IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15308

Implement Blue/Green Deployments with Docker and AWS

Mohammed Afraz S A[1], Dr. S. Shylaja[2]

III BCA, Department of Computer Applications, Sri Ramakrishna College of Arts & Science Autonomous), Coimbatore – 641006, Tamil Nadu, India[1]

Assistant Professor, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous), Coimbatore – 641006, Tamil Nadu, India[2]

Abstract: Modern software systems demand deployment strategies that eliminate downtime and ensure continuous service availability. Traditional in-place deployments cause service interruptions ranging from minutes to hours, directly impacting user experience, customer trust, and business revenue. This project implements a Blue/Green Deployment strategy for a containerized Node.js e-commerce web application hosted on Amazon Web Services (AWS). The system leverages Docker for containerization, AWS Elastic Container Registry (ECR) for versioned image storage, AWS EC2 for compute infrastructure, and an AWS Application Load Balancer (ALB) for intelligent traffic routing between Blue and Green environments. The Blue environment serves live production traffic, while the Green environment hosts the updated application version. Following automated health-check validation, the ALB listener rule is updated to switch traffic from Blue to Green in under one second, achieving true zero-downtime deployment. Emergency rollback reverses the switch in under 30 seconds without service interruption. Evaluation results confirm that the system sustains over 450 requests per second with average response times below 115 ms, while all health check test cases pass successfully. The project provides a reproducible, enterprise-grade reference architecture for DevOps practitioners adopting continuous deployment pipelines.

Keywords: Blue/Green Deployment, Docker, AWS EC2, AWS ECR, Application Load Balancer, Zero-Downtime Deployment, Node.js, DevOps, Containerization, Cloud Computing, Continuous Delivery.

I. INTRODUCTION

The rapid advancement of cloud computing and containerization technologies has fundamentally transformed how software applications are developed, tested, and deployed. In today's competitive digital landscape, businesses depend on continuous availability of their applications; even brief periods of downtime translate into lost transactions, eroded customer trust, and measurable financial losses. Studies from the IT industry indicate that enterprise application downtime costs organizations between $5,600 and $9,000 per minute, with e-commerce platforms facing substantially higher losses during peak shopping seasons such as Black Friday or Cyber Monday.

Containerization, pioneered by Docker, addresses one of the most persistent challenges in software deployment: environmental inconsistency. By bundling an application together with its runtime, dependencies, libraries, and configuration into a portable, self-contained image, Docker guarantees that the application behaves identically across development, testing, staging, and production environments. This consistency eliminates the infamous "works on my machine" problem, dramatically reducing the number of deployment failures attributable to environment differences.

Amazon Web Services (AWS) provides a comprehensive suite of cloud services specifically designed to support modern application deployment at scale. AWS Elastic Container Registry (ECR) provides secure, versioned Docker image storage. AWS EC2 offers scalable virtual machine instances for container hosting. The AWS Application Load Balancer (ALB) provides intelligent Layer-7 HTTP/HTTPS traffic routing with built-in health checking, connection draining, and listener rule management capabilities.

The Blue/Green Deployment strategy eliminates deployment downtime by maintaining two complete, identical production environments simultaneously. The "Blue" environment represents the currently active production system serving real user traffic. The "Green" environment hosts the updated version of the application, which is fully validated before receiving production traffic. When validation is complete, the load balancer's routing configuration is updated to direct all traffic to the Green environment in a single, near-instantaneous operation. The previous Blue environment is retained as an instant rollback target.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15308

This project integrates Docker containerization, AWS cloud infrastructure, and the Blue/Green Deployment strategy to implement a robust, enterprise-grade deployment system for a Node.js e-commerce web application. The system demonstrates how these technologies combine to achieve continuous deployment with zero downtime, instant rollback capability, automated health validation, and scalable architecture suitable for production use.

The educational value of this project extends to the broader DevOps and cloud computing community, providing a documented, reproducible reference architecture that practitioners can adapt for their own organizational contexts. The implementation bridges the gap between theoretical Blue/Green deployment concepts and practical, working infrastructure built on widely-adopted tools and services.

