



Set Up Kubernetes Deployment



Nicolás Aversa

```
[ec2-user@ip-172-31-47-42 nextwork-flask-backend]$ ls
Dockerfile  README.md  app.py  requirements.txt
[ec2-user@ip-172-31-47-42 nextwork-flask-backend]$ docker build -t nextwork-flask-backend .
[+] Building 10.6s (7/9)
=> => resolve docker.io/library/python:3.9-alpine@sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fabe
=> => sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fabe 10.29kB / 10.29kB
=> => sha256:1f0ac3206adabff32f2ade424580287f39907a5c532187acd1bde6390954a4f 1.73kB / 1.73kB
=> => sha256:ff8b7ec5b703652e918ce1752a8709bd1ded7e2135f1c8d8bf8f56c6e3196ee4 5.07kB / 5.07kB
=> => sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870 3.64MB / 3.64MB
=> => sha256:6efc7c91101bd06c8dfbc2c0cf770afd96b5168c833323a8c93c10545537a5be 458.62kB / 458.62kB
=> => sha256:a9948e85d045a6a02a32794cfac4ba5d9f271f65aed76114458b1bb124fd4ea3 14.87MB / 14.87MB
=> => extracting sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870
=> => sha256:e323708b3d96d783ff338254085ca725fb5d3c1a8603100dc474063ca4aaf192 248B / 248B
=> => extracting sha256:6efc7c91101bd06c8dfbc2c0cf770afd96b5168c833323a8c93c10545537a5be
=> => extracting sha256:a9948e85d045a6a02a32794cfac4ba5d9f271f65aed76114458b1bb124fd4ea3
=> => extracting sha256:e323708b3d96d783ff338254085ca725fb5d3c1a8603100dc474063ca4aaf192
=> [internal] load build context
=> => transferring context: 42.40kB
=> [2/5] WORKDIR /app
=> [3/5] COPY requirements.txt requirements.txt
=> [4/5] RUN pip3 install -r requirements.txt
```



Introducing Today's Project!

In this project, I will take a backend application from GitHub and prepare it for Kubernetes deployment because I need to package it properly before my cluster can run it. This means I'll clone the repository, build it into a Docker container image, and push that image to an Amazon ECR repository.

Tools and concepts

I used Amazon EKS, Git, Amazon EC2, Docker, and Amazon ECR to deploy a backend app on Kubernetes. Key steps include: 1. Setting up EKS to host the Kubernetes cluster. 2. Cloning the GitHub repo with the Flask backend code. 3. Building a Docker image of the app and pushing it to Amazon ECR.

Project reflection

This project took me approximately 90 minutes. The most challenging part was troubleshooting all the EC2 terminal errors. My favourite part was when I viewed my container image deployed in ECR.

Something new that I learned from this experience was how seamlessly Docker and Kubernetes work together to deploy applications. They act as a great duo.



What I'm deploying

To set up today's project, I launched a Kubernetes cluster. Steps I took to do this included: 1. Launching an EC2 instance. 2. Connecting to my instance using EC2 Instance Connect. 3. Run a command in my EC2 instance's terminal to download, extract, and install eksctl from a GitHub repo. 4. Attaching an IAM role 'nextwork-eks-instance-role' to my EC2 instance, which has AdministratorAccess. 5. Run a command to create my EKS cluster; specifying name, nodegroup name, node type, number of nodes (desired, min and max).

I'm deploying an app's backend

Next, I retrieved the backend that I plan to deploy. An app's backend means the part where the app processes user requests and stores and retrieves data. Unlike frontend code, which is what users see and interact with, backend code works on the server side (i.e. in the background) to make sure the app behaves as expected (e.g. loads a new page) when a user does things like clicking on buttons. I retrieved backend code by cloning the nextwork-flask-backend repository from GitHub.



```
[ec2-user@ip-172-31-47-42 ~]$ 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.
remote: Total 18 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (18/18), 6.14 KiB | 6.14 MiB/s, done.
Resolving deltas: 100% (4/4), done.
[ec2-user@ip-172-31-47-42 ~]$ ls
nextwork-flask-backend
```

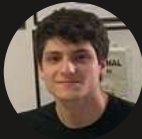


Building a container image

Once I cloned the backend code, my next step is to build a container image of the backend. This is because Kubernetes needs to pull the app from a container image that it can access. This container image was built using the data in the Dockerfile. Then, the container image lets Kubernetes set up multiple, identical containers so the application runs consistently across different environments.

When I tried to build a Docker image of the backend, I ran into a permissions error because `ec2-user`, which is the user I've used to access this instance, doesn't have permission to run Docker commands. When Docker was installed, it was set up for the root user, while `ec2-user` is kind of like logging into an AWS account as an IAM admin user—you've got a lot of control but not the absolute top level unless you're using `sudo` to run the commands.

To solve the permissions error, I added `ec2-user` to the Docker group. The Docker group is a group in Linux systems that gives users the permission to 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.



