

# Case Study: Deploying a Containerized Web Application on AWS ECS Fargate

---

**Submitted by:**

**Group 1**

- Anju Doot (100959389)
- Muskaan Fatima (101005853)
- Hushang Fikrat Muhibullah (101012042)
- Mohammad Kaif (101001476)

**Due Date: 14th/Aug/2025**

## Table of Contents

1. Introduction .....	3
Team Responsibilities:.....	4
2. Background & Motivation: .....	4
3. Deployment Architecture .....	5
3.1 Core AWS Services Used:.....	5
3.2 High-Level Workflow:.....	5
3.3 Flow Chart: .....	6
3.4 Architecture Diagram: .....	8
4. Implementation Steps .....	9
4.1 Docker Installation and testing on Local machine.....	9
4.2 VPC & Networking Setup.....	12
4.3 ECR Repository Creation .....	17
4.4 Image Upload to ECR.....	18
4.5 ECS Cluster Creation.....	20
4.6 Task Definition Creation.....	21
4.7 Service Creation .....	23
5. Security Considerations .....	24
6. Challenges Faced .....	26
7. Outcomes & Benefits.....	26
8. Future Improvements .....	28
9. Conclusion.....	29
10 References:.....	31

# Case Study: Deploying a Containerized Web Application on AWS ECS Fargate

---

## 1. Introduction

This case study provides a detailed account of how the **Oshawa Public Library (OPL)** web application was deployed using **Amazon Elastic Container Service (ECS)** with the **AWS Fargate** launch type. The deployment follows a **microservices architecture**, meaning the application is broken down into smaller, independent services that can be developed, deployed, and scaled separately.

To manage and store container images securely, the solution uses **Amazon Elastic Container Registry (ECR)**, a private and fully managed Docker image repository. Networking is handled through a **custom Amazon VPC** (Virtual Private Cloud) designed with both **public** and **private subnets**, ensuring that sensitive workloads remain isolated while still allowing controlled external access where required.

The project's **primary objectives** were threefold:

1. **Deploy a Dockerized web application** in a **serverless container environment**—eliminating the need to manage underlying servers.
2. **Guarantee security and scalability** by using AWS-native networking, IAM-based permissions, and automatic scaling capabilities, all while staying within budget constraints.
3. **Create a repeatable deployment process** that could be seamlessly integrated into a **CI/CD pipeline** (such as GitHub Actions) to enable automated, reliable deployments in the future.

In essence, the case study captures both the **technical design** and **operational workflow** for deploying a cloud-native application that is secure, cost-efficient, and ready for future automation.

### Team Responsibilities:

- **Hushng Fikrat Muhibullah (Team Leader):** Managing and assigning tasks, Docker installation, containerization, security design and CI/CD workflow draft (with Kaif), architectural diagram and CI/CD documentation complete (with Anju), final document compilation and review (leads), presentation rehearsal and finalization (with all), submit document and presentation (oversees).
- **Kaif:** Create ECS and ECR, security design and CI/CD workflow draft (with Hushng).
- **Muskaan:** Network settings (VPC, Subnet, Internet Gateway, Security Groups), initial architecture draft (with Anju), presentation rehearsal and finalization (with all).
- **Anju:** All documentation and presentation preparation, initial architecture draft (with Muskaan), architectural diagram, flow chart and CI/CD documentation complete (with Hushng), presentation slide deck draft, final document compilation and review (with all), presentation rehearsal and finalization (with all), submit document and presentation (with all).

## 2. Background & Motivation:

The Oshawa Public Library (OPL) web application required a deployment strategy that balanced **security, scalability, and cost-effectiveness** while supporting a microservices-based architecture. To meet these needs, the technical team chose to leverage **Amazon ECS with AWS Fargate** for serverless container hosting, along with **Amazon ECR** for private image storage and a **custom VPC** design for controlled network access.

The motivation behind this deployment approach was to:

- **Streamline deployment** of a Dockerized application without managing underlying servers.
- **Improve security** by isolating workloads in private subnets and restricting public access where possible.
- **Ensure scalability** to handle varying user traffic, such as during community events or seasonal spikes in library usage.
- **Maintain budget discipline** by using a pay-as-you-go model and avoiding overprovisioned infrastructure.
- **Establish a repeatable process** that could integrate with CI/CD pipelines for faster, automated deployments in the future.

By documenting this deployment process, OPL aimed to create a **clear, reusable blueprint** that could be followed by internal teams or external partners, ensuring consistent configurations, easier troubleshooting, and better alignment with AWS best practices.

### 3. Deployment Architecture

#### 3.1 Core AWS Services Used:

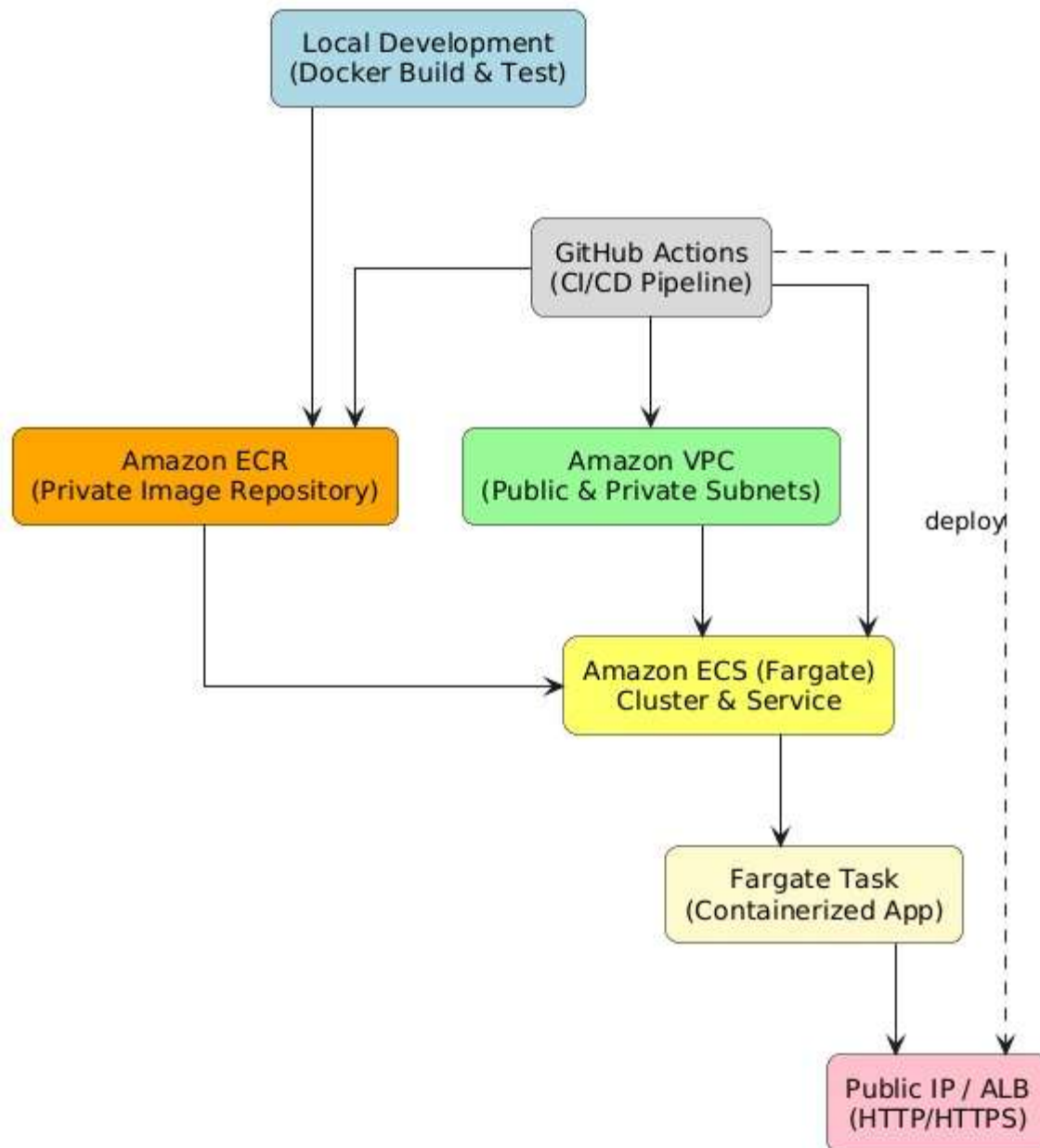
- Amazon ECS (Fargate): Serverless container hosting.
- Amazon ECR: Private Docker image repository.
- Amazon VPC: Custom networking with public/private subnets.
- AWS IAM: Role-based access control for ECS tasks and CI/CD pipelines.
- Amazon CloudWatch: Logging and monitoring.

#### 3.2 High-Level Workflow:

- Local build & test of the application using Docker.
- Push image to ECR for secure storage.
- Create VPC with two public and two private subnets.
- Deploy ECS cluster and service with a single Fargate task.
- Assign public IP for testing (future plan: ALB + HTTPS).

- (Optional) Integrate CI/CD using GitHub Actions for automated deployments.

### 3.3 Flow Chart:



#### Explanation:

This flow diagram illustrates the end-to-end process for deploying the OPL web application to **Amazon ECS with AWS Fargate**, integrating local development, image storage, networking, and deployment steps.

### **Local Development (Docker Build & Test)**

- The development team builds and tests the web application locally using Docker.
- This ensures the containerized application works as expected before pushing it to the cloud.

### **Amazon ECR (Private Image Repository)**

- Once validated, the Docker image is pushed to **Amazon Elastic Container Registry (ECR)**.
- ECR securely stores the application's container image in a private repository, ready for deployment.

### **Amazon VPC (Public & Private Subnets)**

- A **custom Virtual Private Cloud (VPC)** is created with both **public** and **private** subnets.
- Public subnets allow controlled access from the internet, while private subnets host ECS tasks for improved security.

### **GitHub Actions (CI/CD Pipeline)** *(optional/future integration)*

- A CI/CD workflow can automate the build, test, and deployment steps.
- GitHub Actions triggers deployments directly to ECS Fargate after code changes are committed.

### **Amazon ECS (Fargate) Cluster & Service**

- The application is deployed to **Amazon ECS** using the **Fargate launch type**, which removes the need to manage EC2 servers.
- ECS manages the container orchestration, scaling, and availability.

