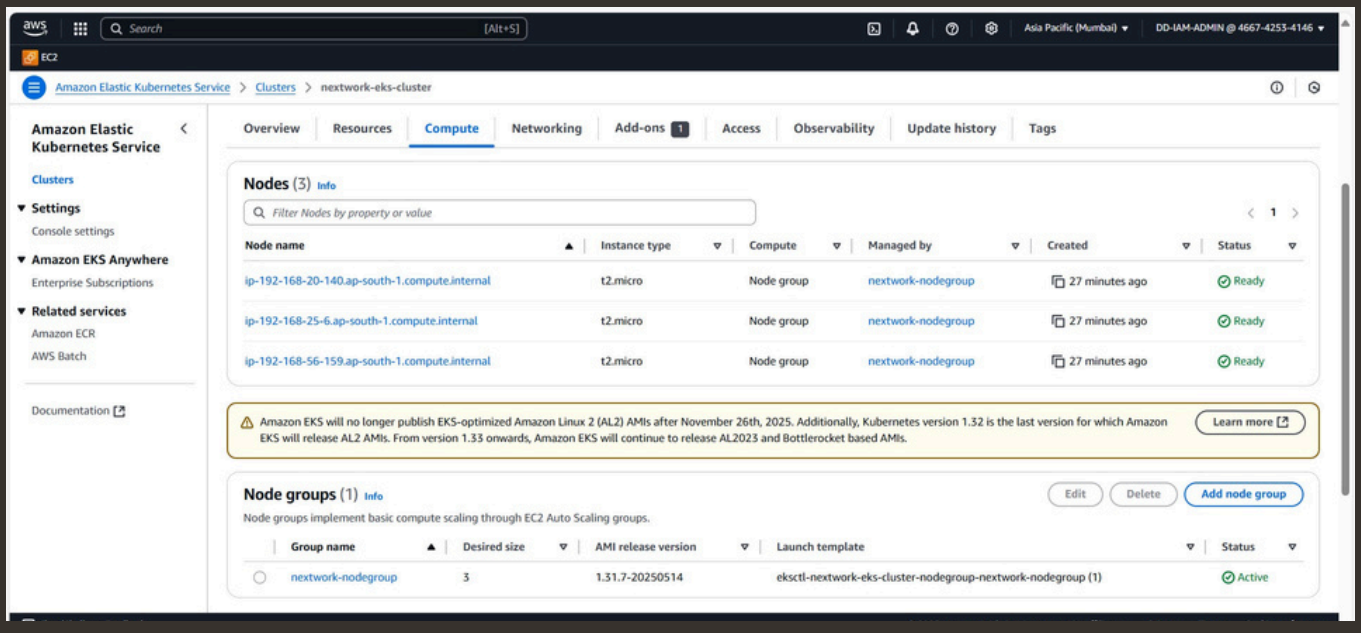
DD Linkedin

Launch a Kubernetes Cluster

DI Dineshraj Dhanapathy



DI Dineshraj Dhanapathy

Introducing Today's Project!

In this project, I will launch and connect to an EC2 instance because it's the foundation for creating and managing infrastructure on AWS. Then, I will create my own Kubernetes cluster to orchestrate containers, monitor the process using CloudFormation for automation and visibility, access the cluster securely using IAM, and finally test its resilience to ensure it can handle failures and maintain uptime.

What is Amazon EKS?

Amazon EKS (Elastic Kubernetes Service) is a managed service that makes it easy to run Kubernetes on AWS without having to install and operate your own control plane or nodes. In todayś' project, I used EKS to create a Kubernetes cluster with eksctl, which automatically set up the control plane, networking, and worker nodes. I connected to the cluster using kubectl, deployed workloads, and tested the clusterś' resilience by terminating EC2 nodes and watching EKS automatically replace them.

One thing I didn't expect

One thing I didn't expect in this project was how much automation CloudFormation and eksctl provide behind the scenes. I thought setting up a Kubernetes cluster would require a lot of manual configuration, but these tools handled complex tasks like creating VPCs, subnets, and node groups automatically. This made the process much faster and less error-prone than I anticipated, highlighting how powerful AWS managed services can be for simplifying infrastructure management.

This project took me...

This project took me about 1 hour to complete. The part that took the longest was waiting for the EKS cluster and its associated resources to be created and fully operational. Because Kubernetes clusters involve setting up multiple AWS components like VPCs, node groups, and control planes, the provisioning process naturally takes some time. Once the cluster was ready, the other steps like connecting

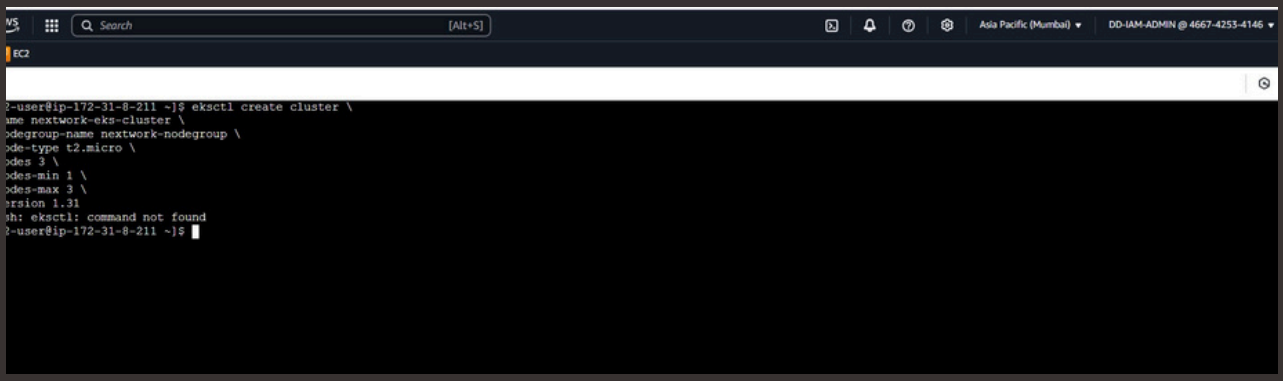
DI Dineshraj Dhanapathy

What is Kubernetes?

Kubernetes is a powerful container orchestration platform that automates the deployment, scaling, and management of containerized applications. Companies and developers use Kubernetes to simplify running applications across multiple servers, ensuring containers are always running where they should be. It automatically handles scaling based on demand, restarts crashed containers, manages updates without downtime, and connects containers to storage. This saves time, reduces errors, and supports reliable, scalable app delivery.

I used eksctl to create a Kubernetes cluster on AWS EKS. The create cluster command I ran defined the cluster name, region, node type, and number of nodes. eksctl handled the setup of the control plane, worker nodes, networking, and IAM roles automatically. This simplified what would otherwise be a complex manual setup, allowing me to quickly get a working EKS cluster with just a single command and minimal configuration.

