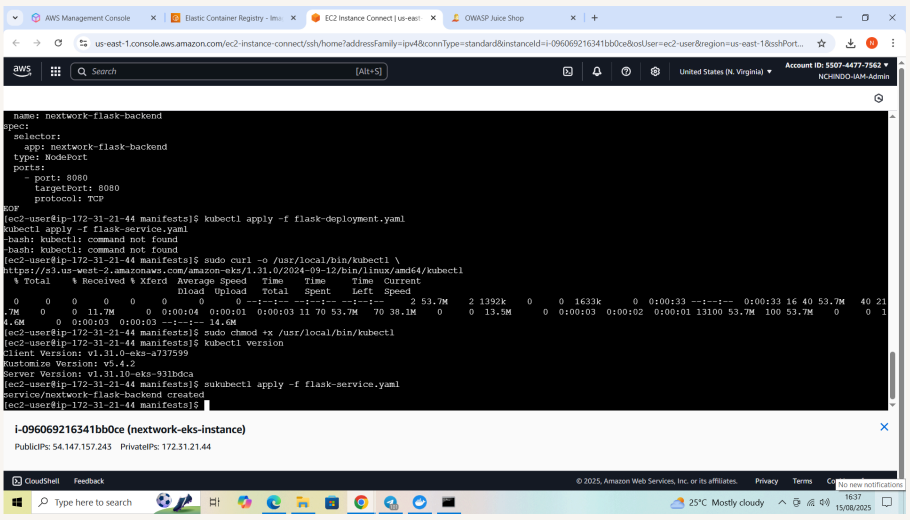
nextwork.org

Deploy Backend with Kubernetes

Nchindo Boris



Nchindo Boris
NextWork Student

Introducing Today's Project!

In this project, I will; - Set up the backend of an app for deployment. - Install kubectl. - Deploy the backend on a Kubernetes cluster

Tools and concepts

I used Kubernetes, ECR, kubectl, and Docker to build, store, and deploy my backend application in a scalable and consistent way. Key concepts include using manifests to define how Kubernetes should deploy and expose my app, containerising the app with Docker for portability, pushing container images to Amazon ECR for centralised storage, and using kubectl to manage Kubernetes resources in the cluster.

Project reflection

This project took me approximately 1 hour to complete. The most challenging part was setting up the correct IAM permissions and configuring access between AWS and Kubernetes, as it involved understanding both AWS security and Kubernetes roles. My favourite part was seeing the application successfully deploy and run in the EKS cluster.

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 run and manage the backend of the app by handling container orchestration, scaling, and networking. I did this by creating an EC2 instance, installing eksctl, attaching an IAM role with admin access, and running a command to create the EKS cluster

Backend code

I retrieved backend code by installing Git on my EC2 instance and cloning the nextwork-flask-backend repository from GitHub. Pulling code is essential to this deployment because it gives me access to the backend files, like app.py, Dockerfile, and requirements.txt, that I need to build, run, and deploy the application on my Kubernetes cluster.

Container image

Once I cloned the backend code, I built a container image because Kubernetes needs that image to create and run the app in containers across the cluster. Without an image, it would be difficult for Kubernetes to consistently deploy and manage the app, since it wouldn't have a standardised package containing the code, dependencies, and environment setup. The image ensures everything runs the same way, no matter where it's deployed.

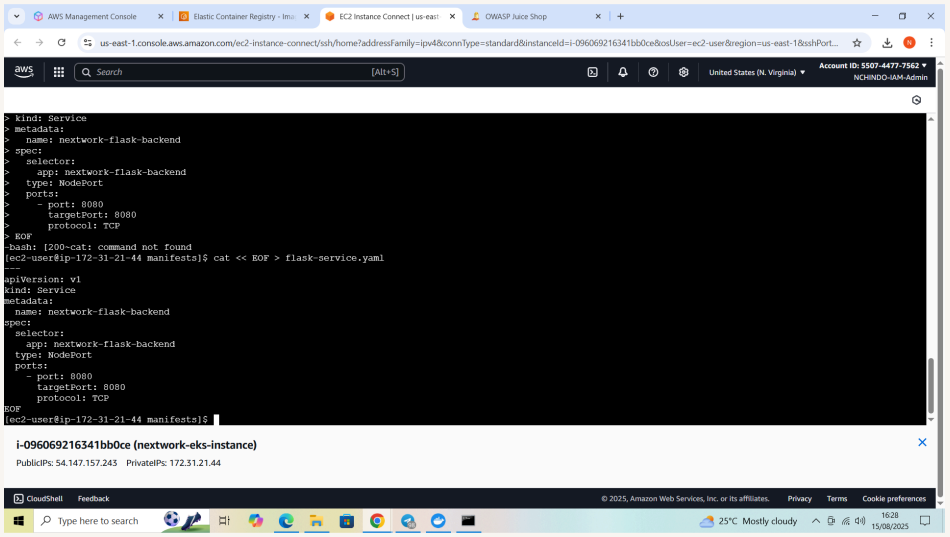
I also pushed the container image to a container registry, which is Amazon ECR. ECR facilitates scaling for my deployment because it provides a central, secure place where Kubernetes can easily pull the exact container image version it needs. This means the app can be deployed consistently across all nodes in the cluster without manually updating each one, making scaling and updates much smoother.

