TechNova DevOps Pipeline

Authors:

Dhruv Joshi

Raman Boora

Diya Tomar

Hardik Yadav

Sidharth Maalpani

Jayvardhan Singh

Here Organization:

Your Organization or Institution Name

Date:

July 3, 2025

Abstract

TechNova Solutions, a rapidly growing e-commerce startup, faced critical challenges in maintaining deployment efficiency due to manual infrastructure setup, inconsistent environments, and delayed release cycles.

To resolve these issues, our project focuses on building a fully automated DevOps pipeline that enables consistent, scalable, and secure deployment of web applications.

The pipeline starts with a version-controlled application repository hosted on GitHub, containing a Node.js-based dynamic web application and a static frontend built using HTML, CSS, and JavaScript. A custom Dockerfile is used to containerize the application to ensure environment consistency.

We implemented a CI/CD pipeline using GitHub Actions, which automates the process of building Docker images and pushing them to Docker Hub upon every code commit. This automation ensures that application updates are continuously built, tested, and deployed without manual intervention.

For infrastructure provisioning, Terraform scripts automate the setup of AWS services such as ECS, EC2, and IAM roles, making the cloud deployment process reproducible and scalable.Monitoring and centralized logging are handled through Amazon CloudWatch, enabling real-time visibility into system and application performance.Security is enforced through Docker image scanning using Trivy, ensuring that container images are free from known vulnerabilities before deployment.

This project demonstrates a complete end-to-end DevOps workflow that combines containerization, continuous integration and delivery, infrastructure as code, monitoring, and security. As a result, TechNova can deploy faster, reduce errors, scale efficiently, and maintain high application reliability.

1. Title of the Project

Technova Devops Pipeline

2. Introduction

2.1 History

Traditionally, software deployment was done manually — setting up servers, copying files, and configuring environments by hand. This often led to **inconsistencies**, **errors**, and **deployment delays**.

With growing application complexity, the industry shifted toward **DevOps practices**, which introduced automation, containerization, and continuous delivery to make deployments faster and more reliable.

Our project reflects this modern approach by implementing a **DevOps pipeline** using tools like **GitHub Actions, Docker, and Terraform**, helping automate code integration, infrastructure provisioning, and deployment processes for a web application.

2.2 Problem Statement

TechNova Solutions, a growing e-commerce startup, faces frequent deployment issues due to manual server configuration, inconsistent environments, and the absence of automated infrastructure provisioning and monitoring tools.

---

3. Objective

3.1 Main Objective

To design and implement a fully automated CI/CD pipeline for a Node.js web application using GitHub Actions, Docker, and Terraform. The goal is to provision infrastructure on AWS, containerize the application, and enable automatic deployment on every code commit.

3.2 Sub-Objectives

- Containerize a Node.js application using Docker.
- Set up GitHub Actions to automate Docker builds and push to Docker Hub.

- Write Terraform scripts to provision EC2 instances and required AWS resources.
- Integrate application deployment into the CI/CD workflow.
- Integrate Trivy into GitHub Actions for container image vulnerability scanning.
- Implement Blue-Green and Canary deployment strategies with rollback capability.
- Use SonarCloud with GitHub Actions for continuous code quality checks.

---

4. System Analysis

4.1 Existing System

The existing deployment approach required manual setup of servers and hand-written shell scripts, making updates risky and non-repeatable. Developers needed to manually install dependencies, configure web servers, and restart applications—leading to inconsistent environments and high error rates.

4.2 Gaps Identified

The current system lacks automation, leading to several critical gaps:

- **Inconsistency in Environments**: Manual setup results in variations between development, testing, and production environments, causing deployment failures.
- **High Downtime and Error Rates**: The absence of standardized processes increases the risk of errors during updates, leading to prolonged downtime.
- **Limited Scalability**: Manual dependency management and server configuration hinder the ability to scale infrastructure efficiently.
- **Low Repeatability**: Hand-written scripts lack version control and documentation, making it difficult to replicate deployments reliably.
- **Developer Overhead**: Significant time is spent on repetitive manual tasks, reducing productivity and release velocity.

