APPLICATION DEPLOYMENT - MINDTRACK

TASKS

Objective:

Deploy a Dockerized React application to AWS EKS using a fully automated CI/CD pipeline with GitHub, CodePipeline, CodeBuild, and ECR.

Work Flow:

- Create an EC2 instance with the help of AWS Management Console with linux OS of required configuration.
- Now, Connect an EC2 instance with an help of Windows Terminal or Gitbash or Vbox.
- To connect an EC2 instance the command is:
  - ssh -i "**key_file**" ec2-user@"**Public_IP_address**"

Key_file --- Key file of the instance with the extension .pem

Public_IP_address --- Public IP address of the instance.
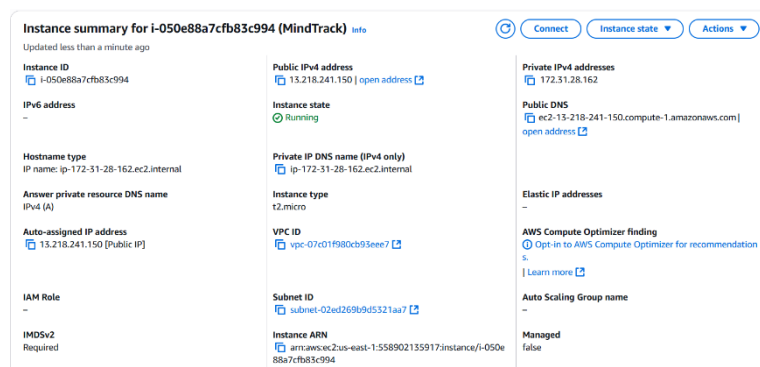
1. Deploy the given React application to a production ready state.

Application:

Step 1: Clone the React Application.

- ✓ To begin the deployment process, the first step is to obtain the source code of the React application.
- ✓ The code is publicly available in a GitHub repository. We use the **git clone** command to download the project to our local machine.
- ✓ Command:
  - **git clone https://github.com/Vennilavan12/Brain-Tasks-App.git**
  - **cd Brain-Tasks-App**
- ✓ This command clones the repository named Brain-Tasks-App from GitHub and navigates into the project directory.

```
[ec2-user@ip-172-31-33-194 ~]$ git clone https://github.com/Vennilavan12/Brain-Tasks-App.git
Cloning into 'Brain-Tasks-App'...
remote: Enumerating objects: 8, done.
remote: Total 8 (delta 0), reused 0 (delta 0), pack-reused 8 (from 1)
Receiving objects: 100% (8/8), 100.04 KiB | 5.56 MiB/s, done.
```

✓ The application files will be used in the following steps to build a Docker image and deploy it to AWS.

```
[ec2-user@ip-172-31-33-194 ~]$ cd Brain-Tasks-App
[ec2-user@ip-172-31-33-194 Brain-Tasks-App]$ ls
dist
[ec2-user@ip-172-31-33-194 Brain-Tasks-App]$ cd dist
[ec2-user@ip-172-31-33-194 dist]$ ls
assets   index.html   vite.svg
[ec2-user@ip-172-31-33-194 dist]$ |
```

Docker:

Step 2: Dockerize the Application

✓ In this step, we containerize the React application using Docker by creating a Dockerfile in the root directory of the project.

✓ Docker Installation:

```
[ec2-user@ip-172-31-86-189 Brain-Tasks-App]$ sudo docker build -t brain-task-app .
[+] Building 0.2s (7/7) FINISHED                                                              docker:default
 => [internal] load build definition from Dockerfile                                                   0.0s
 => => transferring dockerfile: 194B                                                                   0.0s
 => [internal] load metadata for docker.io/library/nginx:alpine                                        0.1s
 => [internal] load .dockerignore                                                                      0.0s
 => => transferring context: 2B                                                                        0.0s
 => [internal] load build context                                                                      0.0s
 => => transferring context: 580B                                                                      0.0s
 => [1/2] FROM docker.io/library/nginx:alpine@sha256:b2e814d28359e77bd0aa5fed1939620075e4ffa0eb20423cc557b375bd5c14ad  0.0s
 => CACHED [2/2] COPY dist/ /usr/share/nginx/html                                                      0.0s
 => exporting to image                                                                                 0.0s
 => => exporting layers                                                                                0.0s
 => => writing image sha256:c81b868b1c808cd31498bc1c26beb843c280ac7cca899c8e19a84c79d3ccb1e1           0.0s
 => => naming to docker.io/library/brain-task-app                                                      0.0s
```

✓ The Dockerfile uses a multi-stage build where the first stage builds the React application using a Node.js base image, and the second stage uses an Nginx base image to serve the static files generated during the build.