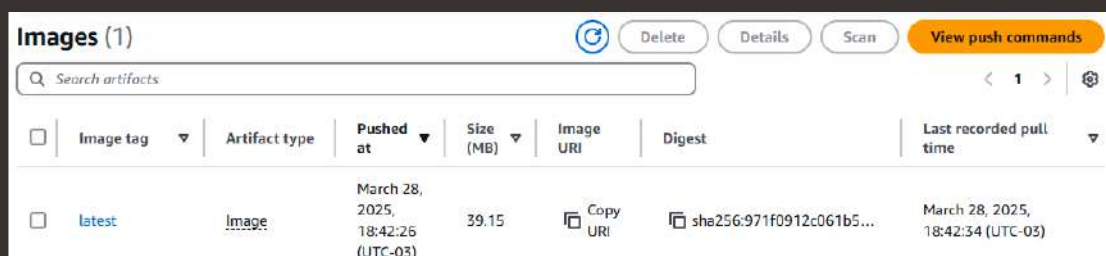
```
[ec2-user@ip-172-31-47-42 nextwork-flask-backend]$ ls
Dockerfile  README.md  app.py  requirements.txt
[ec2-user@ip-172-31-47-42 nextwork-flask-backend]$ docker build -t nextwork-flask-backend .
[+] Building 10.6s (7/9)
=> => resolve docker.io/library/python:3.9-alpine@sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fab
=> => sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fab 10.29kB / 10.29kB
=> => sha256:1f0ac3206adabff32f2ade424580287f39907a5c532187acd1bde6390954a4f 1.73kB / 1.73kB
=> => sha256:ff8b7ec5b703652e918ce1752a8709bd1ded7e2135f1c8d8bf8f56c6e3196ee4 5.07kB / 5.07kB
=> => sha256:f18232174bc91741fd3da96d85011092101a032a93a388b79e99e69c2d5c870 3.64MB / 3.64MB
=> => sha256:6efc7c91101bd06c8dfbc2c0cf770afd96b5168c833323a8c93c10545537a5be 458.62kB / 458.62kB
=> => sha256:a9948e85d045a6a02a32794cfac4ba5d9f271f65aed76114458b1bb124fd4ea3 14.87MB / 14.87MB
=> => extracting sha256:f18232174bc91741fd3da96d85011092101a032a93a388b79e99e69c2d5c870
=> => sha256:e323708b3d96d783ff338254085ca725fb5d3c1a8603100dc474063ca4aaf192 248B / 248B
=> => extracting sha256:6efc7c91101bd06c8dfbc2c0cf770afd96b5168c833323a8c93c10545537a5be
=> => extracting sha256:a9948e85d045a6a02a32794cfac4ba5d9f271f65aed76114458b1bb124fd4ea3
=> => extracting sha256:e323708b3d96d783ff338254085ca725fb5d3c1a8603100dc474063ca4aaf192
=> [internal] load build context
=> => transferring context: 42.40kB
=> [2/5] WORKDIR /app
=> [3/5] COPY requirements.txt requirements.txt
=> [4/5] RUN pip3 install -r requirements.txt
```



Container Registry

I'm using Amazon ECR in this project to securely store, share, and deploy my container image. ECR is a good choice for the job because it's an AWS service, which lets EKS deploy my container image with minimal authentication setup.

Container registries like Amazon ECR are great for Kubernetes deployment because your cluster can pull whatever is the latest image in your repository on demand, which makes deployments stay consistent across all nodes automatically. If you didn't use a container registry, you'd need to preload every node in your Kubernetes cluster with the image. You'd also need to update each node manually with every change to your container image.



The screenshot shows the Amazon ECR console interface. At the top, there's a header with 'Images (1)' and several action buttons: 'Delete', 'Details', 'Scan', and 'View push commands'. Below the header is a search bar labeled 'Search artifacts'. The main content is a table with columns: 'Image tag', 'Artifact type', 'Pushed at', 'Size (MB)', 'Image URI', 'Digest', and 'Last recorded pull time'. There is one row of data for the 'latest' tag, pushed on March 28, 2025, at 18:42:26 (UTC-03), with a size of 39.15 MB. The image URI is 'Copy URI' and the digest is 'sha256:971f0912c061b5...'. The last recorded pull time is March 28, 2025, at 18:42:34 (UTC-03).

| <input type="checkbox"/> | Image tag | Artifact type | Pushed at | Size (MB) | Image URI | Digest | Last recorded pull time |
|--------------------------|-----------|---------------|-----------------------------------|-----------|--------------------------|--------------------------|-----------------------------------|
| <input type="checkbox"/> | latest | Image | March 28, 2025, 18:42:26 (UTC-03) | 39.15 | Copy URI | sha256:971f0912c061b5... | March 28, 2025, 18:42:34 (UTC-03) |



EXTRA: Backend Explained

After reviewing the app's backend code, I've learned that it works like a helpful messenger between users and Hacker News. When someone asks for information about a specific topic, the backend quickly reaches out to Hacker News to find relevant stories. It then carefully selects just the most important details - the story titles and their web links - and packages them up in a clean, easy-to-read format. The whole system is built using straightforward tools that specialize in different tasks: Flask handles the conversations with web browsers, Requests manages the communication with Hacker News, and Docker ensures everything runs smoothly no matter where it's deployed.

Unpacking three key backend files

The requirements.txt file lists the dependencies the application needs to run properly.



The Dockerfile gives Docker instructions on how a Docker image of the backend should be built. Key commands in this Dockerfile include: 1. `WORKDIR /app`: sets `/app` as the working directory in the container, so the commands in the rest of this Dockerfile will be run from there. 2. `COPY requirements.txt: requirements.txt`: copies the `requirements.txt` file from my EC2 instance into the container's `/app` directory. 3. `RUN pip3 install -r requirements.txt`: installs the dependencies in `requirements.txt`.

The `app.py` file contains the core logic for the Flask backend. It sets up an API endpoint (`/contents/<topic>`) that fetches data from the Hacker News Search API when users request a specific topic. The code extracts relevant content (IDs, titles, URLs) from Hacker News, formats it into JSON, and sends it back as a response. Essentially, it acts as a bridge between users and external data, simplifying access to curated information.

Everyone should be in a job they love.

Check out nextwork.org for
more projects

