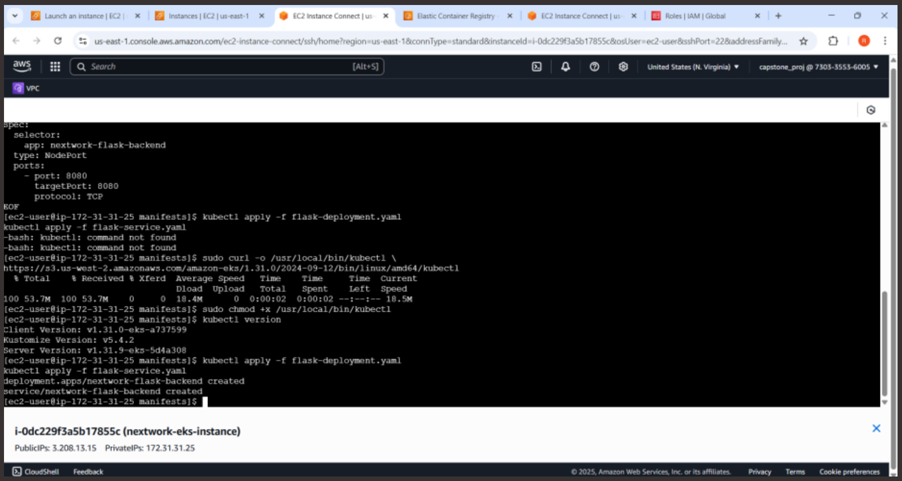
nextwork.org

Deploy Backend with Kubernetes

Rishit Burman



Launch an instance | EC2 |
Instances | EC2 | us-east-1
EC2 Instance Connect | us-
Elastic Container Registry
EC2 Instance Connect | us-
Roles | IAM | Global
us-east-1.console.aws.amazon.com/ec2-instance-connect/ssh/home?region=us-east-1&connType=standard&instanceId=i-0dc229f3a5b17855c&osUser=ec2-user&sshPort=22&addressFamily...
Search
[Alt+S]
United States (N. Virginia)
capstone_proj @ 7303-3553-6005
VPC

```
spec:
  selector:
    app: nextwork-flask-backend
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      protocol: TCP
EOF
[ec2-user@ip-172-31-31-25 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-172-31-31-25 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
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 53.7M  100 53.7M    0     0  18.4M      0  0:00:02  0:00:02 --:--:-- 18.5M
[ec2-user@ip-172-31-31-25 manifests]$ sudo chmod +x /usr/local/bin/kubectl
[ec2-user@ip-172-31-31-25 manifests]$ kubectl version
Client Version: v1.31.0-eks-a737599
Kustomize Version: v5.4.2
Server Version: v1.31.9-eks-5d4a308
[ec2-user@ip-172-31-31-25 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-172-31-31-25 manifests]$
```

i-0dc229f3a5b17855c (nextwork-eks-instance)

PublicIPs: 3.208.13.15 PrivateIPs: 172.31.31.25

Introducing Today's Project!

In this project, I will deploy the backend of an application on a Kubernetes cluster using Amazon EKS. I will create the EKS cluster, building a Docker image of the backend, pushing it to Amazon ECR, and writing manifest files that define how the application should run. Using kubectl, I will apply these manifests to deploy and manage the backend in the EKS environment. This process allows me to take backend code from development to a live, scalable production environment using container orchestration. By doing this, I will gain hands-on experience in deploying real-world cloud-native applications, understanding the connection between infrastructure (EC2, ECR, EKS) and deployment practices, and verifying the deployment through logs and endpoints. Successfully completing this step proves my ability to manage backend deployments at scale in a modern cloud setup.

