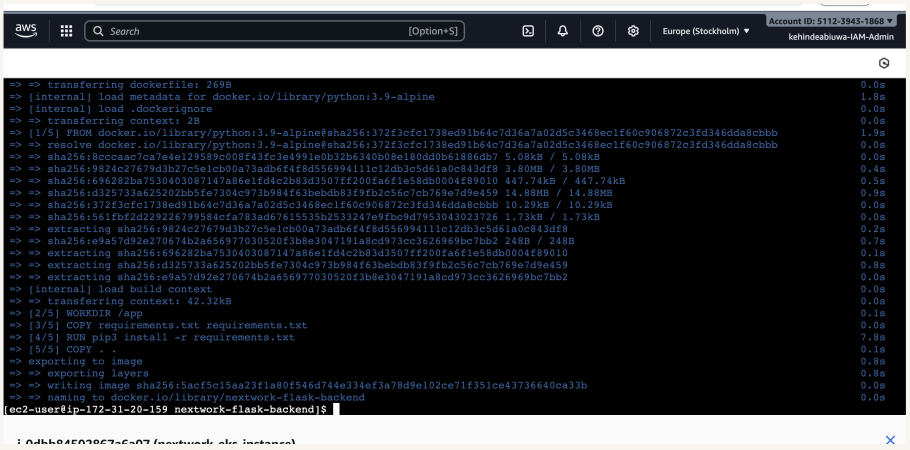
nextwork.org

Set Up Kubernetes Deployment

Kehinde Abiuwa



Kehinde Abiuwa
NextWork Student

Introducing Today's Project!

In this project, I will clone a backend from GitHub, containerize it with Docker, push the image to Amazon ECR, tackle install/config issues like a pro, and explore the backend code—because this gives me hands-on practice with modern DevOps: source control, containerization, cloud image registries, real-world troubleshooting, and understanding the code I'm deploying.

Tools and concepts

I used Amazon EKS, EC2, Git/GitHub, Docker, Amazon ECR, eksctl, and the AWS CLI to containerize a Flask backend and stage it for deployment to Kubernetes. Key steps include: Launch an Amazon Linux EC2 workstation in eu-north-1 and configure AWS credentials. Install eksctl and create the EKS cluster (nextwork-eks-cluster with a managed node group, K8s 1.33). Install Git and clone the repo: NatNextWork1/nextwork-flask-backend. Install/enable Docker and fix the permission issue by adding ec2-user to the docker group. Build the image from the provided Dockerfile and smoke-test locally. Create an Amazon ECR repo, log in, tag the image, and push it for the cluster to pull. Secret mission: review requirements.txt, Dockerfile, and app.py to understand the service (GET /contents/<topic> calling HN Algolia) and its dependencies/ports.

Project reflection

This project took me approximately 60 minutes. The most challenging part was diagnosing Docker permission issues when building the image (Docker installed for root but I was ec2-user).

I fixed it by adding myself to the docker group (sudo usermod -aG docker ec2-user and reloading the session) and double-checking the ECR login/tag syntax before pushing. My favourite part was spinning up the EKS cluster with eksctl and seeing the whole flow click—cloning the repo, building the image from the Dockerfile, and successfully pushing to ECR after reviewing the backend code.

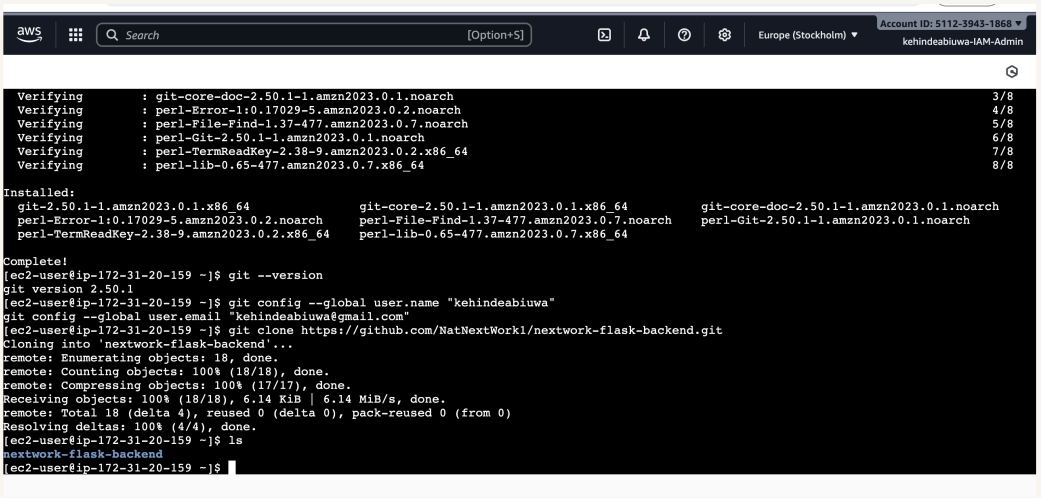
Something new that I learnt from this experience was... How Docker daemon access is controlled by the docker Unix group—and that adding ec2-user fixes “permission denied” without using sudo every time. That eksctl quietly orchestrates CloudFormation, VPC subnets across AZs, nodegroups, and add-ons—so one command can stand up a full EKS control plane. Why Dockerfiles copy requirements.txt first: it enables layer caching and faster rebuilds. The ECR push flow end-to-end (create repo → aws ecr get-login-password → docker login → tag with the repo URI → docker push) and how IAM secures it. Reading the backend helped me spot runtime needs: Flask-RESTX app binding to 0.0.0.0:8080, simple CORS header, and dependency pinning for reproducible builds.