Manifest files

Kubernetes manifests are files that contain instructions telling Kubernetes how to run and manage your application in the cluster. Manifests are helpful because they let you define things like which container images to use, how many copies to run, and how to expose your app; all in a clear, repeatable way. Without manifests, you'd have to manually configure everything each time, which would be slow and prone to mistakes.

A Deployment manifest manages how Kubernetes runs and maintains multiple copies (replicas) of my application to ensure it's always available and can scale smoothly. The container image URL in my Deployment manifest telis Kubernetes exactly which container image to pull from the registry, so it knows what code and environment to run inside each container.

A Service resource exposes an application running inside the Kubernetes cluster to the outside world or other parts of the cluster. My Service manifest sets up a NodePort Service called nextwork-flask-backend that routes external traffic coming to port 8080 on the nodes to the backend containers labelled with app: nextwork-flask-backend, allowing users or other services to access the backend app through the node's IP and assigned port.

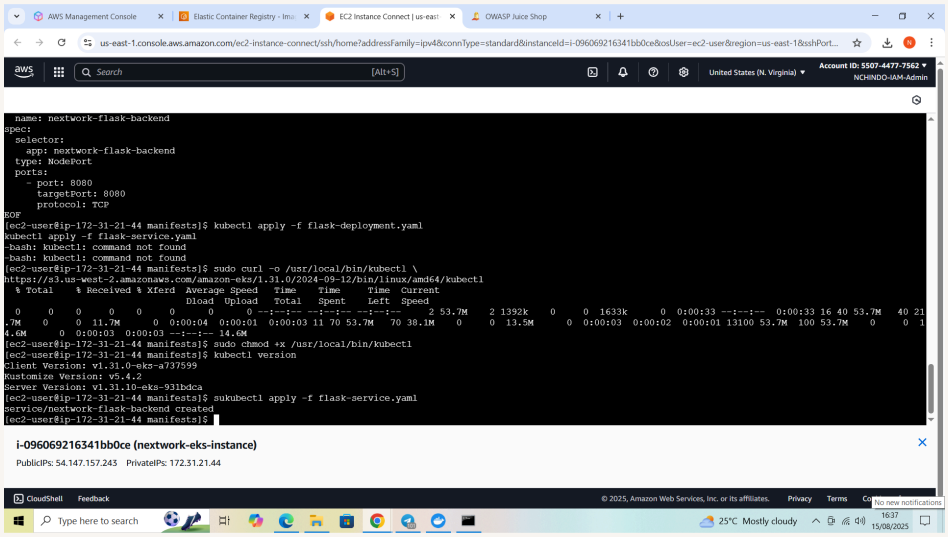

> kind: Service
> metadata:
>   name: nextwork-flask-backend
> spec:
>   selector:
>   app: nextwork-flask-backend
>   type: NodePort
>   ports:
>   - port: 8080
>   targetPort: 8080
>   protocol: TCP
> EOF
-bash: [200~cat: command not found
[ec2-user@ip-172-31-21-44 manifests]$ cat << EOF > flask-service.yaml
---
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-172-31-21-44 manifests]$
i-096069216341bb0ce (nextwork-eks-instance)
PublicIPs: 54.147.157.243 PrivateIPs: 172.31.21.44

Backend Deployment!