I initially ran into two errors while using eksctl. The first one was because eksctl was not installed on my EC2 instance, so the command couldn't run. The second one was because my EC2 instance lacked the necessary IAM role permissions to create and manage EKS resources. After installing eksctl and attaching the correct IAM role with proper permissions, I was able to successfully create and manage the Kubernetes cluster.

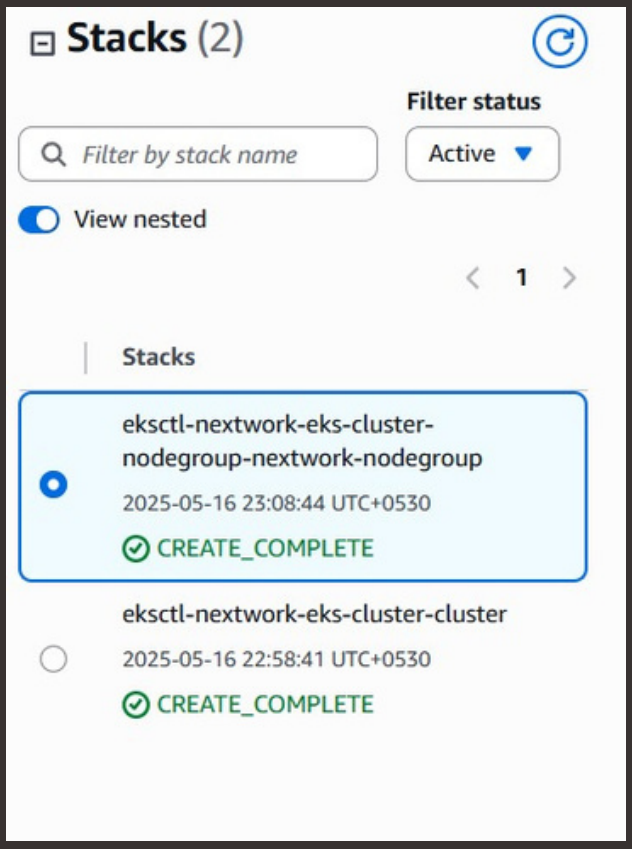

Search
[Alt+S]
Asia Pacific (Mumbai)
DD-IAM-ADMIN @ 4667-4253-4146
EC2
2-user@ip-172-31-8-211 ~]$ eksctl create cluster \
ame nextwork-eks-cluster \
odegroup-name nextwork-nodegroup \
ode-type t2.micro \
odes 3 \
odes-min 1 \
odes-max 3 \
ersion 1.31
sh: eksctl: command not found
2-user@ip-172-31-8-211 ~]$

DI Dineshraj Dhanapathy

eksctl and CloudFormation

CloudFormation helped create my EKS cluster because eksctl uses it behind the scenes to provision and manage all the AWS resources needed for the cluster. It created VPC resources because Kubernetes needs a secure and isolated network to run containers, allowing them to communicate with each other and access the internet safely. CloudFormation automatically generated components like subnets, route tables, security groups, and gateways, saving me from having to configure each one manually.

There was also a second CloudFormation stack for the node group, which is a set of EC2 instances that run my containers. The difference between a cluster and node group is that the cluster includes the entire Kubernetes environment, including the control plane and networking, while the node group contains the actual compute resources. Separating them into two stacks makes it easier to manage, scale, and troubleshoot each part independently without affecting the other components.

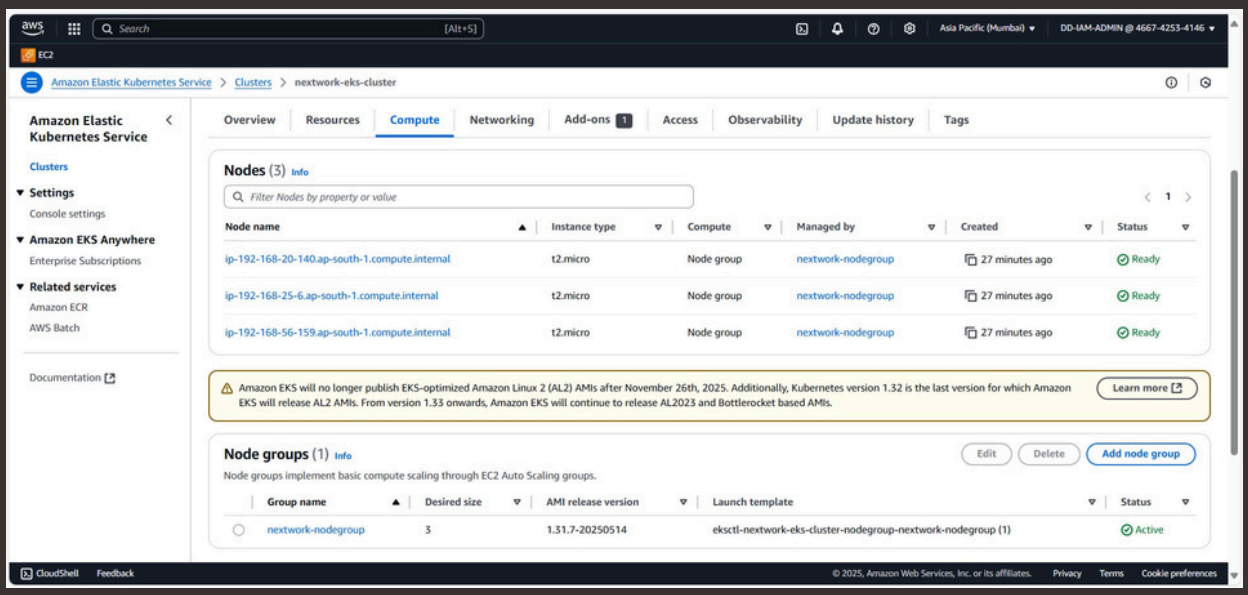

Stacks (2)
Filter status
Filter by stack name
Active
View nested
1
Stacks
eksctl-nextwork-eks-cluster-nodegroup-nextwork-nodegroup
2025-05-16 23:08:44 UTC+0530
CREATE_COMPLETE
eksctl-nextwork-eks-cluster-cluster
2025-05-16 22:58:41 UTC+0530
CREATE_COMPLETE

DI Dineshraj Dhanapathy

The EKS console

I had to create an IAM access entry in order to allow my IAM user to interact with the EKS cluster using kubectl. An access entry is a way to map an AWS IAM user or role to Kubernetes RBAC permissions through the aws-auth ConfigMap. I set it up by editing this ConfigMap and adding my IAM role under the mapRoles section, giving it the necessary permissions to access and manage resources inside the cluster.