### **Fargate Task (Containerized App)**

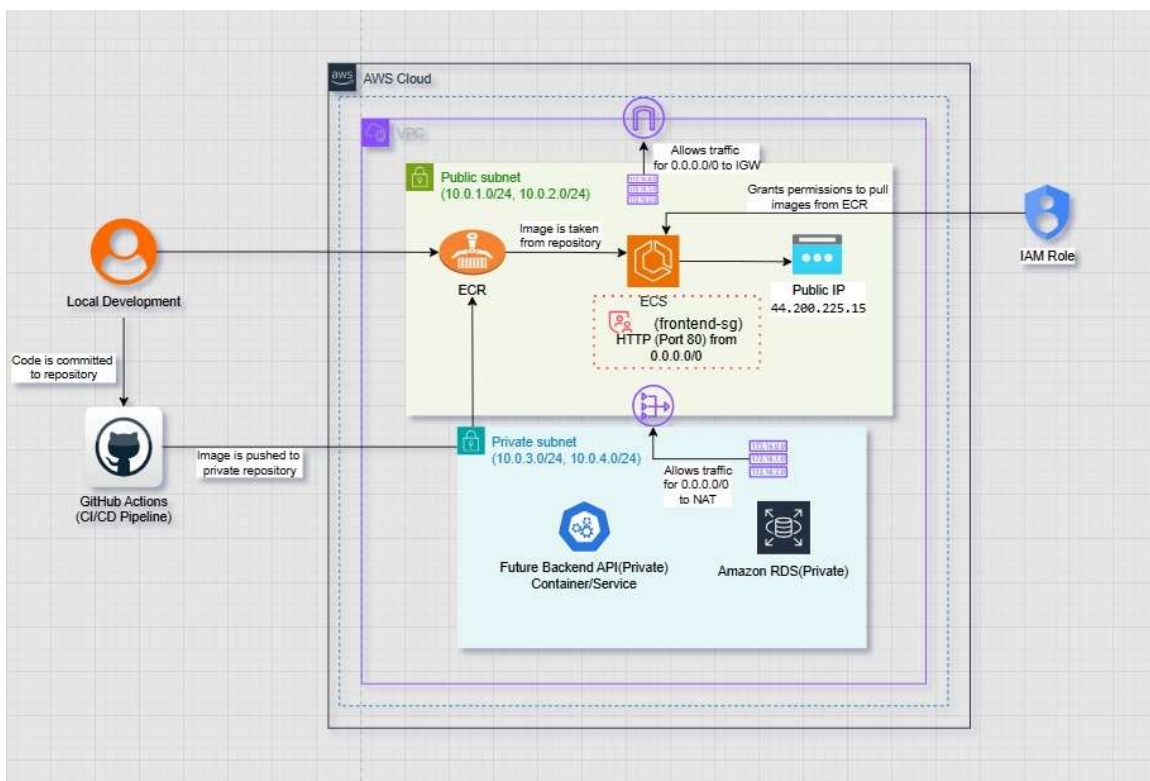
- The ECS service runs one or more **Fargate tasks**, each being an instance of the containerized web application.

### Public IP / ALB (HTTP/HTTPS)

- For testing, a **public IP** can be assigned to access the application directly.
- In production, an **Application Load Balancer (ALB)** with HTTPS is recommended for secure, scalable access.

This flow ensures a **secure, scalable, and repeatable** deployment pipeline, aligning with the case study's goals of minimizing server management, controlling network exposure, and supporting future CI/CD automation.

### 3.4 Architecture Diagram:



This architecture represents a containerized web application deployment on AWS ECS



(Fargate) integrated with a GitHub Actions CI/CD pipeline. The process begins with local development, where code is written and tested before being committed to a GitHub repository. Once committed, the GitHub Actions pipeline automatically builds a Docker image from the source code and pushes it to a private Amazon Elastic Container Registry (ECR). The application is hosted within a custom AWS Virtual Private Cloud (VPC) that contains both public and private subnets.

The public subnet (CIDR blocks 10.0.1.0/24 and 10.0.2.0/24) hosts the ECS service running the frontend container, which is publicly accessible via an Internet Gateway (IGW) and a public IP address (44.200.225.15). The security group for the frontend (frontend-sg) allows HTTP traffic on port 80 from any IP address (0.0.0.0/0). ECS pulls the container image from ECR using permissions granted through an IAM Role, ensuring secure access without embedding credentials.

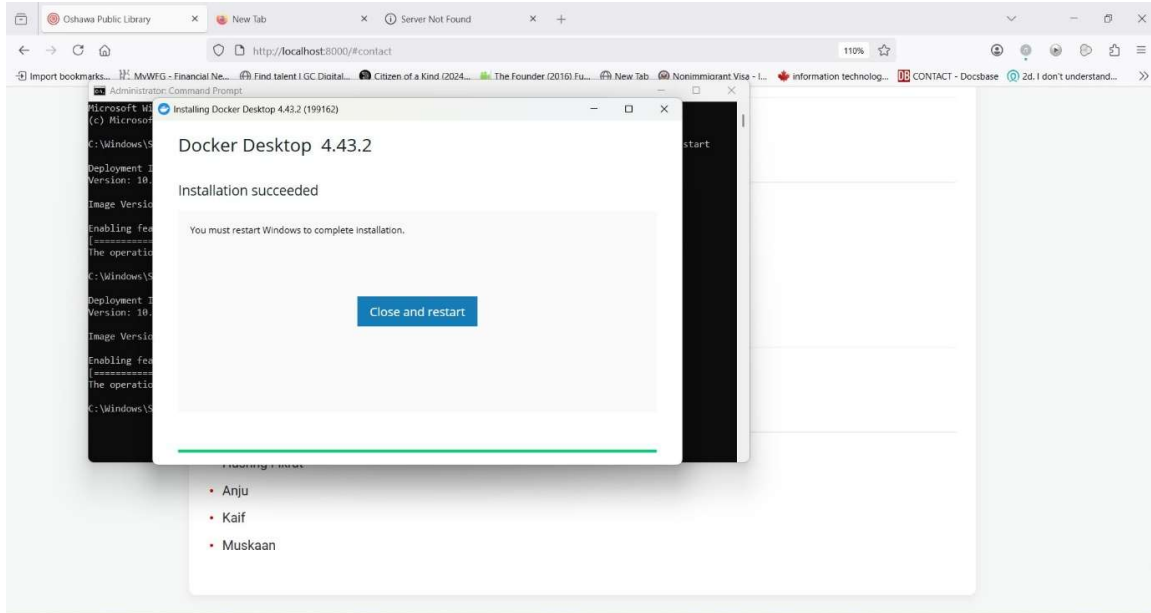
The private subnet (CIDR blocks 10.0.3.0/24 and 10.0.4.0/24) is reserved for future backend API services and an Amazon RDS database, both of which remain inaccessible from the public internet for security purposes. Outbound traffic from the private subnet is routed through a NAT Gateway, allowing access to external resources while keeping backend services private.

This design ensures a clear separation between public-facing and internal components, enabling scalability by adding more ECS services or backend systems in the private subnet. It also follows AWS security best practices by restricting public exposure, enforcing least privilege through IAM Roles, and using security groups to tightly control inbound and outbound traffic.

## 4. Implementation Steps

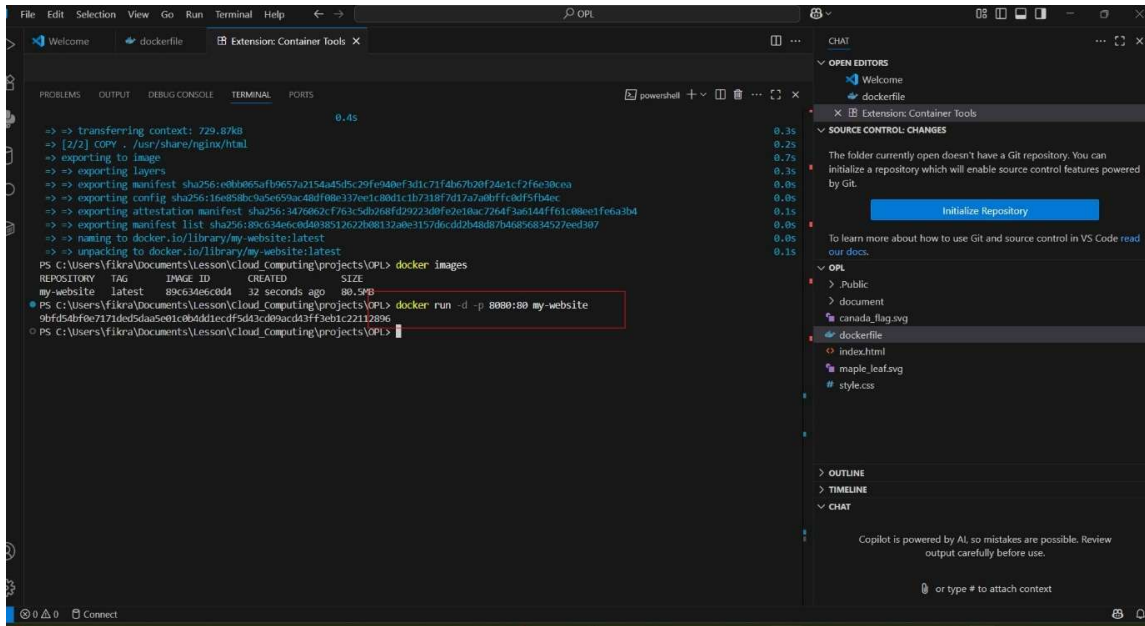
### 4.1 Docker Installation and testing on Local machine.

- Docker Desktop version 4.43.2 has been successfully installed on the system. The installation wizard indicates that a system restart is required to complete the setup. Docker Desktop is essential for building, running, and managing containers locally on Windows. This step is foundational for developers working with containerized applications, as it provides a GUI and CLI for Docker operations



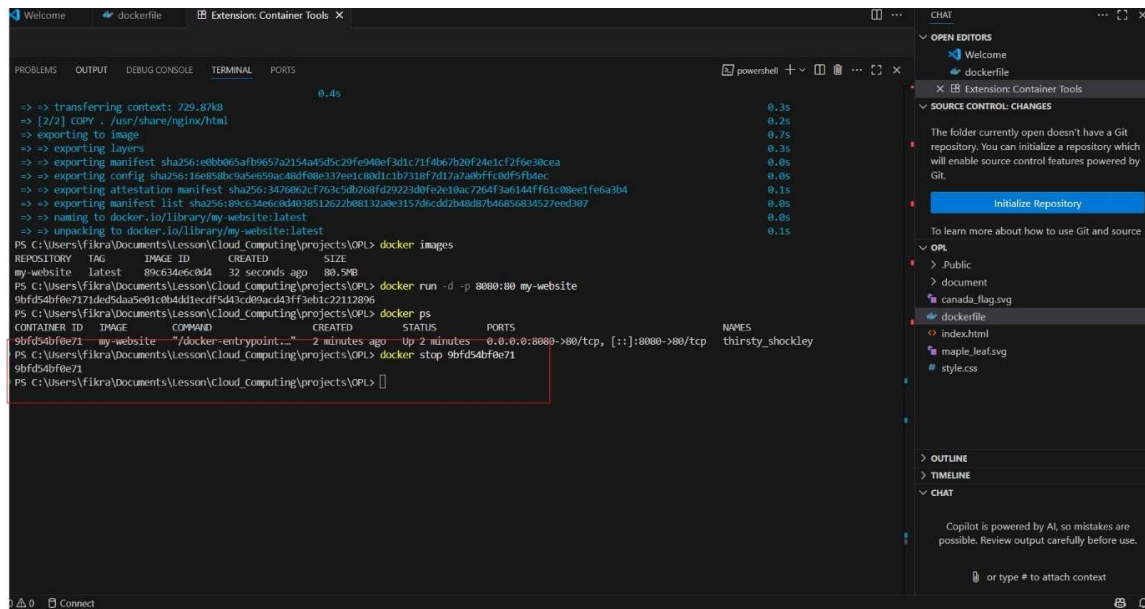
### *Docker Desktop Installation*

- IT shows the process of building and running a Docker image for a static website. The image was successfully named my-website and runs it using the command `docker run -d -p 8080:80 my-website`. This command starts the container in detached mode and maps port 8080 on the host to port 80 in the container, making the website accessible at `http://localhost:8080`. The project includes files such as `index.html`, `style.css`, and SVG images, which are likely served through an NGINX base image defined in the Dockerfile.



