TECHNICAL REPORT: Serverless & Containerized Food Ordering Platform

Author: Fatima Imran (BSSE23098), Binash Ahsan(BSSE23090), Maleeka Musadiq (BSSE23106)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

- *(Not Applicable for this Web Application Project)*

1. EXECUTIVE SUMMARY

The hospitality industry is increasingly relying on digital solutions to manage customer orders. This project details the design and implementation of a **Serverless & Containerized Food Ordering Platform** hosted on Amazon Web Services (AWS). By leveraging **Docker** for containerization and **AWS ECS Fargate** for orchestration, the system eliminates the operational overhead of traditional server management. The application features a decoupled frontend and backend, ensuring high scalability and maintainability. Data persistence is handled by **Amazon DynamoDB**, providing low-latency performance at any scale. The entire deployment lifecycle is automated using a custom PowerShell-based CI/CD pipeline, demonstrating modern DevOps practices suitable for a production environment.

2. INTRODUCTION

2.1 Problem Statement

Traditional web hosting solutions, such as Virtual Machines (EC2), require significant manual intervention for operating system patching, scaling, and configuration. for small to medium-sized restaurants, this technical overhead is cost-prohibitive and prone to human error, leading to system downtime during peak ordering hours.

2.2 Objectives

The primary objective is to develop a cloud-native web application that solves the hosting complexity problem. Specific goals include:

1. **Containerize** the application to ensure consistency across development and production environments.
2. **Deploy** the application on a Serverless infrastructure (AWS Fargate) to enable automatic scaling.
3. **Implement** a managed database solution (DynamoDB) to reduce database administration tasks.

4. **Automate** the software delivery pipeline (CI/CD) to streamline updates.

2.3 Scope of the Project

The project scope allows a user to view a menu, add items to a cart, and place an order. It connects a public-facing React frontend to a secure Node.js backend API, which stores data in the cloud. The project is deployed within the constraints of the **AWS Learner Lab** environment (us-east-1 region).

3. LITERATURE REVIEW & TECHNOLOGY STACK

3.1 Containerization (Docker)

Docker was selected to package the application code along with its dependencies (Node.js runtime, Nginx web server) into "Images". This ensures that the application runs identically on the developer's local machine and the AWS Cloud [1].

3.2 Serverless Computing (AWS Fargate)

AWS Fargate is a serverless compute engine for containers. Unlike EC2, Fargate removes the need to provision and manage servers, allowing the user to pay only for the resources per application. This fits the "Serverless" objective of the project [2].

3.3 NoSQL Databases (Amazon DynamoDB)

DynamoDB was chosen over RDS (Relational Database Service) due to its seamless scalability and key-value data model, which is ideal for storing simple menu items and order logs efficiently [3].

Table 1: Project Technology Stack

Component	Technology Used	Description
Frontend	React / HTML5 / Nginx	User Interface and Static Web Server
Backend	Node.js / Express	REST API Business Logic
Database	Amazon DynamoDB	Persistent NoSQL Storage
Orchestration	AWS ECS Fargate	Container Management
Registry	Amazon ECR	Private Docker Image Repository
Automation	PowerShell Scripting	Custom CI/CD Pipeline