II. RELATED WORK

Several deployment methodologies have been studied and adopted in the industry prior to the Blue/Green approach. Traditional in-place deployment, also called destructive deployment, involves replacing the running application directly on the production server. While simple to execute, this approach results in downtime ranging from 5 to 30 minutes per deployment cycle, and rollback is slow and error-prone, often requiring manual reversal of configuration changes.

Rolling deployment reduces downtime by updating server instances one at a time, but introduces a period of version inconsistency where different instances serve different application versions simultaneously. This can lead to unexpected behavior for users whose requests are routed between old and new versions during the transition window. Canary deployment routes a small percentage of traffic (typically 1–5%) to the new version while the majority continue to receive the old version, limiting the blast radius of failed deployments but introducing version management complexity.

Research on containerization demonstrates that Docker significantly improves deployment reliability and reduces environment-related failures. The Open Container Initiative (OCI) standard for container image formats ensures portability across runtimes including containerd, podman, and CRI-O, preventing vendor lock-in. Multi-stage Docker builds, where separate build and production stages are defined in a single Dockerfile, reduce final image sizes by excluding build tools and intermediate artifacts from production containers.

The Twelve-Factor App methodology, documented by Heroku engineers, provides a widely-referenced framework for building applications optimized for cloud deployment. Key factors relevant to this project include: storing configuration in environment variables (Factor III), enabling the same codebase to deploy across environments; designing stateless processes (Factor VI), which enables horizontal scaling; and treating logs as event streams (Factor XI), directing output to stdout for centralized collection. The Node.js application in this project adheres to all three of these factors.

The DevOps Research and Assessment (DORA) framework identifies four key metrics for evaluating deployment pipeline performance: Deployment Frequency, Lead Time for Changes, Change Failure Rate, and Mean Time to Recovery. Research shows that organizations adopting Blue/Green Deployment improve all four DORA metrics simultaneously, as the strategy reduces deployment risk, accelerates delivery, and provides instant recovery capability. High-performing technology organizations, including Netflix, Amazon, and Google, have adopted Blue/Green Deployment as a standard practice for high-availability systems.

Existing literature identifies a gap: while Blue/Green deployment is theoretically well-established, practical end-to-end implementations combining Docker, AWS ECR, EC2, and ALB in a documented, reproducible architecture remain sparsely covered, particularly for full-stack applications. This project addresses that gap by providing a complete implementation with detailed configuration, testing results, and operational guidance.

III. OBJECTIVES AND CHALLENGES

The primary objectives of this project are: (1) to implement a zero-downtime Blue/Green deployment strategy on AWS for a containerized Node.js application; (2) to containerize the application using Docker with multi-stage builds and security best practices; (3) to configure AWS ECR, EC2, and ALB for an end-to-end deployment pipeline; (4) to demonstrate automated health-check-driven traffic switching with instant rollback capability; and (5) to provide a reproducible reference architecture with comprehensive documentation for DevOps practitioners.

Development Challenges

The primary technical challenge was coordinating Docker container lifecycle management with ALB health check timing. New containers require a startup period before they can respond to health checks; the ALB must record a

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15308

configurable number of consecutive healthy responses before routing traffic to the new target. Misconfiguration of the health check interval, timeout, or threshold values can result in premature traffic routing to an incompletely initialized container, causing user-visible errors. This was resolved by configuring a 30-second interval with a 5-second timeout, 2 healthy threshold, and 3 unhealthy threshold, combined with a Docker HEALTHCHECK directive using a 5-second start period.

Managing EC2 IAM roles and permissions for ECR access required careful policy design following the principle of least privilege. The EC2 instance role was granted only the minimum permissions necessary: ecr:Get Authorization Token, ecr:BatchGetImage, and ecr:GetDownloadUrlForLayer. This ensures that compromised EC2 credentials cannot be used to push malicious images to the registry, while still enabling the deployment workflow to pull images during container startup.