✓ This approach ensures that the final image is lightweight and optimized for production.

```
[ec2-user@ip-172-31-86-189 Brain-Tasks-App]$ sudo docker run -d -p 3000:80 brain-task-app
56f743cdec9084e8821db48160f477b3f16315101f9932d92197d55f8364bce7
```

✓ This confirms that the Dockerized application serves correctly in a production-like containerized environment.

```
[ec2-user@ip-172-31-86-189 Brain-Tasks-App]$ touch Dockerfile
[ec2-user@ip-172-31-86-189 Brain-Tasks-App]$ vi Dockerfile
"Dockerfile" 4L, 98B writtenrain-Tasks-App]$
[ec2-user@ip-172-31-86-189 Brain-Tasks-App]$ ls
Dockerfile  dist
```

✓ The Dockerfile Formate:

```
FROM public.ecr.aws/nginx/nginx:alpine
COPY dist/ /usr/share/nginx/html
EXPOSE 80
CMD [ "nginx", "-g", "daemon off;" ]
~
```

✓ After creating the Dockerfile, we built the Docker image using the docker build command and named the image brain-tasks-app.

✓ To build the dockerfile the command is:

- **docker build -t brain-tasks-app .**
- **docker run -p 3000:80 brain-tasks-app**

✓ Once the image was built successfully, we verified the container by running it locally using the docker run command and mapped it to port 3000.

✓ Now open http://<instance-ip>:3000 in your browser again — the app should load, but this time **inside a Docker container**.
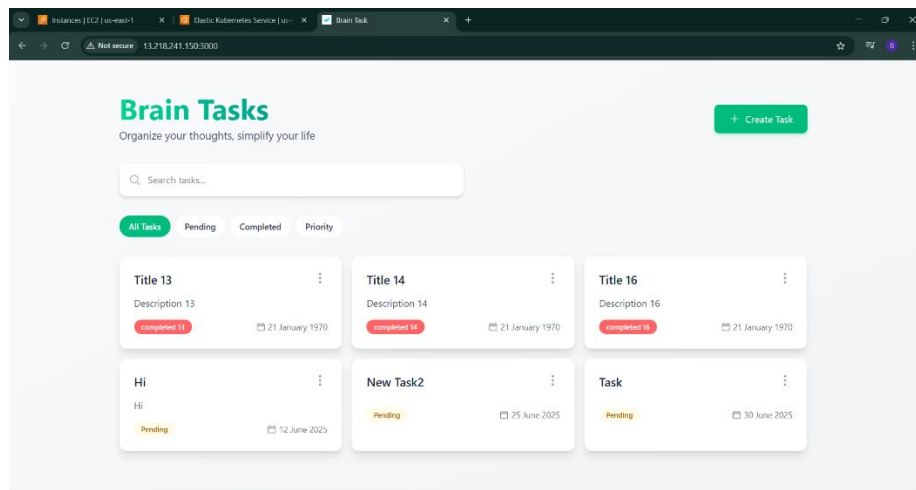
ECR:

Step 3: Push Docker Image to AWS ECR

✓ In this step, we store our Docker image in **Amazon Elastic Container Registry (ECR)** so it can be accessed later by our Kubernetes cluster.

- First, we create an ECR repository named `brain-tasks-app` using the AWS CLI. This repository will hold our Docker image in a secure and scalable manner.

- Next, we authenticate Docker with our AWS ECR registry using the `get-login-password` command. This ensures that we have permission to push images.

```
[ec2-user@ip-172-31-86-189 Brain-Tasks-App]$ aws configure
AWS Access Key ID [None]: AKIAYEIJJZRWXC6Y5OHW
AWS Secret Access Key [None]: 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
Default region name [None]: us-east-1
Default output format [None]: json
```

```
[ec2-user@ip-172-31-86-189 Brain-Tasks-App]$ aws ecr get-login-password --region us-east-1 |
cr.us-east-1.amazonaws.com
WARNING! Your password will be stored unencrypted in /home/ec2-user/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

- After building the Docker image locally, we tag it with the ECR repository URI so that Docker knows where to push the image.

