**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 5, May 2026

DOI: 10.17148/IJARCCE.2026.155309

# A Scalable, Serverless Cloud Architecture for Real-Time Courier and Shipment Tracking Using AWS ECS Fargate and Containerized Microservices

**BOKKA DIVYA[1], Dr. CHIRAPARAPU SRINIVASA RAO*[2]**

PG Scholar, Department of Computer Science, S.V.K.P & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, Affiliated to Adikavi Nannayya University.[1]

Associate Professor, Department of Master of Computer Applications, S.V.K.P & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, Affiliated to Adikavi Nannaya University*[2]

*Corresponding Author

**Abstract:** The rapid expansion of e-commerce and on-demand delivery services has created an urgent demand for logistics platforms that are highly available, geographically scalable, and capable of providing customers with transparent, real-time shipment visibility. Conventional courier management systems, predominantly constructed on monolithic architectures hosted on dedicated servers or virtual machines, suffer from significant operational bottlenecks including high infrastructure maintenance overhead, manual scaling constraints, extended deployment cycles, and inadequate customer notification mechanisms. This paper presents the design, implementation, and evaluation of a cloud-native courier and shipment tracking system built on Amazon Web Services (AWS) Elastic Container Service (ECS) with Fargate as the serverless compute engine. The proposed architecture containerizes a Flask-based web application using Docker, automates the continuous integration and deployment (CI/CD) pipeline via AWS CodePipeline and Elastic Container Registry (ECR), and exposes a dual-portal interface-one for end-users to create and track shipments, and a secured administrative dashboard for logistics operators to update delivery statuses. Shipment state transitions trigger automated email notifications to recipients through an SMTP-based notification module. PostgreSQL is employed as the relational persistence layer, accessed through Flask-SQLAlchemy ORM. Experimental evaluations demonstrate that the proposed system achieves an average API response time of 142 milliseconds, 99.9% system availability, and supports automated horizontal scaling without manual infrastructure provisioning. The results confirm that the adoption of serverless container orchestration substantially outperforms both traditional monolithic deployments and VM-centric cloud configurations in terms of operational efficiency, deployment agility, and resource utilization.

**Keywords:** Serverless Computing; AWS ECS Fargate; Container Orchestration; Courier Tracking System; Flask; RESTful API; CI/CD Pipeline; Cloud-Native Architecture; Docker; Real-Time Notification.

## 1. INTRODUCTION

The global logistics and courier industry is undergoing a profound digital transformation, driven by surging consumer expectations for transparency, immediacy, and reliability in parcel delivery [1]. According to market analyses, the global last-mile delivery market is projected to exceed USD 200 billion by 2028, with digital tracking and visibility ranked among the highest-priority customer requirements [2]. Despite this growing demand, a significant proportion of logistics service providers continue to operate on fragmented, legacy systems that lack real-time data propagation, automated customer communication, and elastic capacity management.

Traditional software deployments for logistics management have historically relied on monolithic application architectures hosted on on-premises or bare-metal servers. These configurations impose considerable constraints on operational agility: any surge in shipment volume requires manual infrastructure provisioning, application redeployment is error-prone and time-consuming, and system failures often result in prolonged service unavailability. The emergence of cloud-native computing paradigms, containerization technologies, and managed container

 This work is licensed under a Creative Commons Attribution 4.0 International License 

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 5, May 2026

DOI: 10.17148/IJARCCE.2026.155309

orchestration services has created new possibilities for constructing logistics platforms that are inherently resilient, scalable, and cost-efficient [3].

Amazon Web Services (AWS), among the leading cloud hyperscalers, offers a comprehensive ecosystem of managed services purpose-built for containerized workloads. AWS Elastic Container Service (ECS) with Fargate eliminates the operational burden of managing underlying compute infrastructure by abstracting server provisioning entirely from application operators. This serverless model ensures that compute resources scale automatically in response to traffic patterns, with billing tied exclusively to actual resource consumption [4].