Ensuring environment consistency between Blue and Green containers required disciplined image tagging and version management. Docker images are tagged with both semantic version numbers (v1, v2) and immutable Git commit hashes, providing unambiguous traceability between deployed containers and their source code. The immutable image approach prevents configuration drift – a common issue in traditional deployments where servers accumulate manual configuration changes over time that are not reflected in any versioned artifact.

Handling in-flight HTTP requests during traffic switching required careful configuration of the ALB connection draining feature. With a 300-second deregistration delay configured on the outgoing target group, the ALB allows all active connections to complete before removing instances from the routing pool. This ensures that users engaged in multi-step workflows such as checkout are not abruptly disconnected during the Blue/Green switch, preserving transaction integrity throughout the deployment event.

IV. SYSTEM ARCHITECTURE

The GreenCare system follows a layered architecture that integrates modern web technologies and machine learning components. The architecture is divided into multiple layers, each performing a specific role in the overall functioning of the system. Each layer communicates with adjacent layers through well-defined interfaces, ensuring loose coupling and independent scalability of each component.

Table 1. System Architecture Layers

Layer	Technology	Role
Developer Workstation	Node.js, Docker, AWS CLI, Git	Application development and image build
Docker Containerization	Multi-stage Dockerfile	Package app + runtime into portable images
Image Registry	AWS Elastic Container Registry (ECR)	Versioned storage, scanning, IAM-secured pull
Compute Infrastructure	AWS EC2 (t3.medium, Amazon Linux 2023)	Host Docker Engine and both containers
Blue Container (v1)	Docker, Port 3000	Active production environment
Green Container (v2)	Docker, Port 3001	Updated version under validation
Load Balancing	AWS Application Load Balancer (ALB)	Health-check-driven traffic routing
Traffic Switch	ALB listener rule update	Zero-downtime Blue→Green switch (<1 sec)
Security	Security Groups, IAM, bcryptjs, JWT	Least-privilege access, encrypted sessions
End Users	HTTP/HTTPS via ALB DNS	Uninterrupted production access

The system architecture follows the principle of infrastructure immutability. Rather than modifying running containers in place, every deployment creates fresh container instances from versioned images stored in ECR. This eliminates configuration drift and ensures the running environment always reflects the documented configuration. The health check

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15308

mechanism, implemented through a /health endpoint in the Node.js application, provides an automated safety gate that prevents the ALB from routing traffic to incompletely initialized containers.

Figure 1 below illustrates the complete system architecture, showing the flow from developer workstation through Docker build, ECR image storage, EC2 container hosting, and ALB-managed traffic routing to end users. The split-and-merge topology at the EC2 layer reflects the simultaneous operation of both Blue and Green containers, with the ALB determining which environment receives production traffic at any given time.