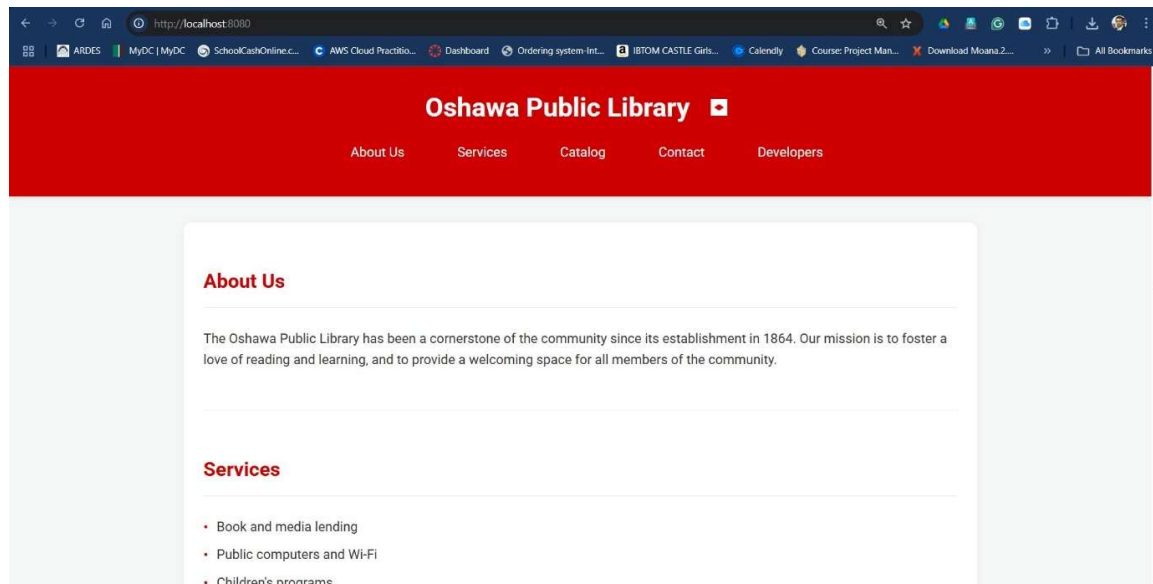
Making the website accessible at <http://localhost:8080>

- Here the docker is running and can be stopped.



Local Docker Container Running and Stopped

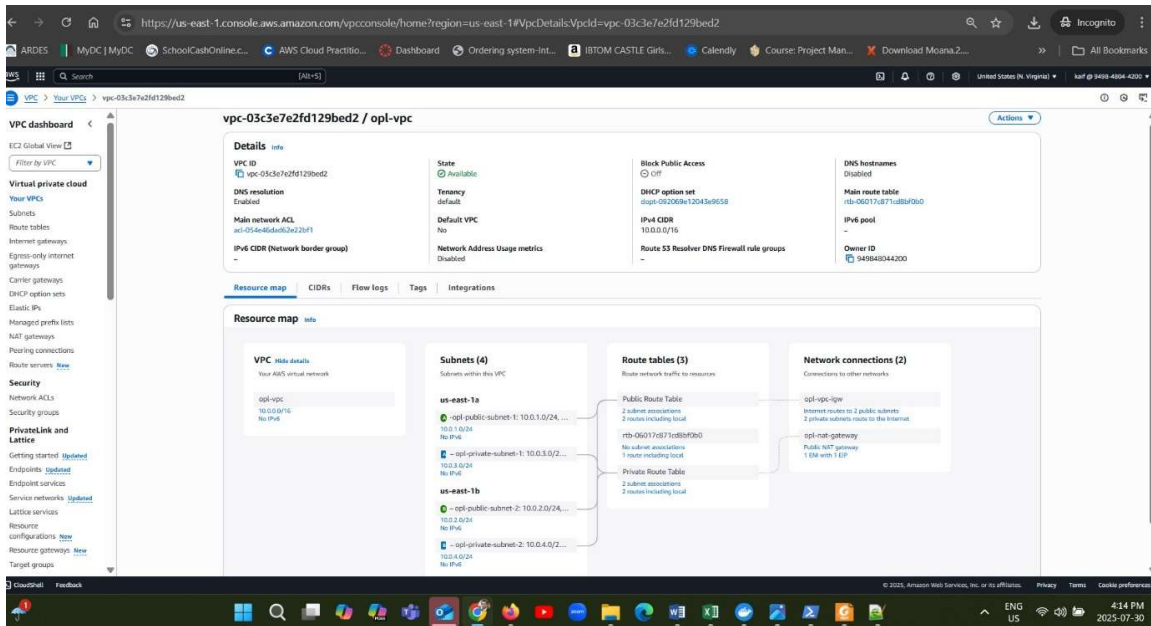
- This step verifies that the Docker image works locally before pushing it to the cloud.



*Accessible at <http://localhost:8080>*

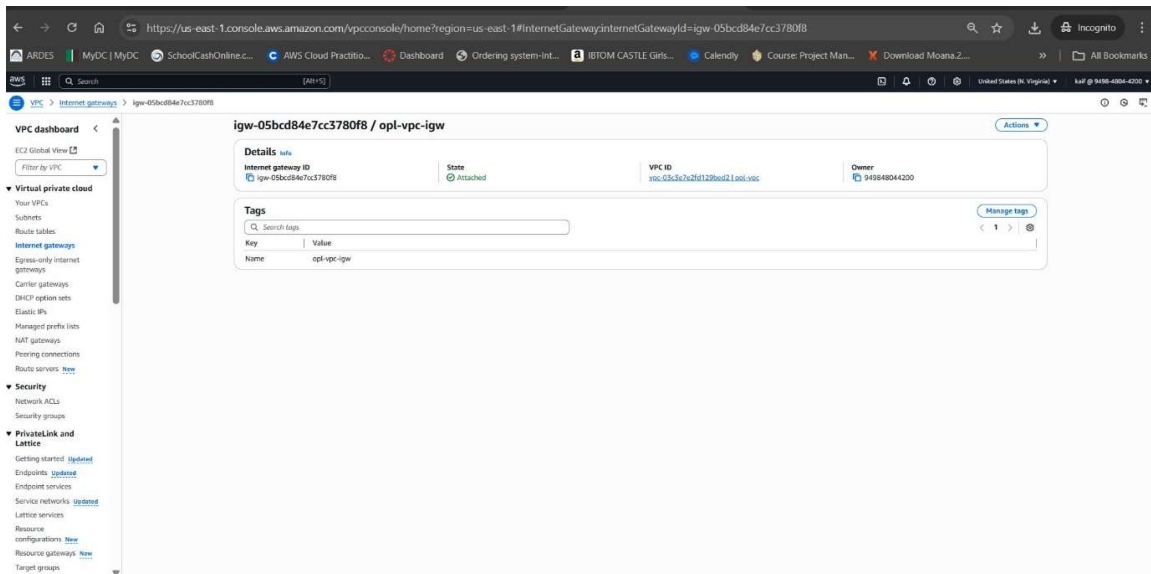
#### 4.2 VPC & Networking Setup

- This shows the VPC (opl-vpc) layout and resources. It contains 4 subnets (2 public and 2 private), along with their corresponding route tables. The VPC has two network connections: one Internet Gateway for public access and a NAT Gateway for enabling outbound traffic from private subnets. The diagram clearly maps how the subnets are associated with the route tables and how the traffic flows within the virtual network.



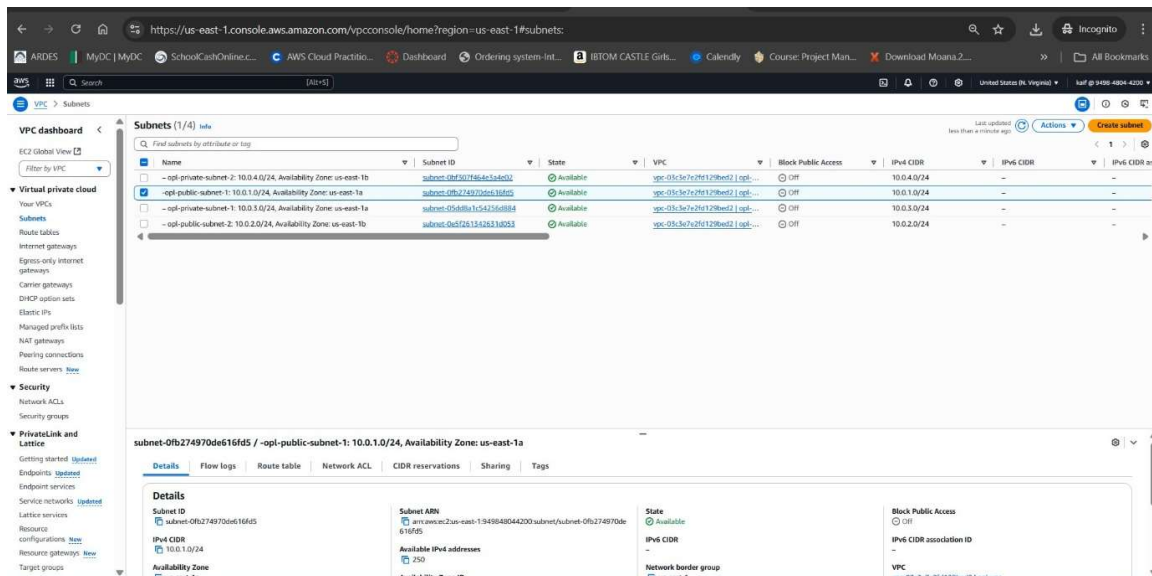
*VPC Resource Map*

- Internet Gateway (opl-vpc-igw) is attached to the user's Virtual Private Cloud (VPC). This component allows communication between instances inside the VPC and the internet. The internet gateway is essential for enabling inbound and outbound access to ECS tasks, especially when combined with Elastic IPs, NAT, and route table configurations for public subnets.



