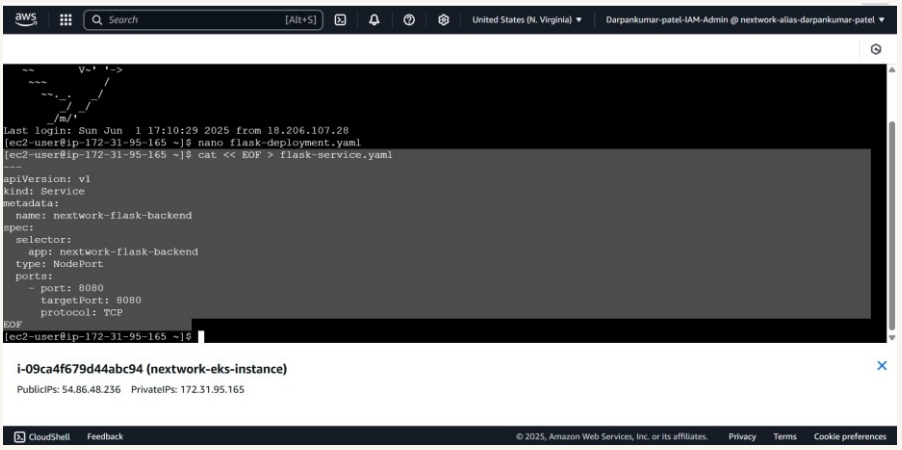
Darpan Patel

Create Kubernetes Manifests



Darpan Patel

Introducing Today's Project!

In this project, I will create Kubernetes manifest files because these files are essential for telling Kubernetes how to deploy and manage my backend application in a reliable and scalable way. By writing Deployment and Service manifests, I can ensure my app runs smoothly on Amazon EKS and is properly exposed to users.

Tools and concepts

I used Amazon EKS, `eksctl`, and `kubectl` to provision and manage a Kubernetes cluster for deploying a containerized application. `eksctl` simplified the creation of the EKS cluster, while `kubectl` allowed me to interact with Kubernetes resources through command-line commands. Key concepts include using manifests to declaratively define Kubernetes objects like Deployments and Services, understanding how Pods are created and managed, and leveraging labels and selectors to control which resources are grouped together. I also learned how to define container specs, expose application ports, and use replica settings to ensure scalability and high availability of my backend application.

Project reflection

I did this project today to deepen my hands-on experience with Kubernetes and Amazon EKS, specifically focusing on how deployments are managed through manifests. I wanted to strengthen my practical understanding of how infrastructure as code works in real-world container orchestration. This project definitely met my goals—it helped reinforce key concepts like Deployments, Pods, replicas, selectors, and container specs. It also improved my confidence using `kubectl` and eksctl to manage and troubleshoot Kubernetes workloads.

This project took me approximately 6 to 8 hours to complete. The most challenging part was understanding how the different layers of the Deployment manifest connect—especially the relationship between selectors, labels, and how Kubernetes matches and manages Pods. It required careful attention to YAML structure and syntax. My

Darpan Patel

Project Set Up

Kubernetes cluster

To set up today's project, I launched a Kubernetes cluster. Steps I took to do this included launching an EC2 instance named `nextwork-eks-instance` using the Amazon Linux 2023 AMI. I installed `eksctl`, a CLI tool to simplify EKS cluster creation. Then, I created an IAM role (`nextwork-eks-instance-role`) with `AdministratorAccess` and attached it to my EC2 instance to grant it the necessary permissions. After that, I used the `eksctl create cluster` command to spin up a Kubernetes cluster named `nextwork-eks-cluster`. This cluster was configured with 3 `t2.micro` nodes, running Kubernetes version 1.31 in my chosen AWS region. This setup provides a managed environment to run containerized applications using Kubernetes on AWS, allowing easier integration with other AWS services and simplifying complex networking and scaling configurations typically required when setting up Kubernetes from scratch.

Backend code

I retrieved the backend that I plan to deploy by installing Git on my EC2 instance and cloning the project repository from GitHub using the HTTPS URL provided by my team member. **Backend** is the server-side part of the application responsible for processing logic, handling API requests, and managing data storage. After cloning the repository, I confirmed that the backend code—including files like `app.py`, `Dockerfile`, and `requirements.txt`—was successfully downloaded into a new folder named `nextwork-flask-backend` on my instance.

Container image

Once I cloned the backend code, I built a container image because Kubernetes requires a container image to deploy the application consistently across environments. I used Docker to create this image. First, I installed and started Docker on my EC2 instance. Then, I added the `ec2-user` to the Docker group so I could run Docker commands without needing sudo. After reconnecting to the instance to apply these changes, I navigated into the backend project directory and ran the command `docker build -t nextwork-flask-backend .` which used the Dockerfile in the folder to

package the backend code and its dependencies into a reusable container image named `nextwork-flask-backend`.

I also pushed the container image to a container registry, because Kubernetes needs a reliable, centralized place to pull the image from when deploying the app. To push the image to ECR, I first created an Amazon ECR repository named `nextwork-flaskbackend` using the AWS CLI. Then, I authenticated my EC2 instance with ECR using the login command provided by AWS. After that, I tagged my local Docker image with the ECR repository URI to link it to the remote repository. Finally, I pushed the tagged image to the ECR repository using Docker push commands. This ensured my backend container image was securely stored and ready for Kubernetes to access and deploy.

