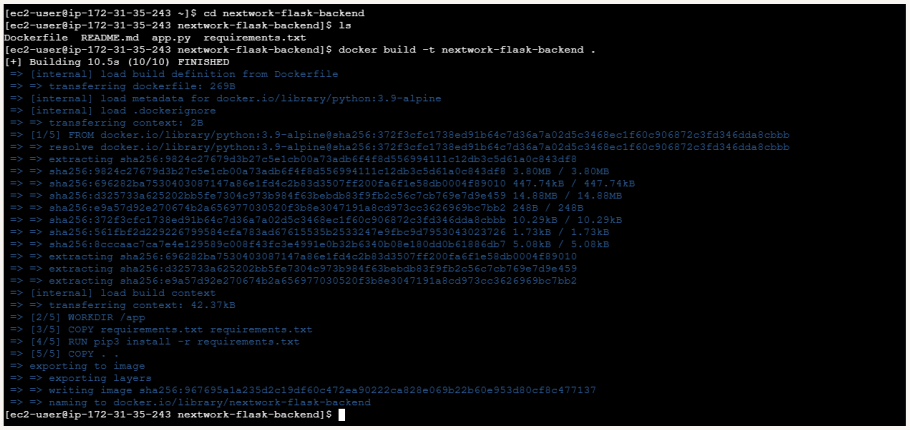
Set Up Kubernetes Deployment

L Louis Moyo



L
Louis Moyo
NextWork Student

Introducing Today's Project!

In this project, I will clone a backend application from GitHub, package it into a Docker image, and push it to Amazon ECR because these are the essential steps before deploying the app into Kubernetes. This project helps me practice real-world DevOps workflows: connecting source code > containerising it > making it available in a registry for Kubernetes to pull and run.

Tools and concepts

I used Amazon EKS to provision a Kubernetes cluster, Git to pull the backend code, Docker to build a container image, and Amazon ECR to store that image. Key steps included launching and connecting to an EC2 instance, installing eksctl and Docker, cloning the backend repository from GitHub, building and tagging the container image, pushing it to ECR, and reviewing the backend code to understand how it works.

Project reflection

This project took me approximately a few hours. The most challenging part was troubleshooting Docker installation and user permissions. My favourite part was seeing the container image successfully built and pushed to ECR, because it showed the workflow coming together from code to deployment-ready image.

L

Louis Moyo
NextWork Student

nextwork.org

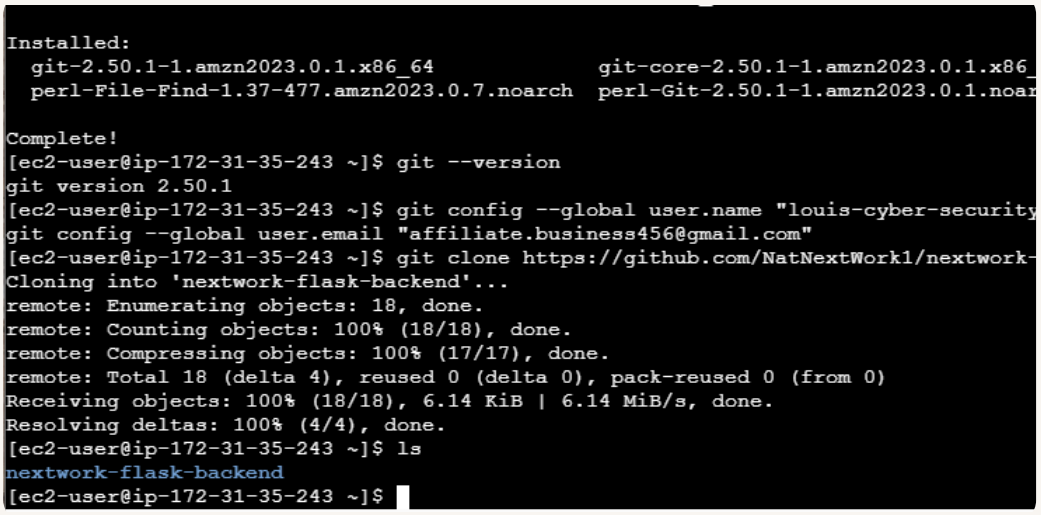
Something new I learnt from this project was how container images and registries work together with Kubernetes. I saw how Docker packages up code with dependencies, how ECR stores images for easy retrieval, and how Kubernetes pulls these images to scale apps consistently. I also gained confidence in troubleshooting permissions errors and understanding the structure of backend code.