Tools and concepts

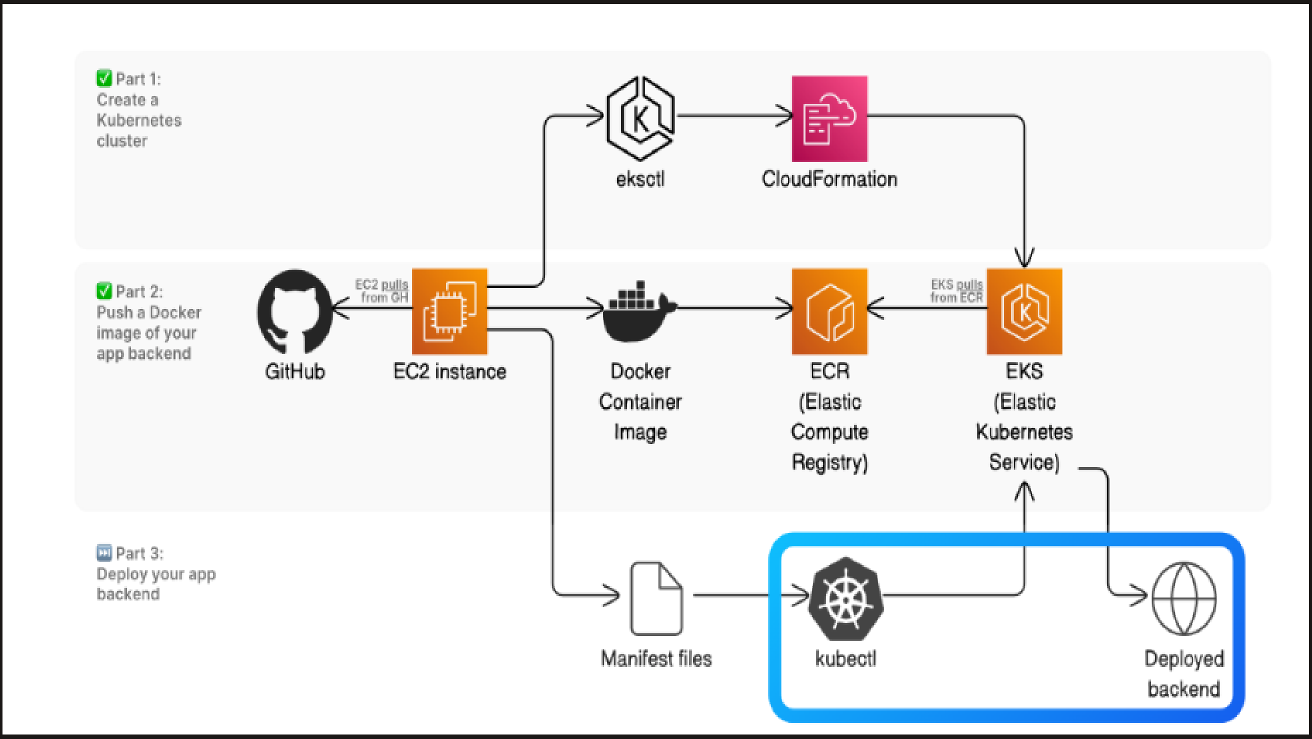
I used Kubernetes, ECR, kubectl, and EC2 to deploy a containerized backend application to an Amazon EKS cluster. Kubernetes managed the orchestration, ECR stored my Docker image, kubectl applied the deployment and service manifests, and EC2 hosted my working environment. Key concepts include using manifests to define desired application states, Deployments to manage replica sets and updates, and Services to expose the application to network traffic. I also learned about pods, the smallest deployable unit in Kubernetes, and how EKS integrates AWS resources to run and scale containerized apps efficiently.

Project reflection

This project took me approximately 3 hours to complete. The most challenging part was configuring IAM roles and permissions correctly to allow kubectl to interact with the EKS cluster, as even a small misconfiguration could break access.

My favourite part was applying the Kubernetes manifests and watching the deployment come alive in the EKS console, especially viewing the pods start up and confirming the backend was running through the Events tab. It was exciting to see all the pieces–from ECR to EC2 to EKS–come together into a working deployment.

