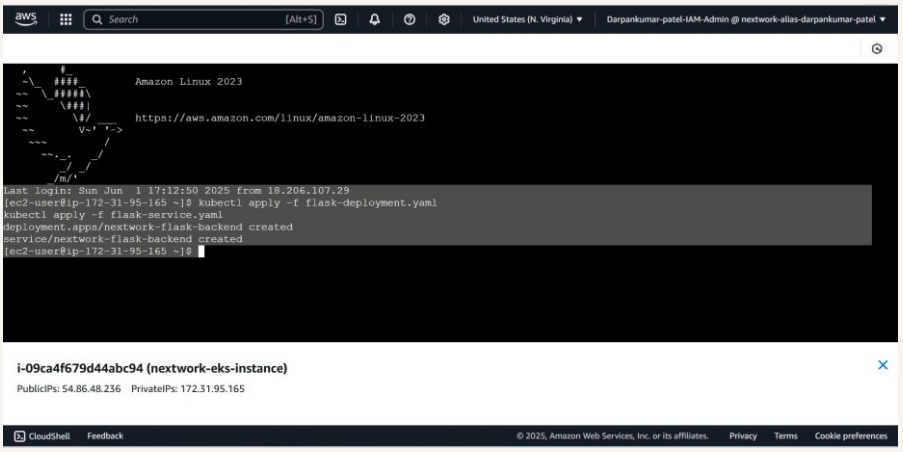
Darpan Patel

Deploy Backend with Kubernetes



Darpan Patel

Introducing Today's Project!

In this project, I will deploy the backend of an application on a Kubernetes cluster using Amazon EKS because this will allow me to manage, scale, and maintain the app efficiently in a cloud-native environment. By doing this, I will gain hands-on experience with Kubernetes deployment processes, including setting up the cluster, writing manifest files, and using kubectl to control the deployment, which are essential skills for modern cloud infrastructure and DevOps roles.

Tools and concepts

I used Kubernetes, ECR, kubectl, and eksctl to deploy and manage a containerized backend application on an EKS (Elastic Kubernetes Service) cluster. Key concepts include using manifests (YAML files) to define Kubernetes resources like Deployments and Services, understanding how pods are the smallest deployable units in Kubernetes, and managing access with IAM and Kubernetes RBAC. I also used `kubectl` to apply manifests and interact with cluster resources, while `eksctl` was used to provision and configure the EKS cluster. Amazon ECR was used to store and pull the container image for deployment.

Project reflection

This project took me approximately 1 hour to complete. The most challenging part was configuring IAM access and ensuring I had the correct permissions to view and interact with nodes inside the EKS cluster, since AWS IAM and Kubernetes RBAC are separate systems and required additional setup. My favourite part was seeing the backend application successfully deploy and run inside a pod, and being able to track its progress through the EKS console events — it made all the effort feel real and rewarding.

Darpan Patel

Project Set Up

Kubernetes cluster

To set up this project, I launched a Kubernetes cluster. The cluster's role in this deployment is to act as the central control plane that manages containerized applications, automating tasks like scheduling, scaling, and networking. I began by launching an EC2 instance to serve as my management environment, then installed **eksctl**, a command-line tool that simplifies creating and configuring EKS clusters on AWS. After creating and attaching an IAM role with the necessary permissions to my EC2 instance, I used **eksctl** to create the EKS cluster, specifying the node type and number of nodes. This cluster provides the infrastructure where Kubernetes schedules and runs the backend application containers efficiently and reliably.

Backend code

I retrieved backend code by cloning the GitHub repository that my team member created using Git. Pulling code is essential to this deployment because it gives me access to the exact backend application files—like the Flask app, Dockerfile, and dependencies—that I need to build, containerize, and deploy on the Kubernetes cluster. Without the backend code, I wouldn't have the core logic or components required to run the application.

Container image

Once I cloned the backend code, I built a container image because Kubernetes requires a consistent, portable, and self-contained package of the application to deploy and manage across its cluster.

