Project Title

ElectroShop MERN stack application – DevOps & Cloud Infrastructure

Project Objective

To design, implement, and manage a secure, scalable, and automated cloud infrastructure for the ElectroShop MERN stack application. This project will establish a complete DevOps lifecycle, from initial containerization and cloud deployment to a fully automated CI/CD pipeline with robust monitoring and security practices on AWS.

Technologies and Tools

- **Containerization:** Docker, Docker Compose
- **Cloud:** AWS (VPC, IAM, ECR, ECS, Fargate, Application Load Balancer, Secrets Manager)
- **CI/CD:** GitHub Actions
- **Monitoring:** Amazon CloudWatch (Logs, Metrics, Dashboards, Alarms), Amazon SNS
- **Security:** AWS WAF, Trivy / Amazon ECR Scanning

Phase 1: Containerization & AWS Deployment (Weeks 1-3)

Deadline: 10/07/2025

Week 1: Project Setup & Containerization

Goal: Encapsulate the application components into portable Docker containers and ensure they can be orchestrated locally.

Tasks:

1. **Backend Containerization:**
    - Create a `Dockerfile` in the `backend` directory.
    - **Requirements:**
        - It must be a multi-stage build to keep the final image size small.
        - The final stage should be based on a lightweight Node.js image (e.g., `node:18-alpine`).
        - It should only install `production` dependencies in the final image.
        - The server's port (e.g., 5000) must be exposed.

        - The container's default command should start the Node.js server (`server.js`).
2. **Frontend Containerization:**
    - Create a `Dockerfile` in the `frontend` directory.
    - **Requirements:**
        - It must be a multi-stage build.
        - The first stage must build the static assets of the React application using `npm run build`.
        - The final stage must use a lightweight web server (e.g., `nginx:stable-alpine`) to serve the built static files.
        - The web server must be configured to listen on port 80.
3. **Local Orchestration:**
    - Create a `docker-compose.yml` file in the project's root directory.
    - **Requirements:**
        - Define two services: `frontend` and `backend`.
        - Configure the services to build from their respective `Dockerfile`.
        - Map the container ports to host ports for local access (e.g., `3000:80` for frontend, `5000:5000` for backend).
        - Implement a way to pass environment variables (like `MONGO_URI`) to the backend service.
4. **Validation:**
    - The team must successfully run the entire application stack locally using a single `docker-compose up` command.
    - Verify that the frontend application at `http://localhost:3000` can communicate with the backend service.

Week 2: AWS Infrastructure Provisioning

Goal: Use the AWS Management Console to provision the core cloud infrastructure required to host the containerized application.

Tasks:

1. **Virtual Private Cloud (VPC) Setup:**
    - Provision a new VPC. It must contain both public and private subnets distributed across at least two Availability Zones to ensure high availability.
    - Configure an Internet Gateway for public subnet access and a NAT Gateway for private subnet outbound access.
    - Set up route tables to correctly manage traffic flow.
2. **Identity and Access Management (IAM) Roles:**
    - Create the necessary IAM roles. This must include an **ECS Task Execution Role** that grants ECS permissions to pull images from ECR and manage logs in CloudWatch.
3. **Elastic Container Registry (ECR) Setup:**
    - Create two private ECR repositories: one for the `electroshop/frontend` image and one for the `electroshop/backend` image.
4. **Elastic Container Service (ECS) Cluster:**

    - Create a new ECS cluster based on the AWS Fargate (serverless) launch type.
5. **Security Group Configuration:**
    - Design and implement a set of Security Groups:
        - One for the Application Load Balancer (ALB), allowing inbound HTTP/HTTPS traffic.
        - One for the ECS services, configured to only accept traffic from the ALB's security group on their respective container ports.

Week 3: Manual Deployment to AWS

Goal: Manually push the container images to ECR and deploy them as running services on the provisioned ECS infrastructure.

Tasks:

1. **Push Container Images:**
    - The team must authenticate their local Docker client with AWS ECR.
    - They need to correctly tag the locally built `frontend` and `backend` images and push them to their respective ECR repositories.
2. **ECS Task Definition Creation:**
    - Create a Fargate Task Definition for the `backend` service. This involves configuring the image URI, task role, CPU/memory allocation, port mappings, and environment variables.
    - Create a separate Fargate Task Definition for the `frontend` service with its corresponding configuration.
3. **Application Load Balancer (ALB) Configuration:**
    - Deploy and configure an internet-facing ALB.
    - Create two Target Groups: one for the frontend service and one for the backend.
    - Configure the ALB listener on port 80 with rules to route traffic:
        - Default traffic should be forwarded to the `frontend` target group.
        - Traffic directed to a specific API path (e.g., `/api/*`) must be forwarded to the `backend` target group.
4. **ECS Service Creation:**
    - Deploy the `frontend` Task Definition as an ECS Service, configuring it to use the ALB and its target group. Ensure it is deployed across multiple Availability Zones.
    - Deploy the `backend` Task Definition as a separate ECS Service, linking it to the backend target group.
5. **Validation:**
    - The team must confirm the deployment is successful by accessing the ALB's public DNS name. The frontend application should load and be fully functional, successfully making API calls to the backend.

Phase 2: CI/CD, Monitoring & Security (Weeks 4-6)

Deadline: 10/07/2025

Objective: To fully automate the deployment pipeline, establish comprehensive monitoring and alerting, and implement security best practices.