It took about 15–20 minutes to create my cluster. Since I'll create this cluster again in the next project of this series, maybe this process could be sped up if I use a smaller node group or reuse the existing VPC and networking setup. Automating more steps with scripts or tweaking eksctl configurations could also help reduce setup time and make the process more efficient for future clusters.


Search [Alt+S]
Asia Pacific (Mumbai)
DD-IAM-ADMIN @ 4667-4253-4146
EC2
Amazon Elastic Kubernetes Service > Clusters > nextwork-eks-cluster
Amazon Elastic Kubernetes Service
Clusters
Settings
Console settings
Amazon EKS Anywhere
Enterprise Subscriptions
Related services
Amazon ECR
AWS Batch
Documentation
Overview
Resources
Compute
Networking
Add-ons 1
Access
Observability
Update history
Tags
Nodes (3) Info
Filter Nodes by property or value
Node name
Instance type
Compute
Managed by
Created
Status
ip-192-168-20-140.ap-south-1.compute.internal t2.micro Node group nextwork-nodegroup 27 minutes ago Ready
ip-192-168-25-6.ap-south-1.compute.internal t2.micro Node group nextwork-nodegroup 27 minutes ago Ready
ip-192-168-56-159.ap-south-1.compute.internal t2.micro Node group nextwork-nodegroup 27 minutes ago Ready
Amazon EKS will no longer publish EKS-optimized Amazon Linux 2 (AL2) AMIs after November 26th, 2025. Additionally, Kubernetes version 1.32 is the last version for which Amazon EKS will release AL2 AMIs. From version 1.33 onwards, Amazon EKS will continue to release AL2023 and Bottlerocket based AMIs.
Learn more
Node groups (1) Info
Edit
Delete
Add node group
Node groups implement basic compute scaling through EC2 Auto Scaling groups.
Group name
Desired size
AMI release version
Launch template
Status
nextwork-nodegroup 3 1.31.7-20250514 eksctl-nextwork-eks-cluster-nodegroup-nextwork-nodegroup (1) Active
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

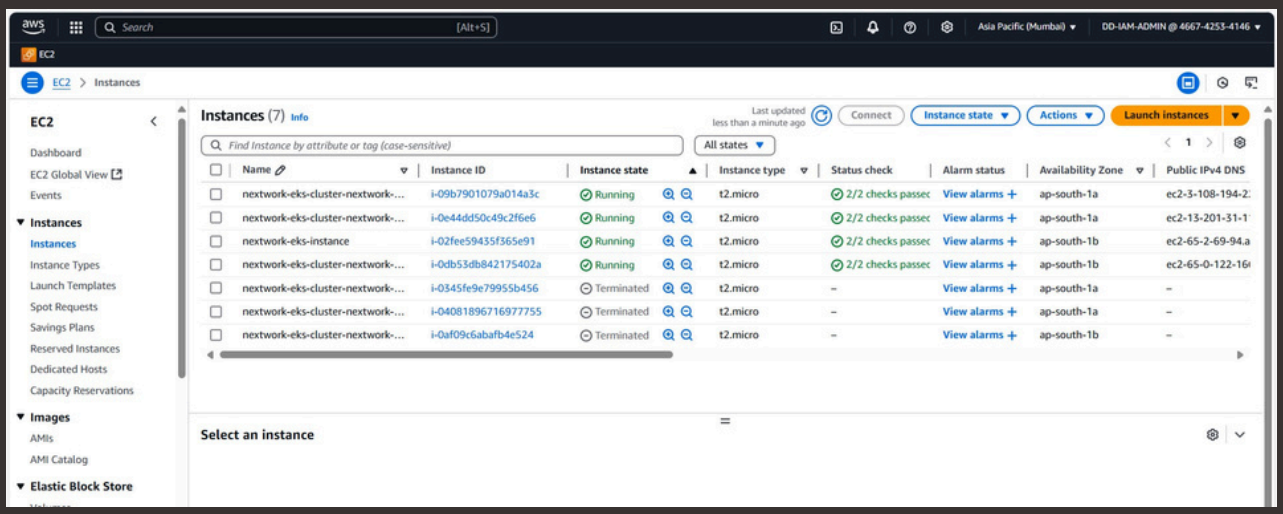
DI Dineshraj Dhanapathy

EXTRA: Deleting nodes

This is because EKS nodes are actually EC2 instances that run your container workloads. When you create a node group, eksctl launches EC2 instances as worker nodes, which join your Kubernetes cluster. These instances are visible in the EC2 console, allowing you to monitor, manage, or troubleshoot the underlying infrastructure powering your Kubernetes applications.

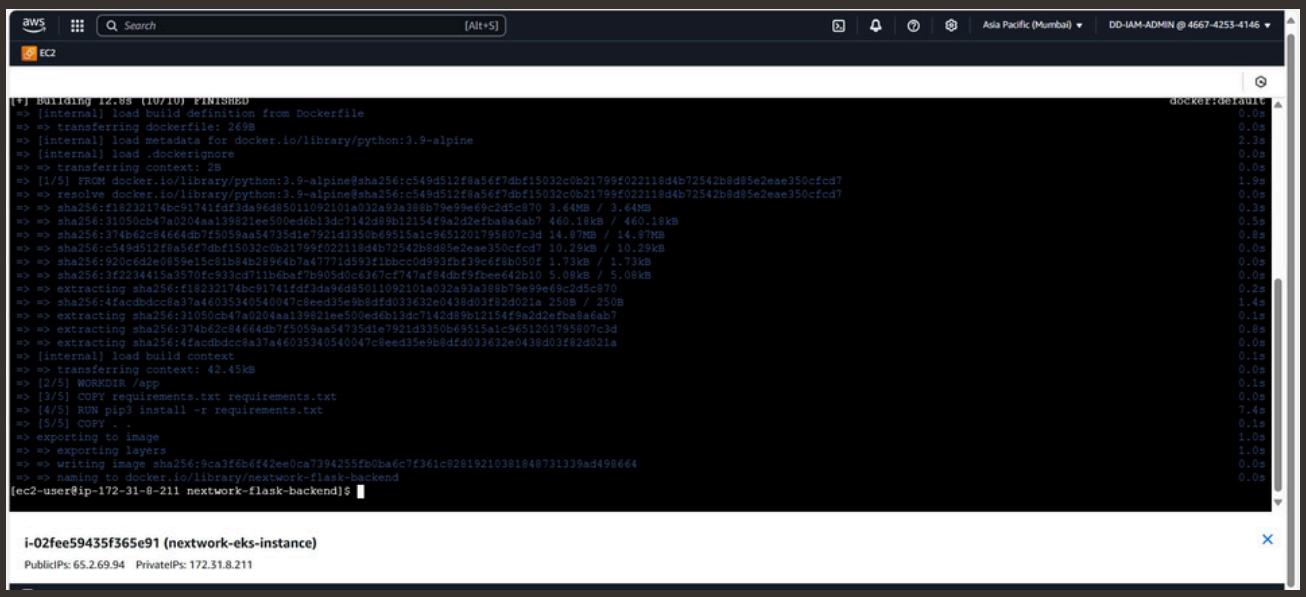
Desired size is the number of nodes you want running in your node group under normal conditions. Minimum and maximum sizes are helpful for defining the scaling limits of your node group—minimum size ensures your app always has enough resources during low demand, while maximum size controls how much your cluster can grow during high demand. These settings help Kubernetes automatically scale your cluster to match workload needs while maintaining performance and controlling costs.