Without an image, it would be difficult for Kubernetes to reliably create identical containers that include all the necessary code, libraries, and dependencies. The container image ensures

that every instance of the backend runs the same way, regardless of the environment, which simplifies scaling, updates, and maintenance of the application.

I also pushed the container image to a container registry, which is Amazon ECR. ECR facilitates scaling for my deployment because it provides a centralized, secure location where Kubernetes can reliably pull the exact same container image across multiple nodes in the cluster. This ensures consistency and makes it easy to manage updates, as Kubernetes can automatically fetch the latest tagged image from ECR whenever it needs to start or scale new containers, without having to manually distribute images to each node.

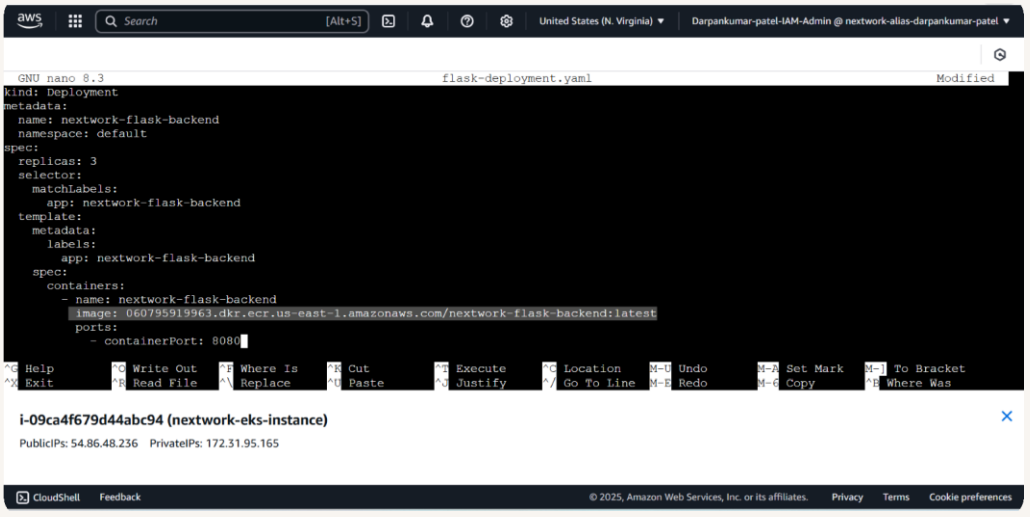
Manifest files

Kubernetes manifests are configuration files, usually written in YAML or JSON, that define the desired state of your application within a Kubernetes cluster. Manifests specify how Kubernetes should deploy, manage, and expose your application— including details like which container images to use, how many replicas to run, resource requirements, networking rules, and more. Manifests are helpful because they provide a clear, repeatable, and version-controlled way to describe your application's infrastructure and behavior. This ensures consistent deployments across different environments (development, testing, production) and simplifies managing complex applications at scale by automating container orchestration.

A Deployment manifest manages how Kubernetes runs and maintains multiple instances of your application across the cluster. It defines important settings like how many copies (replicas) of your app should be running, how to update those copies, and how to handle scaling and fault tolerance automatically. The container image URL in my Deployment manifest tells Kubernetes exactly where to find the container image it needs to deploy— specifically, the location of the image stored in Amazon ECR or another container registry. This ensures Kubernetes pulls the correct version of my app to run inside the cluster.

Darpan Patel

A Service resource exposes an application running in a Kubernetes cluster to network traffic, either from within the cluster or from external sources. It acts as a stable endpoint and a traffic router that directs requests to the appropriate pods based on labels. My Service manifest sets up a Kubernetes Service named `nextwork-flaskbackend` that routes external traffic on port 8080 to the backend pods labeled `app: nextwork-flask-backend`. By using the `NodePort` type, it makes the application accessible outside the cluster via a specific port on each node, allowing users to reach the app through the node's IP address and the assigned port.