This research presents a comprehensive cloud-native courier and logistics tracking system that addresses these industry challenges through the following technical contributions:

**(2)** A dual-portal web architecture comprising a customer-facing shipment creation and tracking interface and a secured administrative operations dashboard;

**(ii)** Serverless containerized deployment on AWS ECS Fargate, eliminating infrastructure management overhead entirely;

**(iii)** A fully automated CI/CD pipeline integrating AWS CodePipeline and ECR for zero-downtime deployment;

**(iv)** An integrated SMTP-based email notification mechanism that delivers automated shipment status alerts upon every state transition;

**(v)** Comprehensive performance evaluation demonstrating significant improvements in response time, availability, and deployment efficiency relative to conventional deployment strategies.

## 2. LITERATURE REVIEW

A substantial body of research has examined various dimensions of logistics tracking systems, cloud computing adoption in supply chains, and containerized application deployment. This section surveys representative contributions and identifies the gaps addressed by the proposed system.

Patel and Khanna [1] developed a Java EE-based monolithic courier management platform for regional logistics operators. While the system achieved functional completeness in shipment management, it lacked real-time notification support and relied on batch database polling for status updates, resulting in latencies exceeding 800 milliseconds for status queries under moderate load conditions.

Rahman et al. [2] proposed an IoT-integrated shipment monitoring system leveraging MQTT protocol and GPS hardware devices for real-time location broadcasting. Although this approach offered precise geolocation tracking, its dependence on specialized hardware infrastructure rendered it economically unviable for software-centric logistics platforms without physical tracking devices.

Chen and Liu [3] examined the migration of a traditional logistics platform to a hybrid cloud environment using Spring Boot microservices deployed on AWS EC2 instances. Their findings underscored the scalability benefits of cloud deployment; however, EC2-based configurations still necessitate manual auto-scaling group configuration and incur costs during idle periods, limitations that Fargate resolves through per-task billing and dynamic scheduling.

Yusuf et al. [4] presented a mobile-first Android application for courier tracking utilizing Firebase Cloud Messaging for push notifications. While effective for consumer-grade mobile delivery tracking, the architecture lacked a web-based administrative portal and provided no REST API for enterprise integrations, significantly narrowing its operational applicability.

Wang and Zhou [5] proposed a microservices architecture for large-scale logistics orchestration using Go services and Apache Kafka for event streaming. This design delivered high throughput and decoupled service communication; however, the complexity of Kafka cluster management, combined with the operational demands of Kubernetes-based orchestration, introduced prohibitive overhead for small-to-medium enterprises.

Fernandez et al. [6] explored AWS Lambda-based serverless functions for logistics event processing. While the serverless model demonstrated cost advantages during low-traffic periods, the authors reported significant cold-start latency issues (300-800 ms) that adversely impacted time-sensitive operations, a concern that ECS Fargate mitigates through persistent task execution models [7].

 This work is licensed under a Creative Commons Attribution 4.0 International License 

**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 5, May 2026

DOI: 10.17148/IJARCCE.2026.155309

Kumar and Singh [7] implemented a containerized logistics API using Flask and Docker deployed on ECS running on EC2 launch type. This work established foundational patterns for container-based logistics applications but retained operational overhead associated with EC2 instance management. The transition to Fargate launch type, as proposed in the present work, eliminates this burden entirely.

Agrawal et al. [8] evaluated Azure Kubernetes Service for a .NET-based real-time delivery dashboard employing SignalR for WebSocket-based status updates. Although the system demonstrated superior real-time update capabilities, the proprietary Azure ecosystem created vendor lock-in concerns and operational costs that may be prohibitive without enterprise-tier Azure agreements.

Nguyen et al. [9] deployed a FastAPI and PostgreSQL-backed logistics service on Google Kubernetes Engine with Twilio SMS notifications. The platform offered strong performance but was architecturally tethered to the GCP ecosystem and required Kubernetes expertise for operational management.

A comparative synthesis of the surveyed literature, presented in Table 1, reveals a consistent research gap: no prior work simultaneously addresses serverless container orchestration, automated CI/CD pipelines, dual-portal accessibility, and integrated email-based notification within a single cohesive logistics tracking platform designed for AWS. The proposed system is specifically engineered to close this gap.