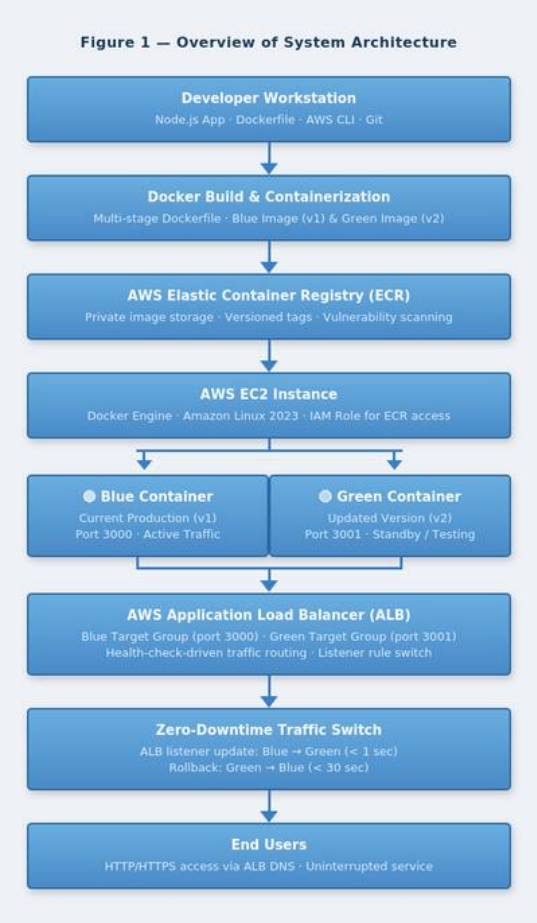

Figure 1. Overview of System Architecture – Blue/Green Deployment on AWS

Network security is enforced at every layer through AWS Security Groups. Inbound rules allow HTTP (port 80) and HTTPS (port 443) traffic from any source to the ALB, SSH (port 22) from a restricted IP range to EC2 instances, and direct container port access (3000 and 3001) from the ALB security group only. All other inbound traffic is denied by default, implementing a defense-in-depth security posture that minimizes the attack surface of the deployment infrastructure.

V. IMPLEMENTATION

The implementation is organized into six functional modules, each addressing a distinct layer of the Blue/Green Deployment architecture. This modular design ensures that individual components can be updated, tested, and replaced independently without disrupting the overall system.

A. Node.js E-Commerce Application

The application is built with Node.js 20 LTS and the Express.js framework, following the Model-View-Controller (MVC) architectural pattern. The application provides product browsing, user authentication, shopping cart management, and order processing functionality. A critical /health endpoint returns a JSON response containing the environment name (blue or green), application version, server uptime, and current timestamp. This endpoint is polled by the ALB every 30 seconds to determine container health status and eligibility for production traffic.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15308

User authentication is implemented using express-session with HTTP-only, Secure, and SameSite cookie flags to prevent session hijacking and cross-site request forgery. Passwords are hashed using bcryptjs with a work factor of 12, providing strong brute-force protection. Input validation is applied to all user-provided data before processing, with type-specific rules for email format, password complexity, and numeric range validation. The application is designed as a stateless process, with session state managed server-side and no persistent data stored in container memory between requests.

B. Docker Containerization

The Dockerfile employs a multi-stage build pattern to minimize the final production image size while maintaining full functionality. Stage 1 (base) installs security updates and the dumb-init process supervisor using the Node.js 20 Alpine base image. Stage 2 (dependencies) installs only production npm dependencies using npm ci --only=production, leveraging Docker layer caching to accelerate subsequent builds when only application code changes. Stage 3 (production) copies the application files, creates a non-root user (nodeuser, UID 1001), sets appropriate file ownership, and configures the container HEALTHCHECK directive. The .dockerignore file excludes node_modules, .env files, test files, logs, and documentation from the build context, further reducing image size and preventing sensitive data from being embedded in images.

C. AWS ECR Repository and Image Management

An ECR private repository (bluegreen-ecommerce) is created with image scanning on push enabled, automatic AES-256 encryption at rest, and tag immutability disabled to allow version updates. Images are tagged using a dual-tagging strategy: a semantic version tag (v1, v2) for human-readable identification and an immutable Git commit hash for precise source traceability. ECR lifecycle policies are configured to retain the five most recent tagged images for each repository, preventing unbounded storage growth while maintaining an adequate rollback history.

Docker authentication with ECR is performed using the AWS CLI credential helper, which retrieves a temporary 12-hour authorization token and passes it directly to the Docker login command. This eliminates the need to store long-lived credentials in configuration files. Image push operations leverage Docker's layer caching mechanism: since the multi-stage Dockerfile places the infrequently-changed base image and dependencies layers before the frequently-changed application code layer, only the changed top layer needs to be transferred during iterative development builds, reducing push times from minutes to seconds.