What I'm deploying

To set up today's project, I launched a Kubernetes cluster. Steps I took to do this included: Spun up an Amazon Linux EC2 instance and connected via the connect button in the ec2 instance console Installed eksctl from the official release and verified it Created the EKS cluster with managed nodes Confirmed from the logs that eksctl Because of these steps, I now have an EKS control plane and an autoscaled t3.micro node group ready for container workloads in eu-north-1.

I'm deploying an app's backend

Next, I retrieved the backend I plan to deploy. An app's backend is the server-side code (APIs, business logic, database access) that runs on servers/containers and powers the frontend. I got the code by cloning the GitHub repo: Saw git: command not found, so I installed Git: sudo dnf update && sudo dnf install git -y Verified and set my identity: git --version → 2.50.1 git config --global user.name "kehindeabiuwa" git config --global user.email "kehindeabiuwa@gmail.com" Cloned the repo: git clone https://github.com/NatNextWork1/nextwork-flask-backend.git Confirmed the download and moved into it: cd nextwork-flask-backend && ls That gave me a local working copy of the backend source to build and deploy.

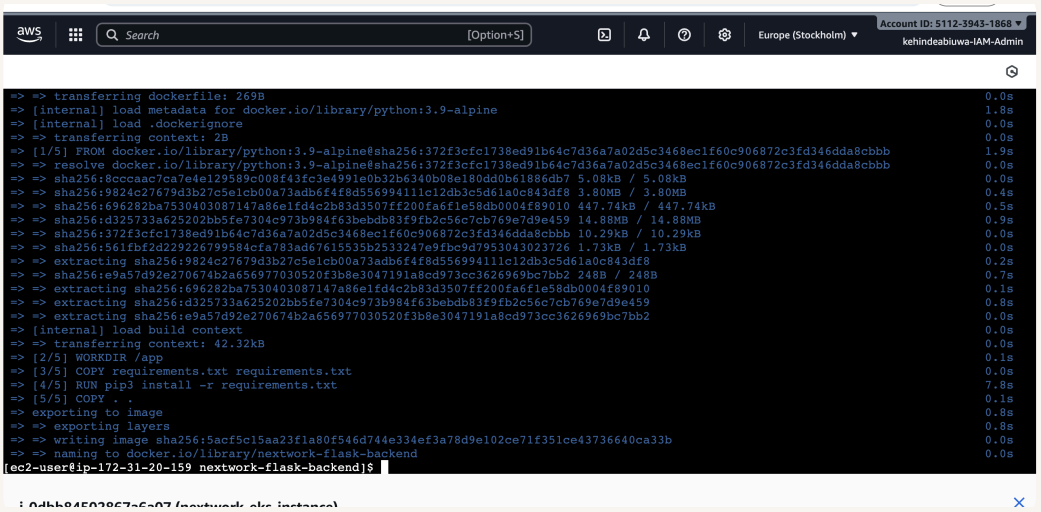

aws
Search
[Option+S]
Europe (Stockholm)
Account ID: 5112-3943-1868
kehindeabiuwa-IAM-Admin
Verifying : git-core-doc-2.50.1-1.amzn2023.0.1.noarch 3/8
Verifying : perl-Error-1:0.17029-5.amzn2023.0.2.noarch 4/8
Verifying : perl-File-Find-1.37-477.amzn2023.0.7.noarch 5/8
Verifying : perl-Git-2.50.1-1.amzn2023.0.1.noarch 6/8
Verifying : perl-TermReadKey-2.38-9.amzn2023.0.2.x86_64 7/8
Verifying : perl-lib-0.65-477.amzn2023.0.7.x86_64 8/8
Installed:
git-2.50.1-1.amzn2023.0.1.x86_64 git-core-2.50.1-1.amzn2023.0.1.x86_64 git-core-doc-2.50.1-1.amzn2023.0.1.noarch
perl-Error-1:0.17029-5.amzn2023.0.2.noarch perl-File-Find-1.37-477.amzn2023.0.7.noarch perl-Git-2.50.1-1.amzn2023.0.1.noarch
perl-TermReadKey-2.38-9.amzn2023.0.2.x86_64 perl-lib-0.65-477.amzn2023.0.7.x86_64
Complete!
[ec2-user@ip-172-31-20-159 ~]$ git --version
git version 2.50.1
[ec2-user@ip-172-31-20-159 ~]$ git config --global user.name "kehindeabiuwa"
git config --global user.email "kehindeabiuwa@gmail.com"
[ec2-user@ip-172-31-20-159 ~]$ git clone https://github.com/NatNextWork1/nextwork-flask-backend.git
Cloning into 'nextwork-flask-backend'...
remote: Enumerating objects: 18, done.
remote: Counting objects: 100% (18/18), done.
remote: Compressing objects: 100% (17/17), done.
Receiving objects: 100% (18/18), 6.14 KiB | 6.14 MiB/s, done.
remote: Total 18 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
Resolving deltas: 100% (4/4), done.
[ec2-user@ip-172-31-20-159 ~]$ ls
nextwork-flask-backend
[ec2-user@ip-172-31-20-159 ~]$

Building a container image

Once I cloned the backend code, my next step is to build a container image of the backend, because a Docker image bundles the app and all its dependencies into an immutable, portable artifact that runs the same everywhere. It gives me a versioned unit I can push to Amazon ECR, scan, roll back, and deploy to EKS reliably—eliminating “works on my machine” issues and making scaling and releases predictable.

When I tried to build a Docker image of the backend, I ran into a permissions error because Docker was installed/running as root, but I was logged in as ec2-user, which wasn’t in the docker group—so non-sudo Docker commands couldn’t talk to the daemon.

To solve the permissions error, I added my login user to the Docker group. The Docker group is a group in Linux systems that gives users the permission to run Docker commands. By default, only the root user can run Docker commands. When you add a user (e.g., ec2-user) to the Docker group, it lets that user run Docker commands without typing sudo every time.