Week 4: CI/CD Pipeline Automation

Goal: Create a continuous integration and deployment (CI/CD) pipeline using GitHub Actions to automate all future deployments.

Tasks:

1. **CI/CD Secrets Management:**
   - Create a dedicated IAM User for GitHub Actions with permissions restricted to ECR and ECS operations.
   - Store this user's `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` as encrypted secrets in the GitHub repository settings.
2. **Workflow Creation:**
   - Develop a GitHub Actions workflow file (`.github/workflows/deploy.yml`) that triggers on every push to the `main` branch.
   - **Workflow Steps:**
     1. The workflow must check out the source code.
     2. It must securely configure AWS credentials using the stored secrets.
     3. It must log in to the Amazon ECR registry.
     4. It must build, tag, and push the `frontend` and `backend` Docker images to their ECR repositories.
     5. It must fetch the latest ECS task definition, update it with the new container image IDs, and deploy the new revision to the appropriate ECS services. The use of official `aws-actions` for these steps is recommended.

Week 5: Monitoring & Logging

Goal: Implement a robust monitoring and logging solution to ensure operational visibility and rapid troubleshooting.

Tasks:

1. **Centralized Logging:**
   - Configure the ECS task definitions to use the `awslogs` log driver, directing all container logs (stdout/stderr) to Amazon CloudWatch Logs.
   - The team should demonstrate the ability to query these logs using CloudWatch Log Insights to debug a simulated application error.
2. **Performance Dashboards:**
   - Create a custom CloudWatch Dashboard.
   - This dashboard must provide a single-pane-of-glass view of the application's health, including key metrics for both ECS services (CPU/Memory Utilization) and the Application Load Balancer (Request Count, Error Rates, Target Health).
3. **Automated Alerting:**
   - Set up a notification system using Amazon SNS.

- Create and configure CloudWatch Alarms that trigger SNS notifications for critical events, such as sustained high CPU, a spike in 5xx server errors from the ALB, or any target group reporting unhealthy hosts.

Week 6: Security Hardening & Final Review

Goal: Enhance the security posture of the application and infrastructure, and conduct a final review of the entire setup.

Tasks:

1. **Secure Secrets Handling:**
    - Refactor the application deployment. All sensitive data (database URIs, API keys, JWT secrets) must be removed from the task definition's environment variables.
    - Store these secrets securely in **AWS Secrets Manager**.
    - Update the ECS Task Execution Role and Task Definitions to allow the containers to fetch these secrets at runtime.
2. **Automated Vulnerability Scanning:**
    - Integrate a container security scanner (such as **Trivy** or the built-in **Amazon ECR scanning**) into the CI/CD pipeline.
    - The pipeline must be configured to fail the build if any high-severity vulnerabilities are detected in the Docker images.
3. **Network Protection:**
    - Deploy **AWS WAF** (Web Application Firewall) and associate it with the Application Load Balancer.
    - Apply the core AWS-managed rule sets to provide baseline protection against common web exploits like SQL injection and XSS.
4. **Final Audit and Documentation:**
    - Conduct a thorough review of all IAM roles and policies, ensuring they adhere to the principle of least privilege.
    - Create clear documentation for the CI/CD pipeline, including instructions on how to manually trigger a rollback if needed.
    - Perform a final end-to-end test of the entire system, from a code push to a live, secured deployment.

Final Deliverables & Submission Guidelines

Required Deliverables

The final submission must include the following components:

1. **GitHub Repository:**
    - A public or private GitHub repository containing the complete source code for the frontend, backend, and all infrastructure/DevOps configuration files (`Dockerfile`, `docker-compose.yml`, `.github/workflows/deploy.yml`, etc.).
    - **Link:** `[Insert GitHub Repository URL Here]`
2. **Technical Documentation:**

    - A detailed `README.md` file in the repository root that explains the project architecture, setup instructions, and deployment process.
    - Architectural diagrams illustrating the AWS infrastructure and CI/CD workflow.
3. **Screenshots or Video Recording:**
    - Screenshots of the running application on AWS.
    - Screenshots of the CloudWatch monitoring dashboard and a successful GitHub Actions pipeline run.
    - (Optional but Recommended) A short video recording (~5-10 minutes) demonstrating the CI/CD pipeline in action and a walkthrough of the deployed infrastructure.
4. **Final Presentation Slides:**
    - A slide deck summarizing the project, architecture, challenges faced, and key achievements.
    - **Link:** `[Insert Link to Google Slides / PowerPoint Here]`

Submission & Collaboration Guide

- **Version Control (Git & GitHub):**
    - All code and configuration must be managed in the designated GitHub repository.
    - Follow the **GitFlow** or **Feature Branch** workflow. Do not commit directly to the `main` branch.
    - Create new branches for each feature or task (e.g., `feature/backend-dockerization`, `fix/alb-routing-rule`).
    - Use Pull Requests (PRs) to merge changes into the `main` branch. Each PR must be reviewed by at least one other team member before merging.
    - Write clear and descriptive commit messages (e.g., `feat: Add Dockerfile for backend service`, `docs: Update README with setup instructions`).

Evaluation Criteria

- End-to-end Application Functionality
- Proper Usage of Docker and Cloud Infrastructure (ECS/S3)
- CI/CD Pipeline Implementation and Automation
- Infrastructure Provisioning using IaC (Terraform)
- Cloud Monitoring & Logging Integration
- Code Quality and Clean Repository Structure
- Completeness of Documentation and Final Presentation Documentation and Final Presentation