Apollo Clothing Web App – DevOps Project Report

Apollo is a feature-rich, modern clothing web application designed to provide users with a seamless and enjoyable online shopping experience. The platform allows customers to browse a curated catalog of apparel, including suits, traditional wear, and casual outfits, each presented with high-resolution images, detailed descriptions, and pricing information.

Key features of Apollo include:

- A visually appealing, responsive user interface that adapts to desktops, tablets, and mobile devices.
- Easy navigation with categorized product listings, search functionality, and product filters.
- Secure user registration and login for personalized shopping and order tracking.
- A shopping cart system that enables users to add, remove, and review items before checkout.
- Integration with payment gateways for secure online transactions.
- An admin dashboard for managing inventory, orders, and customer data.

Apollo is built using PHP and Apache, with static assets for fast loading. The application is fully dockerized, making it easy to deploy, scale, and maintain. It is designed with DevOps best practices in mind, supporting automated CI/CD, infrastructure as code, and comprehensive monitoring to ensure reliability and performance in production environments.

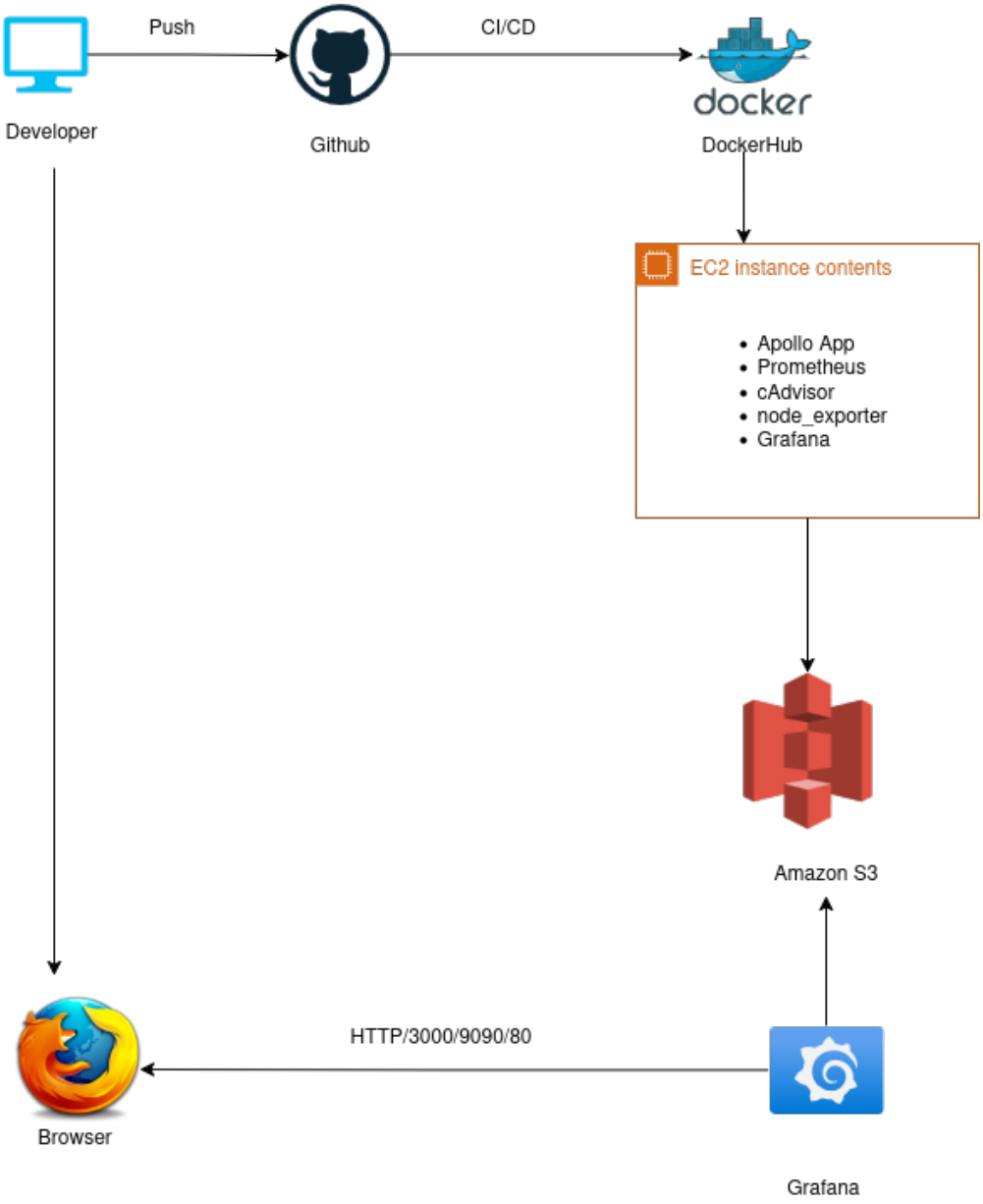
System Architecture

Apollo's system architecture is designed for reliability, scalability, and observability using modern DevOps practices:

