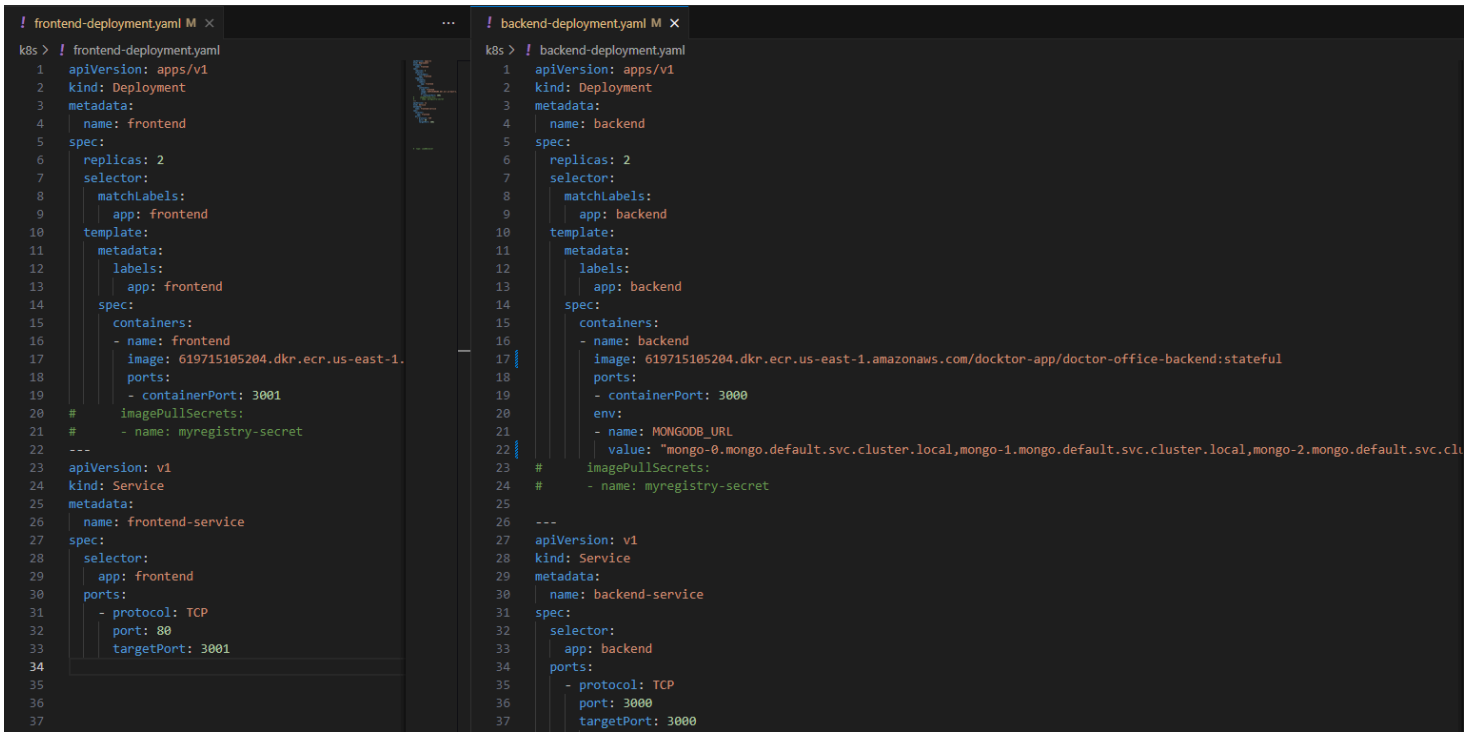
Setup steps for issuing SSL certificates from ACM and Route 53 configuration:

Navigate to the following link for steps to issue an SSL certificate for our web application:
SSL Certificate and Route 53

Kubernetes Deployment/Service/StatefulSet Files

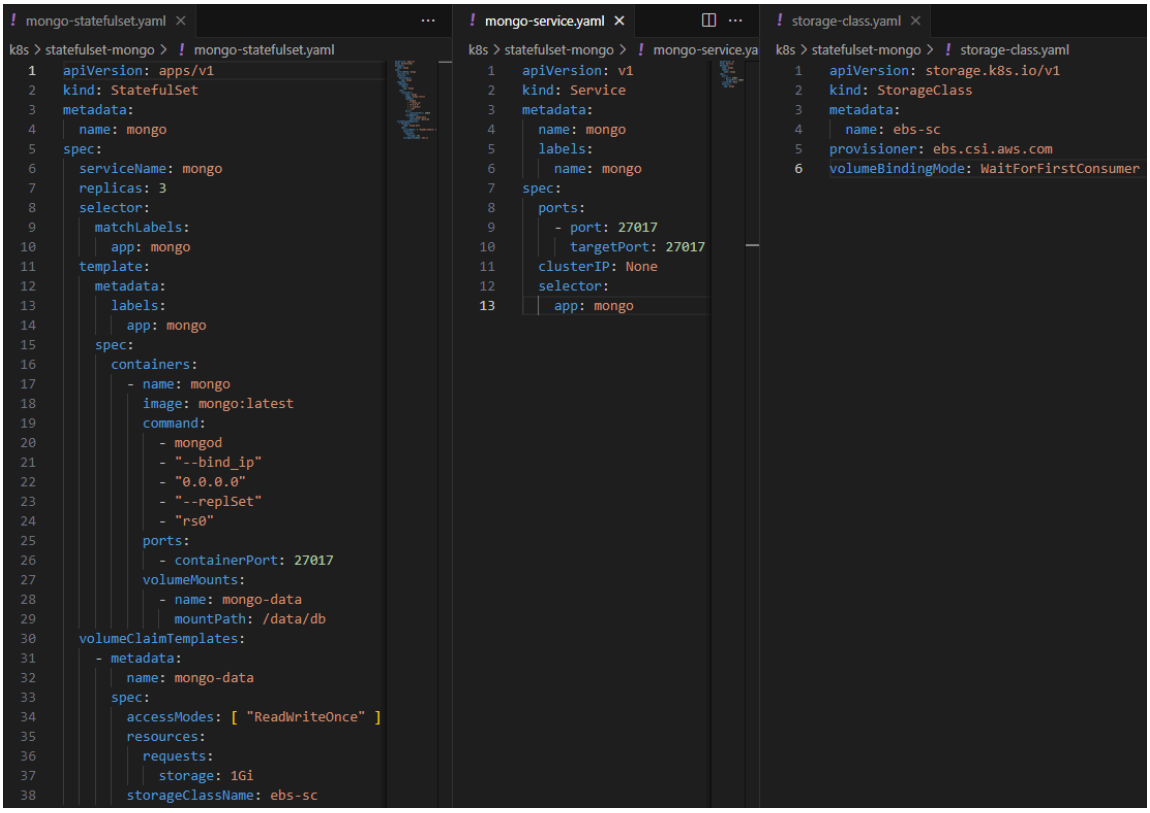
The kubernetes manifest files were updated to use the images pushed in the private ECR registry. The frontend microservice uses the ‘docker-office-frontend’ image and has a service ‘frontend-service’ of ‘ClusterIP’ type that routes traffic incoming on port 80 to the frontend application pods running on port 3001. The nginx proxy running within the pod handles the incoming requests. Requests for the ‘appointments’ (API calls) are internally routed to the backend microservice pods by the nginx proxy.

Similarly, the backend microservice uses the ‘docker-office-backend’ image. The backend microservice pods connect to the MongoDB database microservice for querying or inserting appointment data. This configuration file exposes the port 3000 and enables the pods to listen for incoming requests to the service. Additionally, the URL for the MongoDB StatefulSets is configured as an environment variable and this allows changes to be made without rebuilding the container images.



Repository Link: https://github.com/reubenthomas107/doctor-office-app/tree/main/k8s

The ‘mongodb’ microservice uses the latest mongo container image and exposes port 27017 for incoming database connections. Using StatefulSets for MongoDB service helps us to maintain data persistence, ordered scaling and helps to easily configure replica sets.



mongo-statefulset.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: mongo
  replicas: 3
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
        - name: mongo
          image: mongo:latest
          command:
            - mongod
            - "--bind_ip"
            - "0.0.0.0"
            - "--replSet"
            - "rs0"
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: mongo-data
              mountPath: /data/db
  volumeClaimTemplates:
    - metadata:
        name: mongo-data
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
          requests:
            storage: 1Gi
        storageClassName: ebs-sc
```

mongo-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mongo
  labels:
    name: mongo
spec:
  ports:
    - port: 27017
      targetPort: 27017
  clusterIP: None
  selector:
    app: mongo
```

storage-class.yaml

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-sc
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
```

Ingress Controller:

The ‘ingress-controller.yaml’ file is used to create an Ingress. The annotations in the file are used to configure the application load balancer (ALB) using the ALB controller. We specify the ALB to use the certificate generated by AWS Certificate Manager, and configure the routing rules for the application. The ingress creates an ALB that is configured to listen to client requests on port 80/443, redirect traffic to a secure HTTPS version of the website. The ingress also manages the routing of the traffic to the pods deployed in the cluster.

Ingress-controller.yaml

ingress-controller.yaml

```yaml
# https://kubernetes-sigs.github.io/aws-load-balancer-controller/v2.7/guide/ingress/annotations/
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: docktor-app-ingress
  annotations:
    alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:us-east-1:619715105204:certificate/1cc6df08-c38c-42ed-a89b-f074c4e7e950
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP": 80}, {"HTTPS":443}]'
    alb.ingress.kubernetes.io/ssl-redirect: "443"
    kubernetes.io/ingress.class: "alb"
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/healthcheck-path: /
spec:
  ingressClassName: alb
  rules:
    - #host: doctorapp-group3enpm818r.site
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
```

Deploy the application to the Kubernetes (EKS) cluster

Deploy the MongoDB StatefulSet

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/k8s$ cd statefulset-mongo/
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/k8s/statefulset-mongo$ kubectl apply -f storage-class.yaml
storageclass.storage.k8s.io/ebs-sc created
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/k8s/statefulset-mongo$ kubectl apply -f mongo-statefulset.yaml
statefulset.apps/mongo created
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/k8s/statefulset-mongo$ kubectl apply -f mongo-service.yaml
service/mongo created
```

We will set the mongo-0 pod as primary and the remaining pods as secondary:

```
$ kubectl exec -it mongo-0 -- mongosh
```

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/k8s/statefulset-mongo$ kubectl exec -it mongo-0 -- mongosh
Current Mongosh Log ID: 6727b2d52dc557ff3ffe6910
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.3.2
Using MongoDB:          8.0.3
Using Mongosh:          2.3.2
```

Command to configure the replica set.