To deploy my backend application; - I installed the kubectl command-line tool by downloading it and making it executable. - Then, I verified the installation by checking the kubectl version. - Finally, I applied my Kubernetes manifest files using kubectl apply -f flask-deployment.yarmi and kubectl apply -f flask-service.yaml to create the Deployment and Service resources in my EKS cluster, which launched and exposed my backend application.

kubectl

kubectl is the command-line tool for interacting directly with Kubernetes clusters and managing Kubernetes resources like Deployments and Services. I need this tool to apply, update, and monitor my application manifests inside the cluster. I can't use eksctl for this job because eksctl is designed primarily for creating and managing the AWS EKS cluster itself, not for managing the workloads or resources running inside the Kubernetes cluster.


name: nextwork-flask-backend
spec:
selector:
app: nextwork-flask-backend
type: NodePort
ports:
- port: 8080
targetPort: 8080
protocol: TCP
EOF
[ec2-user@ip-172-31-21-44 manifests]$ kubectl apply -f flask-deployment.yaml
kubectl apply -f flask-service.yaml
-bash: kubectl: command not found
-bash: kubectl: command not found
[ec2-user@ip-172-31-21-44 manifests]$ sudo curl -o /usr/local/bin/kubectl \
https://s3.us-west-2.amazonaws.com/amazon-eks/1.31.0/2024-09-12/bin/linux/amd64/kubectl
[ec2-user@ip-172-31-21-44 manifests]$ sudo chmod +x /usr/local/bin/kubectl
[ec2-user@ip-172-31-21-44 manifests]$ kubectl version
Client Version: v1.31.0-eks-a737599
Kustomize Version: v5.4.2
Server Version: v1.31.10-eks-931bdca
[ec2-user@ip-172-31-21-44 manifests]$ sukubectl apply -f flask-service.yaml
service/nextwork-flask-backend created
[ec2-user@ip-172-31-21-44 manifests]$
i-096069216341bb0ce (nextwork-eks-instance)
PublicIPs: 54.147.157.243 PrivateIPs: 172.31.21.44

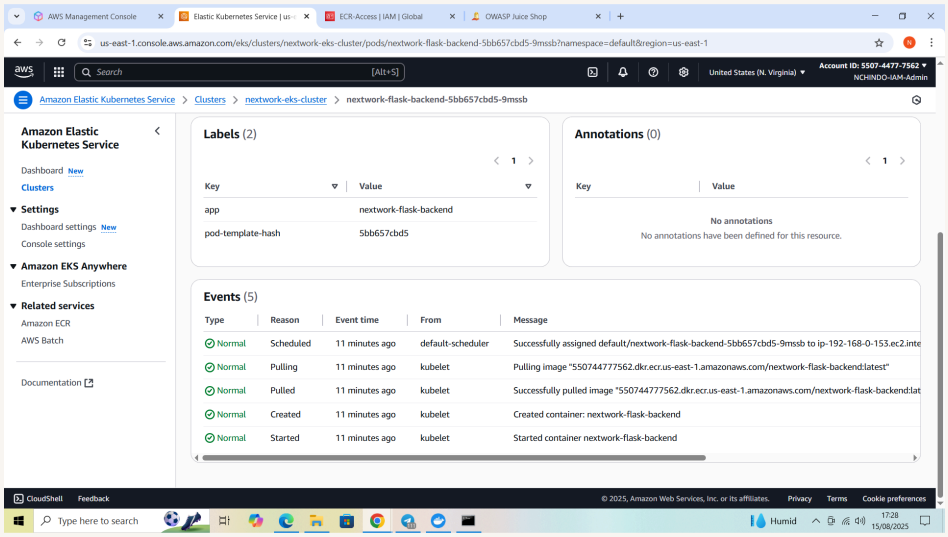
Verifying Deployment

My extension for this project is to use the EKS console to visually monitor and verify the health and status of my cluster and its nodes. I had to set up IAM access policies because, even though I have AdministratorAccess in AWS, Kubernetes manages its access controls separately, so I needed explicit permission to view and manage cluster resources like nodes. I set up access by running the eksctl create iamidentitymapping command with my IAM User ARN, which mapped my AWS user to the Kubernetes system:masters group, giving me admin-level permissions inside the cluster.

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 group one or more containers together so they can operate as a single unit. Containers in a pod share the same network space and storage, allowing them to communicate and share data efficiently within the pod.

The EKS console shows you the events for each pod, where I could see events like the pod being assigned an internal IP, the container image being pulled from Amazon ECR, and the container successfully created and started. This validated that my backend container was correctly deployed, running, and accessible within the cluster's internal network.

Nchindo Boris
NextWork Student



The place to learn & showcase your skills

Check out nextwork.org for more projects