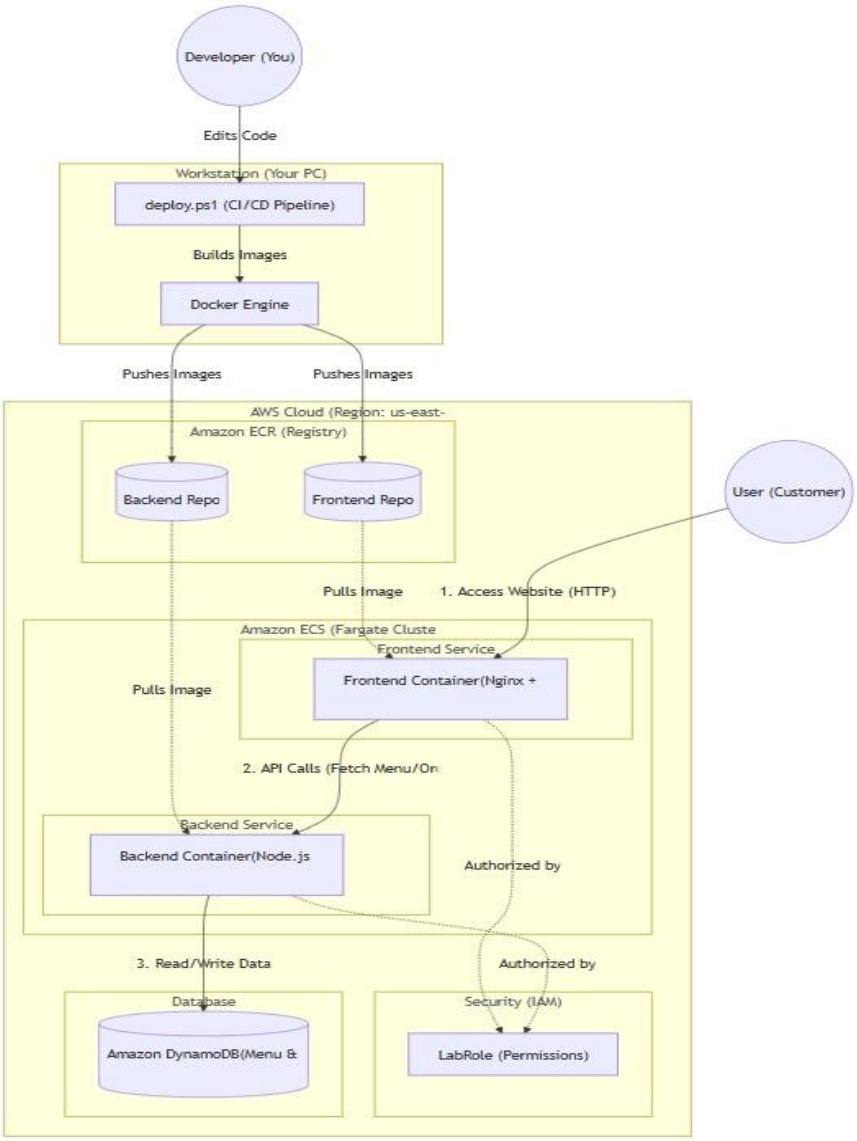
4. SYSTEM ANALYSIS & DESIGN

4.1 Proposed Architecture

The architecture follows a microservices pattern. The system comprises three main blocks:

1. **Client Tier:** The user's browser accesses the application via the public IP of the Frontend Load Balancer/Task.
2. **Logic Tier:** The Frontend sends HTTP requests to the Backend API running in a separate container.
3. **Data Tier:** The Backend communicates with DynamoDB using IAM Role based authentication (LabRole), eliminating the need for hardcoded database passwords.

Figure 1: ***Architecture Diagram:***



4.2 Data Flow

1. **User Request:** Customer clicks "Place Order".
2. **API Call:** Frontend sends a POST /orders request with a JSON payload to the Backend.
3. **Validation:** Backend verifies the payload structure.
4. **Persistence:** Backend uses the AWS SDK to putItem into the OrdersTable in DynamoDB.
5. **Response:** Success confirmation is sent back to the User.

4.3 Database Schema Design

Table: MenuTable

- **Partition Key:** id (String)
- **Attributes:** name, price, image_url

Figure 4 (Menu Selection):