```
[ec2-user@ip-172-31-86-189 Brain-Tasks-App]$ docker push 558902135917.dkr.ecr.us-east-1.amazonaws.com/brain-tasks-app
Using default tag: latest
The push refers to repository [558902135917.dkr.ecr.us-east-1.amazonaws.com/brain-tasks-app]
ed1aea3ee772: Pushed
640e06e412a9: Pushed
bd66bdd2f47f: Pushed
4f313ed230a0: Pushed
36acb230000e: Pushed
2aaacff968bc: Pushed
bbbd2d1aea89: Pushed
7b97c641cb43: Pushed
fd2758d7a50e: Pushed
latest: digest: sha256:1f3c3a0cd352e2a7b6adbc7a434655d99fde4ae5be5f49652caac7a5df82941a size: 2199
```

- Finally, we push the tagged image to ECR. This uploads our image to the cloud and makes it available for deployment in the EKS cluster.

- ✓ By completing this step, we have successfully stored our application image in a cloud-native registry, making it ready for deployment via Kubernetes.

Kubernetes:

Step 4: Setup Kubernetes in AWS EKS

- ✓ To deploy our Dockerized React application in a scalable and managed environment, we set up a Kubernetes cluster using AWS EKS (Elastic Kubernetes Service).

✓ First, we used the AWS Management Console or the eksctl CLI tool to create a new EKS cluster. The cluster was created in a specific region with default settings, and we waited until the cluster status showed as "Active".

✓ After the cluster was created, we configured our local machine to interact with it using kubectl by updating the kubeconfig file through the AWS CLI. This allowed us to run commands against the EKS cluster from our terminal.

✓ We verified the setup by running the command kubectl get svc which confirmed that our EKS cluster was running and accessible.

✓ By completing this step, we now have a managed Kubernetes environment where we can deploy our React application using Kubernetes manifests in the next step.

Step 5: Create Kubernetes Deployment and Service YAML Files

✓ In this step, we define the Kubernetes configuration files required to deploy our Dockerized React application to the EKS cluster.

✓ First, we create a deployment.yaml file which describes the desired state of the application, including the number of replicas, the Docker image to use from ECR, and the

container port. This configuration ensures that the application runs in multiple pods for high availability.

✓ The deployment.yaml file formate:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: brain-tasks-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: brain-tasks
  template:
    metadata:
      labels:
        app: brain-tasks
    spec:
      containers:
      - name: brain-tasks-container
        image: 558902135917.dkr.ecr.us-east-1.amazonaws.com/brain-tasks-app
        ports:
        - containerPort: 80
```

✓ Next, we create a service.yaml file to expose the application to the internet using a LoadBalancer. This allows users to access the application from outside the Kubernetes cluster.

✓ The service.yaml file formate:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: brain-tasks-service
spec:
  type: LoadBalancer
  selector:
    app: brain-tasks
  ports:
    - port: 80
      targetPort: 80
```

✓ After writing both files, we deploy them to the EKS cluster using the kubectl apply command. This command instructs Kubernetes to create the necessary resources based on our configuration.

✓ Commands:

- **kubectl apply -f deployment.yaml**
- **kubectl apply -f service.yaml**

✓ By completing this step, our application is now deployed and accessible through the public IP address provided by the LoadBalancer service.

✓ Command:

- **kubectl get svc**