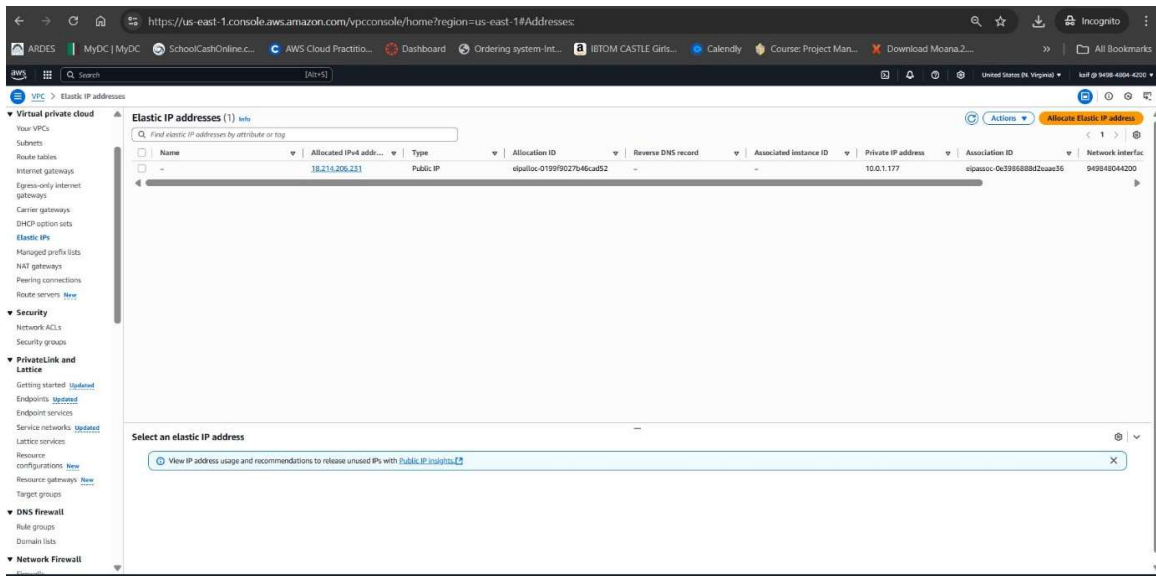
*Attached Internet Gateway*

- Two public and two private—spread across two Availability Zones (us-east-1a and us-east-1b). Each subnet has a CIDR block assigned, and public access is allowed (Block Public Access is off). This multi-AZ configuration improves high availability and fault tolerance. The selected subnet in the image is a public one (10.0.1.0/24)



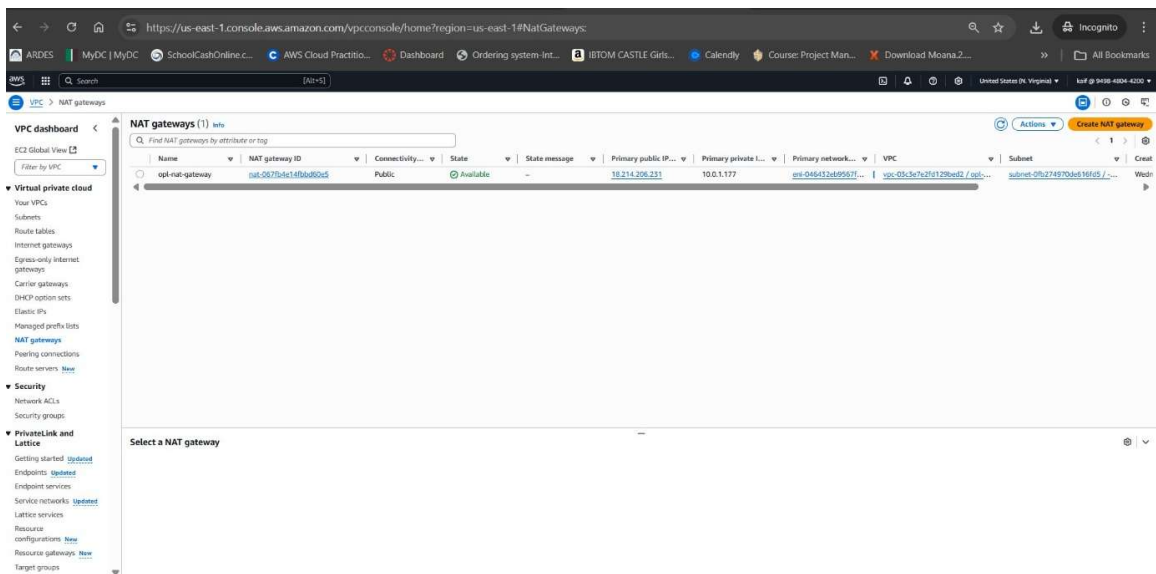
### Subnet Configuration

- Allocating Elastic IP address 18.214.206.231 that is associated with a private IP (10.0.1.177) within the user's VPC. Elastic IPs are static public IPs in AWS that remain consistent even after stopping or restarting services. This one is currently attached and associated with the user's AWS account and likely used to provide public access to services deployed through ECS or EC2 within the configured infrastructure.



### *Elastic IP*

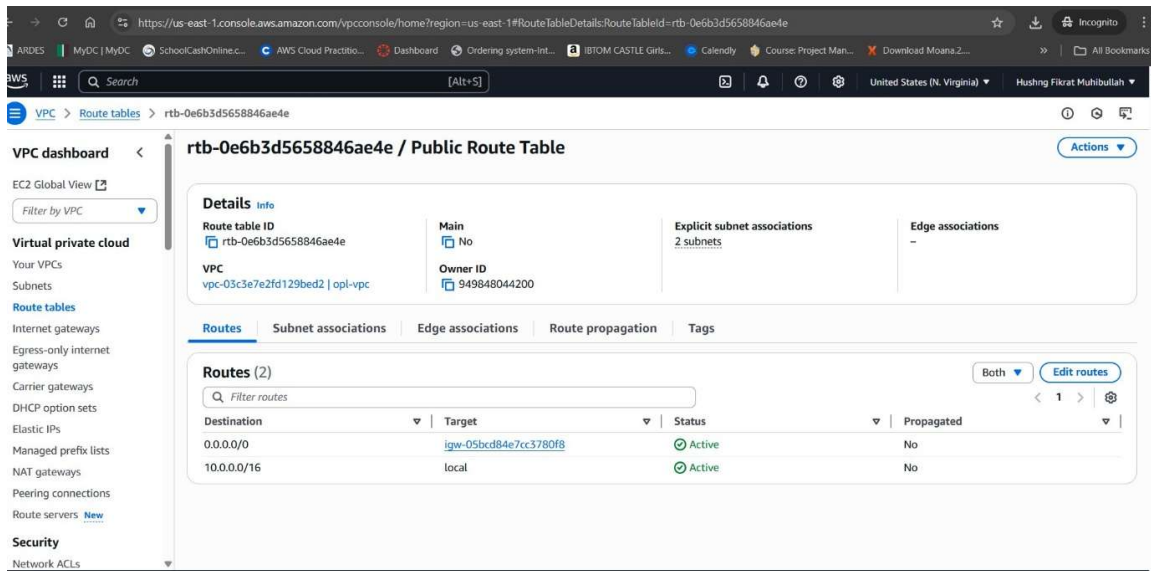
- NAT gateway setup. The gateway allows instances in a private subnet to access the internet (like pulling Docker images or installing updates) while keeping them inaccessible from outside.



### *NAT Gateway Configuration*

- Public route table with two active routes. The default route (0.0.0.0/0) directs traffic to an Internet Gateway, enabling resources in associated subnets to access the internet

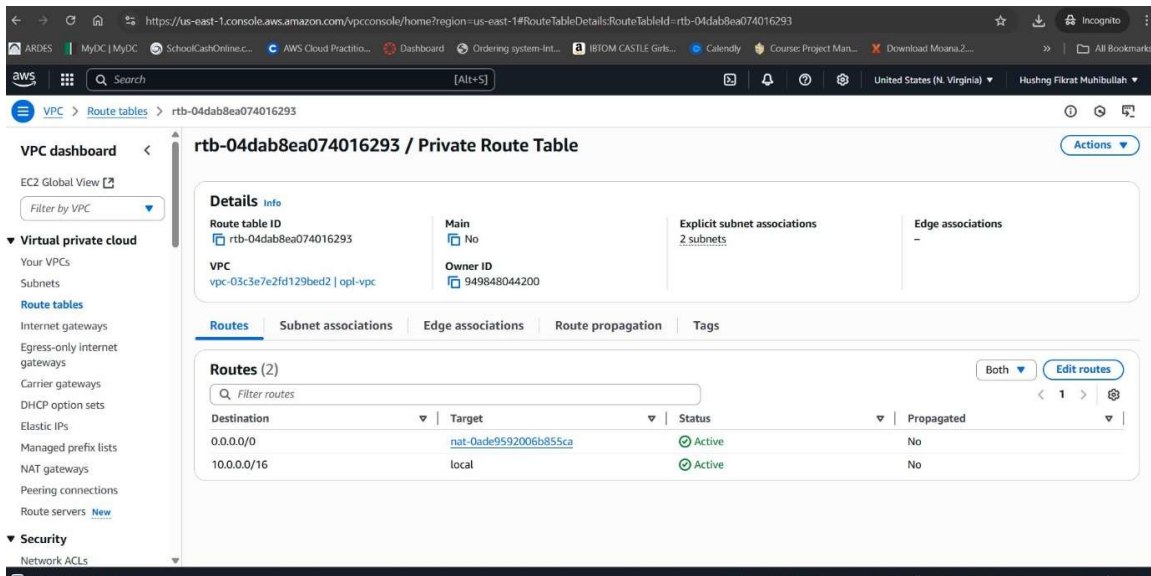
directly. The other route for 10.0.0.0/16 allows internal VPC communication. The table is associated with two subnets, making them public. This setup is for the frontend services like our web server that needs to be accessible from outside the VPC.



### *Public Route Table Configuration*

The private route table is shown with two routes configured. The route 0.0.0.0/0 points to a NAT gateway, allowing instances in private subnets to access the internet securely without exposing them to inbound traffic. The second route 10.0.0.0/16 allows local traffic within the VPC. This route table is linked to two subnets, making them effectively private.