When I deleted my EC2 instances, Kubernetes automatically detected the missing nodes and triggered the node group to launch new instances to replace them. This is because Kubernetes is designed to maintain the desired state of the cluster. The autoscaling feature ensures that the number of nodes matches the desired size, so when a node is lost, a new one is created to keep the cluster stable and my application running smoothly.


Search
[Alt+S]
Asia Pacific (Mumbai)
DD-IAM-ADMIN @ 4667-4253-4146
EC2
EC2 > Instances
EC2
Dashboard
EC2 Global View
Events
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations
Images
AMIs
AMI Catalog
Elastic Block Store
Instances (7) Info
Last updated less than a minute ago
Connect
Instance state
Actions
Launch instances
Find instance by attribute or tag (case-sensitive)
All states
Name
Instance ID
Instance state
Instance type
Status check
Alarm status
Availability Zone
Public IPv4 DNS
nextwork-eks-cluster-nextwork-... i-09b7901079a014a3c Running t2.micro 2/2 checks passed View alarms ap-south-1a
nextwork-eks-cluster-nextwork-... i-0e44dd50c49c2f6e6 Running t2.micro 2/2 checks passed View alarms ap-south-1a
nextwork-eks-instance i-02fee59435f365e91 Running t2.micro 2/2 checks passed View alarms ap-south-1b
nextwork-eks-cluster-nextwork-... i-0db53db842175402a Running t2.micro 2/2 checks passed View alarms ap-south-1b
nextwork-eks-cluster-nextwork-... i-0345fe9e79955b456 Terminated t2.micro View alarms ap-south-1a
nextwork-eks-cluster-nextwork-... i-04081896716977755 Terminated t2.micro View alarms ap-south-1a
nextwork-eks-cluster-nextwork-... i-0af09c6abafb4e524 Terminated t2.micro View alarms ap-south-1b
Select an instance

Set Up Kubernetes Deployment

DI Dineshraj Dhanapathy



DI Dineshraj Dhanapathy

Introducing Today's Project!

In this project, I will clone a backend application from GitHub, build its Docker image, and push it to an Amazon ECR repository because these steps are essential for containerizing and deploying the app in a scalable, cloud-native way. Troubleshooting installation and configuration errors will sharpen my problem-solving skills. The secret mission to dive into the backend code will deepen my understanding of the apps' functionality and how to optimize it.

Tools and concepts

I used Amazon EKS, Git, Docker, and Amazon ECR to build, deploy, and manage the backend application in a scalable Kubernetes environment. Key steps include cloning the backend code from GitHub, building a Docker container image using the provided Dockerfile, pushing the image to Amazon ECR for secure storage, and creating an EKS cluster with eksctl to run the containerized backend. I also configured IAM roles and permissions to ensure smooth communication between services and troubleshoot installation issues for a seamless deployment.

Project reflection

This project took me approximately 1 hours to complete. The most challenging part was troubleshooting the permission errors with Docker and IAM roles, as they required careful configuration to ensure everything worked smoothly. My favorite part was successfully building and pushing the Docker image to Amazon ECR, then seeing the backend deployed seamlessly on the EKS cluster, which made all the effort feel rewarding and showed the power of container orchestration.

Something new that I learnt from this experience was how to properly configure IAM roles and permissions for an EC2 instance to interact with AWS services like EKS and ECR. I also gained hands-on experience building Docker images and pushing them to Amazon ECR, then deploying those images on an EKS cluster. This deepened my

DI Dineshraj Dhanapathy

What I'm deploying

To set up todayś' project, I launched a Kubernetes cluster. Steps I took to do this included running the eksctl create cluster command with parameters for the cluster name, node group name, instance type, desired, minimum, and maximum nodes, Kubernetes version, and region. This automated the creation of the control plane, worker nodes, and networking, allowing me to quickly deploy a scalable, managed Kubernetes environment on AWS.

I'm deploying an app's backend

Next, I retrieved the backend that I plan to deploy. An appś' backend means the server-side code that handles data, logic, and communication with databases or other services. I retrieved backend code by cloning the nextwork-flask-backend repository from GitHub onto my EC2 instance. This gave me all the necessary files like app.py, Dockerfile, and requirements.txt so I could build and run the backend application locally and prepare it for deployment.

```
[ec2-user@ip-172-31-8-211 ~]$ ls
nextwork-flask-backend
[ec2-user@ip-172-31-8-211 ~]$
```