Architecture


✅ Part 1: Create a Kubernetes cluster
eksctl
CloudFormation
✅ Part 2: Push a Docker image of your app backend
GitHub
EC2 pulls from GH
EC2 instance
Docker Container Image
ECR (Elastic Compute Registry)
EKS pulls from ECR
EKS (Elastic Kubernetes Service)
⏭️ Part 3: Deploy your app backend
Manifest files
kubectl
Deployed backend

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 act as the control plane that manages and orchestrates the containerized backend application. I used an EC2 instance to run the eksctl create cluster command, which provisioned an Amazon EKS cluster along with a node group consisting of EC2 instances to serve as worker nodes. The cluster is responsible for scheduling the backend containers, ensuring they stay running, handling networking between services, and automatically scaling the application based on demand. By using Amazon EKS, I was able to skip the manual setup of core components like networking, security groups, and load balancing, allowing me to focus on deploying and managing the application itself. This setup provides a production-ready environment for running my backend with high availability and resilience.

Backend code

I retrieved backend code by cloning the nextwork-flask-backend repository from GitHub using Git on my EC2 instance. Pulling code is essential to this deployment because it provides the source files—like app.py, Dockerfile, and requirements.txt—that define how the backend application runs and what dependencies it needs. These files are necessary to build a Docker image of the backend, which will later be deployed on the Kubernetes cluster. Using Git makes this process efficient and reliable, as it allows me to fetch an exact copy of the code developed by my team member. After installing Git and configuring it with my credentials, I used the git clone command with the HTTPS URL of the repository. Once the cloning was complete, I verified the presence of the project folder using the ls command and explored the contents by navigating into the directory. This ensures that I now have the complete backend code ready for containerization and deployment.

Container image

Once I cloned the backend code, I built a container image because Kubernetes needs a standardized, portable package of the application to deploy and manage it across the cluster. Without an image, it would be difficult for Kubernetes to create consistent containerized environments, as it wouldn't know what code, dependencies, or runtime to use. The container image serves as a blueprint that ensures the application runs the same way regardless of the environment—be it development, staging, or production. I used Docker to build the image using the Dockerfile in the project directory, which included all the necessary instructions to package the backend. Naming the image nextwork-flask-backend helps identify and manage it easily later when uploading to ECR or deploying to EKS. This step is essential to allow Kubernetes to efficiently orchestrate multiple identical containers and scale the app reliably.

Amazon ECR

I also pushed the container image to a container registry, which is Amazon ECR (Elastic Container Registry). ECR facilitates scaling for my deployment because it provides a centralized, secure, and highly available location for storing container images that can be accessed by all nodes in the Kubernetes cluster. This eliminates the need to manually distribute the image to each node, making it much easier to scale horizontally by simply adding more nodes that automatically pull the image. ECR integrates seamlessly with Amazon EKS, reducing authentication overhead and improving deployment speed and consistency. By tagging the image with latest, I ensured that Kubernetes can always reference and pull the most up-to-date version of my backend for deployment.