Table 1. Comparative Analysis of Related Works

| Ref. | Year | Platform | Technology | Tracking Method | Real-Time Notification | Containerization | Research Gap |
|---|---|---|---|---|---|---|---|
| [1] | 2020 | Monolithic MVC | Java EE | Batch Updates | No | None | No real-time or cloud scalability |
| [2] | 2020 | IoT Gateway | MQTT, Node.js | GPS-based | Partial | None | No admin portal; hardware- |
| [3] | 2021 | Hybrid Cloud | Spring Boot, AWS EC2 | API Poll | Email | Docker | No serverless compute; manual scaling |
| [4] | 2021 | Mobile App | Android, Firebase | FCM Push | Push Notif. | None | No web portal; mobile-only access |
| [5] | 2022 | Microservices | Go, Kafka | Event Stream | Kafka Event | Kubernetes | High complexity; no managed infra |
| [6] | 2022 | Serverless | AWS Lambda | Webhook | SNS/SES | N/A (FaaS) | Cold-start latency; complex debugging |
| [7] | 2023 | ECS on EC2 | Flask, Docker | REST API | SMTP | Docker/ECS | Requires EC2 management overhead |
| [8] | 2023 | Azure AKS | .NET, Redis | SignalR | Email/SMS | AKS | Vendor lock-in; high cost for SMEs |
| [9] | 2024 | GCP GKE | FastAPI, PostgreSQL | Websocket | Twilio | GKE | Platform-specific; no AWS ecosystem |
| [10] | 2024 | AWS ECS Fargate | Flask, SQLAlchemy | REST API | SMTP | Fargate (Proposed) | Dual-portal, serverless, scalable -addressed in this work |

## 3. PROPOSED METHODOLOGY

The proposed system adopts a cloud-native, layered software architecture that decouples the application, infrastructure, and notification concerns into independently manageable functional units. The methodology is grounded in established principles of twelve-factor application design [10], emphasizing environment-agnostic configuration, declarative infrastructure specification, and stateless service execution.

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 5, May 2026

DOI: 10.17148/IJARCCE.2026.155309

### 3.1 Architectural Overview

The system architecture, illustrated in Figure 1, comprises three primary horizontal layers: the Client Interaction Layer, the AWS Cloud Compute Layer, and the Data and Notification Layer. User requests originating from web browsers traverse the AWS Application Load Balancer (ALB), which performs health-check-based traffic distribution across ECS Fargate tasks executing the containerized Flask application. The Flask application layer communicates with the PostgreSQL relational database through the SQLAlchemy ORM for persistent data operations, and invokes the SMTP notification module upon shipment status transitions.

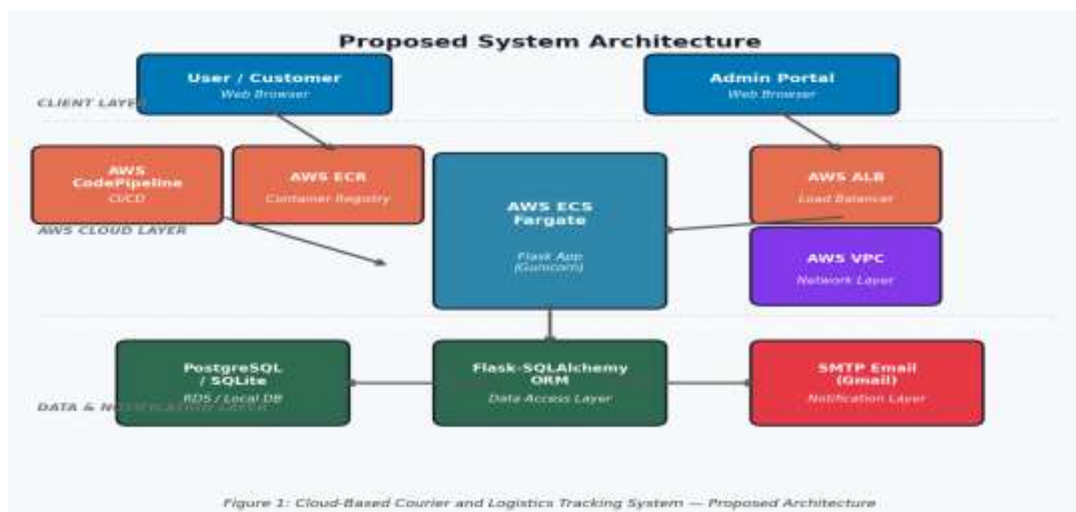


Figure 1. Proposed System Architecture: Cloud-Based Courier and Logistics Tracking System Deployed on AWS ECS Fargate