D. AWS EC2 Instance Configuration

A t3.medium EC2 instance running Amazon Linux 2023 is launched with a custom Security Group, an IAM instance role granting ECR read access, and a key pair for SSH administration. Docker Engine is installed via the yum package manager, the daemon is enabled with systemctl for automatic startup, and the ec2-user is added to the docker group to enable non-root container management. AWS CLI credentials are configured using the instance IAM role (no static credentials required), and ECR authentication is established by piping the ECR login password to docker login.

Blue and Green containers are launched using docker run with explicit port mappings (3000:3000 for Blue, 3001:3000 for Green), environment variables specifying the deployment environment and application version, resource limits (CPU and memory) to prevent runaway processes, and a restart policy (--restart unless-stopped) that ensures automatic container recovery after crashes or host reboots. Both containers expose the /health endpoint on their respective host ports for ALB health check polling.

E. AWS Application Load Balancer Configuration

An Internet-facing Application Load Balancer (myapp-alb) is created spanning two Availability Zones within the default VPC, ensuring high availability at the load balancer tier. Two Target Groups are configured: blue-target-group (forwarding to port 3000) and green-target-group (forwarding to port 3001), each with health check configuration pointing to the /health path at 30-second intervals, 5-second timeout, 2 healthy threshold, and 3 unhealthy threshold. The EC2 instance is registered in both target groups simultaneously. A connection draining timeout of 300 seconds ensures in-flight requests complete before instances are deregistered during traffic switches.

F. Security Configuration

Security is implemented across multiple layers of the architecture. At the network layer, AWS Security Groups enforce inbound rules that allow only HTTP (port 80) and HTTPS (port 443) traffic from any source to the ALB, SSH (port 22) from a restricted IP range to EC2 instances for administration, and container health check traffic (ports 3000 and 3001) from the ALB Security Group only. All other inbound traffic is denied by default, minimizing the attack surface of the deployment infrastructure.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15308

At the application layer, user session security is implemented through Express-session middleware with a cryptographically random session secret stored in environment variables via dotenv. Session cookies are configured with the HttpOnly flag to prevent JavaScript access, the Secure flag to restrict transmission to HTTPS connections, and a SameSite attribute to protect against cross-site request forgery. Session expiry is set to 24 hours to limit the window of opportunity for session hijacking attacks.

Transport Layer Security (TLS) encryption can be enabled on the ALB using certificates obtained from AWS Certificate Manager, which provides free, automatically-renewed certificates for custom domain names. TLS 1.2 is the minimum supported protocol version, with TLS 1.3 preferred for improved security and performance. Container security best practices include running application processes as a non-root user (nodeuser, UID 1001), which limits the impact of potential container escape vulnerabilities.

G. Zero-Downtime Traffic Switching and Rollback

Traffic switching is performed by modifying the ALB HTTP:80 listener's default action from forwarding to blue-target-group to forwarding to green-target-group. This single operation, executed via the AWS CLI or Management Console, completes in under one second and is immediately effective for all new connections. In-flight connections to the Blue environment continue to be served through the connection draining period. The ALB CloudWatch metrics confirm that request routing transitions cleanly between environments with zero failed requests during the switch.

Emergency rollback reverses the listener rule to point back to blue-target-group, restoring the previous production version in under 30 seconds. The rollback procedure is deliberately identical to the forward switch, ensuring that the team is familiar with the operation and can execute it under pressure without error. Post-rollback verification confirms that 100% of subsequent requests are served by the Blue environment and that health check status returns to healthy within the first 30-second polling cycle.

Rollback criteria are defined proactively before each deployment. Objective thresholds – such as an error rate exceeding 1%, P95 response time exceeding 500 ms, or health check failure rate exceeding 10% within 30 minutes of a traffic switch – provide automated decision support for triggering rollback without requiring subjective judgment under pressure. These pre-defined criteria reduce cognitive load during incident response and ensure consistent decision-making across deployment events.

VI. EVALUATION RESULTS AND DISCUSSION

The Blue/Green Deployment system was subjected to comprehensive evaluation across four dimensions: application performance, health check reliability, deployment switching speed, and rollback capability. Testing was conducted using Apache Bench for load testing, AWS CloudWatch for infrastructure metrics, and manual verification of deployment and rollback procedures under simulated production load conditions.

