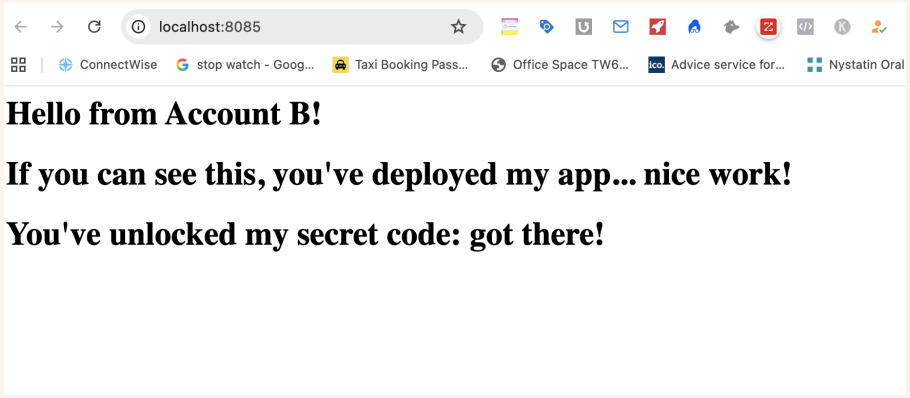
nextwork.org

Deploy an App Across Accounts

Louis Moyo



L
Louis Moyo
NextWork Student

Introducing Today's Project!

In this project, I will demonstrate how to build a Docker image in Account A, push it to Amazon ECR, and then securely share and deploy it from Account B. I'm doing this project to learn how cross-account deployments work and how teams use multiple AWS accounts for security and organisation.

What is Amazon ECR?

Amazon ECR (Elastic Container Registry) is a fully managed container image registry that makes it easy to store, manage, and deploy Docker container images. It is useful because it integrates directly with AWS services like ECS, EKS, and Elastic Beanstalk, meaning I can push my container image once and deploy it anywhere within AWS. I used ECR in this project to store the Docker image created by Account A and then allow my Elastic Beanstalk environment to pull it securely during deployment.

One thing I didn't expect...

One thing I didn't expect was how permissions and policies could block access even when I had administrator privileges. Specifically, the Elastic Beanstalk environment and EC2 instances could not pull the Docker image until I explicitly updated the ECR repository policy with the correct IAM role ARNs. This showed me how important resource-based policies are in AWS, and how caching or previous configurations can sometimes make debugging more confusing.

Louis Moyo
NextWork Student

This project took me...

This project took me several hours, mainly because I worked account A and B at the same time, as well as troubleshooting required with IAM roles and ECR repository permissions. My biggest learning was understanding the difference between IAM user permissions and resource-based policies - even if a user has admin rights, a deny in a resource policy overrides it. I also gained hands-on practice with Elastic Beanstalk, IAM roles, and ECR cross-account access.

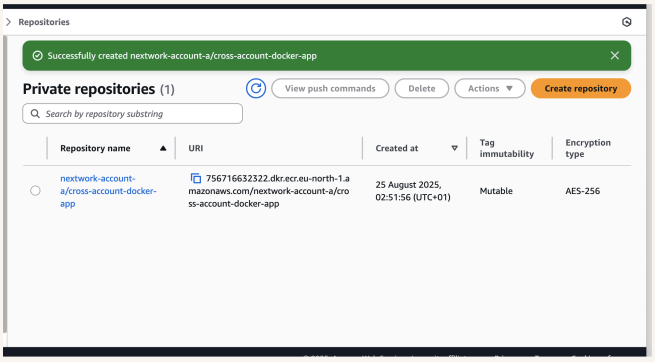
Creating a Docker Image

I set up a Dockerfile and an index.html file. Both files are needed because the Dockerfile gives Docker the instructions to build my image, and index.html is the custom web page that the container will serve.

My Dockerfile tells Docker to start from the Nginx image and copy in my custom index.html file. This way, when the container runs, it uses Nginx to serve my own web page instead of the default one.

I also set up an ECR repository

ECR stands for Amazon Elastic Container Registry. It's AWS's managed, private Docker image registry with IAM-controlled access, encryption, and optional scan-on-push-giving a secure, versioned, regional home for images and easy cross-account pulls.