What I'm deploying

To set up my Kubernetes cluster, I launched an Amazon Linux 2023 EC2 instance, installed eksctl, and attached an IAM role with AdministratorAccess. Then I ran eksctl create cluster to provision an EKS cluster with a t3.micro node group (3 nodes, auto-scaled). After 15 minutes the cluster was ready, and I verified nodes were active and connected for Kubernetes workloads.

I'm deploying an app's backend

Next, I retrieved the backend that I plan to deploy. An app's backend is the server-side code that processes requests and manages data. I retrieved the backend code by installing Git on my EC2 instance and cloning the project repository directly from GitHub, giving me a working copy to build and deploy to my EKS cluster.

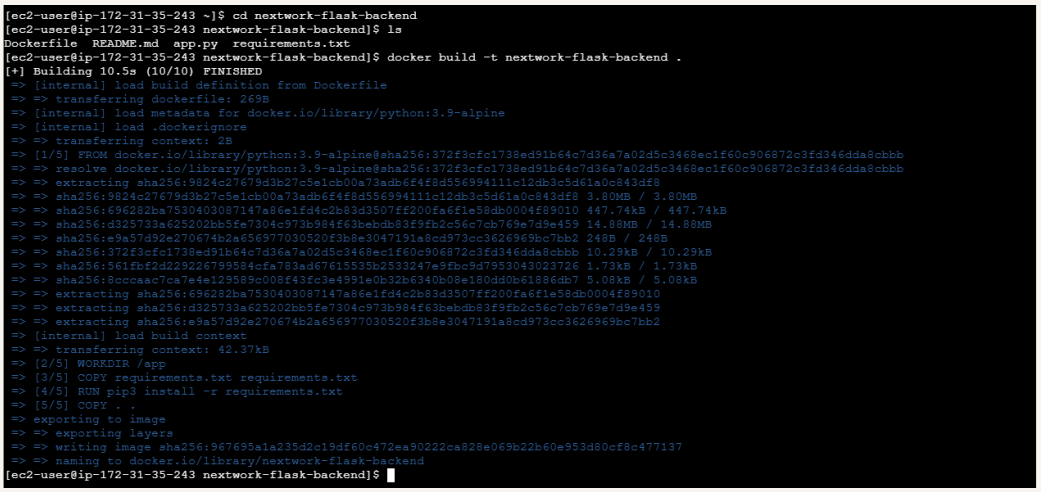


Building a container image

Once I cloned the backend code, my next step is to build a container image of the backend because Kubernetes needs an image it can pull when deploying. The image bundles the backend code and its dependencies into one portable file, ensuring my app runs the same in development, testing, and production. By tagging it as nextwork-flask-backend, I'll have a consistent reference when I push it to ECR and later deploy it on my EKS cluster.

When I tried to build a Docker image of the backend, I ran into a permissions error because the ec2-user I was logged in as didn't have access to the Docker engine. Docker was installed for the root account, so without adding ec2-user to the Docker group, my commands were blocked unless I used sudo.

To solve the permissions error, I added the ec2-user to the Docker group using sudo usermod -a -G docker ec2-user. The Docker group gives non-root users permission to run Docker commands, so this let me build and run containers without needing sudo every time. After restarting my EC2 session, the changes took effect and I could use Docker normally.