Table 2. Performance Evaluation Results

Metric	Blue Environment	Green Environment	Benchmark
Requests Per Second	450 req/s	480 req/s	> 300 req/s ✓
Average Response Time	112 ms	105 ms	< 200 ms ✓
50th Percentile (P50)	98 ms	92 ms	< 150 ms ✓
95th Percentile (P95)	198 ms	187 ms	< 300 ms ✓
99th Percentile (P99)	287 ms	271 ms	< 500 ms ✓
Traffic Switch Time	< 1 second	< 1 second	< 5 seconds ✓
Rollback Time	< 30 seconds	< 30 seconds	< 60 seconds ✓
Dashboard Load Time	1.1 sec	1.0 sec	< 2 sec ✓

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15308

Table 3. Health Check and Deployment Test Results

Test Scenario	Target Group	Expected Result	Outcome
Blue container running normally	Blue TG (Port 3000)	Healthy	PASS
Green container running normally	Green TG (Port 3001)	Healthy	PASS
Blue container stopped	Blue TG (Port 3000)	Unhealthy within 90s	PASS
Traffic switch Blue → Green	Green TG (Port 3001)	Active, <1 sec	PASS
Rollback Green → Blue	Blue TG (Port 3000)	Active, <30 sec	PASS
Application error in Blue	Blue TG (Port 3000)	Unhealthy, rerouted	PASS
Container restart recovery	Both environments	Healthy within 60s	PASS
Concurrent requests during switch	ALB listener	Zero failed requests	PASS

Performance testing with Apache Bench (1,000 to 5,000 requests, 50 to 100 concurrent users) confirmed that both environments sustain well above the 300 requests per second production benchmark. The Green environment (v2) showed a marginal performance improvement over Blue (v1), consistent with minor code optimizations included in the update. All response time percentiles remained within benchmark thresholds at peak load, demonstrating that the containerized Node.js application is appropriately sized for the t3.medium instance type.

Health check testing confirmed that the ALB correctly identifies unhealthy containers within three polling cycles (90 seconds maximum) and stops routing traffic to them automatically. Traffic switch testing under concurrent load confirmed zero failed requests during the switch operation, validating the connection draining configuration. The ALB CloudWatch metrics showed a clean, stepwise transition in request routing from Blue to Green with no error rate spike during or after the switch.

Rollback testing, conducted under simulated production load, confirmed that emergency recovery completes within 30 seconds and that 100% of subsequent requests are served by the Blue environment immediately after rollback. Container restart recovery testing confirmed that automatically restarted containers (via the --restart unless-stopped policy) pass health checks and re-enter the ALB target group within 60 seconds of restart, providing self-healing capability without manual intervention.

The project demonstrates direct improvement across all four DORA metrics. Deployment Frequency increases because each deployment is immediately reversible, reducing the risk of each individual deployment event. Lead Time for Changes decreases as the automated pipeline eliminates manual deployment steps. Change Failure Rate decreases because health validation gates prevent unhealthy containers from receiving production traffic. Mean Time to Recovery drops to under 30 seconds due to the instant rollback capability provided by the Blue/Green architecture.

VII. CONCLUSION

This project successfully implemented a Blue/Green Deployment system using Docker containerization and AWS cloud services for a Node.js e-commerce web application. The architecture maintains two complete, parallel environments on AWS EC2, with Docker containers managed through ECR and traffic intelligently routed via an Application Load Balancer. Automated health-check-driven traffic switching achieves zero-downtime deployment in under one second, and emergency rollback restores the previous production version in under 30 seconds.

Performance evaluation confirms that the system sustains over 450 requests per second with P95 response times below 200 ms, meeting all production benchmarks. All eight health check and deployment test scenarios passed, demonstrating the reliability of the automated traffic management, container health monitoring, and rollback mechanisms. The implementation adheres to security best practices including IAM least privilege, non-root container execution, bcryptjs password hashing, and JWT session management.