```
$ rs.initiate({_id: "rs0",members: [{_id: 0, host:
"mongo-0.mongo.default.svc.cluster.local:27017" },{_id: 1, host:
"mongo-1.mongo.default.svc.cluster.local:27017" },{_id: 2, host:
"mongo-2.mongo.default.svc.cluster.local:27017" },]});
```

```
test> rs.initiate({_id: "rs0",members: [{_id: 0, host: "mongo-0.mongo.default.svc.cluster.local:27017" },{_id: 1, host: "mongo-1.mongo.default.svc.cluster.local:27017
" },{_id: 2, host: "mongo-2.mongo.default.svc.cluster.local:27017" },]});
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1730654954, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1730654954, i: 1 })
}
```

Deploy the ‘frontend’ and ‘backend’ deployments

```
$ kubectl apply -f ./k8s/
```

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app$ kubectl apply -f k8s/
deployment.apps/backend created
service/backend-service created
deployment.apps/frontend created
service/frontend-service created
ingress.networking.k8s.io/docktor-app-ingress created
```

This will deploy all the manifest files to our EKS cluster. To view our resources, we run:

`$ kubectl get all`

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app/k8s$ kubectl get all
NAME                                READY   STATUS    RESTARTS   AGE
pod/backend-69cc9fb6f8-bnhlb        1/1     Running   0          36m
pod/backend-69cc9fb6f8-xsfl8        1/1     Running   0          36m
pod/frontend-856cc4bf89-b8mcw       1/1     Running   0          21m
pod/frontend-856cc4bf89-w9nvx       1/1     Running   0          21m
pod/mongo-0                         1/1     Running   0          41m
pod/mongo-1                         1/1     Running   0          41m
pod/mongo-2                         1/1     Running   0          40m

NAME                       TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)     AGE
service/backend-service    ClusterIP   10.100.27.93     <none>        3000/TCP    36m
service/frontend-service   ClusterIP   10.100.193.131   <none>        80/TCP      21m
service/kubernetes         ClusterIP   10.100.0.1       <none>        443/TCP     56m
service/mongo              ClusterIP   None             <none>        27017/TCP   40m

NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend    2/2     2            2           36m
deployment.apps/frontend   2/2     2            2           21m

NAME                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-69cc9fb6f8    2         2         2       36m
replicaset.apps/frontend-856cc4bf89   2         2         2       21m

NAME                     READY   AGE
statefulset.apps/mongo   3/3     41m
```

We see all the running pods, deployments, and services for our application.

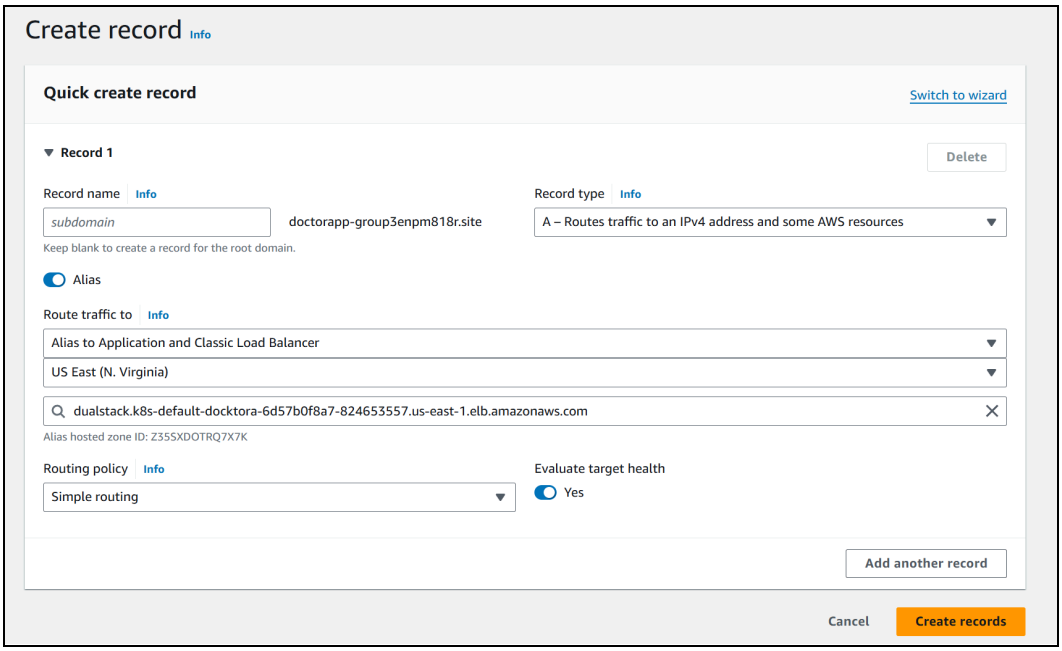
Run the following command to get the address of the ingress controller:

`$ kubectl get ingress`