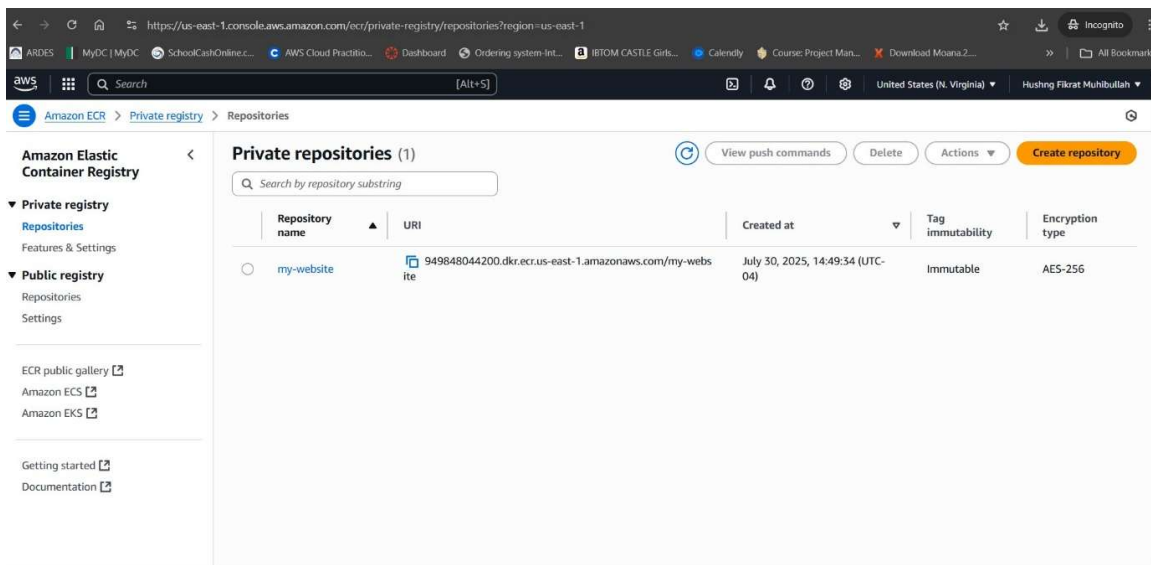




### Private Route Table Configuration

#### 4.3 ECR Repository Creation

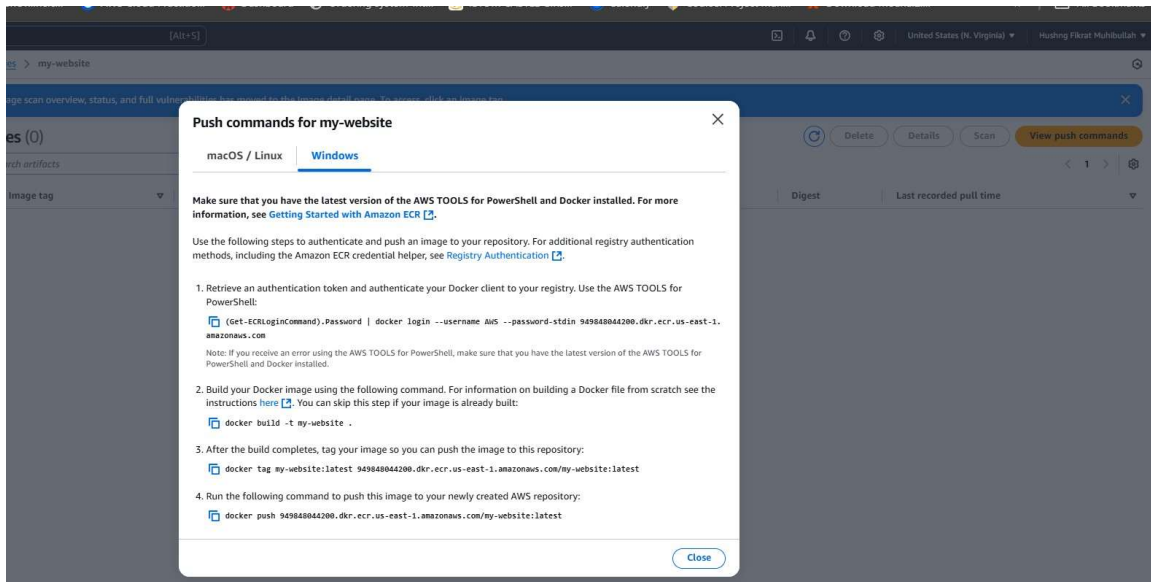
- Private ECR repository named my-website, created under the specified AWS account. The repository URI is clearly displayed and follows the AWS ECR format. The repository is configured to be immutable (preventing image overwrites) and uses AES-256 encryption, ensuring both operational integrity and data security. This repository serves as the storage location for Docker images that will be pulled by ECS or other services.



## ECR Repository Overview

### 4.4 Image Upload to ECR

- Detailed instructions for pushing a Docker image to Amazon ECR (Elastic Container Registry) using PowerShell on Windows. The steps include retrieving a login token using `Get-ECRLoginCommand`, building the Docker image, tagging it with the ECR repository URI, and finally pushing it. These commands automate the authentication and image upload process, ensuring seamless deployment of container images to AWS infrastructure.



### ECR Push Commands Guide

- The Docker build `-t my-website`. The command builds a Docker image from the Dockerfile. Then, `docker tag` renames it with the fully qualified AWS ECR URI. This prepares the image for pushing to Amazon's Elastic Container Registry. The successful build includes pulling `nginx:alpine` and copying project files into the container image.

```

Administrator: Windows PowerShell
Get-ECRLoginCommand : No region specified or obtained from persisted/shell defaults.
At line:1 char:2
+ (Get-ECRLoginCommand).Password | docker login --username AWS --passwo ...

+ CategoryInfo          : InvalidOperation: (Amazon.PowerShell.Cmdlets.GetECRLoginCommandCmdlet) [Get-ECRLoginCommand], InvalidOperationEx
+ FullyQualifiedErrorId : InvalidOperationException,Amazon.PowerShell.Cmdlets.ECR.GetECRLoginCommandCmdlet

PS C:\Users\fikra\Documents\Lesson\Cloud_Computing\projects\OPL> (Get-ECRLoginCommand -Region us-east-1).Password | docker login --username AWS --
password-stdin 949848044200.dkr.ecr.us-east-1.amazonaws.com
Login Succeeded
PS C:\Users\fikra\Documents\Lesson\Cloud_Computing\projects\OPL> docker build -t my-website .
[+] Building 2.5s (8/8) FINISHED                                docker:desktop-linux
=> [internal] load build definition from dockerfile              0.1s
=> => transferring dockerfile: 951B                             0.0s
=> [internal] load metadata for docker.io/library/nginx:alpine  0.7s
=> [auth] library/nginx:pull token for registry-1.docker.io     0.0s
=> [internal] load .dockerignore                                0.0s
=> => transferring context: 2B                                    0.0s
=> [internal] load build context                                0.3s
=> => transferring context: 1.28MB                                0.3s
=> CACHED [1/2] FROM docker.io/library/nginx:alpine@sha256:d67ea0d64d518b1bb04acde3b00f722ac3e9764b3209a9b0a98924ba35e4b779 0.1s
=> => resolve docker.io/library/nginx:alpine@sha256:d67ea0d64d518b1bb04acde3b00f722ac3e9764b3209a9b0a98924ba35e4b779 0.1s
=> [2/2] COPY . /usr/share/nginx/html                          0.1s
=> exporting to image                                           0.8s
=> => exporting layers                                           0.3s
=> => exporting manifest sha256:81609c9484aefb09d9df6f184f53a63c3f352abbf656b573b6e8ab37252c969b 0.0s
=> => exporting config sha256:5c0d32a9a5c56672ae35be57cb99d1dc8479b1dfad20f20aa4ecf885fc495153 0.0s
=> => exporting attestation manifest sha256:cf2d7448f84eed68a2bc2fb22ed7f94fb11bf6d27ea1bad0d9b25faccdb3cbcd 0.1s
=> => exporting manifest list sha256:f34339d13c69af81ba468dec9c2b4457f6b5d75816e8d9b035447ca16daf2d46 0.0s
=> => naming to docker.io/library/my-website:latest            0.0s
=> => unpacking to docker.io/library/my-website:latest         0.2s
PS C:\Users\fikra\Documents\Lesson\Cloud_Computing\projects\OPL>

```

### *Push Command: Docker Build and Tagging*

- This screenshot shows the actual push of the Docker image to Amazon ECR. The command `docker push <ECR URI>` uploads all image layers to the private repository. Each layer is marked as Pushed, confirming the image is now hosted on ECR and ready to be used by AWS ECS for deployment. The final line confirms the digest of the image that has been uploaded.

```

Administrator: Windows PowerShell
=> [internal] load metadata for docker.io/library/nginx:alpine 0.7s
=> [auth] library/nginx:pull token for registry-1.docker.io 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load build context 0.3s
=> => transferring context: 1.28MB 0.3s
=> CACHED [1/2] FROM docker.io/library/nginx:alpine@sha256:d67ea0d64d518b1bb04acde3b00f722ac3e9764b3209a9b0a98924ba35e4b779 0.1s
=> => resolve docker.io/library/nginx:alpine@sha256:d67ea0d64d518b1bb04acde3b00f722ac3e9764b3209a9b0a98924ba35e4b779 0.1s
=> [2/2] COPY . /usr/share/nginx/html 0.1s
=> exporting to image 0.8s
=> => exporting layers 0.3s
=> => exporting manifest sha256:81609c9484aefb09d9df6f184f53a63c3f352abbf656b573b6e8ab37252c969b 0.0s
=> => exporting config sha256:5c0d32a9a5c56672ae35be57cb99d1dc8479b1dfad20f20aa4ecf885fc495153 0.0s
=> => exporting attestation manifest sha256:cf2d7448f84eed68a2bc2fb22ed7f94fb11bf6d27ea1bad0d9b25faccdb3cbcd 0.1s
=> => exporting manifest list sha256:f34339d13c69af81ba468dec9c2b4457f6b5d75816e8d9b035447ca16daf2d46 0.0s
=> => naming to docker.io/library/my-website:latest 0.0s
=> => unpacking to docker.io/library/my-website:latest 0.2s
PS C:\Users\fikra\Documents\Lesson\Cloud_Computing\projects\OPL> docker tag my-website:latest 949848044200.dkr.ecr.us-east-1.amazonaws.com/my-website:latest
PS C:\Users\fikra\Documents\Lesson\Cloud_Computing\projects\OPL> docker push 949848044200.dkr.ecr.us-east-1.amazonaws.com/my-website:latest
The push refers to repository [949848044200.dkr.ecr.us-east-1.amazonaws.com/my-website]
5585638209e: Pushed
f23865b38cc6: Pushed
9a8b492aa4a: Pushed
958a74d6a238: Pushed
28fa206d77b: Pushed
8824c27679d3: Pushed
fd372c3c84a2: Pushed
bdaad27fd04a: Pushed
1d2dc189e38: Pushed
47ca43bfacc4: Pushed
latest: digest: sha256:f34339d13c69af81ba468dec9c2b4457f6b5d75816e8d9b035447ca16daf2d46 size: 856
PS C:\Users\fikra\Documents\Lesson\Cloud_Computing\projects\OPL>

```

### *Docker Push to Amazon ECR*

- Lists multiple image versions of the my-website repository that have been successfully pushed to ECR. Each image is tagged with a timestamp and has a corresponding SHA256 digest. The sizes are around 22–24 MB.