Repositories
Successfully created nextwork-account-a/cross-account-docker-app
Private repositories (1)
View push commands
Delete
Actions
Create repository
Search by repository substring
Repository name
URI
Created at
Tag immutability
Encryption type
nextwork-account-a/cross-account-docker-app
756716632322.dkr.ecr.eu-north-1.amazonaws.com/nextwork-account-a/cross-account-docker-app
25 August 2025, 02:51:56 (UTC+01)
Mutable
AES-256

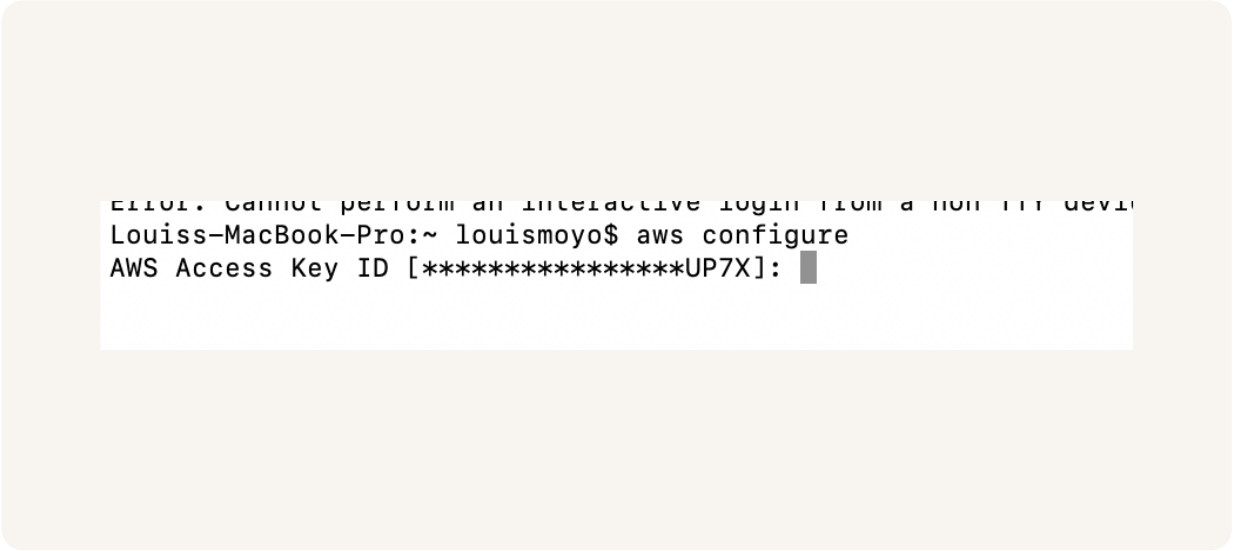
Set Up AWS CLI Access

AWS CLI can let me run ECR commands

AWS CLI is the Amazon Web Services Command Line Interface, a tool that lets me manage AWS resources directly from the terminal. The CLI asked for my credentials because it needs my access keys to authenticate and allow secure actions like creating repositories and pushing images to ECR.

To enable CLI access, I created an IAM user in Account A with ECR push/pull permissions (e.g., AmazonEC2ContainerRegistryFullAccess, least-privilege for this project). I generated an access key + secret, saved them with aws configure (not using the root user, MFA on), which means my CLI can securely authenticate and push/pull images to ECR.

To pass my credentials to the AWS CLI, I ran aws configure and entered the access key, secret key, region, and output format. This lets the CLI authenticate and target the correct account when pushing or pulling images from ECR.


Louiss-MacBook-Pro:~ louismoyo$ aws configure
AWS Access Key ID [****************UP7X]:

Louis Moyo
NextWork Student

Pushing My Image to ECR

Push commands are the Docker and AWS CLI commands that let me upload a container image from my local computer into Amazon ECR. They include authenticating Docker with ECR, tagging the image with the repository URI, and then pushing it so it's stored securely in the registry.

There are three main push commands

To authenticate Docker with my ECR repo, I used: aws ecr get-login-password --region eu-north-1 | docker login --username AWS --password-stdin 756716632322.dkr.ecr.eu-north-1.amazonaws.com This command gets a temporary password from AWS and logs Docker into my ECR registry so I can push images to it.