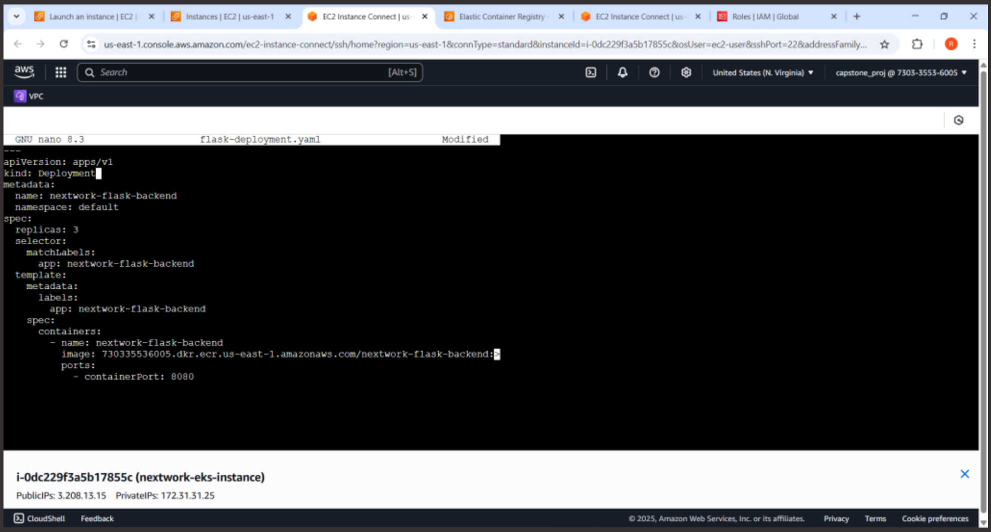
Manifest files

Kubernetes manifests are YAML configuration files that define the desired state of objects within a Kubernetes cluster, such as Deployments, Services, ConfigMaps, and more. Manifests tell Kubernetes what containers to run, how many replicas to create, what resources to allocate, and how to expose the application. They serve as the blueprints for automating application deployment, updates, and management within the cluster. Manifests are helpful because they provide a declarative, repeatable, and version-controlled way to manage infrastructure. Instead of manually configuring settings every time, you can apply the same manifest to reproduce an environment consistently, which improves scalability, reliability, and collaboration across teams.

A Deployment manifest manages how Kubernetes runs and maintains instances of a containerized application across the cluster. It defines the desired state—such as the number of replicas (pods), container image to use, labels, and ports—and ensures that the actual state matches it. Kubernetes uses this manifest to create, update, and monitor the backend pods, replacing them automatically if they crash or during updates. The container image URL in my Deployment manifest tells Kubernetes exactly where to pull the application's image from, which in this case is Amazon ECR. This ensures that Kubernetes uses the correct version of my backend, enabling consistent behavior across all replicas and environments.

A Service resource exposes a group of pods in a Kubernetes cluster and provides a stable way to access them, even if the underlying pods change. It acts like a traffic router, making sure that network requests reach the correct application instances. My Service manifest sets up a NodePort-type Service that allows external traffic to reach the backend application through port 8080.

It uses a selector (app: nextwork-flask-backend) to find and forward traffic to the matching pods, which are defined in the Deployment manifest. This ensures users can access the backend via the node's IP and assigned port, even as pods are replaced or scaled by Kubernetes.


us-east-1.console.aws.amazon.com/ec2-instance-connect/ssh/home?region=us-east-1&connType=standard&instanceId=i-0dc229f3a5b17855c&osUser=ec2-user&sshPort=22&addressFamily...
United States (N. Virginia)
capstone_proj @ 7303-3553-6005

```
  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: 730335536005.dkr.ecr.us-east-1.amazonaws.com/nextwork-flask-backend:>
          ports:
            - containerPort: 8080
```