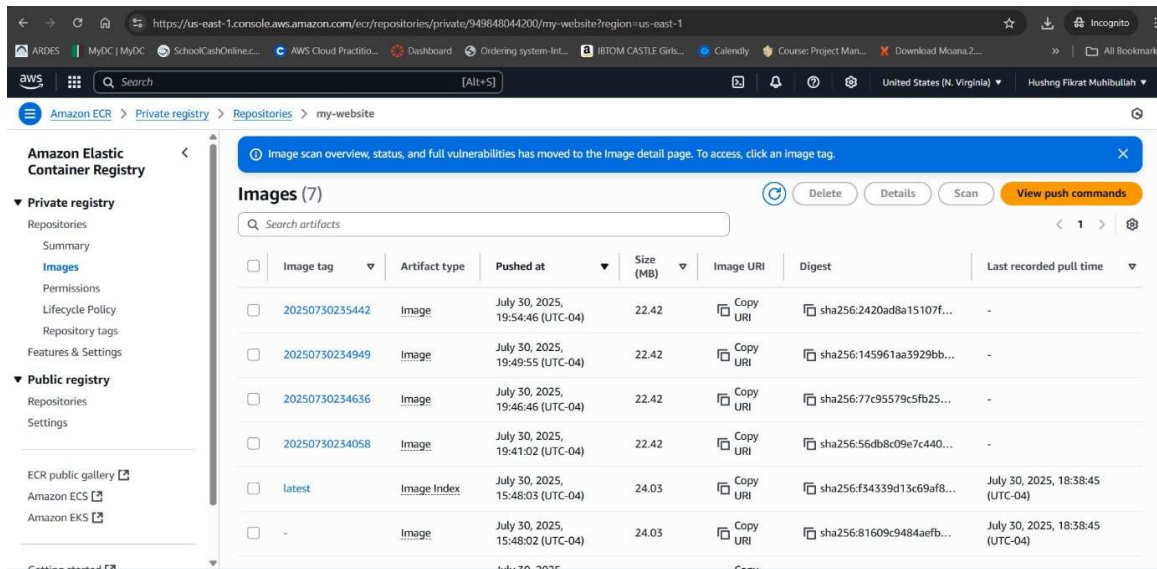
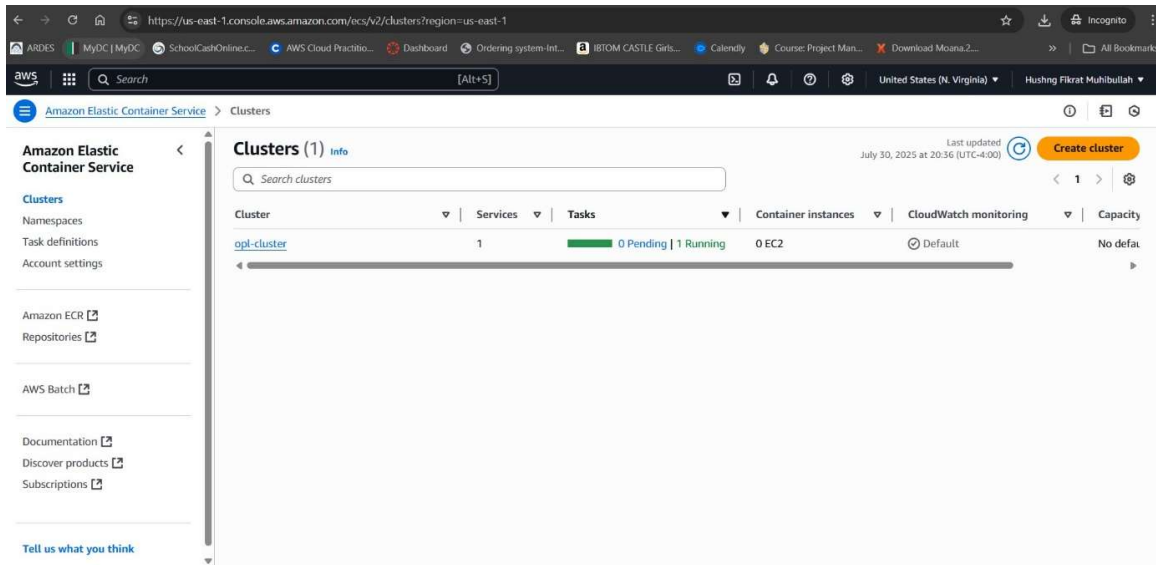


Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest	Last recorded pull time
20250730235442	Image	July 30, 2025, 19:54:46 (UTC-04)	22.42	Copy URI	sha256:2420ad8a15107f...	-
20250730234949	Image	July 30, 2025, 19:49:55 (UTC-04)	22.42	Copy URI	sha256:145961aa3929bb...	-
20250730234636	Image	July 30, 2025, 19:46:46 (UTC-04)	22.42	Copy URI	sha256:77c95579c5fb25...	-
20250730234058	Image	July 30, 2025, 19:41:02 (UTC-04)	22.42	Copy URI	sha256:56db8c09e7c440...	-
latest	Image Index	July 30, 2025, 15:48:03 (UTC-04)	24.03	Copy URI	sha256:f34339d13c69af8...	July 30, 2025, 18:38:45 (UTC-04)
-	Image	July 30, 2025, 15:48:02 (UTC-04)	24.03	Copy URI	sha256:81609c9484aefb...	July 30, 2025, 18:38:45 (UTC-04)

*ECR Images Pushed*

#### 4.5 ECS Cluster Creation

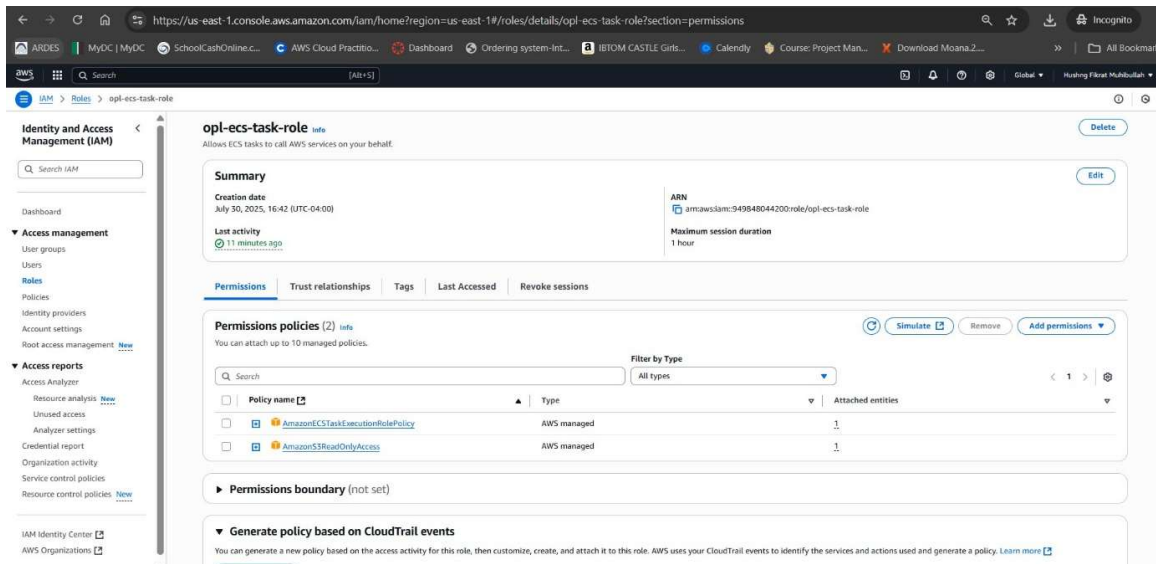
- Created a cluster named opl-cluster. It indicates that the cluster is active with one running task and no pending tasks, which confirms that a service is successfully deployed and running. The cluster is not using EC2 instances, suggesting it's likely running on AWS Fargate, a serverless compute engine. There is also one service associated with the cluster.



### ECS Cluster Overview

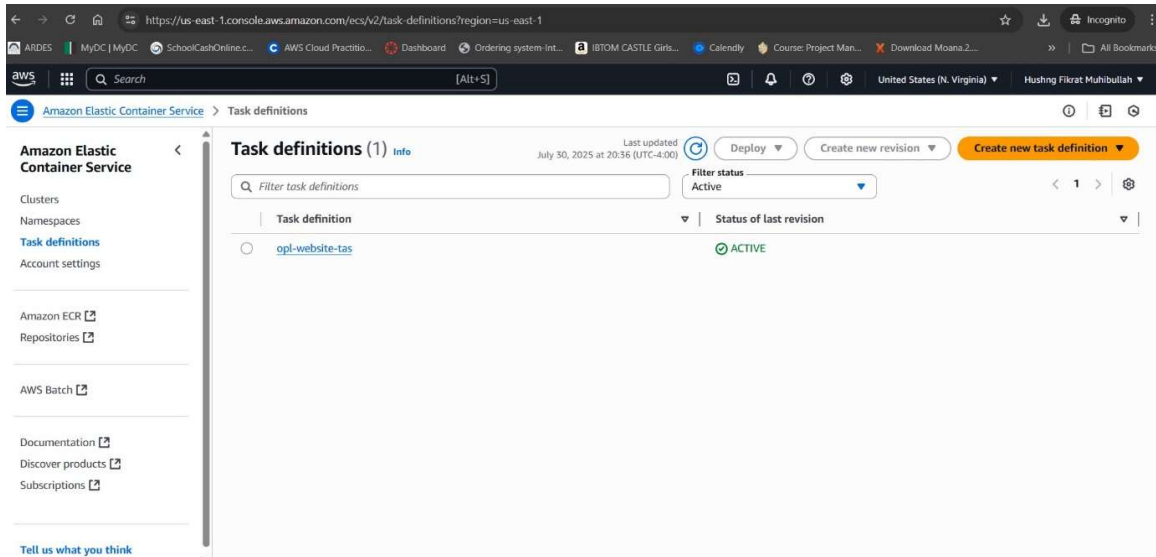
#### 4.6 Task Definition Creation

- IAM role named *opt-ecs-task-role* configured in AWS. It has two policies attached: *AmazonECSTaskExecutionRolePolicy* and *AmazonS3ReadOnlyAccess*. These allow ECS tasks to interact with necessary services like pulling container images from Amazon ECR and, optionally, reading objects from S3. This role is essential for enabling ECS to run and manage container workloads securely using the principle of least privilege.



### IAM Role - *opt-ecs-task-role*

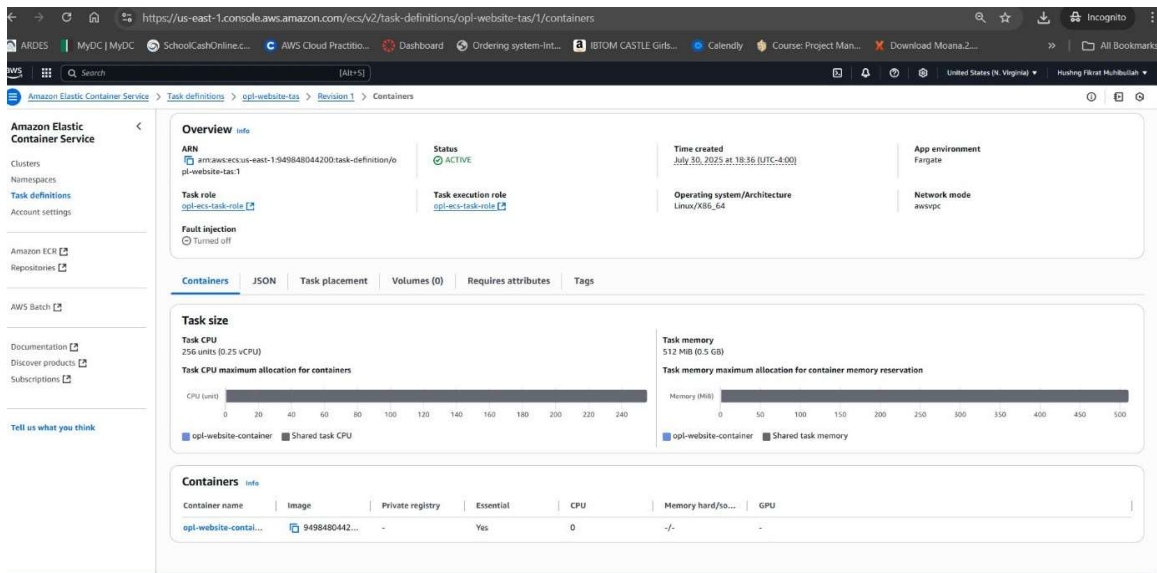
- Task definitions section of the ECS console, where the user has defined a task named opl-website-tas. The status of this task definition is marked as "ACTIVE", meaning it's currently being used by an ECS service. A task definition in ECS serves as a blueprint for running containers, specifying details such as the Docker image to use, memory and CPU requirements, networking mode, IAM roles, and port mappings. This is a crucial part of deploying containerized applications on ECS.