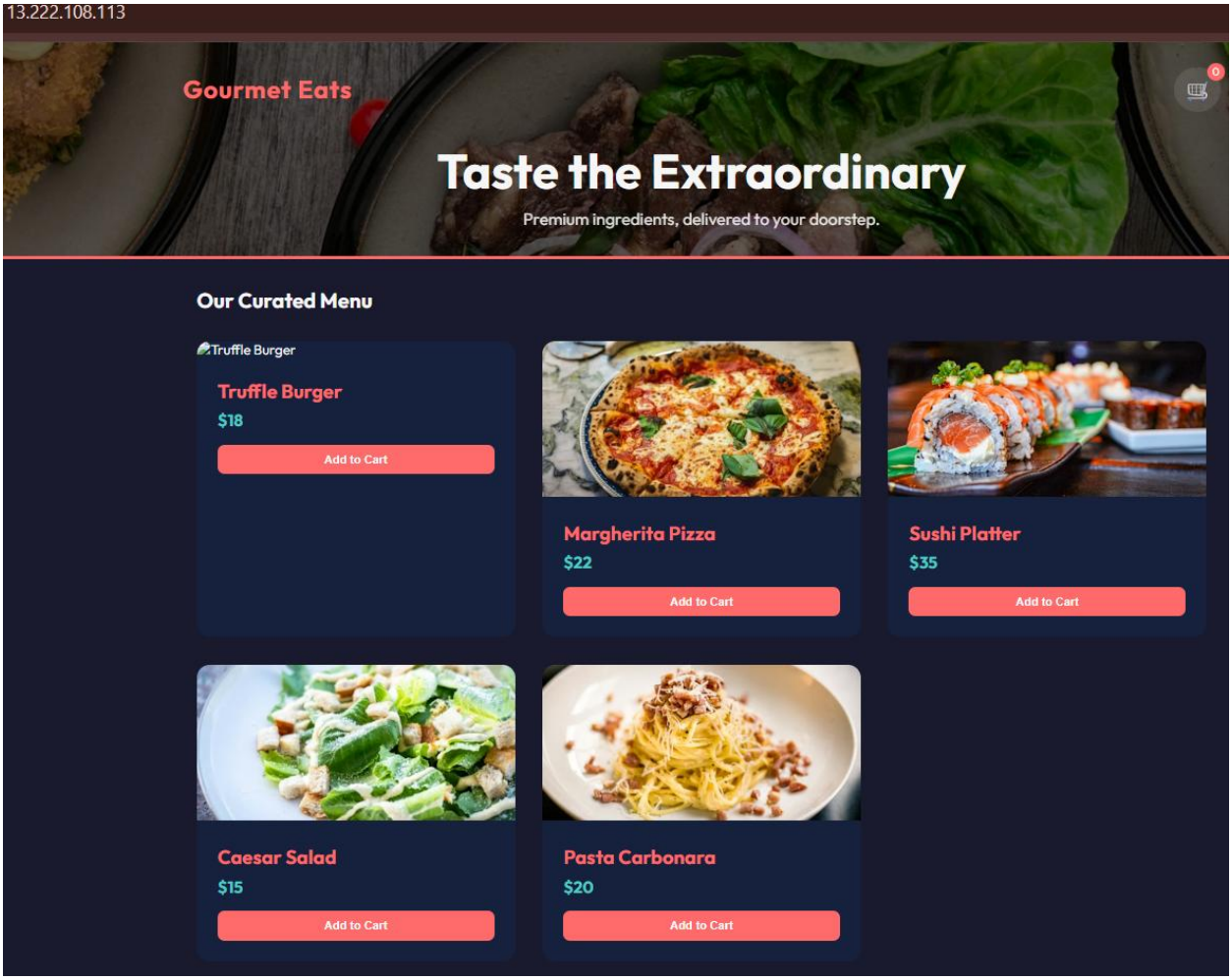
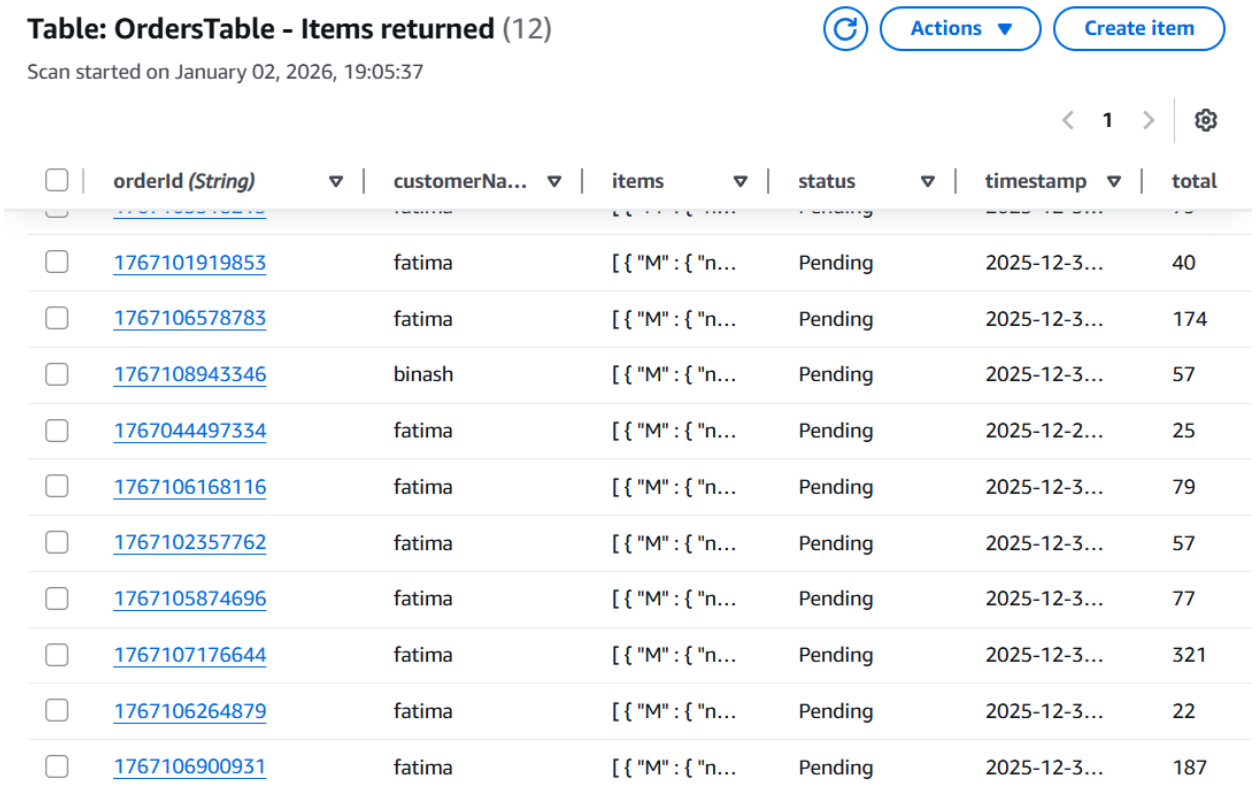