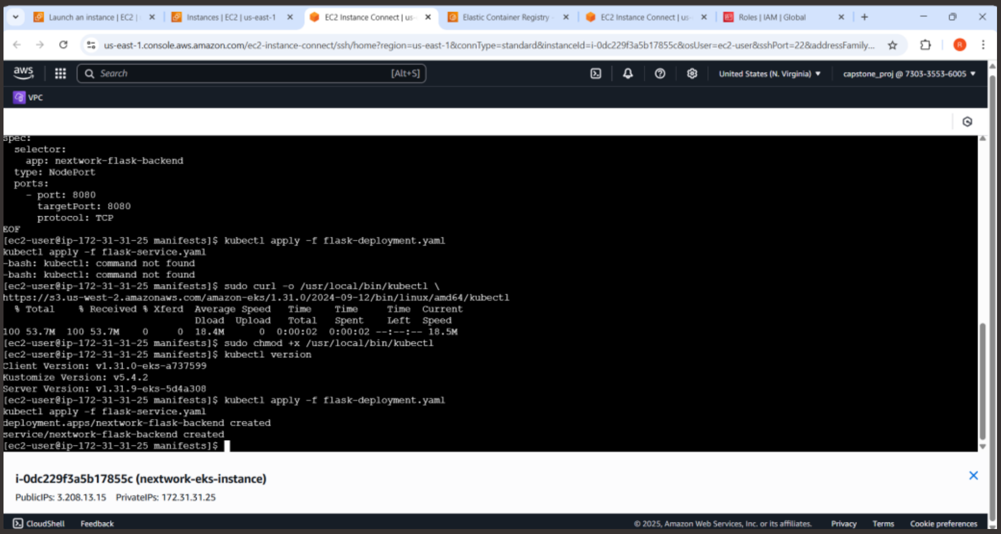
i-0dc229f3a5b17855c (nextwork-eks-instance)
PublicIPs: 3.208.13.15 PrivateIPs: 172.31.31.25

Backend Deployment!

To deploy my backend application, I installed kubectl, the command-line tool used to manage Kubernetes resources. Then, I granted execution permissions to kubectl so I could use it. After confirming the installation with kubectl version, I applied my Kubernetes manifest files using the commands kubectl apply -f flask-deployment.yaml and kubectl apply -f flask-service.yaml. These commands told Kubernetes to create a Deployment resource (which runs multiple replicas of my backend app) and a Service resource (which exposes my app and routes traffic to the backend containers). This completed the deployment of my backend into the EKS cluster.

kubectl

kubectl is the command-line tool for interacting with Kubernetes clusters. I need this tool to apply, update, and manage resources like Deployments and Services within the cluster. It lets me communicate directly with Kubernetes to deploy applications, check pod status, scale services, and troubleshoot issues. I can't use eksctl for the job because eksctl is primarily used for creating, deleting, and configuring EKS clusters, not for managing the workloads running inside them. Once the cluster is set up with eksctl, I switch to kubectl to control the actual applications and services running within the cluster.



```
spec:
  selector:
    app: nextwork-flask-backend
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      protocol: TCP
EOF
[ec2-user@ip-172-31-31-25 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-172-31-31-25 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
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 53.7M  100 53.7M    0     0  18.4M      0  0:00:02  0:00:02 --:--:-- 18.5M
[ec2-user@ip-172-31-31-25 manifests]$ sudo chmod +x /usr/local/bin/kubectl
[ec2-user@ip-172-31-31-25 manifests]$ kubectl version
Client Version: v1.31.0-eks-a737599
Kustomize Version: v5.4.2
Server Version: v1.31.9-eks-5d4a308
[ec2-user@ip-172-31-31-25 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-172-31-31-25 manifests]$
```

i-0dc229f3a5b17855c (nextwork-eks-instance)