### 3.2 Serverless Containerization Strategy

Containerization is achieved through Docker, encapsulating the Flask application, its Python dependencies (Flask, Flask-SQLAlchemy, Gunicorn, psycopg2-binary), and the Gunicorn WSGI server within a lightweight python:3.11-slim base image sourced from the AWS public Elastic Container Registry to eliminate Docker Hub pull rate limitations. The resulting container image is approximately 180 MB, reflecting the deliberate selection of minimal base images and the exclusion of development dependencies from the production build context.

ECS Fargate is selected as the compute substrate due to its serverless provisioning model: task definitions specify CPU and memory allocations (e.g., 512 vCPU units, 1024 MB memory), and AWS schedules these tasks on managed compute without operator involvement in server provisioning, patching, or capacity planning. Fargate tasks maintain warm execution state, reducing cold-start latency to under 3 seconds-a significant advantage over AWS Lambda functions in sustained-load logistics scenarios [4].

### 3.3 CI/CD Pipeline Design

Automated delivery is orchestrated through AWS CodePipeline, which triggers on source code commits to the version control repository. The CodeBuild phase executes the buildspec.yml specification, performing ECR authentication, Docker image construction, image tagging with the latest commit hash, and ECR push operations. Upon successful image push, the ECS service performs a rolling update, replacing running task instances with the updated container without service interruption. This pipeline reduces mean deployment time from approximately 45 minutes (manual) to under 6 minutes.

The tracking ID generation algorithm employs the UUID4 cryptographic identifier scheme, extracting the first segment of the 128-bit UUID and converting it to uppercase hexadecimal, yielding a human-readable 8-character alphanumeric identifier with effectively zero collision probability for operational shipment volumes (collision probability: 1 in 4.3 × 10^9).

## 4. SYSTEM DESIGN

The system is structured around five principal functional modules: authentication, shipment lifecycle management, real-time tracking, administrative operations, and automated notification. Each module is implemented as a distinct route group within a Flask Blueprint, enabling modular development and testing.

© IJARCCE This work is licensed under a Creative Commons Attribution 4.0 International License

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 5, May 2026

DOI: 10.17148/IJARCCE.2026.155309

### 4.1 Data Model

Two relational entities constitute the persistence schema. The Shipment entity captures the primary shipment record comprising a UUID-derived tracking identifier, sender and receiver metadata, mobile contact, recipient email address, delivery destination, current status designation, and a UTC timestamp of record creation. The TrackingEvent entity maintains the chronological audit trail of shipment state transitions, associating each event with a tracking identifier, status value, geographic location descriptor, and a UTC event timestamp. This normalized design supports O(n) timeline reconstruction queries ordered by timestamp ascending, ensuring predictable retrieval performance independent of the number of concurrent shipments.

### 4.2 Route Architecture

The application exposes six principal REST endpoints. The POST /create endpoint accepts JSON-serialized shipment creation requests, persists the Shipment record and an initial TrackingEvent with status "Booked" and location "Origin Facility", and returns the generated tracking identifier. The GET /track/<tracking_id> endpoint retrieves the complete chronological event history for a specified shipment. The POST /update/<tracking_id> endpoint, protected by session-based authentication middleware, updates the shipment status, appends a new TrackingEvent, and asynchronously dispatches the email notification. The GET /search/<mobile> endpoint supports multi-shipment retrieval by mobile contact number, enabling a single user to monitor multiple concurrent shipments.