Manifest files

Kubernetes manifests are configuration files written in YAML or JSON that define the desired state and settings of your Kubernetes resources, such as deployments, services, and pods. Manifests are helpful because they provide clear, repeatable instructions to Kubernetes on how to deploy and manage your application — specifying things like which container images to use, how many replicas to run, resource limits, and networking rules. This allows you to automate and version control your deployments, ensuring consistency across different environments.

A Kubernetes deployment manages the desired state of your application by ensuring that the specified number of container replicas are running and healthy across the cluster. It automates tasks like creating, updating, and rolling back containers, so your app stays available and consistent even during changes or failures. The container image URL in my Deployment manifest tells Kubernetes exactly which container image to pull from the registry (like Amazon ECR) when it needs to create or update pods running my application. This ensures Kubernetes always uses the correct version of my app when deploying or scaling it.

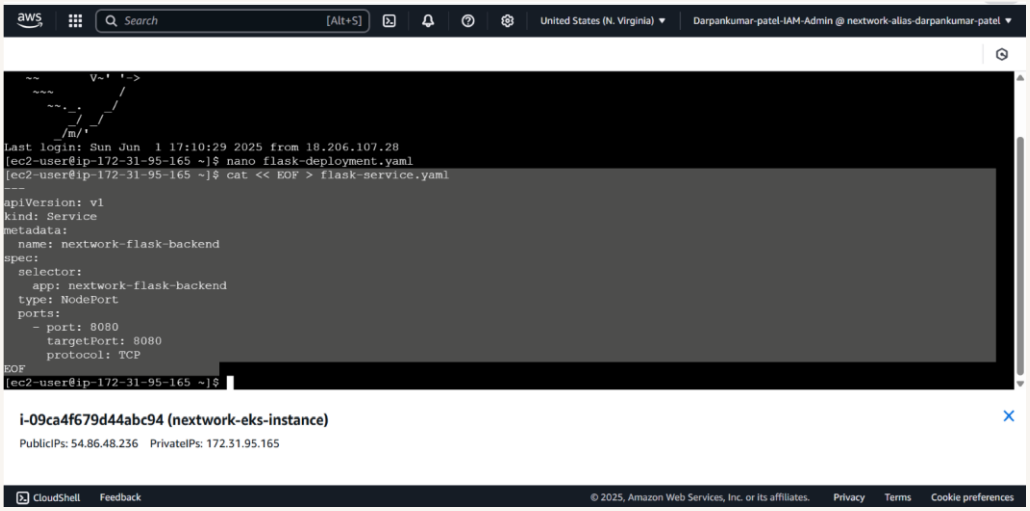
Darpan Patel

Service Manifest

A Kubernetes Service exposes your application to network traffic, allowing other parts of your system or external users to communicate with the pods running your app. You need a Service manifest to tell Kubernetes how to route this traffic, specifying which pods to send it to, what ports to use, and how the traffic should be handled (e.g., internally within the cluster or externally). This ensures your app is reachable and can scale reliably without clients needing to know the details of individual pod IP addresses.

My Service manifest sets up a Kubernetes Service named `nextwork-flask-backend` that routes network traffic to pods labeled with `app: nextwork-flask-backend`. It

exposes port 8080 on the cluster and forwards that traffic to the same port on the backend pods using the TCP protocol. The Service type is `NodePort`, which makes the application accessible externally through a port on each node in the Kubernetes cluster.



Deployment Manifest

Annotating the Deployment manifest helped me understand how Kubernetes manages applications from top to bottom because it broke down each section into meaningful roles. Starting with the number of replicas, I learned how Kubernetes ensures high availability by maintaining a set number of Pods. The selector section showed me how matchLabels connect the Deployment to the correct Pods, which is critical for tracking and management. The template section gave me a clear idea of how each Pod is created, using consistent labels and structure. Diving deeper into the Pod spec and containers section, I understood how the image and containerPort determine what application runs and how it communicates. Annotating helped me connect the dots between the desired state and actual runtime behavior. It turned the YAML from something abstract into a clear blueprint that I can read, explain, and modify with confidence. Overall, it was a hands-on way to learn Kubernetes architecture and improve my Skills.

Darpan Patel

A notable line in the Deployment manifest is the number of replicas, which means how many identical copies of your application Pods Kubernetes should run at any given time. Pods are relevant to this because they are the smallest deployable units in Kubernetes that hold your containerized application. Each replica corresponds to one Pod running your app container(s). Kubernetes ensures that the specified number of Pods (replicas) are always running and healthy, replacing any that fail or get terminated to maintain your desired state.

One part of the Deployment manifest I still want to know more about is the `strategy` field under the Deployment `spec` because while I understand the default behavior is to use a rolling update, I'm not fully clear on how to customize it for different deployment strategies like `Recreate`, or how the `maxUnavailable` and `maxSurge` settings affect application uptime during updates.

Knowing more about this would help me better control how new versions of my application are rolled out and reduce downtime or potential service interruptions during deployments.