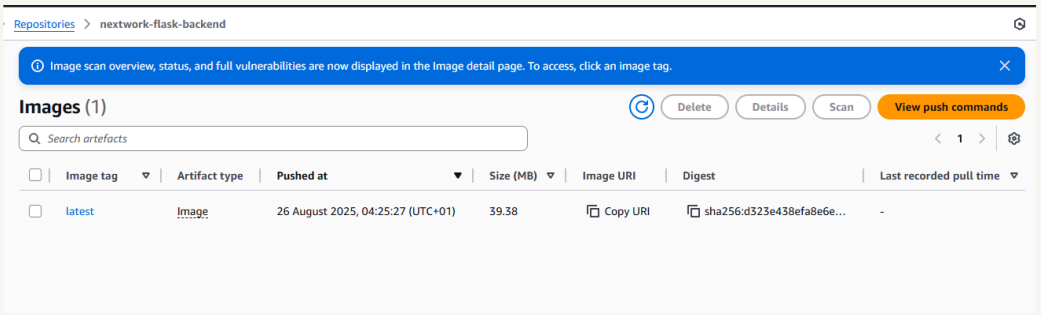

[ec2-user@ip-172-31-35-243 ~]$ cd nextwork-flask-backend
[ec2-user@ip-172-31-35-243 nextwork-flask-backend]$ ls
Dockerfile README.md app.py requirements.txt
[ec2-user@ip-172-31-35-243 nextwork-flask-backend]$ docker build -t nextwork-flask-backend .
[+] Building 10.5s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 269B
=> [internal] load metadata for docker.io/library/python:3.9-alpine
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/python:3.9-alpine@sha256:372f3cfc1738ed91b64c7d36a7a02d5c3468ec1f60c906872c3fd346dda8cbbb
=> => resolve docker.io/library/python:3.9-alpine@sha256:372f3cfc1738ed91b64c7d36a7a02d5c3468ec1f60c906872c3fd346dda8cbbb
=> => extracting sha256:9824c27679d3b27c5e1cb00a73adb6f4f8d556994111c12db3c5d61a0c843df8
=> => sha256:9824c27679d3b27c5e1cb00a73adb6f4f8d556994111c12db3c5d61a0c843df8 3.80MB / 3.80MB
=> => sha256:696282ba7530403087147a86e1fd4c2b83d3507ff200fa6f1e58db0004f89010 447.74kB / 447.74kB
=> => sha256:d325733a625202bb5fe7304c973b984f63bebdb83f9fb2c56c7cb769e7d9e459 14.88MB / 14.88MB
=> => sha256:e9a57d92e270674b2a656977030520f3b8e3047191a8cd973cc3626969bc7bb2 248B / 248B
=> => sha256:372f3cfc1738ed91b64c7d36a7a02d5c3468ec1f60c906872c3fd346dda8cbbb 10.29kB / 10.29kB
=> => sha256:561fbf2d229226799584cfa783ad67615535b2533247e9fbc9d7953043023726 1.73kB / 1.73kB
=> => sha256:8cccaac7ca7e4e129589c008f43fc3e4991e0b32b6340b08e180dd0b61886db7 5.08kB / 5.08kB
=> => extracting sha256:696282ba7530403087147a86e1fd4c2b83d3507ff200fa6f1e58db0004f89010
=> => extracting sha256:d325733a625202bb5fe7304c973b984f63bebdb83f9fb2c56c7cb769e7d9e459
=> => extracting sha256:e9a57d92e270674b2a656977030520f3b8e3047191a8cd973cc3626969bc7bb2
=> [internal] load build context
=> => transferring context: 42.37kB
=> [2/5] WORKDIR /app
=> [3/5] COPY requirements.txt requirements.txt
=> [4/5] RUN pip3 install -r requirements.txt
=> [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => writing image sha256:967695a1a235d2c19df60c472ea90222ca828e069b22b60e953d80cf8c477137
=> => naming to docker.io/library/nextwork-flask-backend
[ec2-user@ip-172-31-35-243 nextwork-flask-backend]$

Louis Moyo
NextWork Student

Container Registry

I'm using Amazon ECR in this project to store and manage my Docker container image so that Kubernetes can pull it securely during deployment. ECR is a good choice because it's tightly integrated with AWS services like EKS, provides secure image storage with encryption, supports automated vulnerability scanning, and eliminates the "works on my machine" issue by ensuring every container is launched from the same consistent image.

Container registries like Amazon ECR are great for Kubernetes deployment because they provide a secure, central place to store and manage container images. A registry ensures every node in the cluster can pull the same version of an image, keeping deployments consistent across environments. It also supports tagging and version control, vulnerability scanning, and integration with services like EKS, making it easy to scale quickly and reliably when more containers are needed.



EXTRA: Backend Explained

After reviewing the app's backend code, I've learnt that it's a lightweight Flask application designed to take in a search term, query the Hacker News Search API, and return the results as JSON. The backend relies on dependencies listed in requirements.txt (like Flask, Flask-RESTx, and Requests), builds consistently with a Dockerfile, and runs through the logic in app.py. In simple terms, the backend acts as the "brain" of the app - it takes input, fetches relevant data from an external source, and sends back clean, structured results.

Unpacking three key backend files

The requirements.txt file lists all the Python libraries and versions that the backend app needs to run. When Docker builds the image, it installs everything in this file so the app always has the right dependencies. This ensures anyone running the app has the same setup, avoiding errors from missing or mismatched packages.

The Dockerfile gives Docker instructions on how to package the backend into a container image. It defines the environment (Python 3.9 on Alpine Linux), copies in the code and requirements, installs dependencies, and sets the command to start the Flask app. Key commands include FROM (base image), COPY (bring files into the image), RUN (install dependencies), and CMD (start the app).

The app.py file contains the main backend code that makes the application work. It sets up a Flask web server, defines the routes (endpoints) for the API, and handles requests from users or other applications. In this project, app.py takes a user's input, sends it as a query to the Hacker News API, processes the results, and then formats the output as JSON data that can be returned to the client. In short, app.py is the "brain" of the backend, controlling how requests are processed and responses are generated.

The place to learn & showcase your skills

Check out nextwork.org for more projects