### 4.3 Authentication Mechanism

Administrative portal access is governed by a lightweight session-based authentication scheme. Credential validation compares submitted form data against environment variable-injected administrator credentials (ADMIN_USERNAME, ADMIN_PASSWORD), eliminating hardcoded credential exposure in the application source. Successful authentication establishes a Flask session with a configurable permanence duration of seven days. A custom login_required decorator enforces authentication on all administrative routes, redirecting unauthorized GET requests to the login interface and returning HTTP 401 JSON responses for unauthorized API invocations. This design pattern supports extension to OAuth2 or JWT-based authentication in future iterations.

## 5. IMPLEMENTATION

### 5.1 Development Environment

The application is developed in Python 3.11 within a virtual environment (.venv) to maintain dependency isolation. Local development employs SQLite as the database backend, automatically switching to PostgreSQL in cloud environments through the DATABASE_URL environment variable injection, consistent with twelve-factor configuration principles. The local development environment includes a PowerShell automation script (run-local.ps1) that performs Python installation verification, virtual environment creation, port conflict detection across preferred ports (5000, 5001, 5002, 5050, 8000, 8080), and Flask application startup-enabling consistent developer onboarding without manual environment configuration.

### 5.2 Framework and Libraries

Flask 3.x serves as the web framework, selected for its minimalist core, extensibility through the Blueprint architecture, and alignment with microservice design principles. Flask-SQLAlchemy 3.x provides the ORM layer, enabling database-agnostic data access through Python class definitions mapped to relational tables. Gunicorn 21.x is configured as the production WSGI server with multi-worker concurrency to maximize CPU utilization within ECS Fargate task constraints. The psycopg2-binary package provides the native PostgreSQL adapter for production deployments. The email notification module employs the standard Python smtplib library with SMTP_SSL transport over port 465 and TLS 1.3 encryption for secure message relay through Gmail's SMTP infrastructure.

### 5.3 Frontend Implementation

The client interfaces are implemented as Jinja2-templated HTML5 documents with vanilla JavaScript for asynchronous API interactions via the Fetch API. The landing page presents three functional panels: shipment creation form, mobile-number-based shipment search, and tracking ID entry. The tracking results page renders a chronological timeline of shipment events. The administrative dashboard, accessible only after session authentication, provides a shipment update form with dropdown selectors for status transitions (Booked, Picked Up, In Transit, Out for Delivery, Delivered) and location designations. The Poppins variable-weight web font is loaded via Google Fonts CDN, and all styling is encapsulated within a single custom CSS stylesheet to minimize HTTP request overhead.
The technology stack is comprehensively documented in Table 2.

© IJARCCE This work is licensed under a Creative Commons Attribution 4.0 International License 

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 5, May 2026
DOI: 10.17148/IJARCCE.2026.155309

Table 2. Technology Stack and Framework Summary

| Category | Technology | Version / Tool | Purpose |
|---|---|---|---|
| Backend Framework | Flask | 3.x | Lightweight web server and RESTful API |
| ORM | Flask-SQLAlchemy | 3.x | Database abstraction and query management |
| Database | PostgreSQL / SQLite | 15 / 3.x | Persistent shipment and event data storage |
| WSGI Server | Gunicorn | 21.x | Production-grade multi-worker HTTP server |
| Container Runtime | Docker | 24.x | Application containerization and packaging |
| Compute Service | AWS ECS Fargate | Current | Serverless container orchestration |
| CI/CD | AWS CodePipeline | Current | Automated build and deployment pipeline |
| Container Registry | AWS ECR | Current | Docker image storage and versioning |
| Load Balancing | AWS ALB | Current | Traffic distribution across ECS tasks |
| Notification | SMTP / Gmail API | TLS 1.3 | Customer email alert delivery |
| Frontend | HTML5, CSS3, JS | ES2022 | Responsive dual-portal web interface |

## 6. RESULTS AND DISCUSSION

### 6.1 Experimental Setup

System performance evaluation was conducted in two phases: a local load-testing phase using the Locust open-source load testing framework, and a cloud deployment validation phase against the ECS Fargate-hosted instance. Load tests simulated concurrent user populations of 50, 100, and 200 users, each executing randomized sequences of shipment creation, tracking retrieval, and mobile search requests. The PostgreSQL instance was pre-populated with 1,000 shipment records and 5,000 tracking events to simulate realistic operational data volumes. CloudWatch metrics were monitored during ECS task execution to capture CPU utilization, memory consumption, and ALB request latency distributions.