- Source code is managed in GitHub, with GitHub Actions automating CI/CD to build, scan, and push Docker images to DockerHub and deploy to AWS.
- Infrastructure is provisioned using Terraform, which creates an EC2 instance, S3 bucket, and security groups.
- The EC2 instance runs multiple Docker containers: the Apollo web app, Prometheus (metrics collection), cAdvisor (container metrics), node_exporter (system metrics), and Grafana (dashboard visualization).
- Security groups restrict access to only necessary ports (HTTP, SSH, monitoring).
- Prometheus scrapes metrics from the system and containers, while Grafana provides real-time dashboards for monitoring.

- DevSecOps is integrated with Trivy image scanning and secure secret management.

This architecture ensures Apollo is easy to deploy, monitor, and maintain, with automated workflows and strong security controls. The image below highlights the system architecture of Apollo.



CI/CD Pipeline

The CI/CD pipeline for Apollo automates the process of building, testing, securing, and deploying the application:

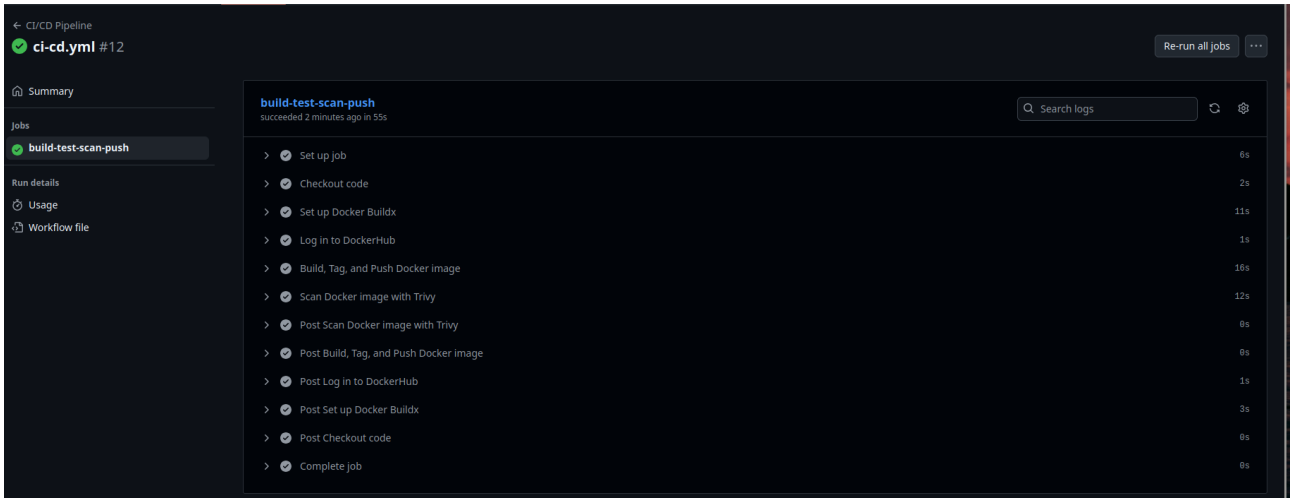
•Code changes are pushed to GitHub, triggering a GitHub Actions workflow.

•The workflow runs automated tests (if present) to ensure code quality.

•A Docker image of the Apollo app is built and scanned for vulnerabilities using Trivy.

•If the image passes security checks, it is pushed to DockerHub.
•The pipeline then deploys the latest image to an AWS EC2 instance, ensuring the server always runs the most up-to-date and secure version of Apollo.
•All secrets (like DockerHub and AWS credentials) are managed securely using GitHub Secrets.

This pipeline ensures fast, reliable, and secure delivery of new features and fixes to production. The configuration files are located in the appendices section and GitHub repository. The image below highlights the successful pipeline tests.