### *ECS Task Definition*

- The task is running on AWS Fargate with a Linux/X86\_64 architecture and uses the awsvpc network mode for networking. It allocates 256 CPU units and 512 MiB of memory to a container named opl-website-container. The task is associated with the IAM role opl-ecs-task-role, which grants it the necessary permissions to perform AWS service actions.

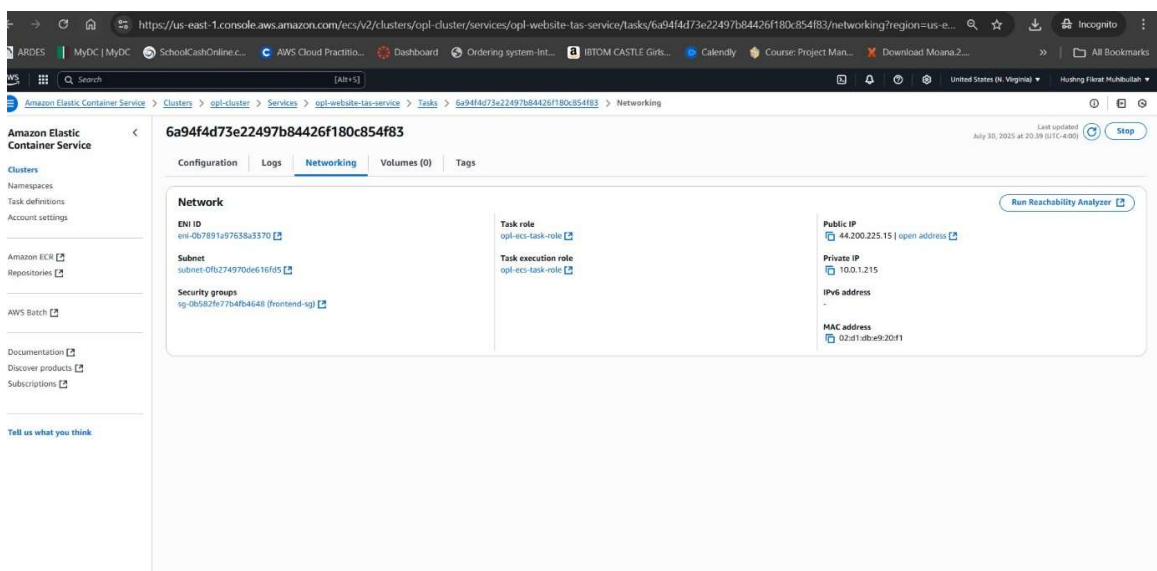




## ECS Task Definition Overview

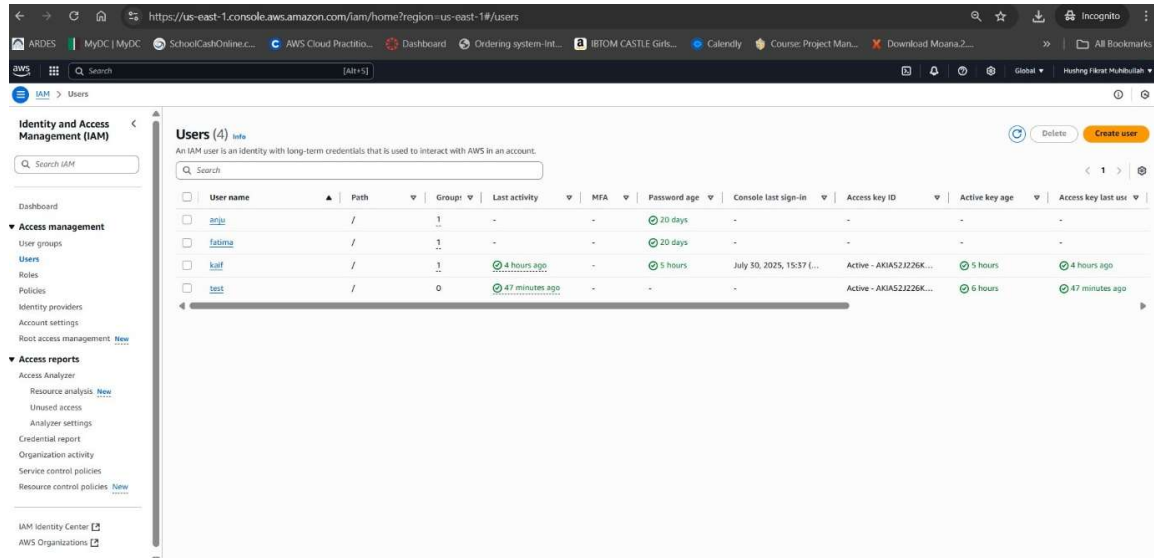
### 4.7 Service Creation

- Networking configuration of the running ECS task. It shows that the task is assigned to a specific subnet and security group named frontend-sg, ensuring controlled access. The task has been assigned both a private IP (10.0.1.215) and a public IP (44.200.225.15), indicating that the application is publicly accessible. The networking tab confirms successful deployment with internet reachability.



## 5. Security Considerations

List of IAM users configured in the AWS account. There are four users—anju, fatima, kaif, and test. The last activity column shows when each user last interacted with the AWS environment.

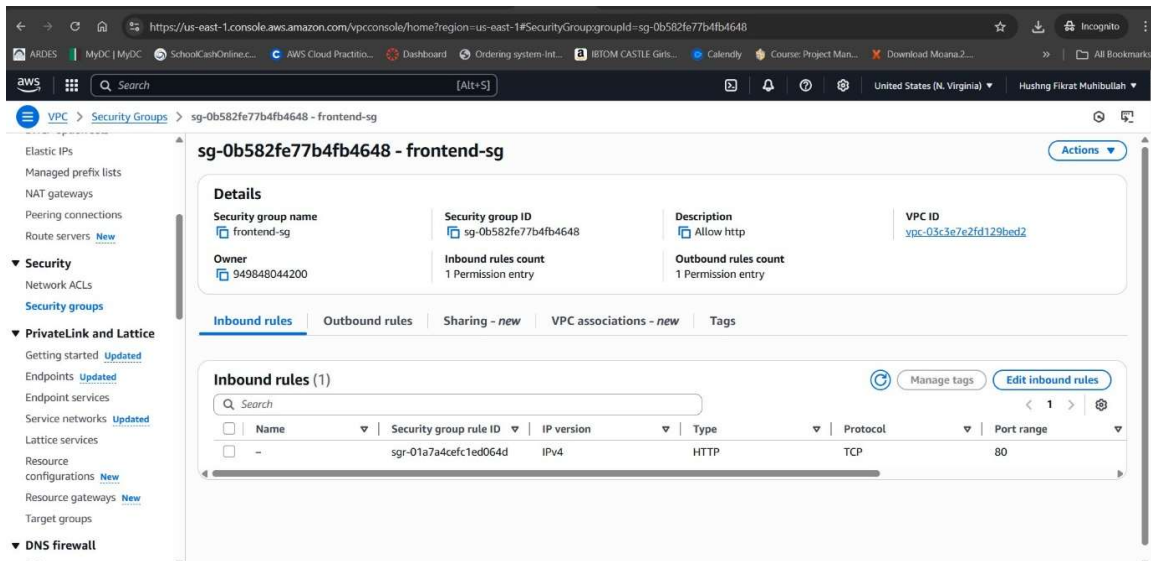


<input type="checkbox"/>	User name	Path	Group	Last activity	MFA	Password age	Console last sign-in	Access key ID	Active key age	Access key last use
<input type="checkbox"/>	anju	/	1	-	-	20 days	-	-	-	-
<input type="checkbox"/>	fatima	/	1	-	-	20 days	-	-	-	-
<input type="checkbox"/>	kaif	/	1	4 hours ago	-	5 hours	July 30, 2025, 15:37 (...)	Active - AKIAS2J226K...	5 hours	4 hours ago
<input type="checkbox"/>	test	/	0	47 minutes ago	-	-	-	Active - AKIAS2J226K...	6 hours	47 minutes ago

*IAM Users List*

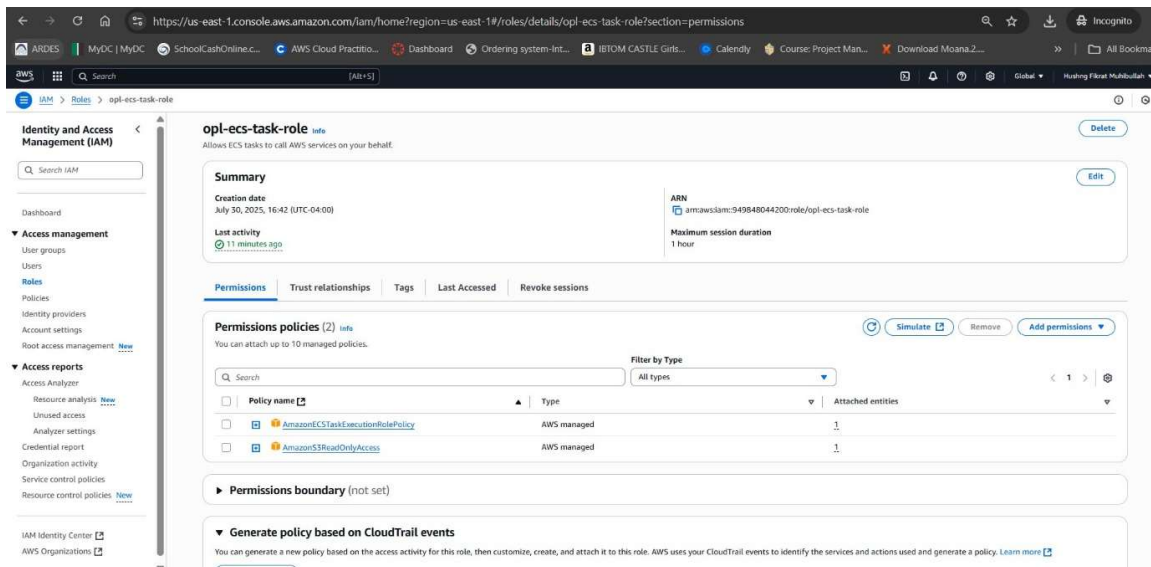
- Security group named frontend-sg, which allows HTTP (port 80) traffic. The inbound rule accepts TCP traffic on port 80 from any IPv4 address, which is typical for publicly accessible web services. This security group is essential to allow incoming connections to the frontend container or EC2 instance. The outbound rule (not shown) likely allows all traffic, which is the AWS default.