These gaps highlight the need for an automated, standardized, and scalable deployment process to address the inefficiencies of the existing system.

4.3 Motivation

To minimize downtime, reduce manual overhead, and standardize environments, the team decided to automate the entire deployment cycle—from code push to production deployment—using modern DevOps tools. This approach improves release velocity, consistency, and team productivity.

4.4 Proposed System

The proposed system introduces the following tools and technologies to achieve a robust, secure, and automated DevOps pipeline:

- **GitHub** – Used for version control and workflow automation, enabling collaborative development and traceable code management.
- **GitHub Actions** – Powers the CI/CD pipeline, automating tasks such as code checkout, Docker image building, vulnerability scanning, and deployment.
- **Docker** – Ensures consistency across environments by packaging the application and its dependencies into lightweight, portable containers.
- **Terraform** – Used to provision AWS infrastructure (ECS, EC2, IAM roles, and networking) in a reproducible and scalable manner through Infrastructure as Code (IaC).
- **AWS EC2 & ECS** – Serve as the primary deployment platforms. ECS handles container orchestration, while EC2 supports infrastructure flexibility.
- **Amazon CloudWatch** – Provides centralized monitoring and logging to observe system health, track application logs, and receive alerts in case of anomalies.
- **SonarQube (via SonarCloud)** – Performs static code analysis during CI to identify bugs, vulnerabilities, and maintain code quality throughout the development lifecycle.

- **Trivy** – Integrated into the CI pipeline to perform Docker image vulnerability scanning, enhancing container security before deployment.

- **Blue-Green and Canary Deployment** – Implemented using ECS and Load Balancers to enable safe, zero-downtime deployments with traffic splitting and rollback capabilities.

---

5. Design

5.1 UI/UX Design

A basic Node.js web application serves the frontend of our website, featuring a welcome page with navigation to 2–3 additional pages. While simple, it effectively demonstrates multi-page routing and supports our CI/CD pipeline testing and deployment.