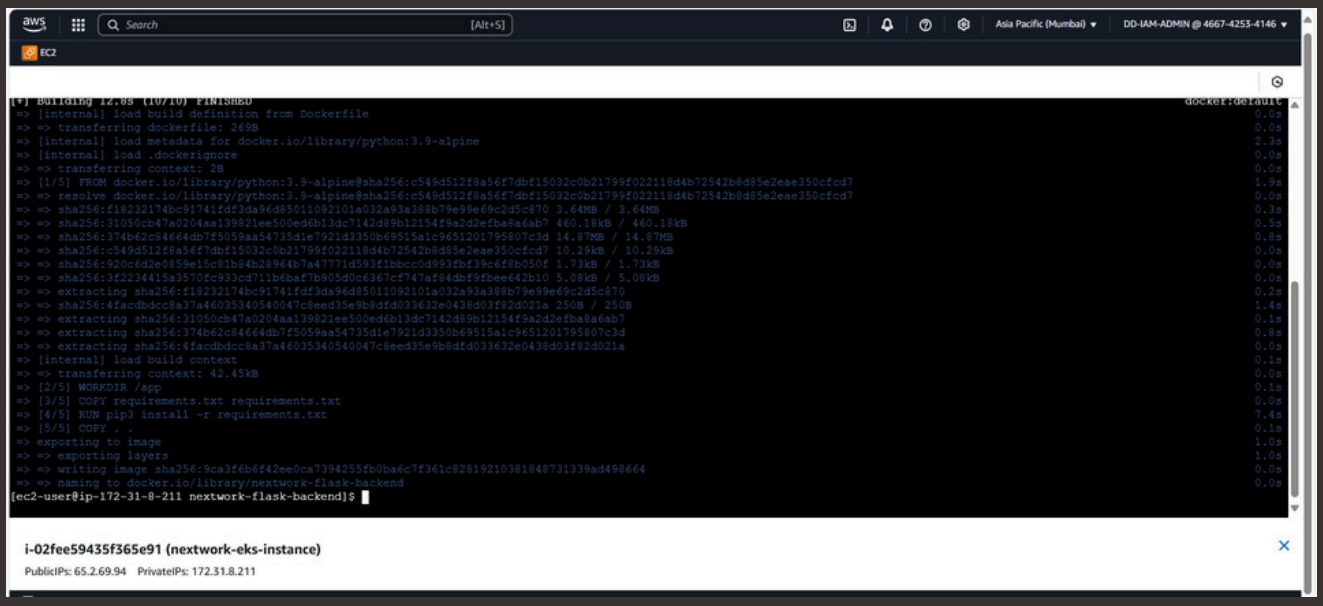
DI Dineshraj Dhanapathy

Building a container image

Once I cloned the backend code, my next step is to build a container image of the backend. This is because building the image packages the application and all its dependencies into a single, portable unit. The Dockerfile provides instructions for this process, ensuring the image is consistent and reproducible. This container image allows Kubernetes to deploy multiple identical containers across environments, guaranteeing the app runs reliably whether in development, testing, or production.

When I tried to build a Docker image of the backend, I ran into a permissions error because the ec2-user account I'm logged in with doesn't have permission to run Docker commands directly. Docker requires root privileges to build images and manage containers. Previously, I used sudo to run Docker commands with elevated rights, but without sudo, the permission is denied. To fix this, I need to add ec2-user to the Docker group to grant proper access without using sudo every time.

When I tried to build a Docker image of the backend, I ran into a permissions error because the ec2-user account I'm logged in with doesn't have permission to run Docker commands directly. Docker requires root privileges to build images and manage containers. Previously, I used sudo to run Docker commands with elevated rights, but without sudo, the permission is denied. To fix this, I need to add ec2-user to the Docker group to grant proper access without using sudo every time.

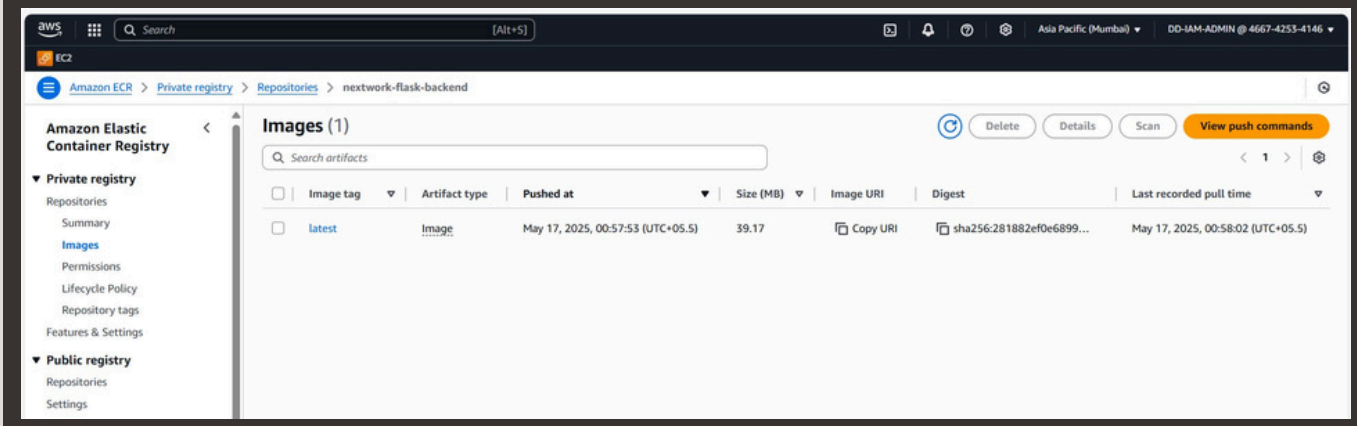

aws
Search
[Alt+S]
Asia Pacific (Mumbai)
DD-IAM-ADMIN @ 4667-4253-4146
EC2
[+] Building 12.8s (10/10) FINISHED
docker:default
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 269B
=> [internal] load metadata for docker.io/library/python:3.9-alpine
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/python:3.9-alpine@sha256:c549d512f8a56f7dbf15032c0b21799f022118d4b72542b8d85e2eae350cfcd7
=> => resolve docker.io/library/python:3.9-alpine@sha256:c549d512f8a56f7dbf15032c0b21799f022118d4b72542b8d85e2eae350cfcd7
=> => sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870 3.64MB / 3.64MB
=> => sha256:31050cb47a0204aa139821ee500ed6b13dc7142d89b12154f9a2d2efba8a6ab7 460.18kB / 460.18kB
=> => sha256:374b62c84664db7f5059aa54735d1e7921d3350b69515a1c9651201795807c3d 14.87MB / 14.87MB
=> => sha256:c549d512f8a56f7dbf15032c0b21799f022118d4b72542b8d85e2eae350cfcd7 10.29kB / 10.29kB
=> => sha256:920c6d2e0859e15c81b84b28964b7a47771d593f1bbcc0d993fbf39c6f8b050f 1.73kB / 1.73kB
=> => sha256:3f2234415a3570fc933cd711b6baf7b905d0c6367cf747af84dbf9fbee642b10 5.08kB / 5.08kB
=> => extracting sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870
=> => sha256:4facdbdcc8a37a4603534054004 7c8eed35e9b8dfd033632e0438d03f82d021a 250B / 250B
=> => extracting sha256:31050cb47a0204aa139821ee500ed6b13dc7142d89b12154f9a2d2efba8a6ab7
=> => extracting sha256:374b62c84664db7f5059aa54735d1e7921d3350b69515a1c9651201795807c3d
=> => extracting sha256:4facdbdcc8a37a4603534054004 7c8eed35e9b8dfd033632e0438d03f82d021a
=> [internal] load build context
=> => transferring context: 42.45kB
=> [2/5] WORKDIR /app
=> [3/5] COPY requirements.txt requirements.txt
=> [4/5] RUN pip3 install -r requirements.txt
=> [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:9ca3f6b6f42ee0ca7394255fb0ba6c7f361c828192103818487313 39ad498664
=> => naming to docker.io/library/nextwork-flask-backend
[ec2-user@ip-172-31-8-211 nextwork-flask-backend]$
i-02fee59435f365e91 (nextwork-eks-instance)
PublicIPs: 65.2.69.94 PrivateIPs: 172.31.8.211

DI Dineshraj Dhanapathy

Container Registry

I'm using Amazon ECR in this project to securely store and manage my backendś' Docker container image. ECR is a good choice for the job because it integrates seamlessly with AWS services like EKS, allowing easy and secure image deployment without complex authentication. It ensures my container images are available, scalable, and protected, making the deployment process smoother and more reliable within the AWS ecosystem.