```
[ec2-user@ip-172-31-28-162 Brain-Tasks-App]$ kubectl get svc
NAME                 TYPE          CLUSTER-IP      EXTERNAL-IP                                                                      PORT(S)        AGE
brain-tasks-service  LoadBalancer  10.100.228.44   afefd1637eae7498bb44ba9bc5525793-642403127.us-east-1.elb.amazonaws.com          80:32396/TCP   44h
kubernetes           ClusterIP     10.100.0.1      <none>                                                                           443/TCP        46h
[ec2-user@ip-172-31-28-162 Brain-Tasks-App]$
```

✓ It should show an **EXTERNAL-IP** for the LoadBalancer — that's the URL to access your app!

- **afefd1637eae7498bb44ba9bc5525793-642403127.us-east-1.elb.amazonaws.com**
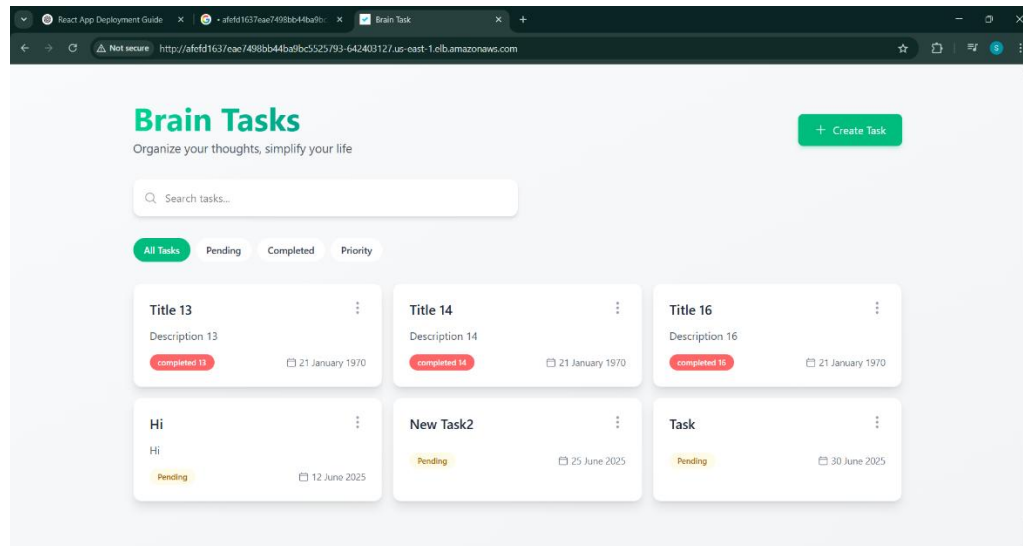
CodeBuild:

Step 6: Set Up AWS CodeBuild

✓ In this step, we automate the process of building and pushing the Docker image to Amazon ECR using AWS CodeBuild.
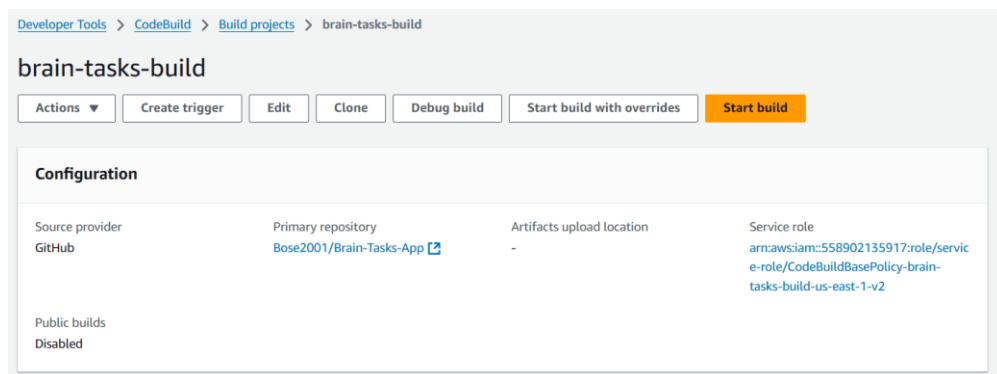
✓ First, a buildspec.yml file is created in the root directory of the project. This file defines the build phases and the commands required to build the Docker image and push it to the ECR repository.

✓ The buildspec.yml file includes the following phases:

- **Pre-Build Phase**: Authenticates Docker with Amazon ECR using the AWS CLI.
- **Build Phase**: Builds the Docker image from the Dockerfile and tags it with the ECR repository URL.
- **Post-Build Phase**: Pushes the tagged Docker image to the ECR repository.

✓ The buildspec.yml file format:

```
version: 0.2

phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 558902135917.dkr.ecr.us-east-1.amazonaws.com/brain-task
s-app
  build:
    commands:
      - echo Building the Docker image...
      - docker build -t brain-tasks-app .
      - docker tag brain-tasks-app 558902135917.dkr.ecr.us-east-1.amazonaws.com/brain-tasks-app
  post_build:
    commands:
      - echo Pushing the Docker image...
      - docker push 558902135917.dkr.ecr.us-east-1.amazonaws.com/brain-tasks-app
      - echo Updating Kubernetes deployment...
      - aws eks update-kubeconfig --region us-east-1 --name brain-tasks-cluster
      - kubectl set image deployment/brain-tasks-deployment brain-tasks-container=558902135917.dkr.ecr.us-east-1.amazonaws.com/brain-tasks-app
      - kubectl rollout status deployment/brain-tasks-deployment
```

✓ After creating the buildspec.yml file, an AWS CodeBuild project is set up in the AWS Management Console.

✓ The source is connected to the GitHub repository where the project code is hosted. The environment is configured to use a managed image (e.g., Amazon Linux 2 with Docker support).

✓ Finally, CodeBuild is triggered to execute the instructions in buildspec.yml, which results in a new Docker image being pushed to ECR.

✓ This step ensures that every time changes are pushed to the repository, the latest version of the application is automatically built and stored in ECR for deployment.
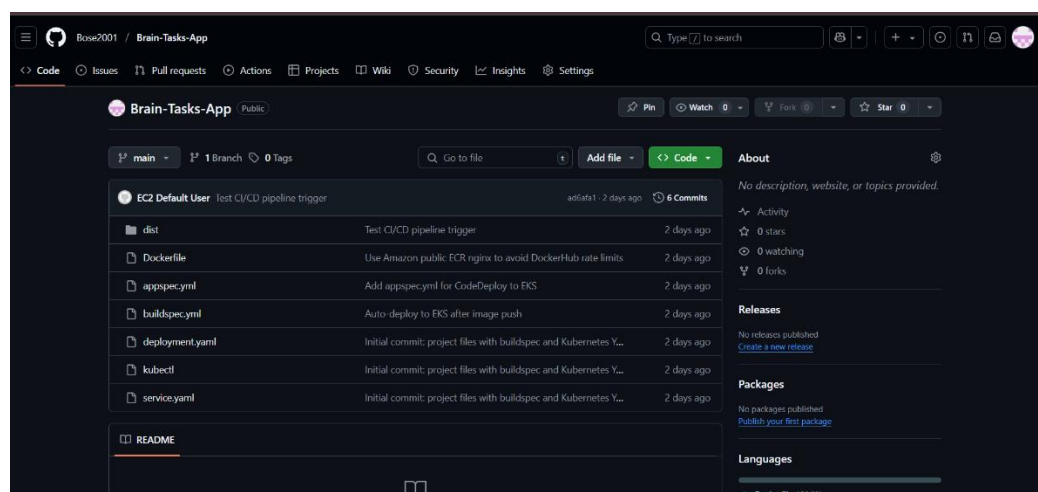
CodeDeploy:

Step 7: Configure CodeDeploy for EKS

✓ In this step, we configure AWS CodeDeploy to deploy our application to the EKS cluster using Kubernetes manifests.

✓ We begin by creating an appspec.yml file, which defines how CodeDeploy will handle the deployment process. This file specifies the target EKS cluster, the namespace (default in our case), and the Kubernetes resource files that need to be applied (i.e., deployment.yaml and service.yaml).

✓ The appspec.yml acts as the deployment blueprint for CodeDeploy and ensures that the application is rolled out to the EKS cluster in a structured and repeatable manner.

✓ This configuration is necessary for CodeDeploy to locate and apply the appropriate manifests to the correct cluster environment.

✓ By linking CodeDeploy with EKS using this setup, we enable automated and consistent deployment of containerized applications directly into our Kubernetes cluster.

<u>Version Control:</u>

Step 8: Push the Project Code to GitHub

- ✓ In this step, the complete project codebase, including the React application, Dockerfile, Kubernetes deployment files, and CI/CD configuration files (like buildspec.yml and appspec.yml), was pushed to a GitHub repository.
- ✓ The process began by initializing a Git repository in the project folder using the git init command.
- ✓ The remote repository was then added using the git remote add origin command, pointing to the desired GitHub repository URL. After staging all the files using git add . and committing the changes with git commit -m "Initial commit", the project was successfully pushed to the main branch using git push -u origin main.