```
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app$ kubectl get ingress
NAME                CLASS   HOSTS   ADDRESS                                                                  PORTS   AGE
docktor-app-ingress   alb     *       k8s-default-docktora-6d57b0f8a7-824653557.us-east-1.elb.amazonaws.com   80      49s
reuben@LAPTOP-NCSNGKOD:~/doctor-office-app$
```

The load balancer DNS name/address is now visible here. The application load balancer was provisioned by the ALB controller and will now be ready to accept the requests from the client. The SSL certificate configured in the 'ingress-controller.yaml' config file will be attached to this load balancer. We **create a new 'A' record** within our Route53 Hosted Zone's domain configuration to route incoming traffic for our domain - '**doctorapp-group3enpm818r.site**' to the Application Load Balancer:

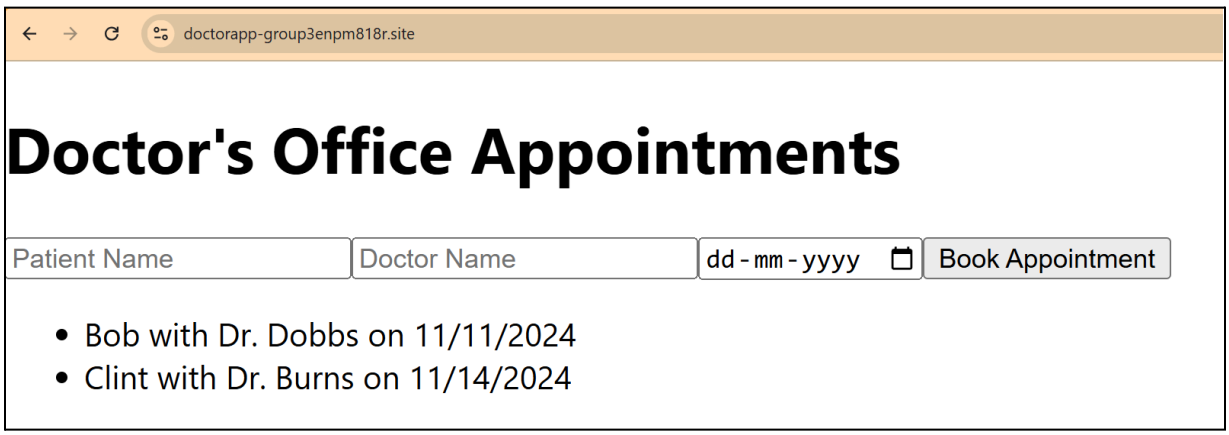
Navigate to: **Route 53 > Hosted zones > doctorapp-group3enpm818r.site**



Accessing the 'Doctor Appointment' application

Let us now visit the deployed web application on https://doctorapp-group3enpm818r.site/

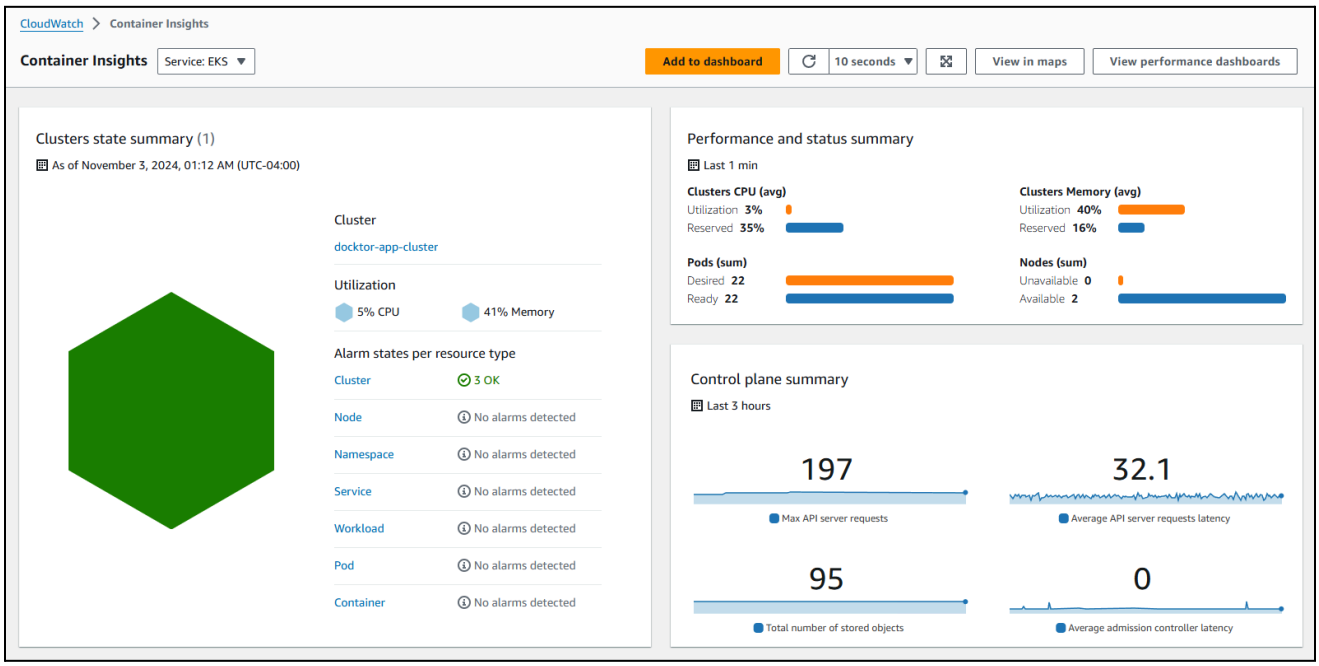
Booking appointments:

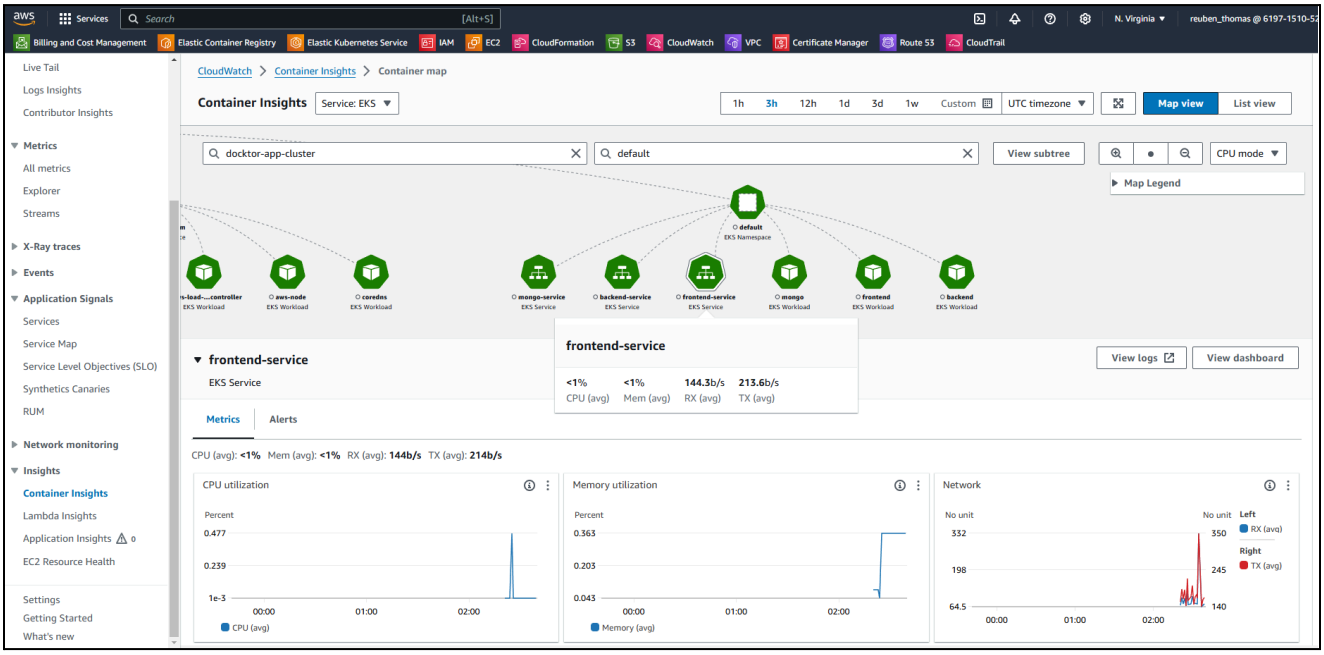


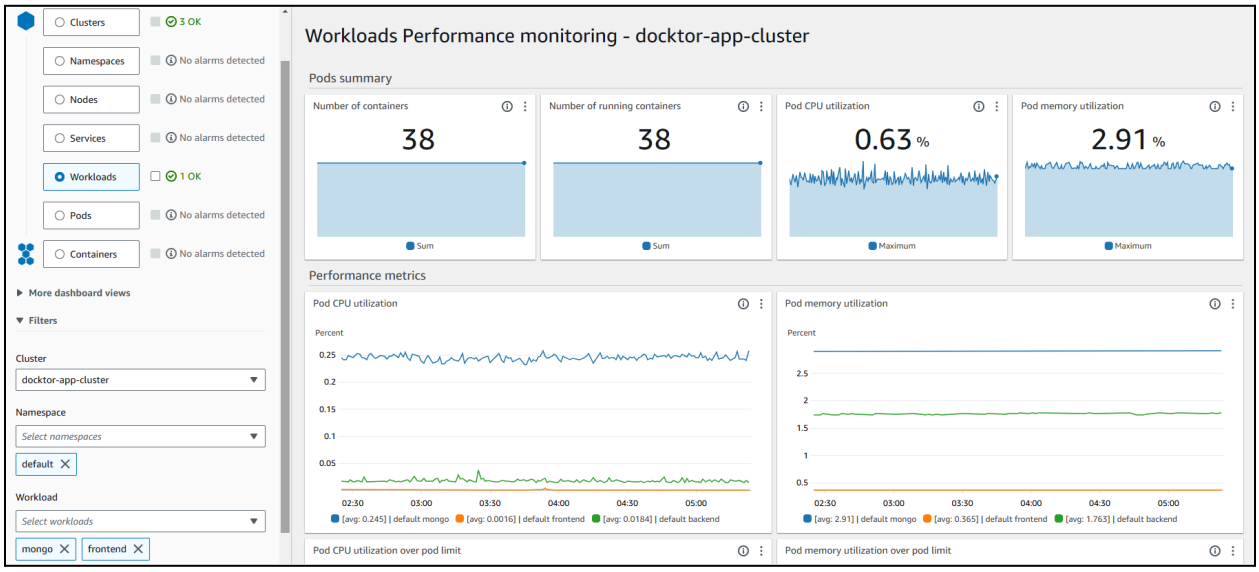
Monitoring our pods/Application with CloudWatch

EKS cluster logging within CloudWatch was configured in the EKS setup configuration file. Additionally, by installing the CloudWatch Observability EKS add-on, we enable **Container Insights**, which is a feature that provides greater visibility into our kubernetes environment. This add-on installs a Fluent-bit agent and CloudWatch agent on the cluster which helps report and obtain logs, metrics, performance data from individual pods, namespaces, deployments, etc. in the cluster. As an optional feature, we also configure the Application Signals which helps us monitor the operational status, metrics, and health of some of our deployed microservices.

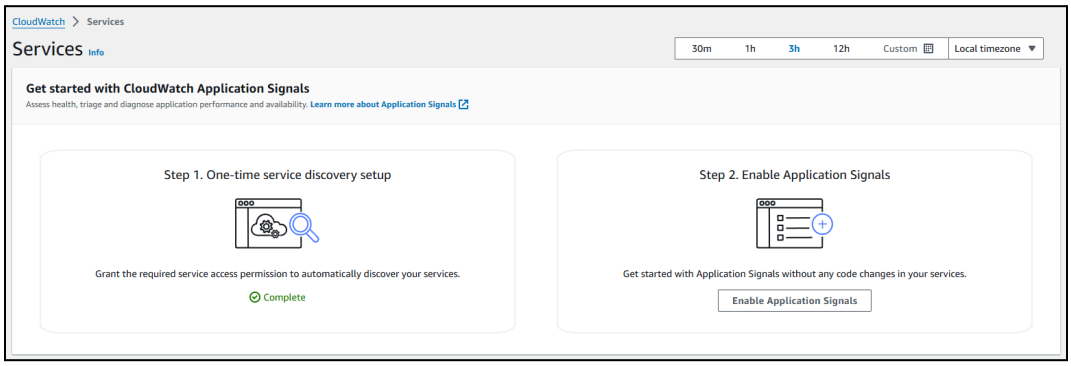
CloudWatch Monitoring and Container Insights



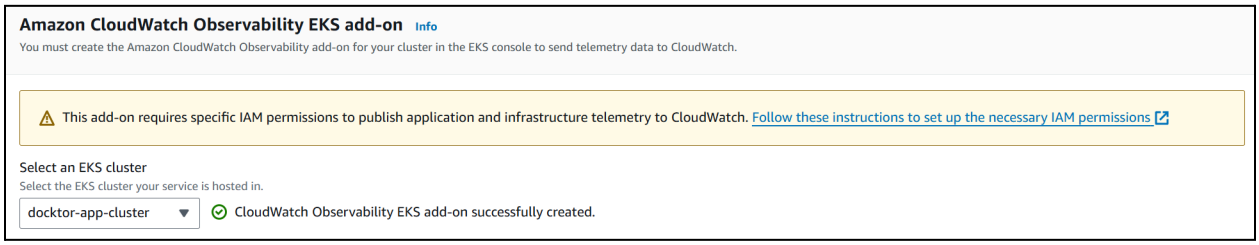




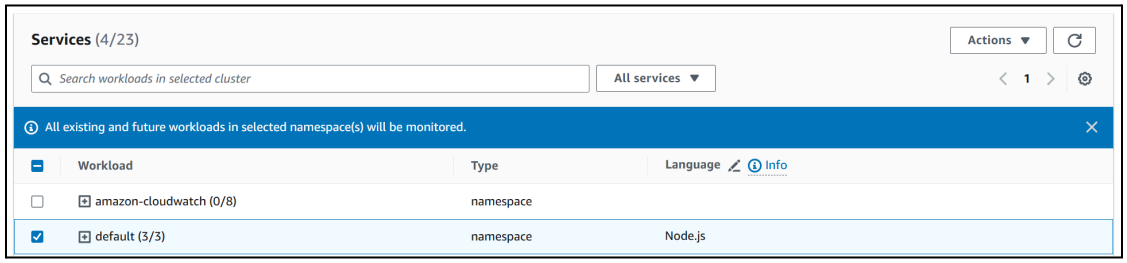
Configuration for Application Signals



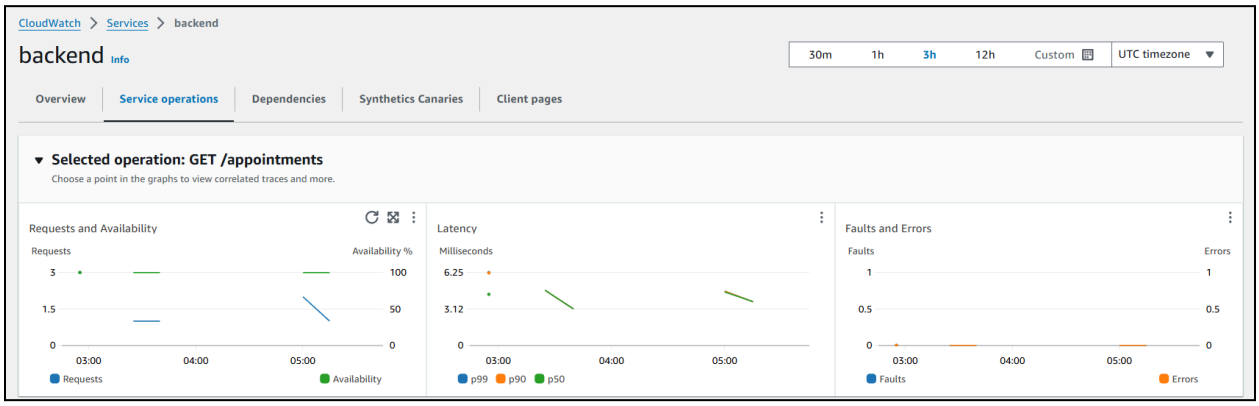
1. Specify the platform as “**EKS**”
2. Select the EKS Cluster - “**docktor-app-cluster**”

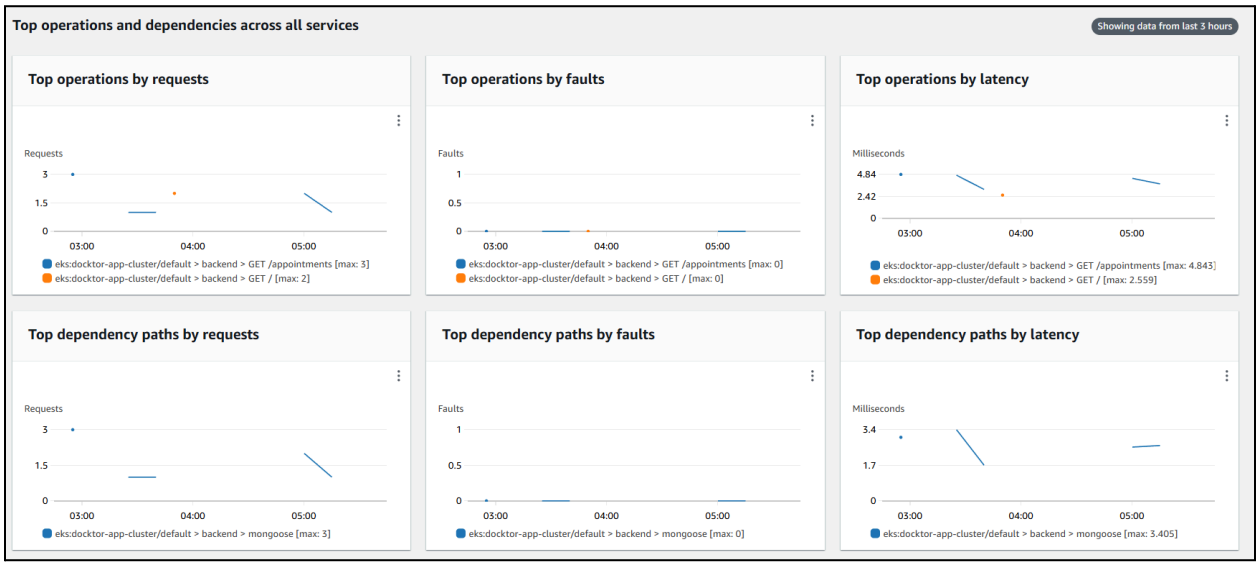


3. Select the application namespace - “**default**” (application’s namespace)
4. Select programming language of the ‘backend’ microservice - “**Node.js**”



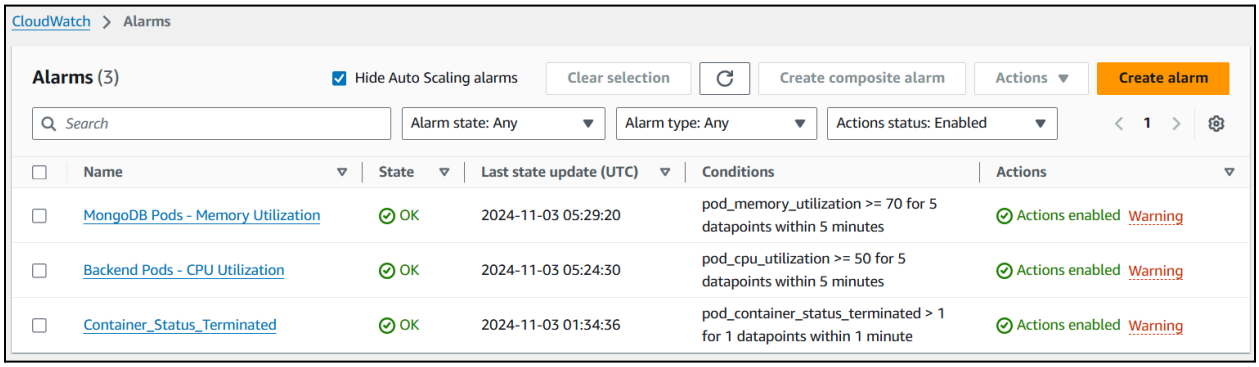
After creating the application signal, we will be able to see the operational status of the ‘backend’ microservice, which can help us determine the overall load on the service and can help us in making scaling decisions.





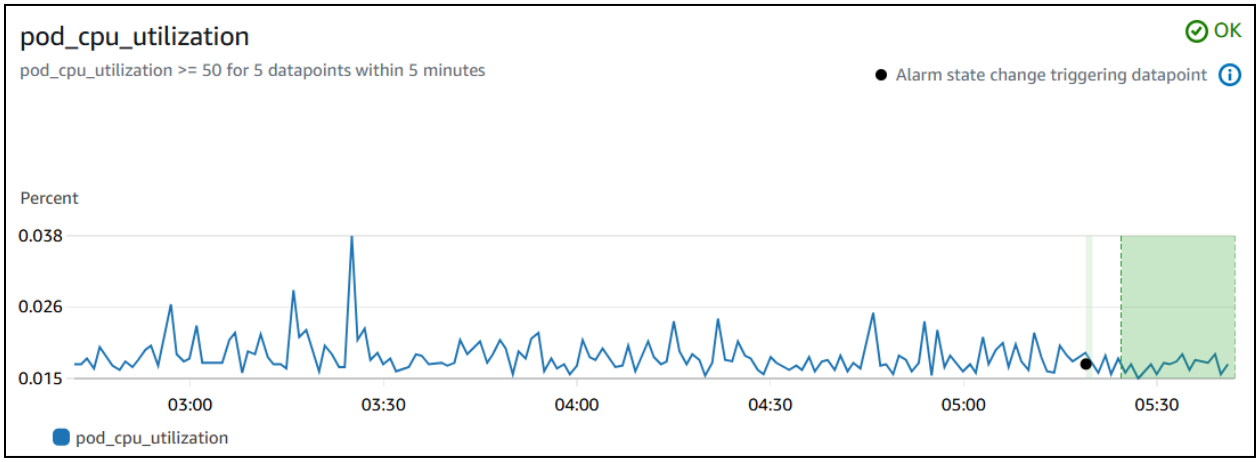
Setting Up Alerts for Monitoring Container Performance

Configuring alarms in CloudWatch to monitor container performance. Alarms for CPU utilization, memory usage and container termination errors were configured to alert when custom thresholds for each metric are met. Key alerts that were setup include:



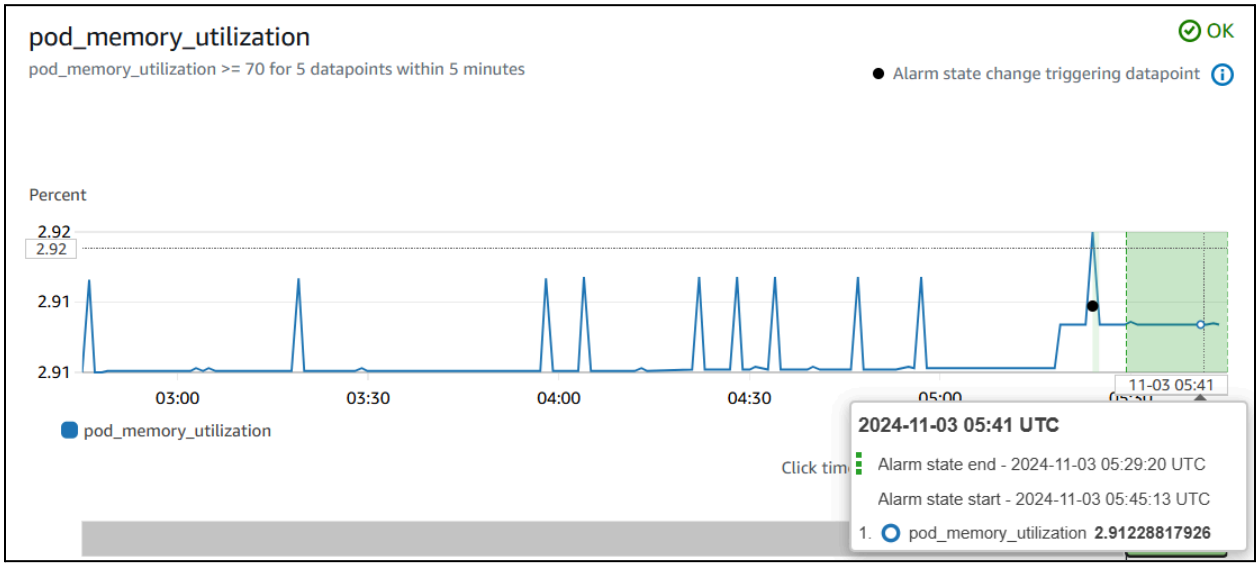
CPU Utilization Alarm:

- Setting a threshold for CPU usage to be greater than or equal to 50% for 5 datapoints. If the ‘backend’ pods exceed this threshold over a specified period (5 minutes), an alarm triggers.
- The alert can indicate an over-utilization of resources on the backend pods which may need to be scaled up.



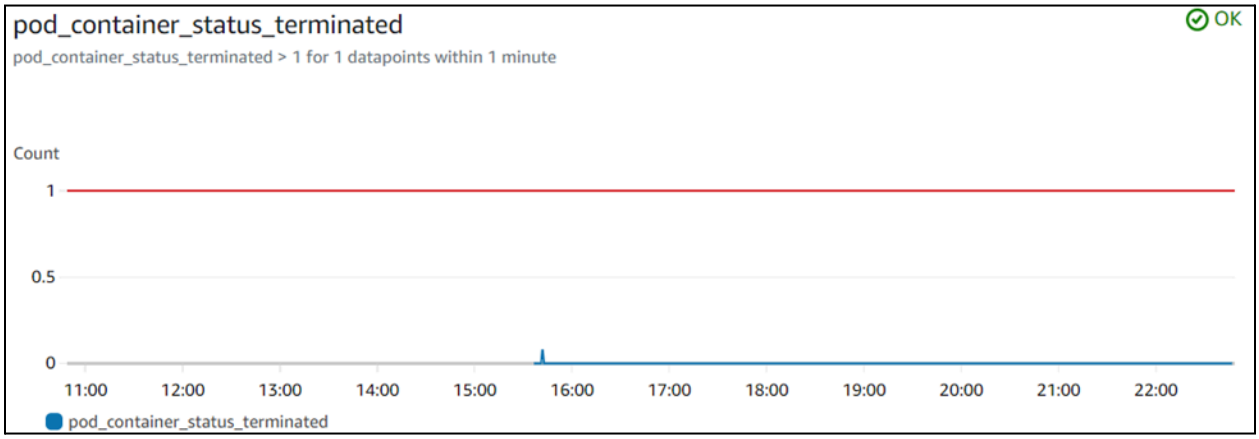
Memory Usage Alarm:

- Configuring an alert to trigger if the memory usage on the MongoDB pods exceeds a set threshold of 70% for 5 minutes.High memory usage alerts can help identify memory leaks or containers that need more memory allocation.

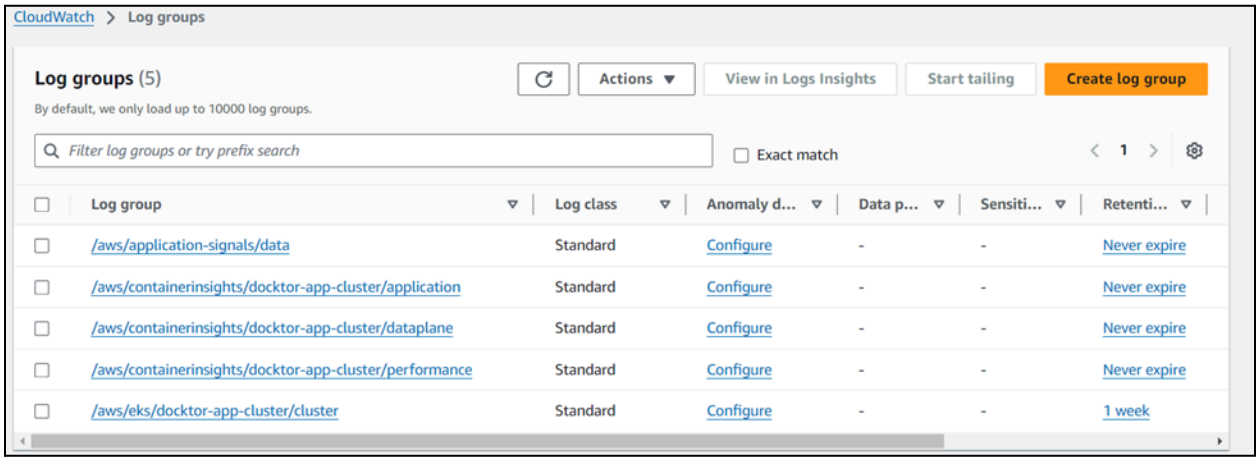


Container Termination Errors Alarm:

- Monitor for abnormal container terminations, which could indicate application errors or resource constraints.
- The alert is configured to detect a specific error rate or count within a certain time frame (e.g., more than 5 termination errors in 1 hour).
- By detecting termination patterns early, teams can proactively investigate and address issues before they affect end-users.



We implemented CloudWatch logging from the outset, ensuring all logs from EKS cluster are sent to CloudWatch for monitoring. This configuration provides access to both real-time and historical logs, facilitating easy activity tracking and swift issue troubleshooting. By consolidating logs in CloudWatch, we've enhanced our visibility into our services' performance and health.



Horizontal pod autoscaling in AWS EKS

The **Horizontal Pod Autoscaler (HPA**) in Amazon EKS is essential for dynamically managing application workloads based on real-time CPU utilization, enabling efficient use of resources and cost savings. Our goal was to implement HPA to automatically adjust the number of running pods in response to varying traffic demands, ensuring optimal performance without manual intervention. This setup used the Kubernetes **Metrics Server** to collect resource metrics and applied the **HPA** to a backend deployment in EKS. Testing confirmed that the system successfully scaled pods up during high CPU load and scaled them back down when the load decreased, demonstrating HPA's effectiveness in maintaining application stability and resource efficiency in a cloud environment.

Prerequisites

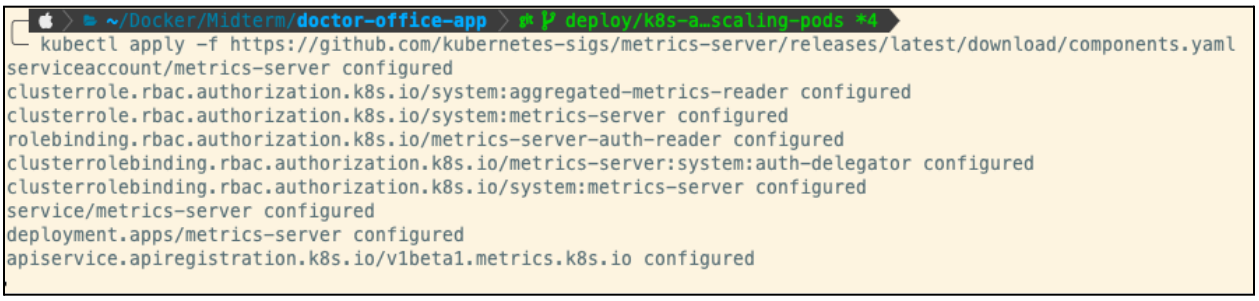
- An AWS account with permissions to create and manage EKS clusters and nodes
- AWS CLI configuration with access to the account
- Kubectl installed and configured to communicate with EKS clusters

Steps to configure autoscaling

1. **Set up Metrics Server**
   The Metrics Server is required to collect resource metrics from the Kubernetes API server, which Horizontal Pod Autoscaler(HPA) uses to make scaling decisions. To install the Metrics Server, follow these steps:

   Deploy the Metric Server:
   `$ kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml`

   https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml



2. **Configure Horizontal Pod Autoscaler(HPA)**
   The HPA automatically scales the number of pods in a deployment based on observed CPU utilization.

Create the HPA:

```
~/Docker/Midterm/doctor-office-app  deploy/k8s-a…scaling-pods *4 !1                docktor-app-cluster  11:48:29 PM
kubectl apply -f k8s/backend-hpa.yaml
horizontalpodautoscaler.autoscaling/backend-hpa created
```

The *backend-hpa.yaml* should contain specifications.

k8s > ! backend-hpa.yaml

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend  # Deployment name to apply HPA
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 20  # CPU usage threshold

```

3. **Modify individual target YAML files**
   Ensure the deployment YAML files for your applications specify the appropriate resource requests and limits, which allow the HPA to function correctly. For example:

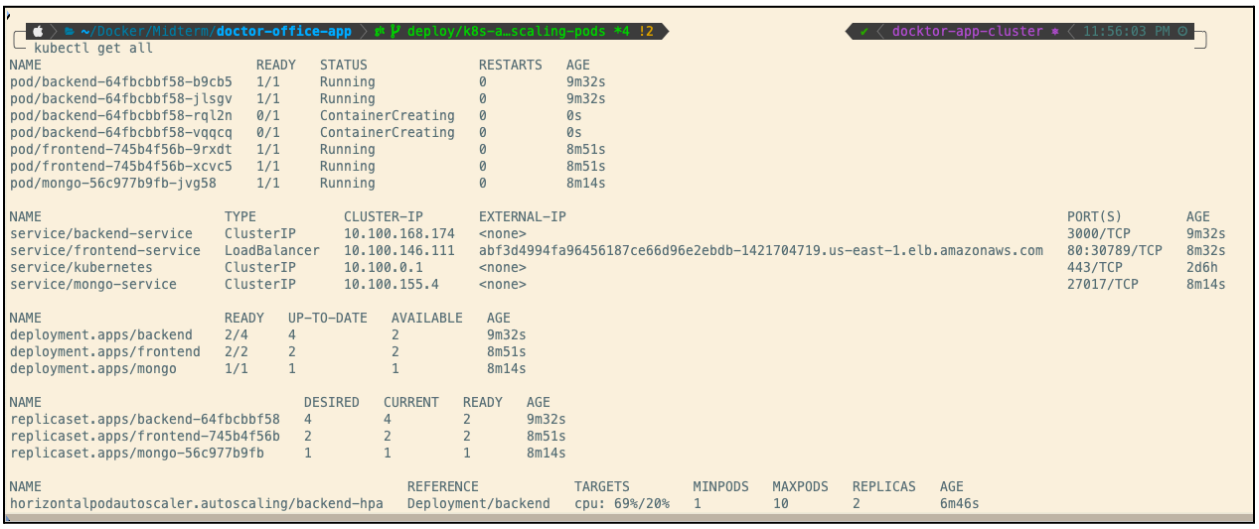
k8s > ! backend-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        # image: 913524912273.dkr.ecr.us-east-1.amazonaws.com/dock
        image: reubenthomas107/doctor-office-backend
        resources:
          requests:
            cpu: "100m"   # min CPU requests
            memory: "256Mi"   # min memory requests
          limits:
            cpu: "200m"   # max CPU requests
            memory: "512Mi"   # max memory requests
        ports:
        - containerPort: 3000
        env:
        - name: MONGODB_URL
          value: "mongo-service.default.svc.cluster.local:27017"
```

Right after deploy all the applications (frontend, backend, mongo, hap, metrics server) using:

```
$ kubectl apply -f <deployment-target>.yaml
```

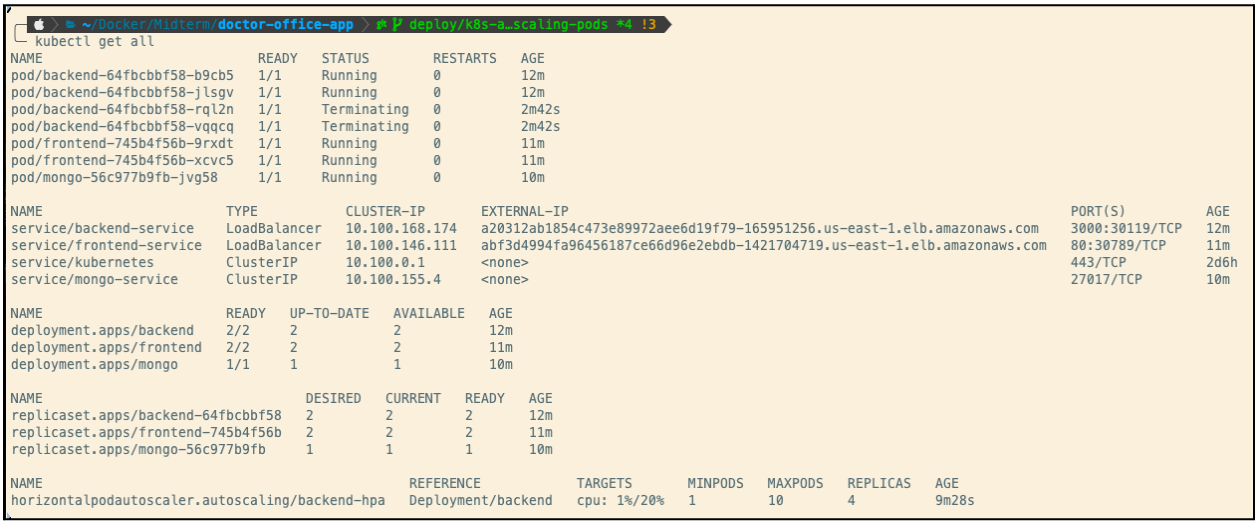
The pods initially started with 2 replicas, but since CPU usage exceeded the threshold(20%) and reached 69%, it's now creating 2 more replicas.



```
kubectl get all
NAME                          READY   STATUS              RESTARTS   AGE
pod/backend-64fbcbbf58-b9cb5  1/1     Running             0          9m32s
pod/backend-64fbcbbf58-jlsgv  1/1     Running             0          9m32s
pod/backend-64fbcbbf58-rql2n  0/1     ContainerCreating   0          0s
pod/backend-64fbcbbf58-vqqcq  0/1     ContainerCreating   0          0s
pod/frontend-745b4f56b-9rxdt  1/1     Running             0          8m51s
pod/frontend-745b4f56b-xcvc5  1/1     Running             0          8m51s
pod/mongo-56c977b9fb-jvg58    1/1     Running             0          8m14s

NAME                       TYPE           CLUSTER-IP       EXTERNAL-IP                                                              PORT(S)        AGE
service/backend-service    ClusterIP      10.100.168.174   <none>                                                                   3000/TCP       9m32s
service/frontend-service   LoadBalancer   10.100.146.111   abf3d4994fa96456187ce66d96e2ebdb-1421704719.us-east-1.elb.amazonaws.com   80:30789/TCP   8m32s
service/kubernetes         ClusterIP      10.100.0.1       <none>                                                                   443/TCP        2d6h
service/mongo-service      ClusterIP      10.100.155.4     <none>                                                                   27017/TCP      8m14s

NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend  2/4     4            2           9m32s
deployment.apps/frontend 2/2     2            2           8m51s
deployment.apps/mongo    1/1     1            1           8m14s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-64fbcbbf58  4         4         2       9m32s
replicaset.apps/frontend-745b4f56b  2         2         2       8m51s
replicaset.apps/mongo-56c977b9fb    1         1         1       8m14s

NAME                                                REFERENCE           TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/backend-hpa     Deployment/backend  cpu: 69%/20%  1         10        2          6m46s
```

```
NAME                                                REFERENCE           TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/backend-hpa     Deployment/backend  cpu: 69%/20%  1         10        2          6m54s
```

Now that the replicas have increased to 4, the CPU usage is going down, and 2 of the pods are terminating



```
kubectl get all
NAME                          READY   STATUS        RESTARTS   AGE
pod/backend-64fbcbbf58-b9cb5  1/1     Running       0          12m
pod/backend-64fbcbbf58-jlsgv  1/1     Running       0          12m
pod/backend-64fbcbbf58-rql2n  1/1     Terminating   0          2m42s
pod/backend-64fbcbbf58-vqqcq  1/1     Terminating   0          2m42s
pod/frontend-745b4f56b-9rxdt  1/1     Running       0          11m
pod/frontend-745b4f56b-xcvc5  1/1     Running       0          11m
pod/mongo-56c977b9fb-jvg58    1/1     Running       0          10m

NAME                       TYPE           CLUSTER-IP       EXTERNAL-IP                                                              PORT(S)          AGE
service/backend-service    LoadBalancer   10.100.168.174   a20312ab1854c473e89972aee6d19f79-165951256.us-east-1.elb.amazonaws.com   3000:30119/TCP   12m
service/frontend-service   LoadBalancer   10.100.146.111   abf3d4994fa96456187ce66d96e2ebdb-1421704719.us-east-1.elb.amazonaws.com   80:30789/TCP     11m
service/kubernetes         ClusterIP      10.100.0.1       <none>                                                                   443/TCP          2d6h
service/mongo-service      ClusterIP      10.100.155.4     <none>                                                                   27017/TCP        10m

NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend  2/2     2            2           12m
deployment.apps/frontend 2/2     2            2           11m
deployment.apps/mongo    1/1     1            1           10m

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-64fbcbbf58  2         2         2       12m
replicaset.apps/frontend-745b4f56b  2         2         2       11m
replicaset.apps/mongo-56c977b9fb    1         1         1       10m

NAME                                                REFERENCE           TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/backend-hpa     Deployment/backend  cpu: 1%/20%   1         10        4          9m28s
```

```
NAME                                                REFERENCE           TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/backend-hpa     Deployment/backend  cpu: 1%/20%   1         10        4          9m28s
```

```
~/Docker/Midterm/doctor-office-app  deploy/k8s-a…scaling-pods *4 !3
kubectl get all
NAME                          READY   STATUS              RESTARTS   AGE
pod/backend-64fbcbbf58-25rb2  0/1     ContainerCreating   0          0s
pod/backend-64fbcbbf58-b9cb5  1/1     Running             0          12m
pod/backend-64fbcbbf58-jlsgv  1/1     Running             0          12m
pod/backend-64fbcbbf58-q8sx9  0/1     ContainerCreating   0          0s
pod/backend-64fbcbbf58-rql2n  1/1     Terminating         0          2m45s
pod/backend-64fbcbbf58-vqqcq  1/1     Terminating         0          2m45s
pod/frontend-745b4f56b-9rxdt  1/1     Running             0          11m
pod/frontend-745b4f56b-xcvc5  1/1     Running             0          11m
pod/mongo-56c977b9fb-jvg58    1/1     Running             0          10m

NAME                       TYPE           CLUSTER-IP       EXTERNAL-IP                                                               PORT(S)          AGE
service/backend-service    LoadBalancer   10.100.168.174   a20312ab1854c473e89972aee6d19f79-165951256.us-east-1.elb.amazonaws.com    3000:30119/TCP   12m
service/frontend-service   LoadBalancer   10.100.146.111   abf3d4994fa96456187ce66d96e2ebdb-1421704719.us-east-1.elb.amazonaws.com   80:30789/TCP     11m
service/kubernetes         ClusterIP      10.100.0.1       <none>                                                                    443/TCP          2d6h
service/mongo-service      ClusterIP      10.100.155.4     <none>                                                                    27017/TCP        10m

NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend    2/4     4            2           12m
deployment.apps/frontend   2/2     2            2           11m
deployment.apps/mongo      1/1     1            1           10m

NAME                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-64fbcbbf58   4         4         2       12m
replicaset.apps/frontend-745b4f56b   2         2         2       11m
replicaset.apps/mongo-56c977b9fb     1         1         1       10m

NAME                                                    REFERENCE            TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/backend-hpa   Deployment/backend   cpu: 1%/20%   1         10        2          9m31s
```