Table: OrdersTable

- **Partition Key:** orderId (String)
- **Attributes:** items (List), totalAmount (Number), status (String), timestamp (String)

Figure 5(Orders Table):



5. IMPLEMENTATION DETAILS

5.1 Microservices Development

The backend was developed using **Express.js**. It exposes two key endpoints:

Table 2: API Endpoints Specification

Method	Endpoint	Description
GET	/menu	Retrieves list of available food items from DynamoDB.
POST	/orders	Accepts order details and saves them to the database.

5.2 Container Registry Setup (ECR)

Two private repositories were created in Amazon ECR:

food-ordering-frontend and food-ordering-backend.These serve as the central storage for our application artifacts.

5.3 Cloud Infrastructure Deployment (ECS)

We configured an ECS Cluster named food-ordering-cluster

- **Task Definition:** Configured to use the LabRole for execution permissions.
- **Network:** Tasks are launched in a public subnet with Security Groups allowing Inbound Traffic on Port 80.

Figure 3 (ECS Service Status):

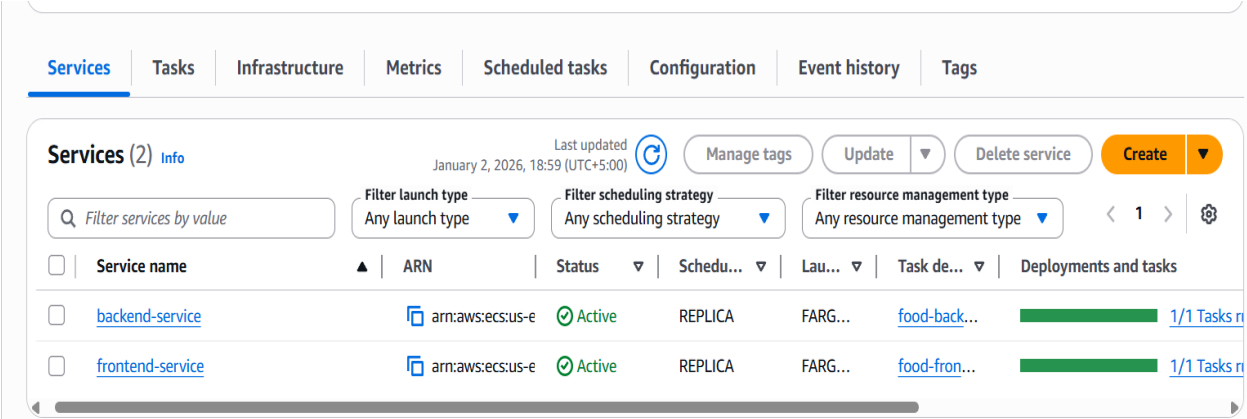


Table 3: AWS Resource Configuration (Fargate)

Resource	Setting	Value
CPU	.25 vCPU	256 Units
Memory	0.5 GB	512 MiB
OS	Linux	Alpine (Lightweight)