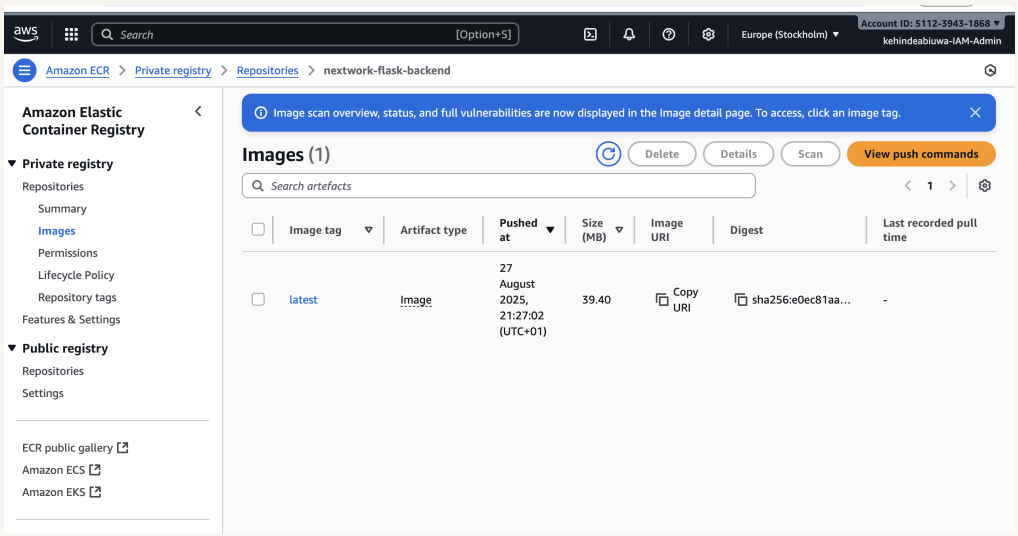

aws
Search
[Option+S]
Europe (Stockholm)
Account ID: 5112-3943-1868
kehindeabiuwa-IAM-Admin
=> => transferring dockerfile: 269B 0.0s
=> [internal] load metadata for docker.io/library/python:3.9-alpine 1.8s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [1/5] FROM docker.io/library/python:3.9-alpine@sha256:372f3cfc1738ed91b64c7d36a7a02d5c3468ec1f60c906872c3fd346dda8cbbb 1.9s
=> => resolve docker.io/library/python:3.9-alpine@sha256:372f3cfc1738ed91b64c7d36a7a02d5c3468ec1f60c906872c3fd346dda8cbbb 0.0s
=> => sha256:8cccaac7ca7e4e129589c008f43fc3e4991e0b32b6340b08e180dd0b61886db7 5.08kB / 5.08kB 0.0s
=> => sha256:9824c27679d3b27c5e1cb00a73adb6f4f8d556994111c12db3c5d61a0c843df8 3.80MB / 3.80MB 0.4s
=> => sha256:696282ba7530403087147a86e1fd4c2b83d3507ff200fa6f1e58db0004f89010 447.74kB / 447.74kB 0.5s
=> => sha256:d325733a625202bb5fe7304c973b984f63bebdb83f9fb2c56c7cb769e7d9e459 14.88MB / 14.88MB 0.9s
=> => sha256:372f3cfc1738ed91b64c7d36a7a02d5c3468ec1f60c906872c3fd346dda8cbbb 10.29kB / 10.29kB 0.0s
=> => sha256:561fbf2d229226799584cfa783ad67615535b2533247e9fbc9d7953043023726 1.73kB / 1.73kB 0.0s
=> => extracting sha256:9824c27679d3b27c5e1cb00a73adb6f4f8d556994111c12db3c5d61a0c843df8 0.2s
=> => sha256:e9a57d92e270674b2a656977030520f3b8e3047191a8cd973cc3626969bc7bb2 248B / 248B 0.7s
=> => extracting sha256:696282ba7530403087147a86e1fd4c2b83d3507ff200fa6f1e58db0004f89010 0.1s
=> => extracting sha256:d325733a625202bb5fe7304c973b984f63bebdb83f9fb2c56c7cb769e7d9e459 0.8s
=> => extracting sha256:e9a57d92e270674b2a656977030520f3b8e3047191a8cd973cc3626969bc7bb2 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 42.32kB 0.0s
=> [2/5] WORKDIR /app 0.1s
=> [3/5] COPY requirements.txt requirements.txt 0.0s
=> [4/5] RUN pip3 install -r requirements.txt 7.8s
=> [5/5] COPY . . 0.1s
=> exporting to image 0.8s
=> => exporting layers 0.8s
=> => writing image sha256:5acf5c15aa23f1a80f546d744e334ef3a78d9e102ce71f351ce43736640ca33b 0.0s
=> => naming to docker.io/library/nextwork-flask-backend 0.0s
[ec2-user@ip-172-31-20-159 nextwork-flask-backend]$

Kehinde Abiuwa
NextWork Student

Container Registry

I'm using Amazon ECR in this project to store and version the Docker image of my backend so EKS can pull it during deployment. ECR is a good choice for the job because it's AWS-native, private and regional, integrates seamlessly with EKS/EC2/CI, supports encryption, image scanning and lifecycle cleanup, and provides fast, no-egress pulls inside the same region.