To push my container image, I ran: docker buildx build --platform linux/amd64,linux/arm64 -t <ECR-URI>:latest --push . This command built my image for multiple platforms and pushed it to ECR in Account A, making it available for pulling in other accounts.

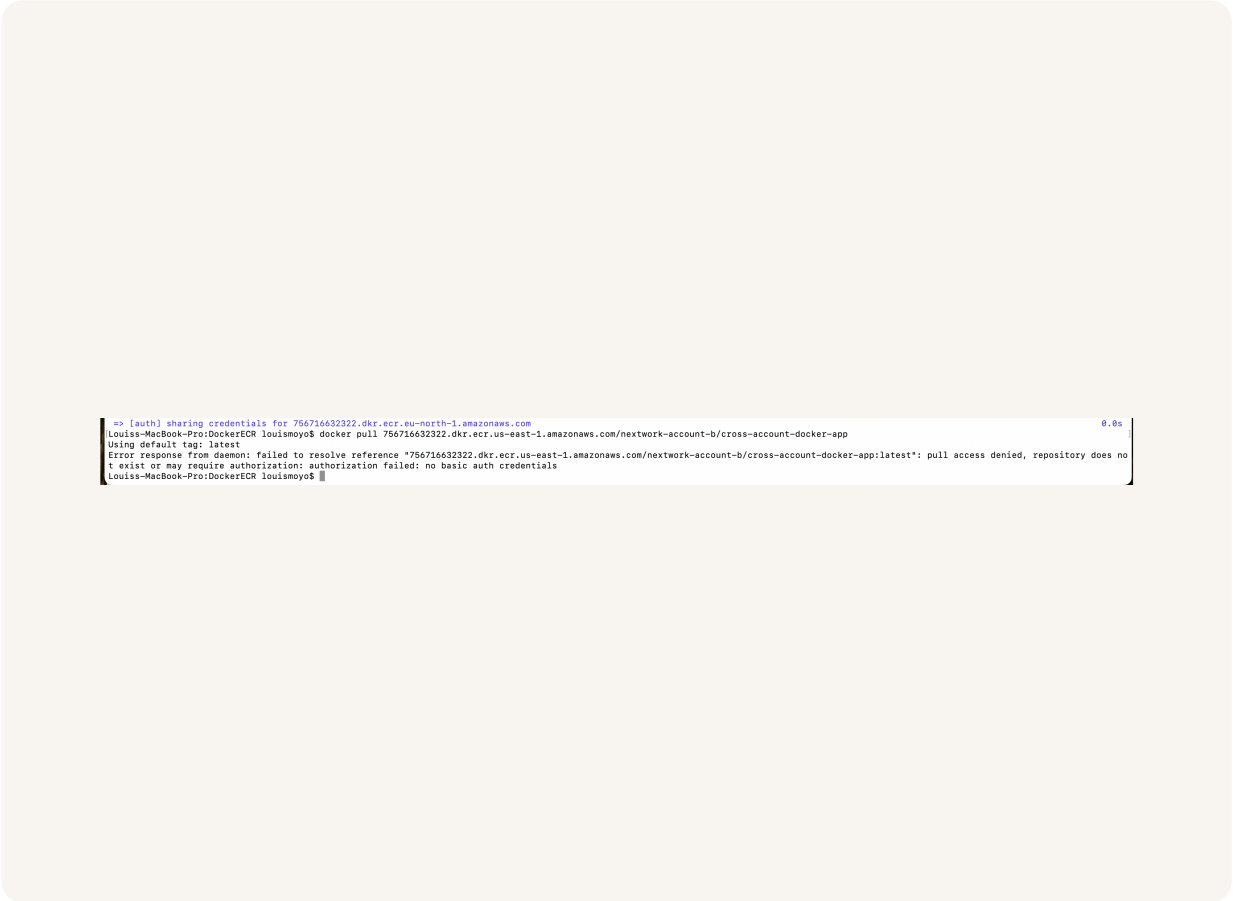
When I built my image, I tagged it with the label latest. This means it's marked as the most up-to-date version. Any deployments that pull the image with the latest tag will always get the newest build without needing a version number.