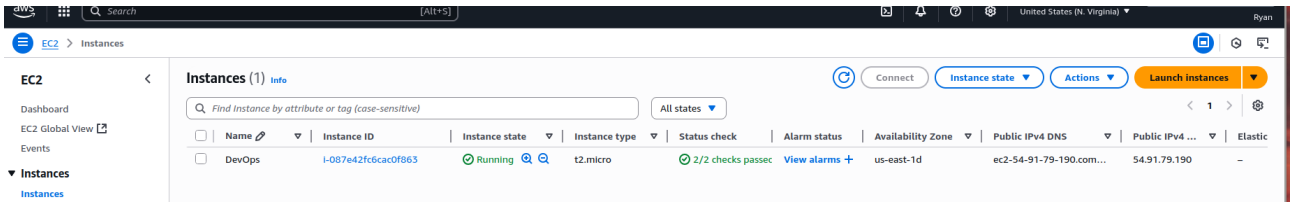


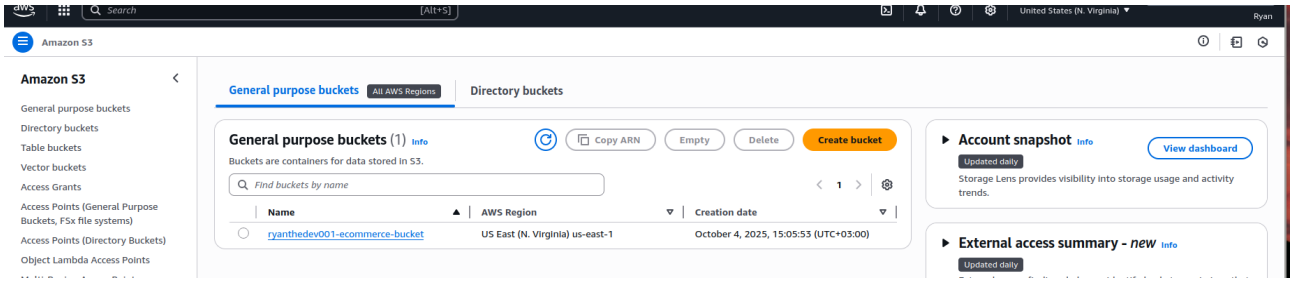
Infrastructure as Code (IaC)

Terraform is used to define, provision, and manage all cloud infrastructure for Apollo. The infrastructure is described in code, making it version-controlled, audit-able, and reproducible. Terraform scripts automatically create and configure AWS resources, including the EC2 instance (for app and monitoring containers), S3 bucket (for storage), and security groups (for network access control). This approach ensures consistent, repeatable deployments and enables rapid scaling or recovery by simply reapplying the code.

Containerization and Deployment

Apollo is packaged as a Docker container using a Dockerfile, ensuring consistency across development, testing, and production environments. Multi-service orchestration is managed with Docker Compose, allowing the Apollo app, Prometheus, cAdvisor, node_exporter, and Grafana to run as isolated containers on the same EC2 instance. Deployment is automated after building and pushing the Docker image to DockerHub, the latest version is pulled and run on the EC2 server. This approach enables rapid, reliable deployments, easy scaling, and simplified maintenance. The following images show the running instances of EC2 and S3.





Monitoring & Observability

The monitoring stack is designed to provide comprehensive visibility into both system and container performance. Prometheus serves as the central metrics collector, gathering data from node_exporter for host-level metrics such as CPU, memory, disk, and network usage. It also collects information from cAdvisor for real-time container resource utilization. If the application offers metrics endpoints, these are integrated as well for end-to-end observability. Grafana connects to Prometheus and converts raw metrics into interactive dashboards, facilitating proactive monitoring, troubleshooting, and capacity planning. This setup ensures that infrastructure and workloads are continuously monitored, providing actionable insights through customizable visualizations. The screenshots below are taken from Grafana's project dashboard.

Devops Practices

Security is integrated throughout the development and deployment lifecycle to ensure a robust and resilient system. Automated vulnerability scanning is performed on all Docker images using Trivy within the CI/CD pipeline, helping to identify and remediate known security issues before deployment. Sensitive credentials and configuration values are managed securely using GitHub Secrets for CI/CD and AWS IAM roles and policies for cloud resources, following the principle of least privilege. This approach minimizes the risk of unauthorized access and data exposure. Regular reviews and updates of dependencies, combined with infrastructure security best practices, further strengthen the overall security posture of the platform.

Conclusion

This project showcases the successful implementation of a modern DevOps pipeline, integrating automation, security, and observability across the entire software lifecycle. By leveraging containerization, infrastructure as code, and continuous integration/deployment, the system achieves rapid, reliable, and repeatable delivery to the cloud. Comprehensive monitoring and alerting ensure operational visibility, while DevSecOps practices proactively address vulnerabilities and protect sensitive data. The result is a resilient, scalable, and secure platform that exemplifies industry best practices for deploying and managing web applications in a cloud-native environment.

Appendices

CI/CD pipeline configuration file

ci-cd.yml

.github > workflows > ci-cd.yml

```yaml
name: CI/CD Pipeline

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build-test-scan-push:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to DockerHub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}


      - name: Build, Tag, and Push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ryanthedev001/ecommercetest:latest

      - name: Scan Docker image with Trivy
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: 'ryanthedev001/ecommercetest:latest'
          scan-type: 'image'
          exit-code: '0'
          severity: 'CRITICAL,HIGH'
          format: 'table'

```