```
NAME                                                    REFERENCE            TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/backend-hpa   Deployment/backend   cpu: 1%/20%   1         10        2          9m31s
```

4. Test the HPA

Before sending API requests:

```
~/Docker/Midterm/doctor-office-app  deploy/k8s-a…scaling-pods *4 !3
kubectl get hpa
NAME          REFERENCE              TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
backend-hpa   Deployment/backend     cpu: 1%/20%   1         10        4          12m

~/Docker/Midterm/doctor-office-app  deploy/k8s-a…scaling-pods *4 !3
kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
backend-64fbcbbf58-25rb2    1/1     Running   0          2m41s
backend-64fbcbbf58-b9cb5    1/1     Running   0          14m
backend-64fbcbbf58-jlsgv    1/1     Running   0          14m
backend-64fbcbbf58-q8sx9    1/1     Running   0          2m41s
frontend-745b4f56b-9rxdt    1/1     Running   0          14m
frontend-745b4f56b-xcvc5    1/1     Running   0          14m
mongo-56c977b9fb-jvg58      1/1     Running   0          13m
```

Sent 1,000 API requests to the backend simultaneously:

```
~/Docker/Midterm/doctor-office-app  deploy/k8s-aws-deployment *3 !6 ?2
└ hey -n 1000 -c 100 http://a20312ab1854c473e89972aee6d19f79-165951256.us-east-1.elb.amazonaws.com:3000/appointments

Summary:
  Total:        5.4712 secs
  Slowest:      1.4848 secs
  Fastest:      0.0169 secs
  Average:      0.3984 secs
  Requests/sec: 182.7741

  Total data:   2000 bytes
  Size/request: 2 bytes

Response time histogram:
  0.017 [1]     |
  0.164 [61]    |■■■■■■■■
  0.310 [324]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.457 [299]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.604 [192]   |■■■■■■■■■■■■■■■■■■■■■■■■
  0.751 [80]    |■■■■■■■■■■
  0.898 [17]    |■■
  1.044 [12]    |■
  1.191 [5]     |■
  1.338 [4]     |
  1.485 [5]     |■


Latency distribution:
  10% in 0.1935 secs
  25% in 0.2868 secs
  50% in 0.3787 secs
  75% in 0.4947 secs
  90% in 0.6249 secs
  95% in 0.7196 secs
  99% in 1.1907 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0060 secs, 0.0169 secs, 1.4848 secs
  DNS-lookup:   0.0026 secs, 0.0000 secs, 0.0270 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0007 secs
  resp wait:    0.3919 secs, 0.0165 secs, 1.4847 secs
  resp read:    0.0001 secs, 0.0000 secs, 0.0019 secs

Status code distribution:
  [200] 1000 responses
```

After sending API requests:

```
~/Docker/Midterm/doctor-office-app  deploy/k8s-a…scaling-pods *4 !3
└ kubectl get hpa
NAME          REFERENCE            TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
backend-hpa   Deployment/backend   cpu: 27%/20%   1         10        4          12m

~/Docker/Midterm/doctor-office-app  deploy/k8s-a…scaling-pods *4 !3
└ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
backend-64fbcbbf58-25rb2      1/1     Running   0          2m58s
backend-64fbcbbf58-b9cb5      1/1     Running   0          15m
backend-64fbcbbf58-c8ggq      1/1     Running   0          13s
backend-64fbcbbf58-jlsgv      1/1     Running   0          15m
backend-64fbcbbf58-q8sx9      1/1     Running   0          2m58s
backend-64fbcbbf58-z2j2g      1/1     Running   0          13s
frontend-745b4f56b-9rxdt      1/1     Running   0          14m
frontend-745b4f56b-xcvc5      1/1     Running   0          14m
mongo-56c977b9fb-jvg58        1/1     Running   0          13m

~/Docker/Midterm/doctor-office-app  deploy/k8s-a…scaling-pods *4 !3
└ kubectl get hpa
NAME          REFERENCE            TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
backend-hpa   Deployment/backend   cpu: 12%/20%   1         10        6          12m

~/Docker/Midterm/doctor-office-app  deploy/k8s-a…scaling-pods *4 !3
└ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
backend-64fbcbbf58-25rb2      1/1     Running   0          3m4s
backend-64fbcbbf58-b9cb5      1/1     Running   0          15m
backend-64fbcbbf58-c8ggq      1/1     Running   0          19s
backend-64fbcbbf58-jlsgv      1/1     Running   0          15m
backend-64fbcbbf58-q8sx9      1/1     Running   0          3m4s
backend-64fbcbbf58-z2j2g      1/1     Running   0          19s
frontend-745b4f56b-9rxdt      1/1     Running   0          14m
frontend-745b4f56b-xcvc5      1/1     Running   0          14m
mongo-56c977b9fb-jvg58        1/1     Running   0          14m
```

CPU usage increases, and replicas have increased to 6

Review the test results

During testing, we observed the following:

Initial State: The deployment started with a specified number of replicas.
Scaling Behavior: Upon reaching CPU utilization thresholds, the HPA successfully scaled the number of pods up to the maximum specified.
Restoration: When the CPU usage dropped, the HPA scaled down the pods as expected.

Additional Features

GitHub

In this project, we implemented a structured version control process using **GitHub**, leveraging a branching strategy to facilitate collaboration and effective code management. Each contributor develops features or resolves issues on separate branches, reducing conflicts and maintaining a clear record of changes. We protected the main branch to ensure code integrity and enforce quality standards, requiring all updates to undergo peer review before integration. Pull Requests (PRs) were used for submitting changes, enabling the team to review, discuss, and verify modifications before they are merged into the main branch. Additionally, GitHub Actions powers our continuous integration and deployment (CI/CD) pipeline, automating the build, and deployment processes. This automation enhances productivity and ensures consistent application performance by verifying that each update meets project requirements before integration. This systematic approach ensures secure code sharing, maintains efficient workflows, and promotes reliable application deployment.

Continuous Integration and Continuous Deployment (CI/CD)

To automate the build and deployment process, we implemented CI/CD using GitHub Actions which would trigger a CI/CD pipeline whenever the code is merged into the ‘main’ or ‘deploy/*’ branch. This automation enhances productivity by seamlessly deploying changes to the application in AWS. The jobs in the workflow ‘yaml’ file use secrets and variables defined in the GitHub repository secrets, preventing credentials from being hard-coded in the repository. The CI/CD runner interacts with AWS using an IAM user defined as ‘CI_RUNNER_ROLE’ in GitHub. The pipeline consists of four different jobs namely ‘check-prerequisites’, ‘build’, ‘deploy-eks-cluster’ and ‘prod-k8s-deploy’.

Workflow File (deploy.yaml):
https://github.com/reubenthomas107/doctor-office-app/blob/main/.github/workflows/deploy.yaml

Jobs:

‘check-prerequisites’:
Used to check if all the command-line tools are installed on the runner.

```yaml
check-prerequisites:
  runs-on: ubuntu-latest
  steps:
    - name: "Check/Install Prerequisites"
      run: |
        echo "Checking all the runtime prerequisites"
        aws --version
        kubectl version --client
        helm version
      continue-on-error: false
```

‘build’:
Builds the Docker images for the microservices and pushes it into the AWS ECR registry. The job authenticates with AWS using the IAM user account created for the CI/CD pipeline runner.

```yaml
build:
  runs-on: ubuntu-latest
  needs: check-prerequisites
  steps:
  - name: Checkout Code
    uses: actions/checkout@v4.2.2

  - name: Configure AWS Credentials
    uses: aws-actions/configure-aws-credentials@v4
    with:
      aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
      aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      aws-region: ${{ vars.AWS_REGION }}

  - name: Login to ECR
    run: |
      aws sts get-caller-identity
      aws ecr get-login-password --region $AWS_REGION | \
      docker login --username AWS --password-stdin $ECR_REGISTRY

  - name: Build and Push Frontend Microservice Image to ECR
    run: |
      docker build -t frontend:$IMAGE_TAG ./doctor-office-frontend
      docker tag frontend:$IMAGE_TAG $ECR_REGISTRY/docktor-app/doctor-office-frontend:$IMAGE_TAG
      docker push $ECR_REGISTRY/docktor-app/doctor-office-frontend:$IMAGE_TAG

  - name: Build and Push Backend Microservice Image to ECR
    run: |
      docker build -t backend:$IMAGE_TAG ./doctor-office-backend
      docker tag backend:$IMAGE_TAG $ECR_REGISTRY/docktor-app/doctor-office-backend:$IMAGE_TAG
      docker push $ECR_REGISTRY/docktor-app/doctor-office-backend:$IMAGE_TAG
```

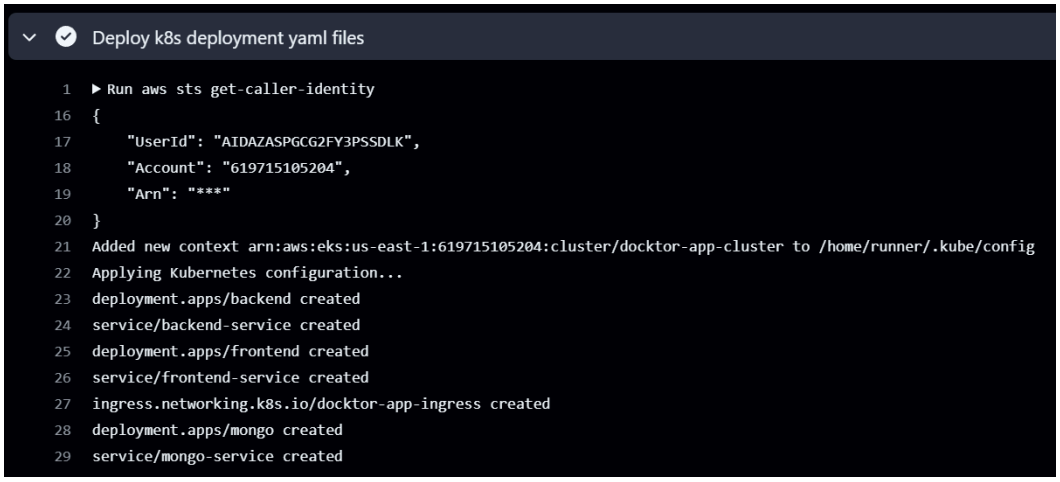
‘deploy-eks-cluster’:
This job deploys the EKS Cluster on AWS and installs the add-ons (application load balancer controller and CloudWatch observability) if the cluster is not already running. However, if the ‘docktor-app-cluster’ is already running, it is configured to map the IAM user of the ‘runner’ to the kubernetes ‘system’ role which gives the “CI_RUNNER_ROLE” user access to the control plane.