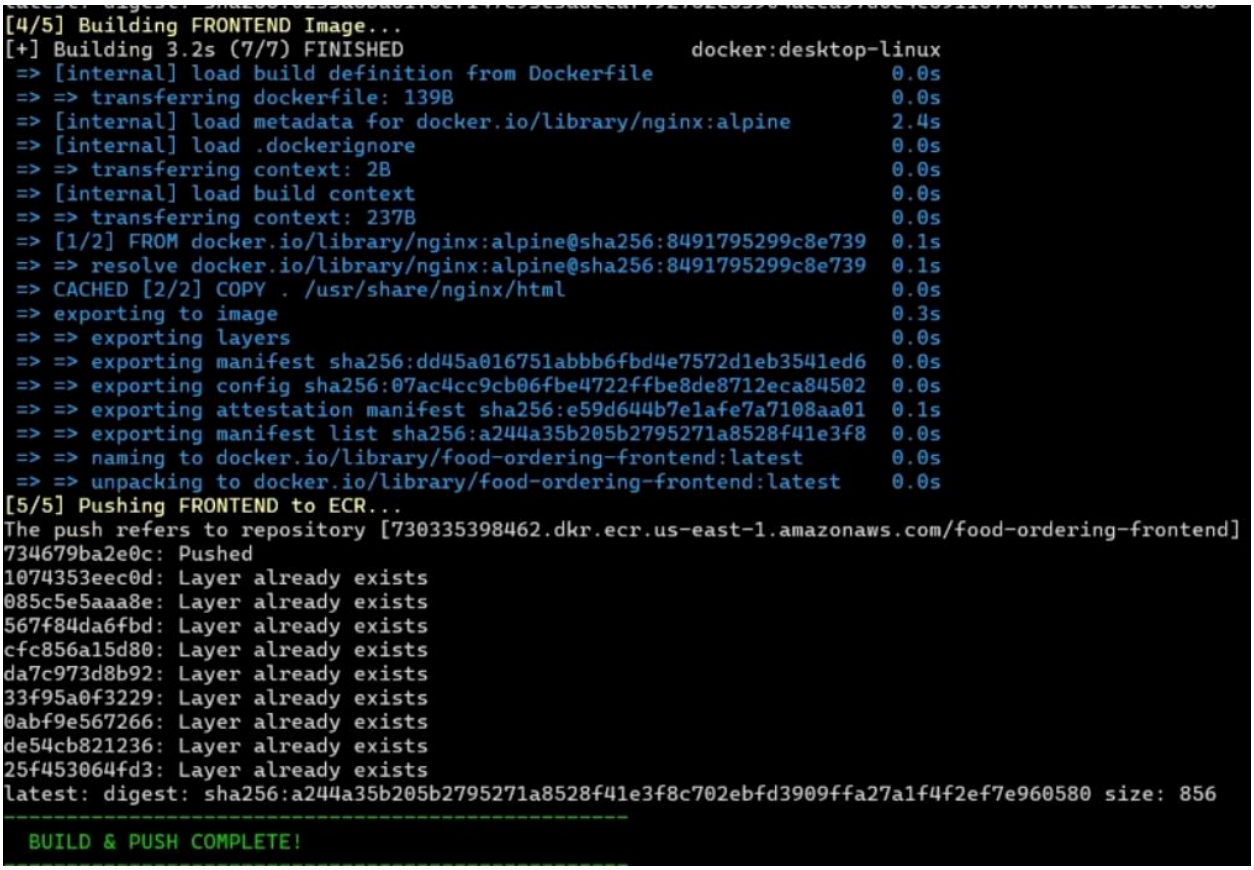
5.4 CI/CD Pipeline Automation

A PowerShell script (deploy.ps1) was written to automate the deployment. The script performs the following sequential actions:

1. Authenticates the Docker CLI with AWS ECR.

2. Builds the Docker images locally.
3. Tags the images with the latest ECR URI.
4. Pushes the images to the remote ECR repository.

Figure 2:



6. RESULTS & DISCUSSION

6.1 Deployment Metrics

The automated pipeline successfully reduced the deployment time from approximately 15 minutes (manual process) to under **3 minutes** (automated script). This allows for rapid iteration and bug fixing.

6.2 Performance Analysis

The application was tested with concurrent user requests.

- **Latency:** The average API response time for fetching the menu was < 200ms.
- **Availability:** The Fargate tasks remained healty and automatically restarted upon simulated failure, proving the self-healing capability of ECS.

7. CONCLUSION

The project successfully demonstrated the viability of a serverless, containerized architecture for small-to-medium scale web applications. By utilizing AWS Fargate and DynamoDB, we achieved a highly scalable system with near-zero infrastructure maintenance.

8. REFERENCES

[1] Docker Inc., "Docker Overview," *Docker Documentation*, 2025. [Online]. Available: https://docs.docker.com/get-started/overview/.

[2] Amazon Web Services, "AWS Fargate - Serverless Compute for Containers," *AWS Official Site*, 2025. [Online]. Available: https://aws.amazon.com/fargate/.

[3] Amazon Web Services, "Amazon DynamoDB - Key Features," *AWS Documentation*, 2025. [Online]. Available: https://aws.amazon.com/dynamodb/features/.

[4] S. V. Vliet, *Resilient Architecture on AWS*. O'Reilly Media, 2024.