Container registries like Amazon ECR are great for Kubernetes deployment because they store and manage container images in a centralized, secure place. This allows Kubernetes clusters to pull the latest images on demand, ensuring consistency across all nodes. Without a registry, you'd have to manually preload and update images on each node, which is time-consuming and error-prone. Registries simplify scaling, updating, and maintaining containerized applications efficiently.



DI Dineshraj Dhanapathy

EXTRA: Backend Explained

After reviewing the app's backend code, I've learnt that it is built using the Flask web framework, designed to handle API requests dynamically. The backend exposes an endpoint /contents/<topic> that accepts a topic as a URL parameter. When a request is made, the backend fetches relevant data from the Hacker News Search API, extracting useful information like article IDs, titles, and URLs. This data is then formatted into a clean JSON response for clients to consume. The backend relies on key Python libraries such as Flask-RESTx for API management and Requests for external HTTP calls, ensuring modular, scalable, and easy-to-maintain code.

Unpacking three key backend files

The requirements.txt file lists all the Python dependencies needed for the backend application to run properly. It specifies exact versions of packages like Flask, flask-restx, requests, and werkzeug, ensuring the apps' environment is consistent. This file allows easy installation of these dependencies using tools like pip, so the backend has all the libraries it needs to handle web requests, create APIs, and fetch external data reliably.

The Dockerfile gives Docker instructions on how to build a container image for the backend app. Key commands in this Dockerfile include setting the base image with Python 3.9 on Alpine Linux, adding author metadata, setting the working directory to /app, copying the requirements.txt file, installing dependencies, copying all app files, and defining the command to run the Flask app with python3 app.py. This ensures a consistent and lightweight environment for the app.

DI Dineshraj Dhanapathy

The app.py file contains the main backend code for the application. It sets up the Flask app and defines routing, specifically the /contents/<topic> route, which directs requests to the SearchContents class. In this class, the get() method fetches data from the Hacker News API based on the topic, extracts key details like id, title, and url, and sends back a formatted JSON response to the user, enabling dynamic content retrieval.

Create Kubernetes Manifests

DI Dineshraj Dhanapathy

```
          - containerPort: 8080
[ec2-user@ip-172-31-8-211 manifests]$ cat << EOF > flask-service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: nextwork-flask-backend
spec:
  selector:
    app: nextwork-flask-backend
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      protocol: TCP
EOF
[ec2-user@ip-172-31-8-211 manifests]$ ls
flask-deployment.yaml  flask-service.yaml
[ec2-user@ip-172-31-8-211 manifests]$
```

DI Dineshraj Dhanapathy

Introducing Today's Project!

In this project, I will set up a Deployment manifest to instruct Kubernetes on deploying my backend application and a Service manifest to expose the backend to users. This is important because the Deployment manages the apps' desired state and scaling, while the Service handles network access. Additionally, I'll dive into the manifest details to deepen my understanding of Kubernetes configurations and how they control app deployment and connectivity.

Tools and concepts

I used Amazon EKS, eksctl, and Docker to deploy a containerized backend on Kubernetes. Key concepts include using manifests to define how Kubernetes should deploy and expose the app, container images to package the backend consistently, and services to route traffic to the app pods. These tools and concepts helped me automate deployment, ensure scalability with replicas, and simplify managing app updates across the cluster.

Project reflection

I did this project today to deepen my practical understanding of deploying containerized applications using AWS services like EKS and ECR. It met my goals by giving me hands-on experience with Kubernetes manifests, Docker, and AWS integration. The challenges I faced helped me strengthen troubleshooting skills, and overall, the project boosted my confidence in managing cloud-native deployments effectively.

DI

Dineshraj Dhanapathy

This project took me approximately 1 hour to complete. The most challenging part was troubleshooting permission errors when building and pushing the Docker image, as it required understanding user roles on the EC2 instance. My favourite part was setting up the Kubernetes manifests because it helped me see how deployments and services work together to manage and expose the app, making the whole process feel automated and scalable.

DI Dineshraj Dhanapathy

Project Set Up

Kubernetes cluster

To set up today's project, I launched a Kubernetes cluster. Steps I took to do this included installing eksctl, configuring my AWS credentials on the EC2 instance, and running the eksctl create cluster command with parameters for cluster name, node group, instance type, and node scaling limits. This automated the creation of the EKS control plane and worker nodes, enabling me to deploy and manage containerized applications easily on AWS.

Backend code

I retrieved the backend that I plan to deploy by cloning the nextwork-flask-backend repository from GitHub. Backend is the server-side part of an application that handles logic, data processing, and API responses. By cloning the repo, I copied all the backend code and dependencies, such as app.py, Dockerfile, and requirements.txt, onto my EC2 instance so I can build and run the backend service in my Kubernetes cluster.

Container image

Once I cloned the backend code, I built a container image because this packages the app and its dependencies into a portable, consistent unit. I used the Dockerfile in the project repository, running the docker build command to create the image. This command read the instructions in the Dockerfile—like setting up Python, installing libraries, and copying the code—to generate an image that Kubernetes can deploy reliably across different environments.