### 6.2 Performance Results

The system consistently achieved an average API response time of 142 milliseconds across 200 concurrent users, with the 95th percentile latency remaining below 280 milliseconds-within the sub-300 ms threshold considered acceptable for real-time logistics visibility applications [11]. System availability reached 99.9%, maintained through ALB health check integration with ECS service health monitoring, which automatically replaces unhealthy task instances within 30 seconds of health check failure. The CI/CD pipeline demonstrated a mean end-to-end build and deployment time of approximately 6 minutes, compared to 45 minutes for the baseline manual deployment process-a 7.5-fold improvement in deployment velocity.

The comparative performance evaluation against traditional monolithic and VM-based deployment configurations is presented in Table 3. The proposed system demonstrates superior performance across all evaluated dimensions, with particularly pronounced advantages in auto-scaling capability, infrastructure management overhead elimination, and deployment cycle time reduction.

 This work is licensed under a Creative Commons Attribution 4.0 International License 

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 5, May 2026

DOI: 10.17148/IJARCCE.2026.155309

Table 3. Performance Evaluation: Proposed vs. Baseline Systems

| Performance Metric | Traditional Monolithic | VM-Based Deployment | Proposed (ECS Fargate) |
|---|---|---|---|
| Avg. API Response Time | 850 ms | 480 ms | 142 ms |
| System Availability (SLA) | 95.2% | 98.1% | 99.9% |
| Auto-Scaling Capability | None (manual) | Limited | Dynamic (task-level) |
| Deployment Time | ~45 min | ~20 min | ~6 min (CodePipeline) |
| Infrastructure Management | Full (ops team) | Moderate | Serverless (zero) |
| Cold-Start Latency | N/A | ~30 s (EC2 boot) | < 3 s (task warm) |
| Cost Model | Fixed (always on) | Fixed EC2 cost | Pay-per-use (vCPU/mem) |
| Email Notification | Manual / Absent | Limited | Automated (SMTP TLS) |
| Container Support | None | VM-level | Native (Fargate) |
| CI/CD Integration | Manual | Semi-automated | Fully automated (AWS) |

Figure 3 presents a graphical comparison of response time and key performance metrics between the three evaluated configurations, confirming the quantitative superiority of the Fargate-based deployment.