### *Security Group for Frontend*

- IAM role named *opl-ecs-task-role* is configured in AWS. This role is essential for enabling ECS to run and manage container workloads securely using the principle of least privilege.



### *IAM Role - opl-ecs-task-role*

## 6. Challenges Faced

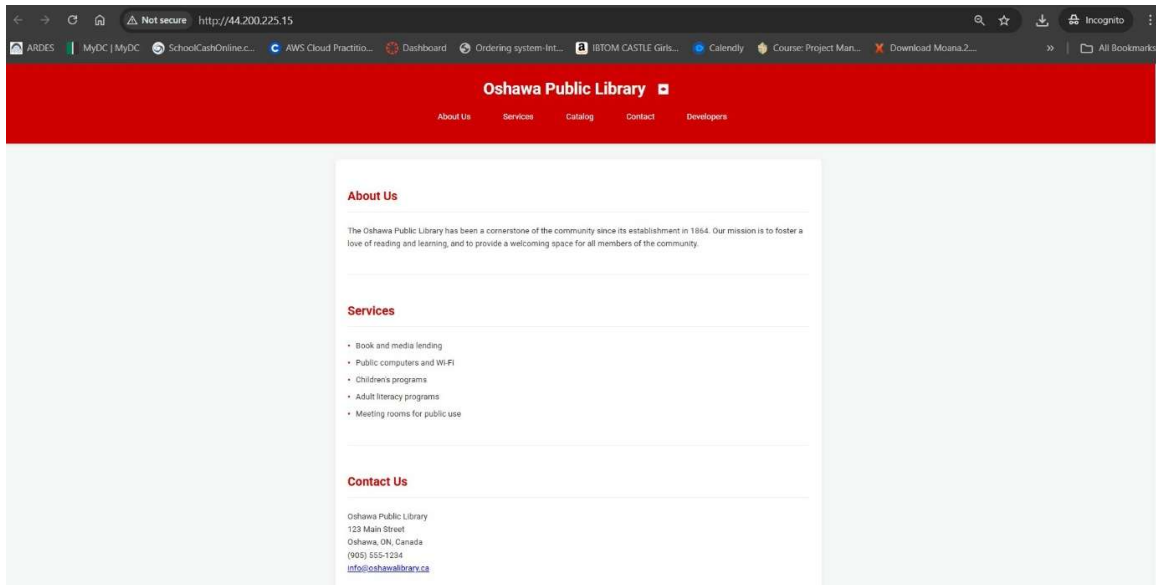
During the deployment process, several challenges were encountered that required troubleshooting and adjustments:

- **ECR Authentication Failures** – At times, Docker failed to authenticate with Amazon ECR, resulting in push and pull errors. This issue was resolved by re-running the AWS CLI authentication command (`aws ecr get-login-password`) to refresh credentials and re-establish the connection between Docker and ECR.
- **Cannot Pull Container Error** – ECS tasks occasionally failed to start due to an inability to pull the container image from ECR. This was resolved by ensuring the ECS task execution role had the Amazon ECS Task Execution Role Policy attached and by verifying that the ECR image URI was correct. In some cases, re-pushing the Docker image to ECR resolved the problem.
- **Access Denied Errors** – Some AWS actions, such as creating ECS services or configuring VPC components, initially failed due to insufficient IAM permissions. These were temporarily resolved by attaching broader permissions like Amazon ECS Full Access, Amazon VPC Full Access, and Amazon EC2 Container Registry Full Access to the deployment role. After deployment, permissions were reviewed and tightened to follow the principle of least privilege.
- **Budget Constraints** – Due to limited budget allocation for the project, certain features such as an Application Load Balancer (ALB), custom domain setup with Route 53, and HTTPS using AWS Certificate Manager were deferred. Instead, the service was exposed using a public IP for testing purposes, with these enhancements planned for future implementation.

## 7. Outcomes & Benefits

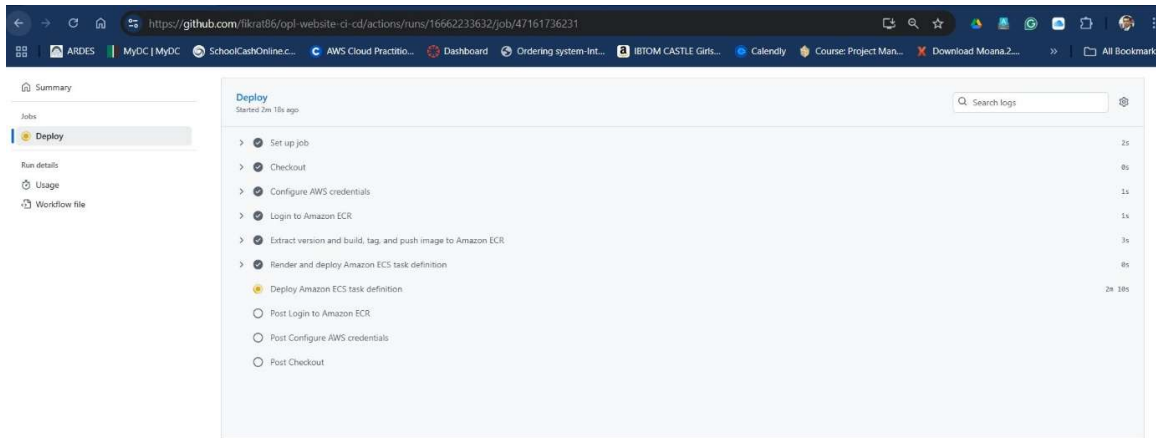
Successful deployment of a static website hosted on AWS infrastructure. The public IP address (*44.200.225.15*) is accessible in the browser, serving the Oshawa Public Library website. This result proves that the ECS is correctly set up in a public subnet with the

right security group and routing. It's the final verification that all cloud networking and containerization steps were done properly.



*Website Running from Public IP*

- This screenshot shows a GitHub Actions workflow executing deployment steps for the ECS service. The workflow includes stages such as checking out the code, configuring AWS credentials, logging into Amazon ECR, building and pushing a Docker image, and deploying an ECS task definition. The highlighted step suggests that deployment to ECS is currently in progress



### *GitHub Actions CI/CD Workflow*

## 8. Future Improvements

The current deployment meets the basic requirements for hosting a public-facing web application; however, several enhancements can significantly improve performance, security, and scalability in the future:

- **Application Load Balancer (ALB) with HTTPS** – Introduce an ALB to route incoming traffic to ECS tasks. This will improve fault tolerance, enable zero-downtime deployments, and provide better integration with AWS WAF. Using AWS Certificate Manager (ACM) to issue and manage SSL/TLS certificates will ensure all traffic is encrypted via HTTPS, improving security and user trust.
- **Autoscaling Policies** – Implement ECS Service Auto Scaling to automatically adjust the number of running tasks based on CPU, memory usage, or custom CloudWatch metrics. This will help maintain performance during peak loads while reducing costs during periods of low traffic.
- **Backend API Integration** – Add a private backend service running in the private subnets of the VPC. This API could handle database interactions, authentication, or

advanced business logic, ensuring that sensitive operations remain isolated from public access.

- **Amazon RDS Integration** – Deploy a managed relational database (such as PostgreSQL or MySQL) within private subnets to provide persistent data storage. RDS will handle backups, patching, and high availability, allowing the application to scale without database performance bottlenecks.
- **CloudWatch Alarms & AWS WAF** – Set up CloudWatch alarms to monitor CPU usage, memory consumption, HTTP error rates, and service health. Integrating AWS WAF will add an additional layer of protection by filtering out malicious traffic, blocking SQL injection, cross-site scripting (XSS), and other web exploits before they reach the application.

## 9. Conclusion

The deployment of the Oshawa Public Library (OPL) web application on **AWS ECS Fargate** successfully meets the immediate objectives of providing a secure, scalable, and cost-effective hosting environment. By leveraging **ECS with Fargate**, the team eliminated the complexity of managing EC2 instances while maintaining the flexibility to run containerized workloads. The integration of **Amazon ECR** ensures secure, centralized storage of Docker images, supporting a clean and controlled build-to-deployment workflow.

The use of a **custom VPC with public and private subnets** enhances network security, allowing sensitive components to remain isolated while still enabling controlled public access. This architecture aligns with AWS best practices for secure application hosting. The inclusion of **role-based IAM policies** further strengthens the security posture, ensuring that permissions are tightly managed.

From a scalability perspective, ECS Fargate enables on-demand task scaling to handle fluctuating workloads, making the solution adaptable to both routine library operations and seasonal spikes in user traffic. The design also anticipates future growth, with

provisions for integrating an **Application Load Balancer (ALB)** and enabling **HTTPS** for secure production traffic.

The deployment process is documented as a **repeatable model**, allowing OPL or other teams to replicate the setup with minimal effort. The optional integration with **CI/CD pipelines** provides a pathway to full automation, supporting faster release cycles and improved development efficiency.

In conclusion, this implementation serves as a practical demonstration of how AWS ECS Fargate can be used to deliver a reliable, secure, and budget-conscious containerized application deployment, while keeping the architecture flexible for future upgrades and technological enhancements.

## 10 References:

- Amazon Web Services, Inc. (2024). Amazon Elastic Container Service developer guide. Retrieved August 2025, from <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html>
- Amazon Web Services, Inc. (2024). AWS Fargate. Retrieved August 2025, from [https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS\\_Fargate.html](https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html)
- Amazon Web Services, Inc. (2024). Amazon Elastic Container Registry user guide. Retrieved August 2025, from <https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html>
- Amazon Web Services, Inc. (2024). Amazon Virtual Private Cloud user guide. Retrieved August 2025, from <https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html>
- Amazon Web Services, Inc. (2024). Amazon Elastic Kubernetes Service user guide. Retrieved August 2025, from <https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html>
- Amazon Web Services, Inc. (2024). Application Load Balancer. Retrieved August 2025, from <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html>
- Amazon Web Services, Inc. (2023). AWS Well-Architected Framework. Retrieved August 2025, from <https://aws.amazon.com/architecture/well-architected/>
- Amazon Web Services, Inc. (2023). AWS security best practices. Retrieved August 2025, from <https://aws.amazon.com/whitepapers/aws-security-best-practices/>
- Amazon Web Services, Inc. (2024). AWS container services overview. Retrieved August 2025, from <https://aws.amazon.com/containers/>