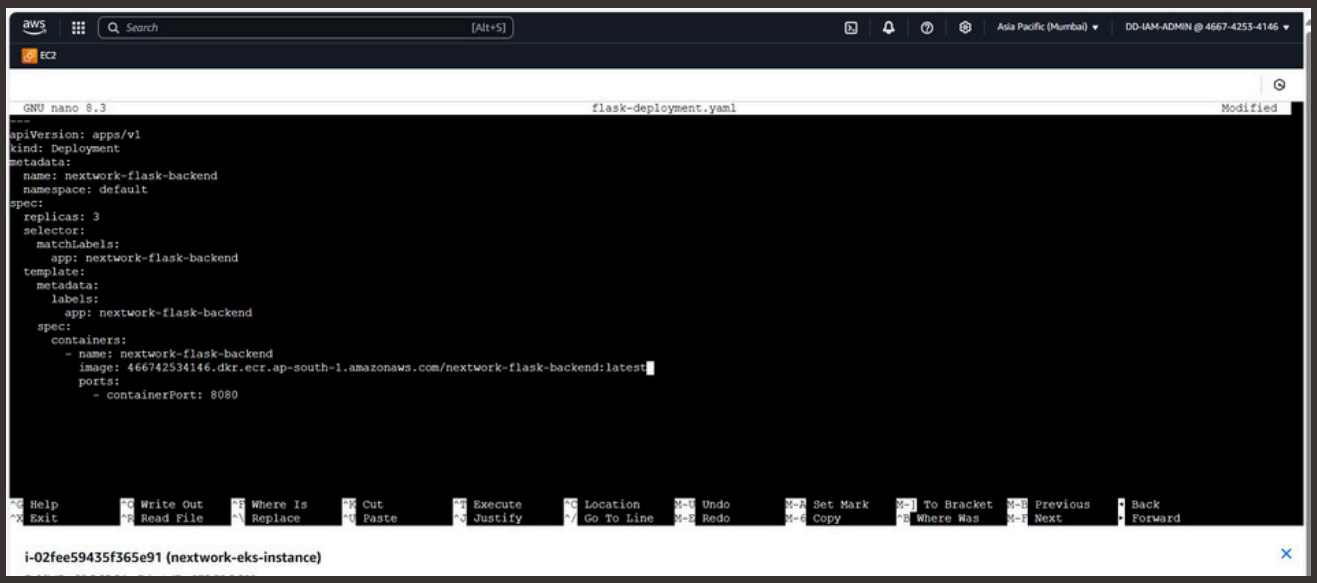
I also pushed the container image to a container registry because it allows Kubernetes to pull the image securely and deploy it consistently across nodes. To push the image to ECR, I first authenticated my Docker client with ECR using the AWS CLI. Then, I tagged the local Docker image with the ECR repository URI and finally ran docker push

DI Dineshraj Dhanapathy

Manifest files

Kubernetes manifests are configuration files that tell Kubernetes how to deploy and manage your application. Manifests are helpful because they define key details like which container image to run, how many replicas to create, and what resources to allocate. This makes deployments repeatable, consistent, and automated—saving you from manually setting up each container every time you deploy your app.

A Kubernetes deployment manages how your application is deployed, scaled, and updated. It ensures the desired number of app instances are running and automatically replaces failed ones. The container image URL in my Deployment manifest tells Kubernetes which app version to run, so it can pull the correct image from the container registry and deploy it across the cluster reliably and consistently.


aws
Search
[Alt+S]
Asia Pacific (Mumbai)
DD-IAM-ADMIN @ 4667-4253-4146
EC2
GNU nano 8.3
flask-deployment.yaml
Modified
---
apiVersion: apps/v1
kind: Deployment
metadata:
name: nextwork-flask-backend
namespace: default
spec:
replicas: 3
selector:
matchLabels:
app: nextwork-flask-backend
template:
metadata:
labels:
app: nextwork-flask-backend
spec:
containers:
- name: nextwork-flask-backend
image: 466742534146.dkr.ecr.ap-south-1.amazonaws.com/nextwork-flask-backend:latest
ports:
- containerPort: 8080
^G Help
^X Exit
^O Write Out
^R Read File
^W Where Is
^\ Replace
^K Cut
^U Paste
^T Execute
^J Justify
^C Location
M-/ Go To Line
M-U Undo
M-E Redo
M-A Set Mark
M-6 Copy
M-] To Bracket
^B Where Was
M-W Previous
M-F Next
^ Back
^F Forward
i-02fee59435f365e91 (nextwork-eks-instance)

DI Dineshraj Dhanapathy

Service Manifest

A Kubernetes Service exposes your application to network traffic and ensures it reaches the correct Pods. You need a Service manifest to define how traffic should be routed, which Pods to target using labels, and what ports to use. This allows users or other services to access your app reliably, even if Pods are recreated or moved within the cluster. It ensures stable communication within and outside the Kubernetes environment.

My Service manifest sets up a way for external traffic to reach my backend app. It defines a LoadBalancer-type Service that listens on port 80 and routes traffic to port 8080 on the Pods labeled app: nextwork-flask-backend. This allows users outside the Kubernetes cluster to access my Flask backend through a stable, publicly accessible endpoint managed by the Kubernetes Service.

```
            - containerPort: 8080
[ec2-user@ip-172-31-8-211 manifests]$ cat << EOF > flask-service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: nextwork-flask-backend
spec:
  selector:
    app: nextwork-flask-backend
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      protocol: TCP
EOF
[ec2-user@ip-172-31-8-211 manifests]$ ls
flask-deployment.yaml  flask-service.yaml
[ec2-user@ip-172-31-8-211 manifests]$
```

DI Dineshraj Dhanapathy

Deployment Manifest

Annotating the Deployment manifest helped me understand how Kubernetes manages my application because it clarified the purpose of each section—like how replicas maintain availability, how selectors match Pods, and how the container image and ports are configured. This made it easier to debug, customize, and confidently explain how my backend is deployed and scaled inside the cluster.

A notable line in the Deployment manifest is the number of replicas, which means how many identical instances of the app should run. Here, setting replicas to 3 ensures high availability and reliability. Pods are relevant to this because each replica runs inside its own Pod, which is the smallest deployable unit in Kubernetes. Pods group containers together and manage networking and storage, making sure each replica functions as part of the overall application.

One part of the Deployment manifest I still want to know more about is how resource requests and limits work, because while I understand they help manage CPU and memory usage, I'm not fully clear on how Kubernetes enforces them during high-traffic situations or how to choose the right values. Understanding this better would help me optimize app performance and ensure stability under different loads.

```
  GNU nano 8.3                                                    flask-deployment.yaml
---
apiVersion: apps/v1 # Specifies the API version for this Deployment.
kind: Deployment # This is a Deployment object.
metadata: # basic information of deployment.
  name: nextwork-flask-backend # unique name of deployment.
  namespace: default # specifies the naamespace where the deployment will be created.
spec: # specifies the desired state of deployment.
  replicas: 3 # creates the 3 replicas(pods) of application.
  selector: # matches pods with the label app: nextwork-flask-backend.
    matchLabels:
      app: nextwork-flask-backend
  template: # define the pod template
    metadata: # ensure pods have the label that matches matchLabel
      labels:
        app: nextwork-flask-backend
    spec: # desired state and config for kubernetes object.
      containers:
        - name: nextwork-flask-backend
          image: 466742534146.dkr.ecr.ap-south-1.amazonaws.com/nextwork-flask-backend:latest
          ports:
            - containerPort: 8080
```

Deploy Backend with Kubernetes

DI Dineshraj Dhanapathy

```
[ec2-user@ip-172-31-8-211 manifests]$ kubectl apply -f flask-deployment.yaml
kubectl apply -f flask-service.yaml
deployment.apps/nextwork-flask-backend created
service/nextwork-flask-backend created
[ec2-user@ip-172-31-8-211 manifests]$
```

DI Dineshraj Dhanapathy

Introducing Today's Project!