- ✓ The commands:
  - git init
  - git remote add origin https://github.com/<your-username>/<your-repo-name>.git
  - git add .
  - git commit -m "Initial commit: Docker, buildspec, and Kubernetes files"
  - git push -u origin main
- ✓ This step ensures the codebase is version controlled and integrated with AWS CodePipeline for automated build and deployment.

<u>CodePipeline:</u>

Step 9: Set Up CodePipeline

- ✓ In this step, we configured AWS CodePipeline to automate the end-to-end CI/CD process, which connects our GitHub repository to AWS CodeBuild and AWS CodeDeploy.

- ✓ First, we selected **GitHub** as the source provider and authorized access to the repository containing our application code, Dockerfile, and deployment manifests.
- ✓ Next, we added a **Build stage** using the previously created **CodeBuild project**, which is responsible for building the Docker image and pushing it to AWS ECR based on the instructions defined in the buildspec.yml file.
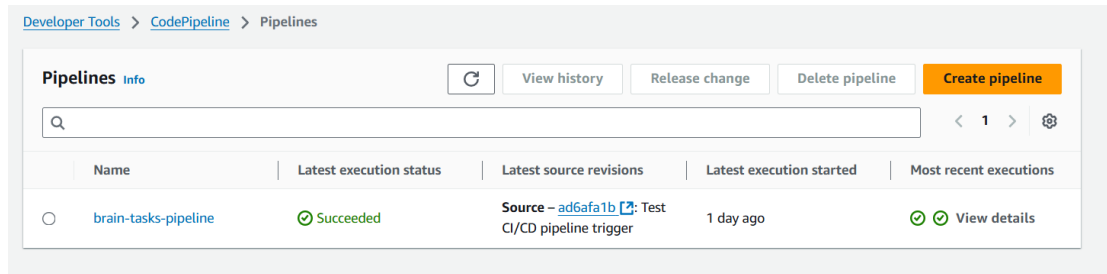
- ✓ Finally, we added a **Deploy stage** by integrating **AWS CodeDeploy**, which uses the appspec.yml to deploy the application to the EKS cluster using the Kubernetes manifests (deployment.yaml and service.yaml).
- ✓ Once the pipeline was created and executed, it automatically triggered a new build and deployment whenever code changes were pushed to the GitHub repository.
- ✓ This ensures that our application remains up-to-date and follows a continuous delivery workflow.
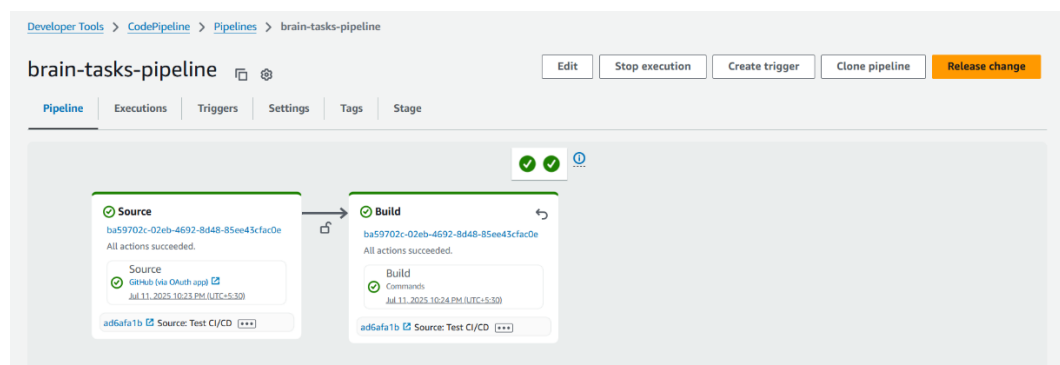
Monitoring:

Step 10: Monitoring with CloudWatch

- ✓ To monitor the application and deployment pipeline, we used **Amazon CloudWatch** for centralized logging and performance tracking.
- ✓ CloudWatch helps in identifying issues during the build, deployment, and runtime of the application.
- ✓ For CodeBuild and CodeDeploy, logs are automatically pushed to CloudWatch, where we can view detailed build and deployment history, error messages, and timestamps to troubleshoot issues.

✓ Additionally, for monitoring the **application running in the EKS cluster**, we used the following command to fetch logs directly from the application pods:

- **kubectl logs -l app=brain-tasks**

✓ This command retrieves the logs from the containers matching the label app=brain-tasks-app, helping to verify that the application is running correctly and to check for any runtime errors.

✓ In summary, CloudWatch provides an essential observability layer to track the health of our CI/CD pipeline and application, ensuring reliable and continuous deployment.

Conclusion:

✓ Successfully deployed the "Brain Tasks App" React application to a production-ready environment on AWS using Docker, ECR, EKS, CodeBuild, CodeDeploy, and CodePipeline. The process is fully automated with monitoring enabled via CloudWatch.

✓ **Application deployed kubernetes Loadbalancer ARN.**

- http://afefd1637eae7498bb44ba9bc5525793-642403127.us-east-1.elb.amazonaws.com

************** **TASK COMPLETED** **************