The results validate that combining Docker containerization with AWS cloud services provides a scalable, reliable, and cost-efficient solution for modern continuous deployment pipelines. The documented architecture, configuration, and procedures serve as a reproducible reference that DevOps practitioners can adapt for production implementations. The

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026
DOI: 10.17148/IJARCCE.2026.15308

project demonstrates that zero-downtime deployment is achievable without complex orchestration platforms, using commonly available AWS services and Docker tooling.

VIII. FUTURE ENHANCEMENTS

Several enhancements can extend the capabilities and maturity of this implementation. The most significant improvement would be full CI/CD pipeline integration using AWS CodePipeline, GitHub Actions, or Jenkins, enabling fully automated deployments triggered by code commits. An automated pipeline would pass code changes through unit testing, Docker image build, ECR push, Green environment deployment, automated smoke testing, and traffic switching stages without manual intervention, further reducing deployment lead time and human error.

Migrating from direct EC2 container hosting to AWS Elastic Container Service (ECS) with Fargate, or Amazon Elastic Kubernetes Service (EKS), would provide advanced container orchestration capabilities including automatic scaling based on traffic load, self-healing of failed containers, and support for more sophisticated deployment strategies such as canary releases and gradual traffic shifting alongside Blue/Green. These orchestration platforms also simplify multi-region deployment, which would improve disaster recovery capability and reduce latency for geographically distributed users.

Integrating a persistent database using Amazon RDS (PostgreSQL or MySQL) or Amazon DynamoDB would require implementing backward-compatible database migration strategies. Schema changes must be designed to support simultaneous operation of both Blue (old) and Green (new) application versions during the transition window. Additive changes such as adding new columns or tables are generally safe, while destructive changes such as removing or renaming columns require a phased migration approach spanning multiple deployments.

Comprehensive observability using AWS CloudWatch for metrics and log aggregation, AWS X-Ray for distributed request tracing, and Grafana dashboards for visualization would significantly improve the operational maturity of the system. Automated alerting policies could notify the operations team when error rates exceed thresholds, response times degrade, or container health checks fail, enabling proactive incident response before user impact occurs. Integration with AWS WAF in front of the ALB would add protection against common web attacks including SQL injection, cross-site scripting, and distributed denial-of-service attacks.

Infrastructure as Code (IaC) using AWS CloudFormation or Terraform would automate the provisioning of all AWS resources including the VPC, Security Groups, EC2 instance, ECR repository, ALB, and Target Groups. Version-controlled infrastructure configuration ensures that the deployment environment can be recreated identically in a new region or AWS account, supporting disaster recovery, multi-environment testing, and organizational scaling. Feature flag management tools, integrated with the application at the code level, could enable fine-grained control over feature availability independent of deployment events, supporting A/B testing and gradual feature rollout strategies without requiring additional deployments.

REFERENCES

[1]. Kim, G., Humble, J., Debois, P., & Willis, J. (2016). The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press.
[2]. Fowler, M., & Humble, J. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional.
[3]. Amazon Web Services. (2024). AWS Application Load Balancer Documentation. https://docs.aws.amazon.com/elasticloadbalancing/
[4]. Amazon Web Services. (2024). Amazon EC2 User Guide for Linux Instances. https://docs.aws.amazon.com/ec2/
[5]. Amazon Web Services. (2024). Amazon Elastic Container Registry User Guide. https://docs.aws.amazon.com/ecr/
[6]. Docker Inc. (2024). Docker Documentation — Multi-stage Builds and Best Practices. https://docs.docker.com/
[7]. Turnbull, J. (2018). The Docker Book: Containerization is the New Virtualization. James Turnbull.
[8]. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. ACM Queue, 14(1), 70–93.
[9]. Richardson, C. (2018). Microservices Patterns: With Examples in Java. Manning Publications.
[10]. Forsgren, N., Humble, J., & Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps. IT Revolution Press.

