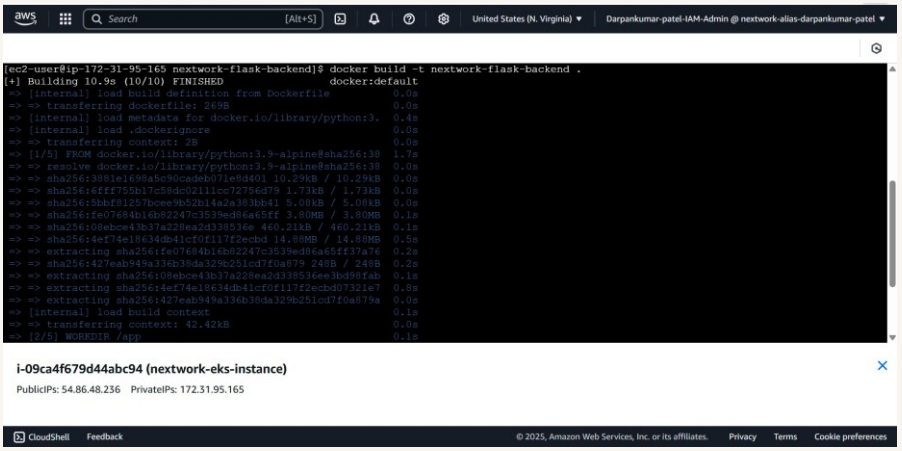
Darpan Patel

Set Up Kubernetes Deployment



Darpan Patel

Introducing Today's Project!

In this project, I will clone a backend application from GitHub, build a Docker image of it, push that image to Amazon ECR, and prepare it for deployment with Kubernetes** because this is a crucial step in modern cloud-native development. It helps ensure that applications are containerized, version-controlled, and ready to be deployed in a scalable and reliable Kubernetes environment using AWS services. This workflow mirrors real-world DevOps pipelines and strengthens my skills in containerization, CI/CD, and cloud orchestration.

Tools and concepts

I used Amazon EKS, Git, Docker, and Amazon ECR to build, containerize, store, and deploy the backend application efficiently. Key steps include cloning the backend repository from GitHub, exploring and understanding the backend code, building a Docker image of the app, creating an Amazon ECR repository to securely store the container image, pushing the Docker image to ECR, and finally using Amazon EKS (Elastic Kubernetes Service) to deploy the containerized backend app in a scalable and consistent environment. This workflow ensures seamless integration, version control, and reliable deployment of the app.

Project reflection

This project took me approximately 1 hour to complete. The most challenging part was setting up and configuring the Amazon ECR repository and ensuring the Docker image pushed correctly with all permissions in place. My favourite part was exploring the backend code and understanding how the Flask app connects with the Hacker News API to fetch and process data, which gave me a clear picture of how the whole system works end-to-end.

Darpan Patel

Something new that I learnt from this experience was how to seamlessly integrate
Docker container images with Amazon ECR and then use those images in a Kubernetes environment like Amazon EKS. I also gained a deeper understanding of

What I'm deploying

To set up today's project, I launched a Kubernetes cluster. Steps I took to do this included launching an EC2 instance as my control hub, installing eksctl, and configuring AWS credentials. Then, I created an IAM role with AdministratorAccess and attached it to my EC2 instance to give it the necessary permissions. Finally, I used eksctl to create an EKS cluster with 3 t2.micro nodes. This setup gave me a fully managed Kubernetes environment where I could deploy and manage containerized applications efficiently.

I'm deploying an app's backend

In this step, I will connect to my EC2 instance again and install Git because I need Git to clone the backend application repository from GitHub. Once Git is installed and configured, I will clone the backend code so that I have the full application source locally on my EC2 instance, which is essential for building and deploying the backend to the Kubernetes cluster.

Darpan Patel

```
[ec2-user@ip-172-31-95-165 ~]$ ls
nextwork-flask-backend
```

Building a container image

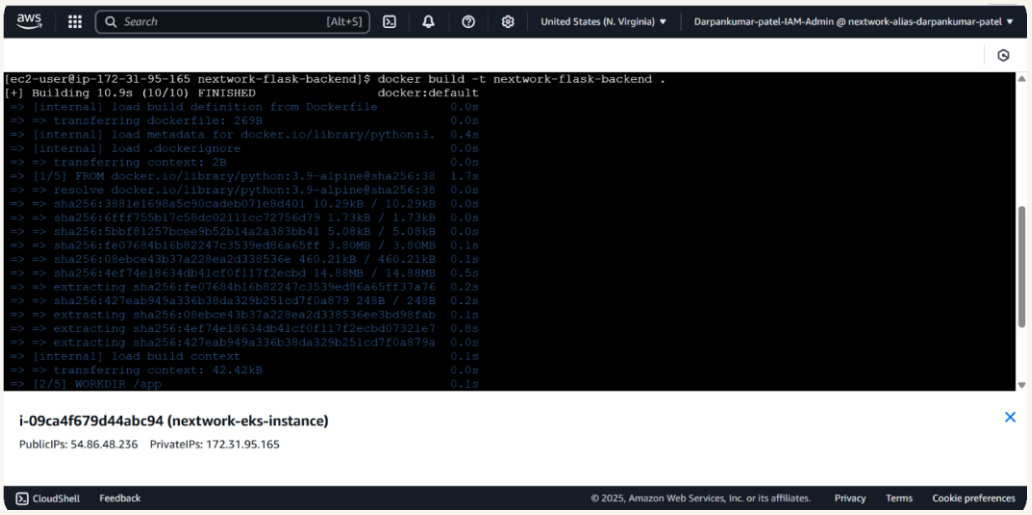
Once I cloned the backend code, my next step is to build a container image of the backend. This is because Kubernetes deploys applications using container images, which package the app's code, dependencies, and environment settings into a single, portable unit. Building the container image ensures that the backend can run consistently and reliably across different environments, whether in development, testing, or production. It also enables Kubernetes to easily create, scale, and manage multiple instances of the backend as needed.