Louis Moyo
NextWork Student

Resolving Permission Issues

When I tried pulling my project buddy's container image for the first time, I saw the error because ECR repositories are private by default. Without the right permissions or authentication, Docker blocks access, which is why the pull failed.

To resolve each access issues, I updated repository permissions in ECR by adding IAM user ARN to the policy for account a and b. This allows cross-account pulls (ecr:BatchGetImage, ecr:BatchCheckLayerAvailability, and ecr:GetDownloadUrlForLayer) so both accounts can access each other's container images.

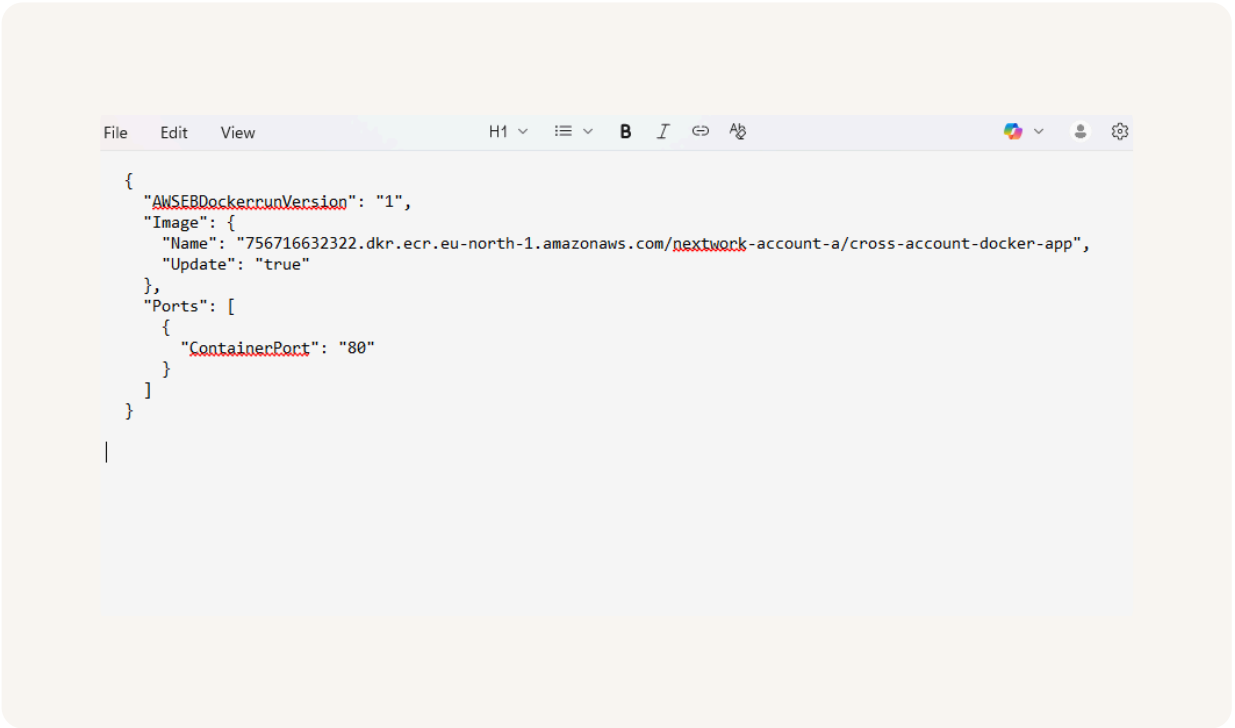


Deploying the App

I used Elastic Beanstalk to deploy my containerised application without having to manually set up and manage the underlying infrastructure. When I set up an Elastic Beanstalk application, I configured settings like the environment type (single instance), EC2 instance profile, public IP, storage size, and monitoring level. Elastic Beanstalk handled creating and managing the EC2 instance, networking, and scaling, while automatically pulling my Docker image from ECR.

While setting up for deployment, I created two new IAM roles. The first was a service role (aws-elasticbeanstalk-service-role) to give Elastic Beanstalk permission to manage AWS resources on my behalf (like EC2, load balancers, and health checks). The second was an instance profile role (ecrInstanceRole) that allows EC2 instances in the Elastic Beanstalk environment to pull Docker images from ECR. I attached the AmazonEC2ContainerRegistryReadOnly policy to both roles so that Elastic Beanstalk and the EC2 instances could access the container images stored in ECR.