Darpan Patel

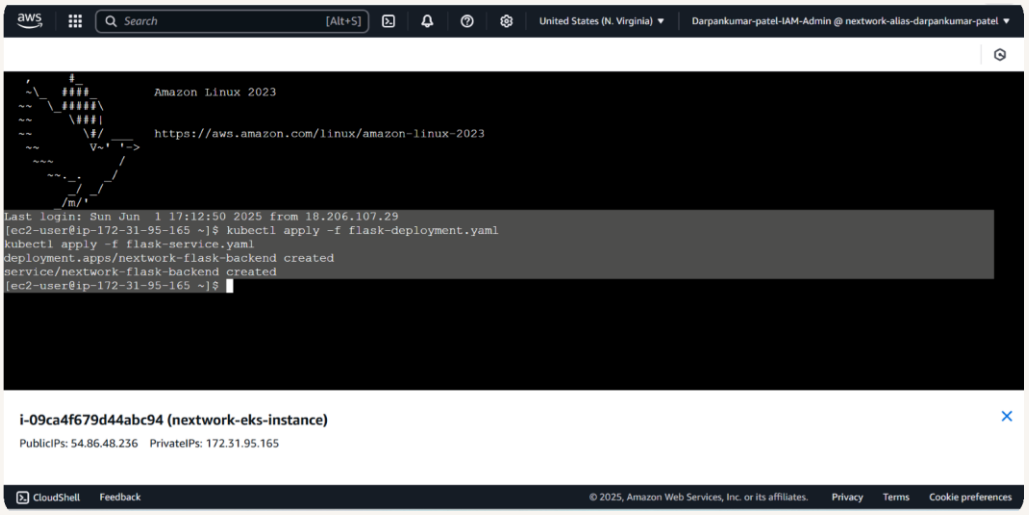
Backend Deployment!

To deploy my backend application, I first installed **kubectl**, the command-line tool needed to interact with my Kubernetes cluster. Then, I used **kubectl apply** commands to apply the Deployment and Service manifests I created. These commands told Kubernetes to create the resources defined in those manifests, which launched my backend application containers and exposed them to network traffic through the configured Service. This process ensured my app was running and accessible within the cluster.

kubectl

kubectl is the command-line tool used to interact with Kubernetes clusters and manage the resources running inside them. I need this tool to deploy, update, and troubleshoot applications within my Kubernetes cluster by applying manifests, checking the status of pods, scaling deployments, and more. I can’t use eksctl for the job because eksctl is primarily designed for creating and managing the EKS cluster itself—not for managing the workloads and resources inside the cluster. Kubernetes resource management is what kubectl specializes in.

Darpan Patel



Verifying Deployment

My extension for this project is to use the EKS console to visually monitor and verify the status of my Kubernetes cluster and its resources. I had to set up IAM access policies because, even though I have AWS AdministratorAccess, Kubernetes uses its own access control system, so AWS permissions alone don’t grant me access inside the cluster. I set up access by mapping my IAM user to the Kubernetes `system:masters` group using the `eksctl create iamidentitymapping` command, which gives me the necessary permissions to view and manage the nodes and other cluster resources through the console.

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 bundle one or more containers together to run as a single unit. They provide an environment where containers can share resources and coordinate closely. Containers in a pod share the same network namespace and storage volumes, which means they can communicate over `localhost` and access shared data efficiently. This setup is especially useful for

applications with tightly coupled components that need to run side by side, such as a main app container and a helper or logging container. Pods allow Kubernetes to manage and scale these containerized workloads more effectively across the cluster.

The EKS console shows you the events for each pod, where I could see a step-bystep log of the pod's lifecycle—including assigning an internal IP, pulling the container image from Amazon ECR, successfully creating the container, and starting it. This validated that my backend application was correctly deployed, the image was pulled successfully, and the container is now up and running inside the pod within my Kubernetes cluster.