5.2 Use Case Flow

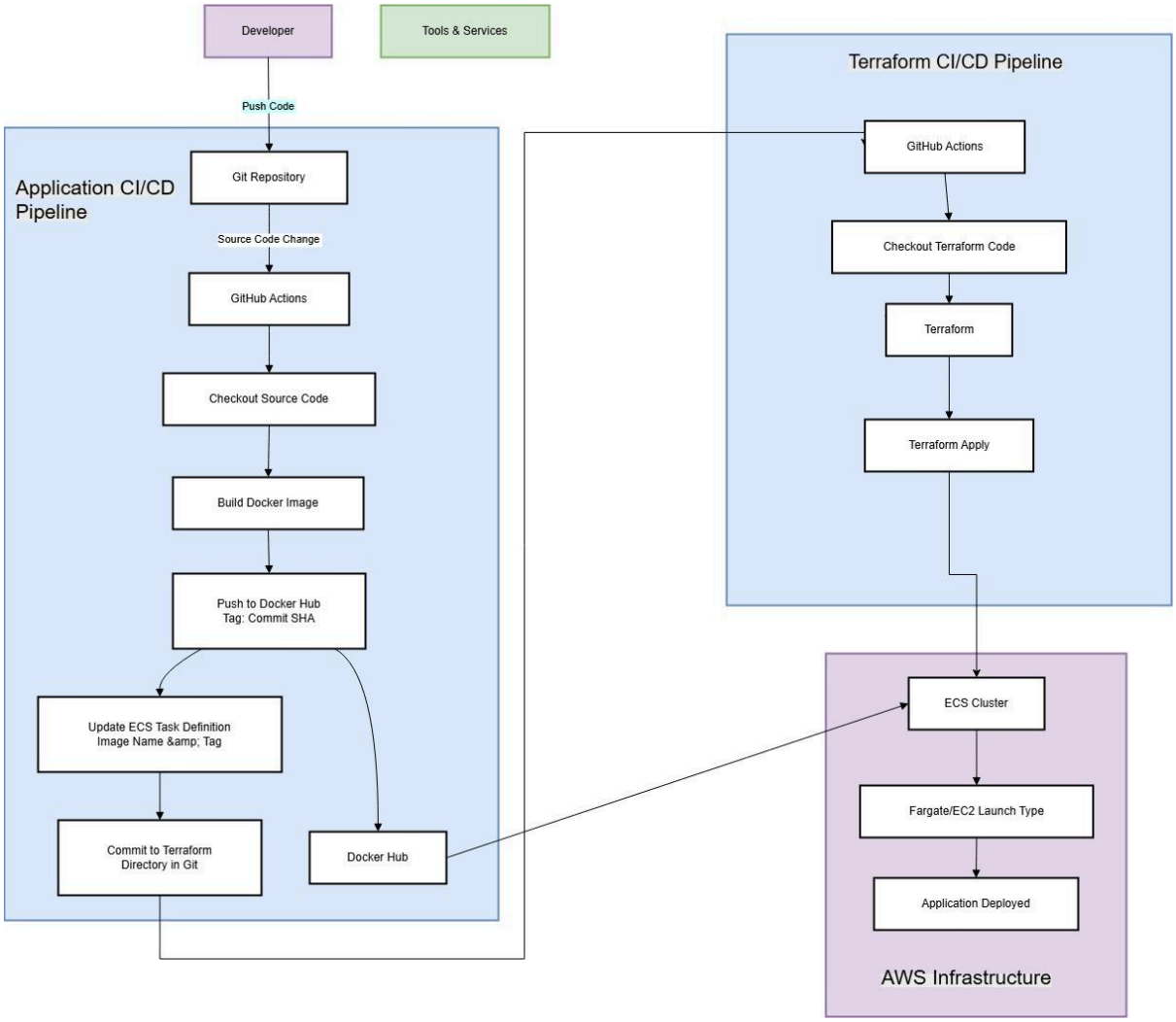
- **Code Push to GitHub**
  Developer pushes the latest code changes to the `main` branch.

- **CI/CD Workflow Triggered**
  GitHub Actions runs CI steps: code checkout, SonarCloud scan, Trivy scan, Docker build, and image push to Docker Hub.

- **Infrastructure Provisioning via Terraform**
  Terraform provisions ECS clusters, networking, IAM roles, and load balancers on AWS.

- **Deployment to ECS**
  GitHub Actions updates the ECS service using the new Docker image, enabling Blue-Green or Canary deployment.

- **Monitoring with CloudWatch**
  CloudWatch collects logs and metrics for real-time performance monitoring and alerting.



---

6. Technical Architecture

6.1 Frontend Layer

The Node.js app serves a static HTML page:

- Built with Express.js
- Styled with plain CSS for simplicity
- Demonstrates runtime web hosting within Docker

6.2 CI/CD Layer

GitHub Actions automates:

- Code checkout from the main branch
- Static code analysis using **SonarCloud**
- Docker image scanning with **Trivy**
- Docker image build and push to **Docker Hub**
- Deployment to **AWS ECS** using Blue-Green or Canary strategy
- Secrets (Docker credentials and tokens) are securely managed in GitHub

6.3 Containerization Layer

- Dockerfile packages the Node.js application and its dependencies
- Containers ensure consistent execution across all environments
- Images are version-tagged and stored in Docker Hub

6.4 Infrastructure Layer

Terraform provisions:

- **AWS ECS Cluster**, Task Definitions, and Load Balancer setup
- **Security Groups**, VPC, and subnets for networking
- **IAM roles and policies** for access control
- All infrastructure is reproducible, scalable, and version-controlled

---

7. Implementation

7.1 Dockerization

- Dockerfile defines how to build and run the Node.js app
- Image tested locally and via CI to ensure consistency