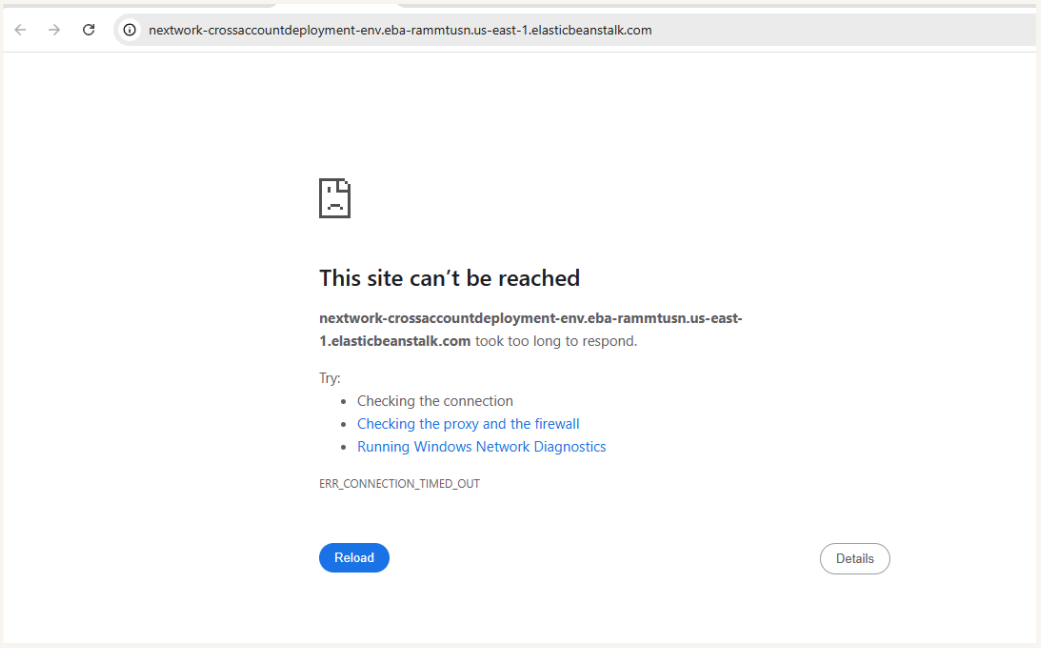
The Dockerrun.aws.json file is a configuration file that Elastic Beanstalk uses to know how to run a Docker container. My file tells Elastic Beanstalk which Docker image to pull (from Amazon ECR), what port the container listens on, and how to deploy it inside the Elastic Beanstalk environment. In short, it acts like the “instructions” for Elastic Beanstalk so it can launch Player A’s app correctly in the cloud.


File
Edit
View
H1
{
"AWSEBDockerrunVersion": "1",
"Image": {
"Name": "756716632322.dkr.ecr.eu-north-1.amazonaws.com/nextwork-account-a/cross-account-docker-app",
"Update": "true"
},
"Ports": [
{
"ContainerPort": "80"
}
]
}

Resolving Deployment Issues

When I visited my environment, the application couldn't be reached because although my IAM user had access to Player A's ECR repository, the Elastic Beanstalk environment and its EC2 instances were not using that IAM user. Instead, they run with their own roles (the Elastic Beanstalk service role and the EC2 instance profile). Since those roles were not yet granted permissions in the ECR repository policy, Elastic Beanstalk and the instances could not pull the Docker image, which caused the site to show the error.

To fix the permissions error, I updated Account A's ECR repository policy so that Account B's Elastic Beanstalk environment and EC2 instances could pull images. Specifically, I added the ARNs of Account B's ecrInstanceRole and aws-elasticbeanstalk-service-role into the ECR permissions.


nextwork-crossaccountdeployment-env.eba-rammtusn.us-east-1.elasticbeanstalk.com
This site can't be reached
nextwork-crossaccountdeployment-env.eba-rammtusn.us-east-1.elasticbeanstalk.com took too long to respond.
Try:
Checking the connection
Checking the proxy and the firewall
Running Windows Network Diagnostics
ERR_CONNECTION_TIMED_OUT
Reload
Details

The place to learn & showcase your skills

Check out nextwork.org for more projects