```
- name: Deploy EKS Cluster
  id: eks_cluster_setup
  run: |
    if ! eksctl get cluster -n $CLUSTER_NAME --region $AWS_REGION; then
      echo "Verify eksctl installation.."
      eksctl version
      echo "Creating EKS Cluster...."
      eksctl create cluster -f ./aws_infra/eks_cluster_setup.yaml
      echo "::set-output name=cluster_exists::false"

    else
      echo "EKS Cluster $CLUSTER_NAME already running in $AWS_REGION...."
      eksctl get cluster -n $CLUSTER_NAME --region $AWS_REGION
      eksctl create iamidentitymapping --cluster $CLUSTER_NAME --arn $CI_RUNNER_ROLE --group system:masters --username admin
      echo "::set-output name=cluster_exists::true"
    fi
  continue-on-error: false
```

‘prod-k8s-deploy’:
The job interacts with the EKS cluster to deploy our application’s kubernetes manifest files.

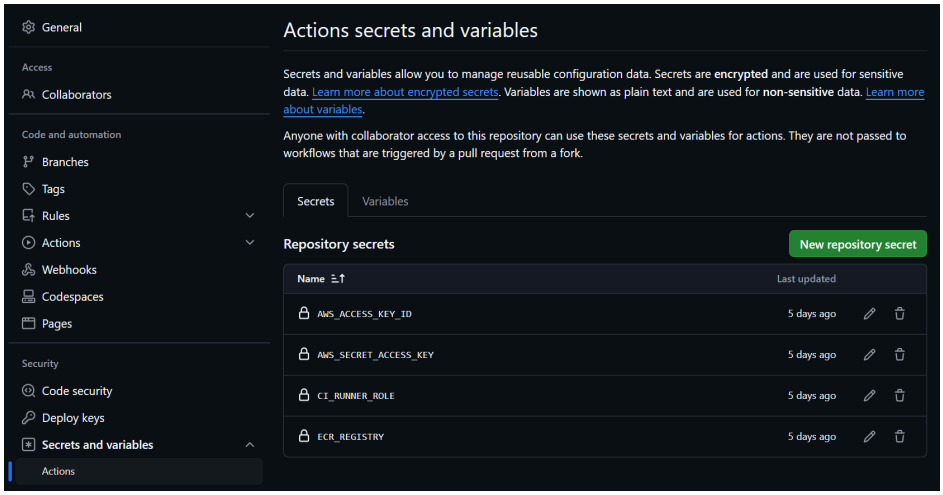
```
prod-k8s-deploy:
  runs-on: ubuntu-latest
  needs: deploy-eks-cluster
  steps:
    - name: Checkout Code
      uses: actions/checkout@v4.2.2

    - name: Configure AWS Credentials
      uses: aws-actions/configure-aws-credentials@v4
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: ${{ vars.AWS_REGION }}

    - name: Deploy k8s deployment yaml files
      run: |
        aws sts get-caller-identity
        aws eks --region $AWS_REGION update-kubeconfig --name $CLUSTER_NAME
        echo "Applying Kubernetes configuration..."
        kubectl apply -f ./k8s/
      continue-on-error: false