PublicIPs: 3.208.13.15 PrivateIPs: 172.31.31.25

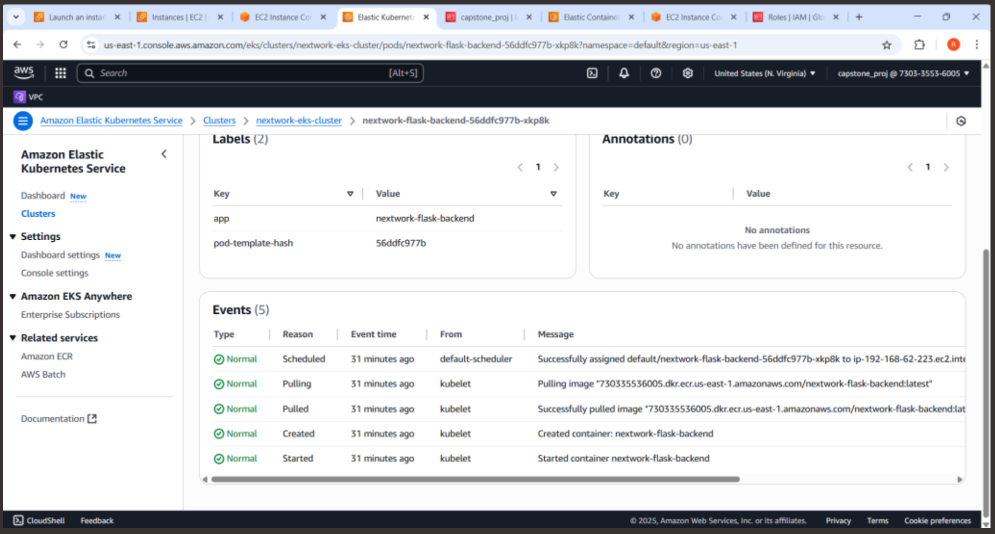
Verifying Deployment

My extension for this project is to use the EKS console to visually verify and monitor the state of my Kubernetes cluster, including checking that my nodes, deployments, and services are all running correctly. I had to set up IAM access policies because EKS needs permission to interact with other AWS services like EC2 and CloudFormation to provision resources securely and correctly. I set up access by creating an EKS cluster role with the necessary policies (like AmazonEKSClusterPolicy and AmazonEKSWorkerNodePolicy), and attached them to the EC2 instances that act as nodes in my EKS cluster. This ensured that both the control plane and the worker nodes could communicate and function within the AWS environment.

Once I gained access into my cluster's nodes, I discovered pods running inside each node. Pods are the smallest deployable units in Kubernetes, and they act as wrappers that bundle one or more containers together so they can operate as a single unit. Containers in a pod share the same network namespace and storage, which means they can communicate with each other directly using localhost and access shared volumes. This setup allows containers in the same pod to collaborate more efficiently, for example, by sharing files or coordinating actions. You can't deploy containers alone in Kubernetes–they must always be deployed within a pod, making pods the fundamental building block for running applications in a Kubernetes cluster.

The EKS console shows you the events for each pod, where I could see the step-by-step progress of my backend pod deployment. This validated that Kubernetes successfully assigned an internal IP, pulled the container image from Amazon ECR, created the container, and successfully started it inside the pod. These events confirmed that my backend application is running properly within the cluster.

By tracking each step–from image pull to container start–I ensured that there were no errors during deployment and that my backend is now accessible inside the Kubernetes network, ready to receive traffic via the configured Service.



us-east-1.console.aws.amazon.com/eks/clusters/nextwork-eks-cluster/pods/nextwork-flask-backend-56ddfc977b-xkp8k?namespace=default&region=us-east-1

Amazon Elastic Kubernetes Service > Clusters > nextwork-eks-cluster > nextwork-flask-backend-56ddfc977b-xkp8k

Labels (2)

Key	Value
app	nextwork-flask-backend
pod-template-hash	56ddfc977b

Annotations (0)

Key	Value

No annotations
No annotations have been defined for this resource.

Events (5)

Type	Reason	Event time	From	Message
Normal	Scheduled	31 minutes ago	default-scheduler	Successfully assigned default/nextwork-flask-backend-56ddfc977b-xkp8k to ip-192-168-62-223.ec2.inte
Normal	Pulling	31 minutes ago	kubelet	Pulling image "730335536005.dkr.ecr.us-east-1.amazonaws.com/nextwork-flask-backend:latest"
Normal	Pulled	31 minutes ago	kubelet	Successfully pulled image "730335536005.dkr.ecr.us-east-1.amazonaws.com/nextwork-flask-backend:lat
Normal	Created	31 minutes ago	kubelet	Created container: nextwork-flask-backend
Normal	Started	31 minutes ago	kubelet	Started container nextwork-flask-backend

Thank You!