When I tried to build a Docker image of the backend, I ran into a permissions error because the `ec2-user` on my instance didn't have the necessary rights to run Docker commands directly. Docker requires root-level privileges to create containers and build images, and since Docker was installed for the root user, running Docker commands without `sudo` caused a permission denied error. This meant I either had to use `sudo` before Docker commands or update the user permissions to allow `ec2user` to run Docker commands without elevated privileges.

To solve the permissions error, I added the `ec2-user` to the Docker group by running the command `sudo usermod -a -G docker ec2-user`. The Docker group is a special user group in Linux that grants its members permission to run Docker commands without needing root (or sudo) privileges. By adding my

Darpan Patel

user to this group, I was able to execute Docker commands directly, avoiding the permissions error.



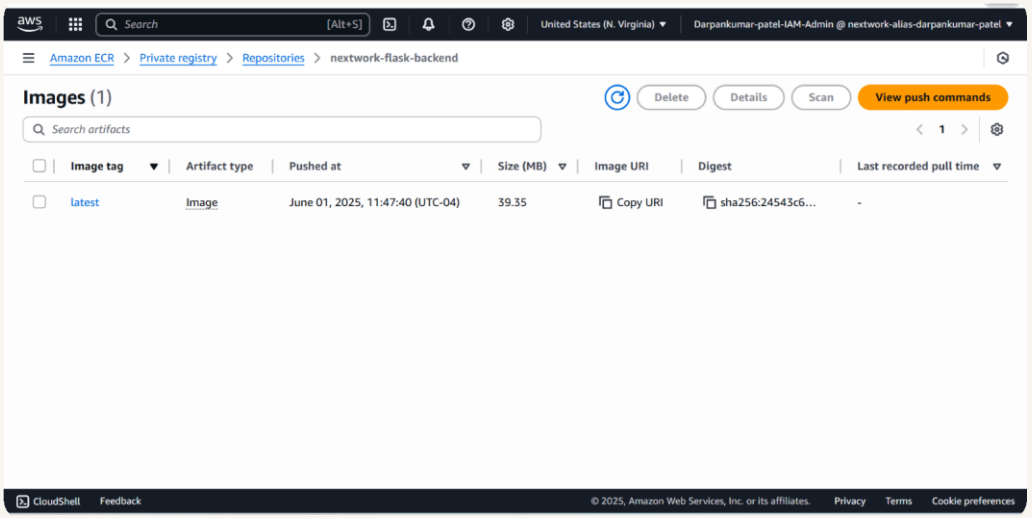
Container Registry

I'm using Amazon ECR in this project to store and manage the Docker container image of the backend application so that it can be pulled and deployed by Amazon EKS. ECR is a good choice for the job because it's tightly integrated with other AWS services like EKS and IAM, making authentication and image access seamless and secure. It also supports features like vulnerability scanning, version control, and high availability, which are essential for deploying scalable and secure containerized applications in a production-ready environment.

Container registries like Amazon ECR are great for Kubernetes deployment because they provide a centralized and secure location to store and manage

Darpan Patel

container images. This allows Kubernetes to pull the correct image version automatically during deployment, ensuring consistency across development, testing, and production environments. ECR also supports image tagging, making it easier to manage versions like `latest` or `v1.0`. Using a registry eliminates the need to preload images manually on each cluster node, simplifying scaling and updates. ECR integrates with AWS IAM for access control and can automatically scan images for vulnerabilities, enhancing security. It also streamlines automation in CI/CD pipelines by allowing seamless image pushing and pulling. Overall, container registries like ECR help maintain a reliable, secure, and efficient container deployment workflow with Kubernetes.



EXTRA: Backend Explained

After reviewing the app's backend code, I've learnt that the backend is a lightweight Flask application designed to serve as an API that takes a user's search topic, queries the Hacker News Search API for related content, processes the results, and returns them in a clean JSON format. It relies on a few key dependencies to handle HTTP requests, routing, and API creation, and the entire backend is packaged into a Docker container for consistent

deployment. This setup ensures the backend can efficiently fetch, process, and serve data to support the frontend or other services in a scalable and reliable way.

Unpacking three key backend files

The requirements.txt file lists all the Python dependencies that the backend application needs to run properly. It specifies the exact versions of libraries like Flask
(for building the web app), Flask-RESTx (for creating the API), Requests (for making
HTTP calls to the Hacker News API), and Werkzeug (which helps with routing in Flask). This file ensures that when the app is deployed or the Docker image is built, all necessary packages are installed consistently across environments.

placeholder

The Dockerfile gives Docker instructions on how to build a container image for the backend application. Key commands in this Dockerfile include: `FROM python:3.9alpine` which sets up a lightweight Python 3.9 environment as the base image. `WORKDIR /app` to set the working directory inside the container. `COPY requirements.txt requirements.txt` to copy the dependencies list into the container. `RUN pip3 install -r requirements.txt` to install all necessary Python packages. `COPY .