    - name: Check Deployment Status
      run: |
        echo "Verifying deployment status...."
        kubectl get all
        kubectl get ingress -o json
```

An example of how the job deploys our configuration files can be seen here:



Secrets Management:

Access tokens, API keys, user identifiers, and other sensitive information were securely stored as repository secrets in GitHub. This adds an additional layer of security as it prevents secrets from being accidentally exposed to the public. Additionally, the use of variables and secrets allows us to manage configurations more effectively as changes can be easily made without having to update the workflows or code files.

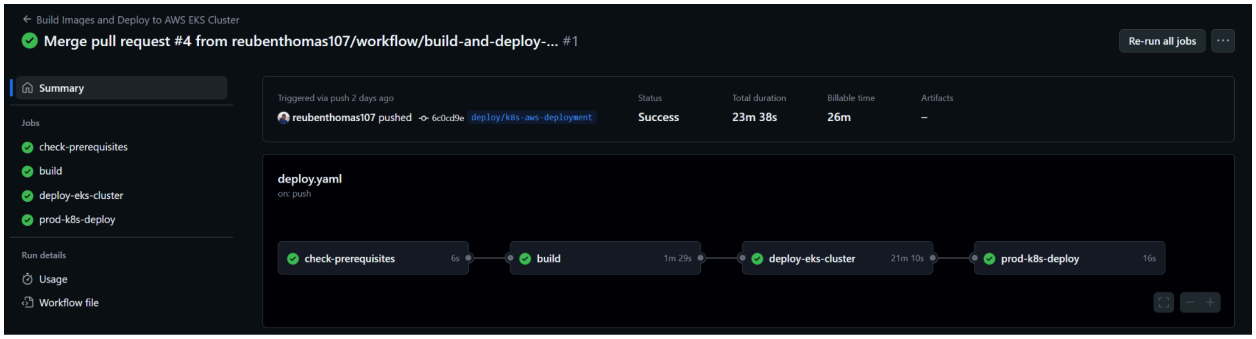


AWS Region and cluster name defined as variables:

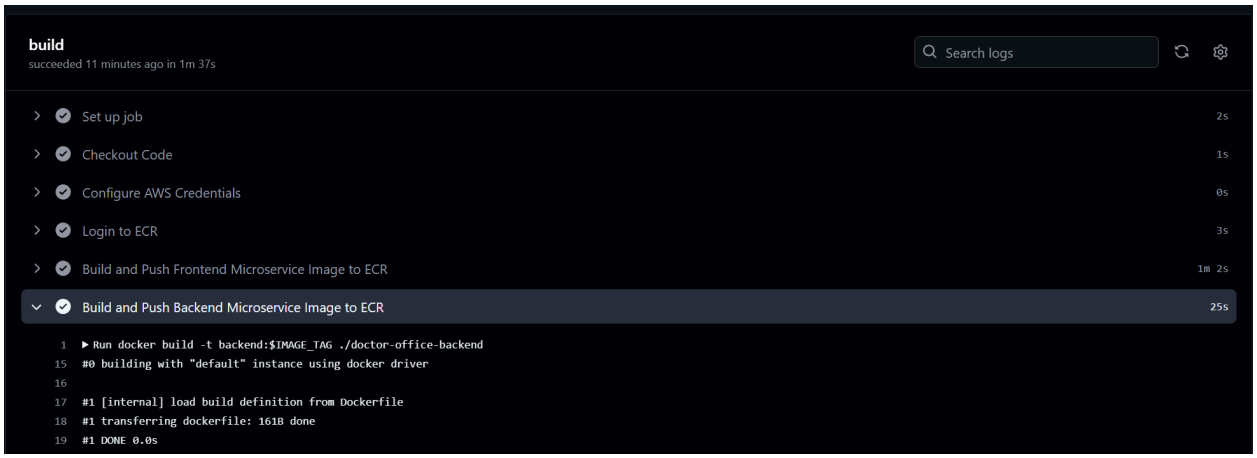
CI/CD Pipeline:

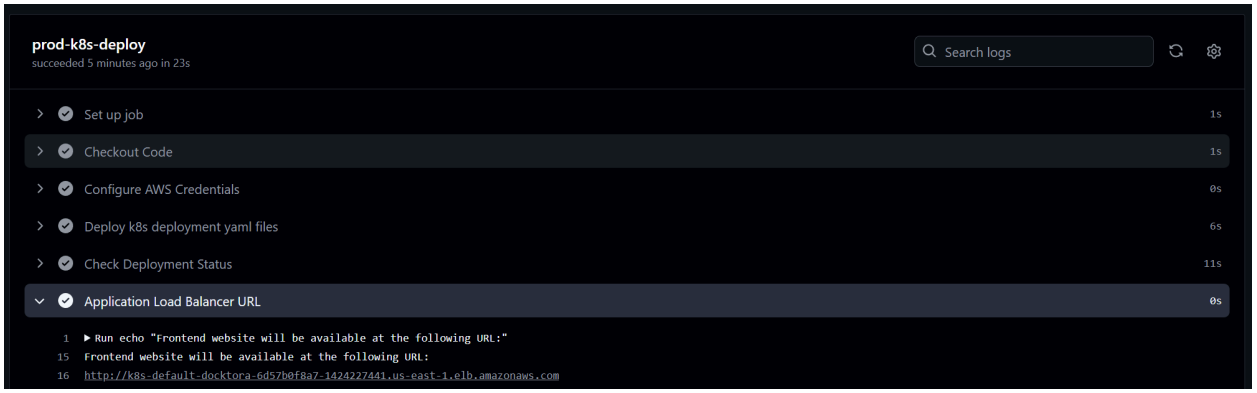
https://github.com/reubenthomas107/doctor-office-app/actions/runs/11654532783/job/32448231038

Pipeline run to build ECR images, create EKS cluster and deploy the application on AWS:



Jobs overview in a successful pipeline run:





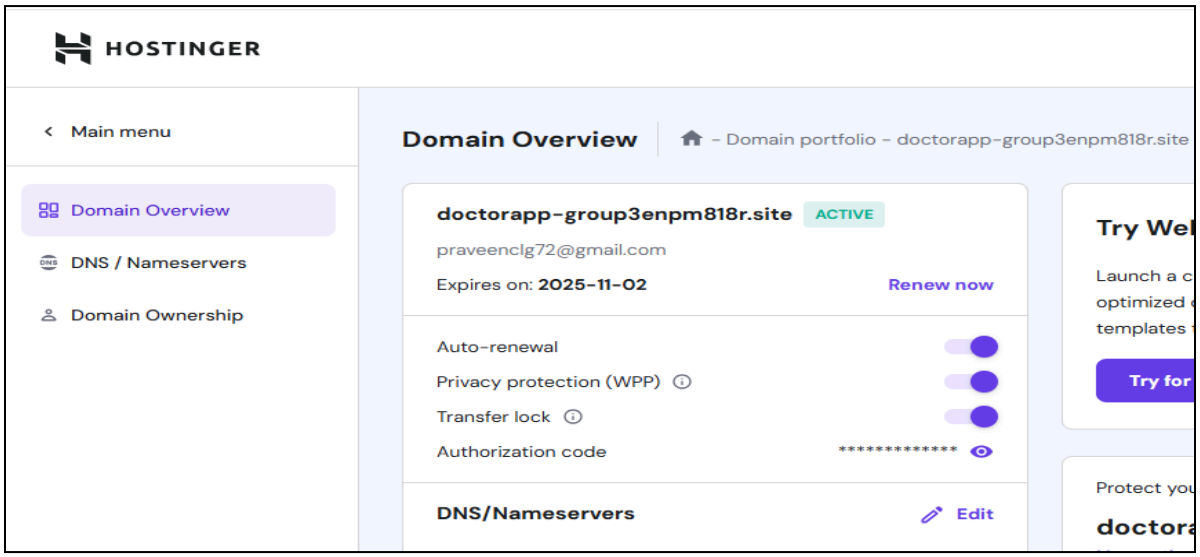
Implementing Secure Communication and Traffic Management with AWS Certificate Manager and Route 53

In this project, we implemented secure communication and optimized traffic routing by utilizing AWS Certificate Manager (ACM) and Amazon Route 53. To enhance the security of our application, we requested an SSL/TLS certificate through ACM. This certificate facilitates encrypted data transmission between client and server, ensuring that sensitive information such as user credentials and personal data remains protected from potential eavesdropping or interception during transit. The use of SSL is critical for maintaining user trust and compliance with data protection standards. The ACM service provides an efficient solution for managing SSL certificates, as it automates certificate renewals and eliminates the need for manual configuration, thus reducing maintenance efforts.

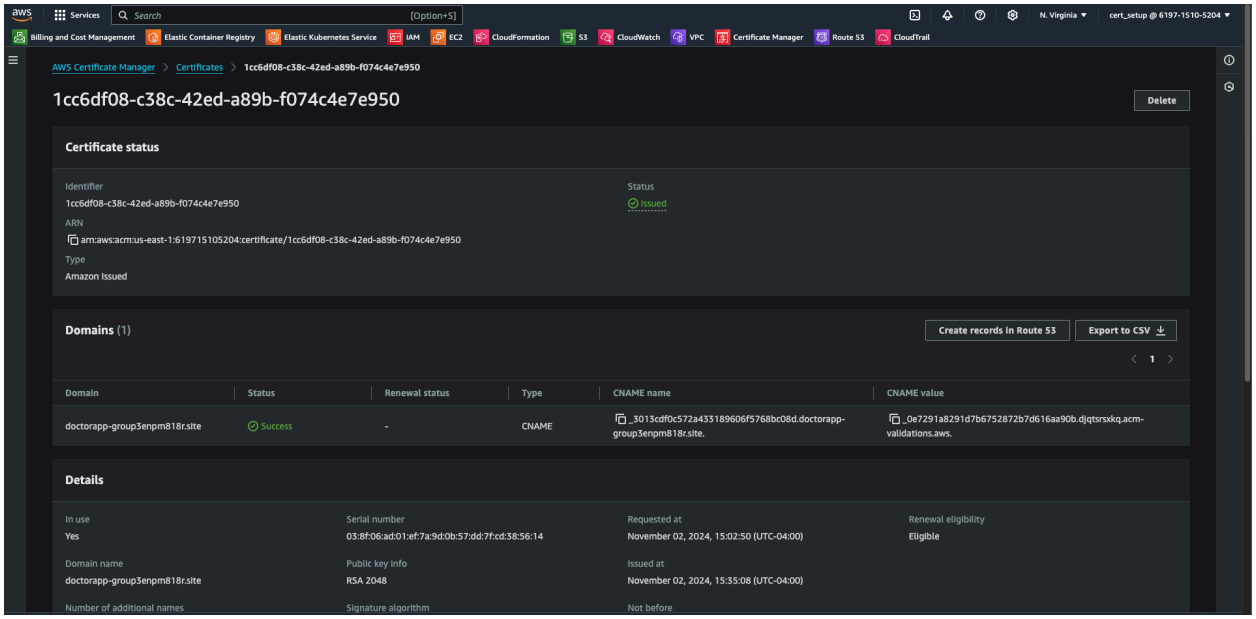
For efficient traffic management, we configured simple routing through Amazon Route 53, AWS's scalable Domain Name System (DNS) service, directing requests to the Application Load Balancer (ALB) associated with our application. This configuration ensured that all requests were accurately routed from our custom domain (***doctorapp-group3enpm818r.site***) to the appropriate resources on the ALB. Additionally, we managed the listener entries of the ALB through an ingress YAML file within Kubernetes, which enabled precise control over how incoming requests were handled. This combined approach with ACM and Route 53 facilitated secure, reliable, and well-managed access for users of our application.

Setting Up SSL Certificate using AWS Certificate Manager

Initially, after navigating to the AWS Certificate Manager console and requesting a public certificate, we used a custom domain named ***doctorapp-group3enpm818r.site*** for website hosting from Hostinger and entered it as our **Fully Qualified Domain Name**, which is essential for proving the identity of our website.

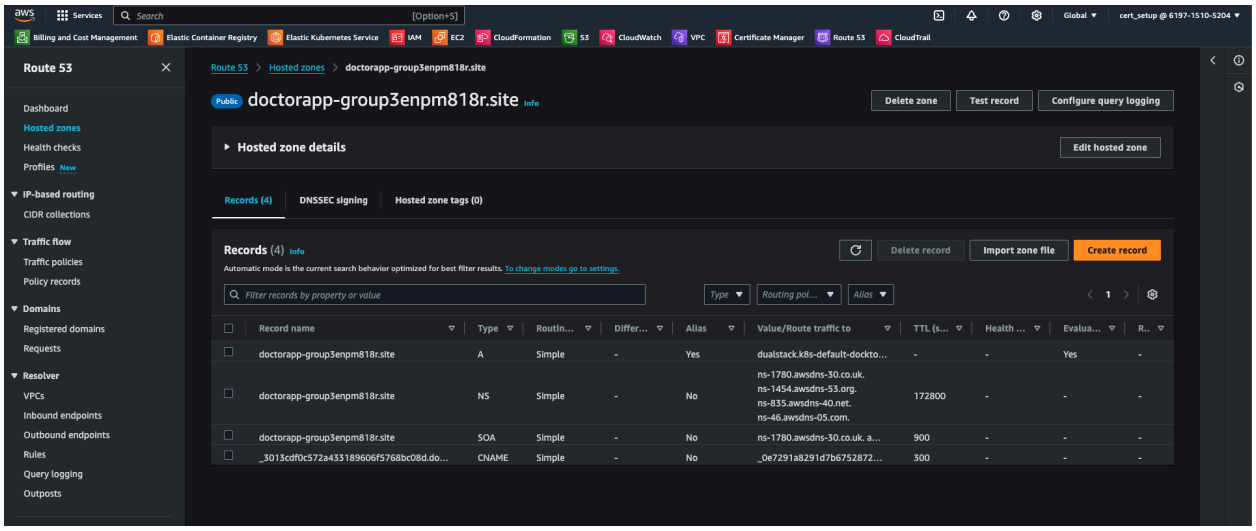


Next, choose DNS validation as your validation method, select the RSA key algorithm, add any necessary tags, and then submit your request.

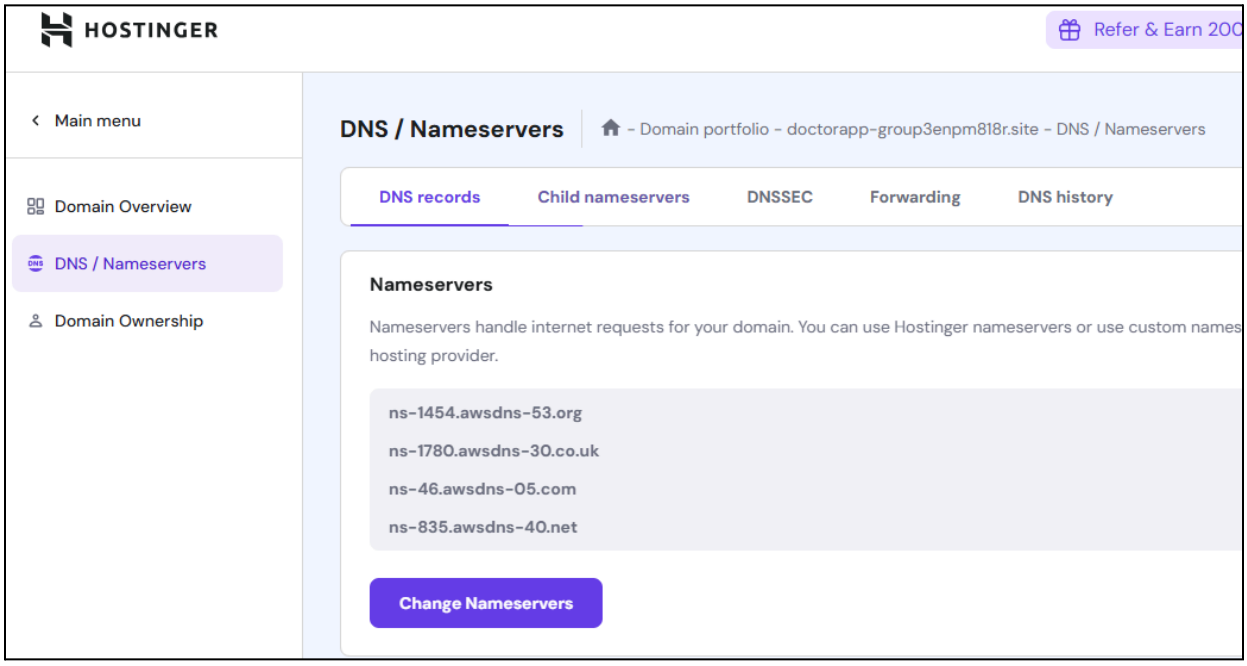


After submitting, we received a notification indicating that the certificate is pending validation. Clicking on the certificate to view its status, where we added the CNAME record (**_3013cdf0c572a433189606f5768bc08d.doctorapp-group3enpm818r.site**) for validation.

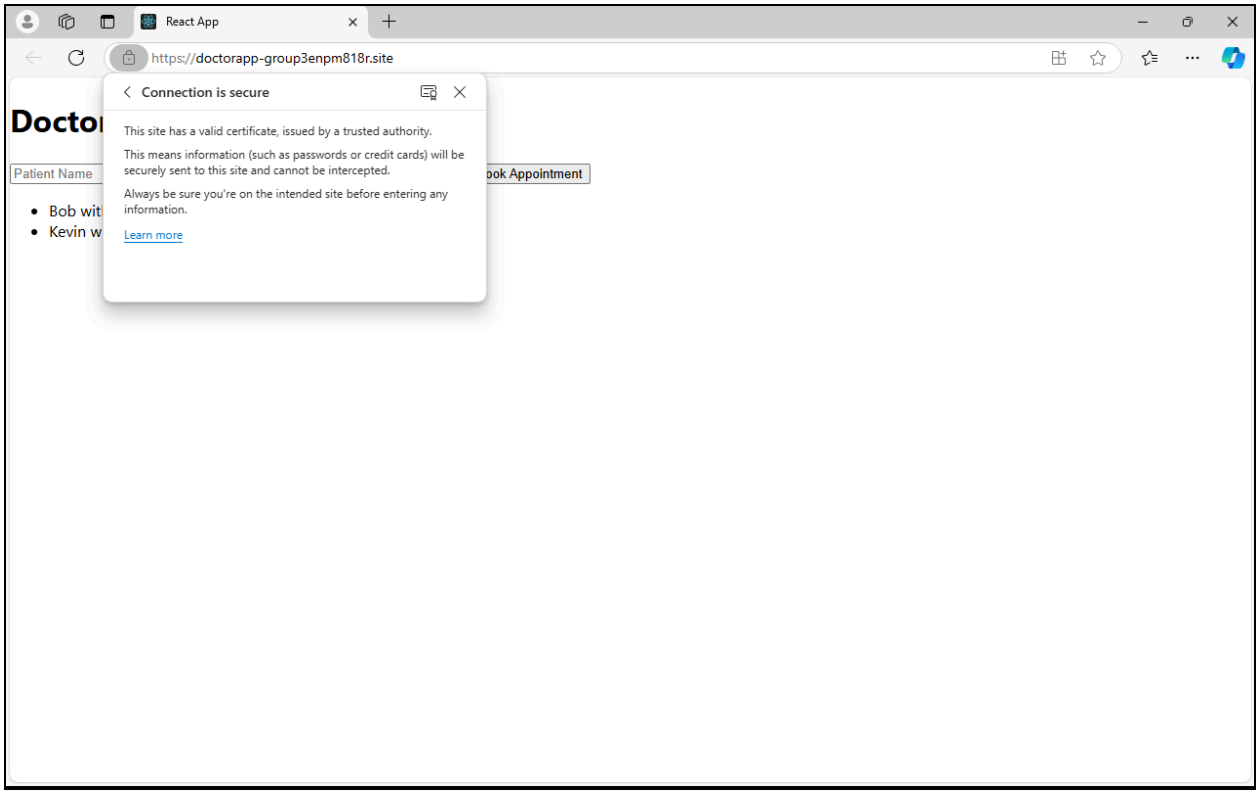
In Route 53, we clicked the option to create the required DNS records. It automatically added the necessary CNAME record to your Route 53 hosted zone.



Next, we configured our web hosting application to use the Route 53 hosted zone DNS and updated the nameservers accordingly as shown below.



We have validated that the certificate is available and could verify the same below.



References

- Set up to use AWS Certificate Manager - AWS Certificate Manager
- Monitor Amazon ECS using CloudWatch - Amazon Elastic Container Service
- Horizontal Pod Autoscaling
- Github - Amazon Elastic Block Store (EBS) CSI driver
- Store Kubernetes volumes with Amazon EBS
- Getting Started - eksctl
- Quickstart: Deploy a web app and store data - Amazon EKS
- Installation Guide - AWS Load Balancer Controller
- Connect kubectl to an EKS cluster by creating a kubeconfig file - Amazon EKS
- Kubernetes Ingress with AWS ALB Ingress Controller
- Set up end-to-end encryption for applications on Amazon EKS using cert-manager and Let's Encrypt - AWS Prescriptive Guidance
- Exposing Kubernetes Applications, Part 1: Service and Ingress Resources | Containers
- Install the Amazon CloudWatch Observability EKS add-on
- Install the CloudWatch agent with the Amazon CloudWatch Observability EKS add-on or the Helm chart - Amazon CloudWatch
- AWS EKS Kubernetes CloudWatch Container Insights - STACKSIMPLIFY
- MongoDB StatefulSet in Kubernetes | by Deeptiman Pattnaik | Medium
- Running MongoDB on Kubernetes with StatefulSets
- Configuring Amazon Route 53 as your DNS service - Amazon Route 53
- aws-ebs-csi-driver/examples/kubernetes/dynamic-provisioning/manifests/storageclass.yaml at master
- https://www.youtube.com/watch?v=5XpPiORNy1o
- https://www.youtube.com/watch?v=S8U7A-eGdOs