Container registries like Amazon ECR are great for Kubernetes deployment because they give you a secure, centralized place to store immutable image versions (tags), enforce IAM access control, run vulnerability scanning/signing, and provide fast, regional pulls. They integrate cleanly with CI/CD and let Kubernetes deploy by simply referencing an image URI—making releases reproducible, rollbacks trivial, and scaling effortless.



EXTRA: Backend Explained

After reviewing the app's backend code, I've learnt that it's a tiny Flask REST service that exposes a single endpoint, GET /contents/<topic>, which calls the Hacker News Algolia API, filters results with both title and url, and returns them as JSON under outcome with permissive CORS. It runs on 0.0.0.0:8080 (debug mode), is stateless (no DB), and depends on Flask, Flask-RESTX, Requests, Werkzeug. A slim python:3.9-alpine Dockerfile installs requirements.txt, copies the code, and starts app.py, making it straightforward to containerize and deploy.

Unpacking three key backend files

The requirements.txt file lists the backend's Python dependencies and exact versions (e.g., Flask 2.1.3, Flask-RESTX 0.5.1, Requests 2.28.1, Werkzeug 2.1.2) so builds can run pip install -r requirements.txt to install them. This makes the environment reproducible across machines/containers and prevents "works on my machine" version mismatches.

The Dockerfile gives Docker instructions on how to build a runnable image for the Flask backend—set the base OS/Python, add dependencies and code in layers, and define how the container starts. Key commands in this Dockerfile include: FROM python:3.9-alpine — start from a lightweight Python 3.9 image. LABEL Author="NextWork" — add image metadata. WORKDIR /app — set the working directory inside the image. COPY requirements.txt requirements.txt — copy dependency list (enables layer caching). RUN pip3 install -r requirements.txt — install Python packages. COPY . . — copy the application source into the image. CMD ["python3","app.py"] — default command to launch the Flask app when the container runs.

The app.py file contains a small Flask REST API. It defines one endpoint, GET /contents/<topic>, using Flask-RESTX. When called, it: Queries the Hacker News Algolia API for the given topic. Filters results to items that have both a title and url. Returns a JSON payload {"outcome": [{id, title, url}, ...]} and adds a permissive CORS header (Access-Control-Allow-Origin: *). If run directly, it starts the Flask server in debug mode on 0.0.0.0:8080.

The place to learn & showcase your skills

Check out nextwork.org for more projects