7.2 GitHub Actions Workflow

- Triggered on every push to the `main` branch
- Runs SonarCloud and Trivy scans
- Builds Docker image and pushes it to Docker Hub with a SHA-based tag
- Deploys the updated image to AWS ECS with Blue-Green or Canary rollout

7.3 Terraform Infrastructure Pipeline

- Terraform scripts define AWS ECS cluster, IAM roles, networking, and load balancer
- Provisioning is automated via GitHub Actions using the Terraform pipeline

7.4 Final Integrations

- ECS deployment fully automated and integrated into CI/CD
- SonarCloud performs static code analysis in every pipeline run
- Trivy scans Docker images for vulnerabilities pre-deployment
- CloudWatch handles centralized logging and application monitoring

---

8. Limitations and Future Enhancements

8.1 Current Limitations

The current system meets our project goals, with all core CI/CD, infrastructure, security, and quality tools integrated. Any further enhancements would focus on advanced application features or ML-based monitoring.

8.2 Future Goals and Enhancement

- **Implement Performance Testing Tools**
  Integrate tools like JMeter or k6 to perform automated load and stress testing.

- **Database Integration and Persistence**
  Extend the backend to include persistent data storage using RDS or DynamoDB, and automate backup strategies.

- **Advanced Monitoring and Alerts**
  Add anomaly detection, cost tracking, and automated alerting for SLA violations using CloudWatch Insights or third-party tools.

- **Custom Dashboard for DevOps Metrics**
  Create a real-time dashboard (e.g., with Grafana or CloudWatch Dashboards) to visualize CI/CD pipeline health, deploy status, and system performance.

---

9. Team Contribution

The development of the TechNova DevOps Pipeline was a collaborative effort, with each team member taking ownership of specific components to ensure a well-structured and end-to-end automated deployment system. The contributions are as follows:

Jayvardhan Singh
Designed and implemented the Dockerfile to containerize the Node.js application. He also developed the frontend interface, ensuring a functional and responsive web UI for deployment and testing.

Diya Tomar & Dhruv Joshi
Collaboratively built and implemented the CI/CD pipeline using GitHub Actions, automating the complete build, test, scan, and deployment process. They also developed the backend functionality by connecting the login and signup pages to a

database (e.g., MongoDB), enabling secure and persistent user data management. Additionally, they were responsible for preparing and organizing the project documentation and report.

Hardik Yadav
Led the implementation of the Application Load Balancer (ALB) using Terraform. He also integrated SonarCloud into the CI pipeline to enable static code analysis and maintain code quality standards.

Sidharth Maalpani
Focused on infrastructure security and networking, including VPC setup, Security Group configuration, and IAM role assignments to ensure secure and controlled deployment access.

Raman Boora
Managed the setup and configuration of the AWS ECS Cluster and its services. He also implemented Blue-Green and Canary deployment strategies and configured Terraform remote state management for collaborative infrastructure provisioning.

---

10. Conclusion

This project successfully established a robust, secure, and fully automated DevOps pipeline for the TechNova application. The Node.js application was containerized using Docker, and a complete CI/CD workflow—powered by GitHub Actions—automates code analysis, image scanning, building, and deployment.

Terraform was used to provision AWS infrastructure, including ECS clusters, networking components, IAM roles, and load balancers, ensuring scalable and reproducible deployments. The application is deployed to AWS ECS using Blue-Green and Canary strategies to enable safe, zero-downtime releases.

Monitoring and centralized logging are handled through Amazon CloudWatch, providing real-time insights into application performance. Static code analysis using SonarCloud and Docker image scanning with Trivy were integrated into the CI pipeline to maintain security and code quality.

This end-to-end solution enables TechNova to deploy faster, scale seamlessly, monitor effectively, and maintain high reliability with minimal operational overhead.
The project also helped the team gain practical experience with industry-grade DevOps tools and workflows, enhancing both technical and collaborative skills.

Overall, this solution is adaptable to real-world production environments and lays a strong foundation for future enterprise-grade DevOps implementations.