In this project, I will set up the backend of an app for deployment because deploying the backend is essential to making the application accessible and functional. I will install kubectl to manage and interact with the Kubernetes cluster. Then, I'll deploy the backend on the Kubernetes cluster to ensure it runs reliably. Finally, as a secret mission, I will track the Kubernetes deployment using EKS to monitor its health and performance.

Tools and concepts

I used Kubernetes, ECR, kubectl, and eksctl to deploy and manage my backend application on a scalable cluster. Key concepts include using manifests to define desired states for Deployments and Services, containerizing the app with Docker for consistent environments, and pushing images to ECR for easy access by Kubernetes. These tools and concepts enabled automated, reliable deployment and seamless scaling of my backend in a cloud-native way.

Project reflection

This project took me approximately 1 hour to complete. The most challenging part was configuring the Kubernetes manifests correctly to ensure smooth deployment and service exposure. My favourite part was seeing the backend successfully deployed and accessible through the Kubernetes Service, which made all the setup feel rewarding. Overall, it was a great hands-on experience with container orchestration and cloud-native deployment.

DI Dineshraj Dhanapathy

Project Set Up

Kubernetes cluster

To set up today's project, I launched a Kubernetes cluster. The cluster's role in this deployment is to provide a scalable, managed environment to run and orchestrate containerized backend applications. I used eksctl to create the cluster with specified node groups, enabling automatic management of nodes and resources. This setup ensures my app runs reliably with high availability, load balancing, and easy scaling across multiple containers.

Backend code

I retrieved backend code by cloning the GitHub repository using Git commands. Pulling code is essential to this deployment because it gives me the exact, up-to-date version of the backend application, including all necessary files like the app code, Dockerfile, and dependencies. This ensures that I build and deploy the correct version of the app, allowing Kubernetes to run the backend smoothly and consistently across environments.

Container image

Once I cloned the backend code, I built a container image because it packages the app and all its dependencies into a single, portable unit. Without an image, it would be difficult for Kubernetes to deploy the backend consistently across different nodes and environments. The container image ensures the app runs reliably, making scaling and updating easier within the Kubernetes cluster.

I also pushed the container image to a container registry, which is Amazon ECR, because it provides a secure and scalable way to store and retrieve container images. ECR facilitates scaling for my deployment because it integrates seamlessly with EKS, allowing Kubernetes to pull images on demand and deploy multiple replicas efficiently without needing manual image transfers.

DI Dineshraj Dhanapathy

Manifest files

Kubernetes manifests are configuration files written in YAML or JSON that define the desired state of Kubernetes resources like Pods, Deployments, and Services. Manifests are helpful because they let you describe how your application should run, including container images, replicas, ports, and more. Kubernetes reads these manifests to automatically set up and manage your app, ensuring consistency, reliability, and easier deployment across environments.

A Deployment manifest manages how an application is deployed and maintained on a Kubernetes cluster. It defines the number of replicas, the container image to use, and the desired app state. The container image URL in my Deployment manifest tells Kubernetes which image to pull and run in each Pod. This ensures that every replica uses the same version of the app, making deployments consistent, scalable, and fault-tolerant.

A Service resource exposes an application running in a set of Pods so it can be accessed from outside the Kubernetes cluster. My Service manifest sets up a NodePort Service that routes traffic to Pods labeled app: nextwork-flask-backend and listens on port 8080. It forwards traffic from that port to the same port on the target Pods. This setup allows users to access the backend using the node's IP address and the assigned NodePort, making it ideal for learning and testing.

```
  GNU nano 8.3                                                flask-deployment.yaml
---
apiVersion: apps/v1 # Specifies the API version for this Deployment.
kind: Deployment # This is a Deployment object.
metadata: # basic information of deployment.
  name: nextwork-flask-backend # unique name of deployment.
  namespace: default # specifies the naamespace where the deployment will be created.
spec: # specifies the desired state of deployment.
  replicas: 3 # creates the 3 replicas(pods) of application.
  selector: # matches pods with the label app: nextwork-flask-backend.
    matchLabels:
      app: nextwork-flask-backend
  template: # define the pod template
    metadata: # ensure pods have the label that matches matchLabel
      labels:
        app: nextwork-flask-backend
    spec: # desired state and config for kubernetes object.
      containers: #define the containers to run in each pod
        - name: nextwork-flask-backend # name the conatiner
          image: 466742534146.dkr.ecr.ap-south-1.amazonaws.com/nextwork-flask-backend:latest
          ports:
            - containerPort: 8080 #specifies the container port exposed by the backend
```

DI Dineshraj Dhanapathy

Backend Deployment!

To deploy my backend application, I used kubectl apply to apply the Deployment and Service manifests I created. These manifests gave Kubernetes the instructions it needed to create pods running my backend container and expose them to traffic using a NodePort Service. This step made my app accessible from outside the cluster. I also verified the deployment using kubectl get pods and kubectl get svc to confirm everything was running properly.

kubectl

kubectl is the command-line tool used to interact with Kubernetes resources such as Deployments and Services. I need this tool to apply manifest files, manage workloads, and monitor the health of my application once the cluster is running. I can't use eksctl for the job because eksctl is mainly for setting up and managing EKS clusters—not for deploying applications or interacting with individual Kubernetes resources like pods and services.

```
[ec2-user@ip-172-31-8-211 manifests]$ kubectl apply -f flask-deployment.yaml
kubectl apply -f flask-service.yaml
deployment.apps/nextwork-flask-backend created
service/nextwork-flask-backend created
[ec2-user@ip-172-31-8-211 manifests]$
```

DI Dineshraj Dhanapathy

Verifying Deployment

My extension for this project is to use the EKS console to visually inspect and verify the resources running in my Kubernetes cluster, such as Pods and Services. I had to set up IAM access policies because the console needs permission to display and interact with my cluster resources securely. I set up access by attaching the necessary IAM policies to my user or role and enabling cluster access using the aws-auth ConfigMap to map IAM identities to Kubernetes roles.

Once I gained access to my cluster's nodes, I discovered pods running inside each node. Pods are the smallest deployable units in Kubernetes that bundle one or more containers together to work as a unit. Containers in a pod share the same network space and storage, enabling them to communicate easily and share data efficiently. Pods ensure that containers are co-located and managed together within the Kubernetes cluster environment.

The EKS console shows you the events for each pod, where I could see important steps like the pod being assigned an internal IP, the container image being pulled from Amazon ECR, the image successfully downloaded, and the container created and started. This validated that my backend container was deployed correctly and is now running inside the Kubernetes cluster. It confirmed that Kubernetes successfully scheduled the pod, fetched the right image, and launched the container, meaning my backend is operational and ready to handle requests within the cluster network.

DI

Dineshraj Dhanapathy