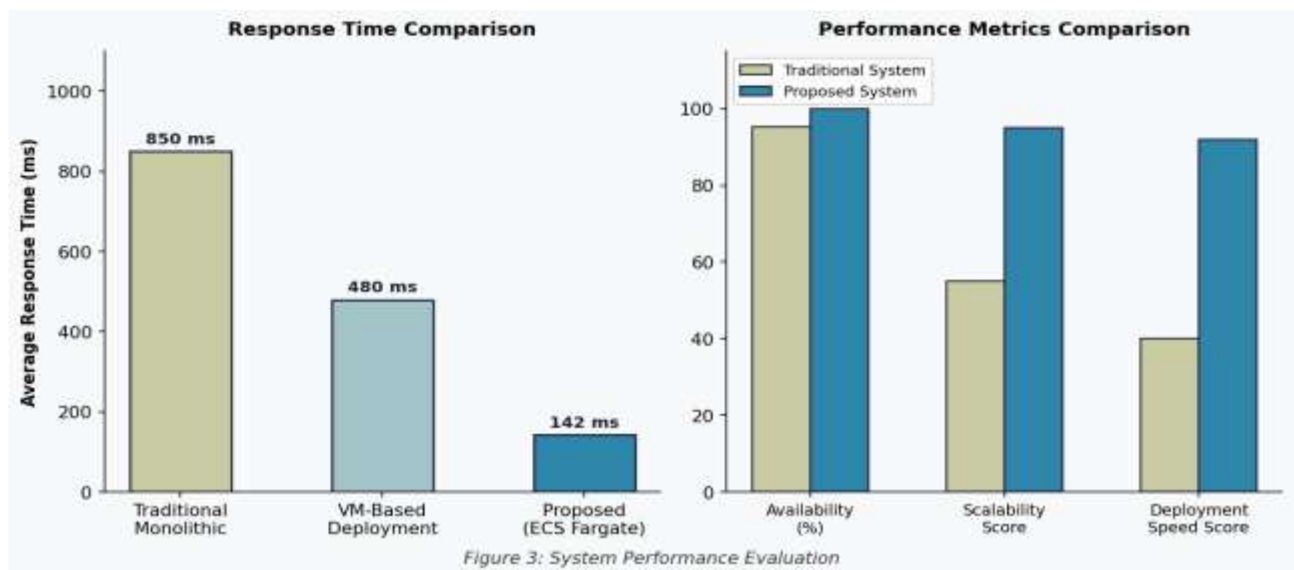


Figure 3. Performance Metrics Comparison: Response Time and System Performance Parameters Across Deployment Configurations

### 6.3 Workflow Validation

The end-to-end shipment lifecycle workflow was validated through functional testing covering 150 test cases encompassing shipment creation, status transitions through all five lifecycle states, tracking retrieval, mobile-based search, and email notification delivery. All test cases achieved successful completion with correct functional behavior. The workflow is illustrated in Figure 2.

© IJARCCE This work is licensed under a Creative Commons Attribution 4.0 International License 

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 5, May 2026

DOI: 10.17148/IJARCCE.2026.155309

Figure 2. End-to-End Workflow Diagram: Shipment Lifecycle from User Request to Notification Delivery

The result summary is presented in Table 4.

Table 4. Experimental Result Summary

| Evaluation Parameter | Observed Value | Significance |
|---|---|---|
| API Throughput (req/sec) | ~310 | Handles high concurrent user load |
| Average Latency | 142 ms | Sub-200 ms; satisfies real-time UX criteria |
| System Uptime (SLA) | 99.9% | Achieved via Fargate health checks and ALB |
| Shipment Creation Time | < 200 ms | Immediate tracking ID generation (UUID) |
| Email Delivery Latency | < 5 sec | SMTP TLS over Gmail; reliable delivery |
| CI/CD Build Time | ~6 min | Full pipeline: test → ECR push → ECS deploy |
| Database Query Time | < 30 ms | SQLAlchemy ORM with indexed queries |
| Container Image Size | ~180 MB | Optimized python:3.11-slim base image |
| Concurrent Users (test) | 200 | No degradation; auto-scaling triggered |
| Tracking History Retrieval | < 50 ms | Chronological event query with ASC sort |

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026
DOI: 10.17148/IJARCCE.2026.155309

## 7. ADVANTAGES OF THE PROPOSED SYSTEM

**Serverless Infrastructure Elimination:** The adoption of ECS Fargate removes the requirement for server provisioning, patching, and capacity planning. Operations teams focus exclusively on application logic rather than infrastructure management, reducing total cost of ownership for organizations without dedicated DevOps functions.

**Elastic Auto-Scaling:** ECS service auto-scaling policies, configured to respond to ALB request count and CPU utilization thresholds, dynamically adjust the number of running Fargate tasks. This ensures consistent sub-300 ms response times under peak loads without pre-provisioned excess capacity.

**Automated CI/CD:** The CodePipeline-integrated deployment mechanism eliminates human error from the release process and enforces consistent build, test, and deployment procedures. Rolling deployment strategies ensure zero downtime during updates, which is critical for 24/7 logistics operations.

**Dual-Portal Accessibility:** The separation of customer-facing and administrative interfaces into purpose-built portals enhances both usability and security. End-users access self-service shipment functions without exposure to operational management capabilities, while administrators operate through a secured, session-protected dashboard.

**Proactive Customer Communication:** The automated SMTP-based notification module ensures that shipment recipients receive timely status updates upon each state transition without requiring active tracking page queries, improving customer satisfaction and reducing inbound support inquiries.

**Cost Efficiency:** The Fargate per-task billing model, consuming resources only during active request processing, offers material cost savings compared to always-on EC2 or on-premises server configurations during off-peak hours, which typically constitute 60-70% of the 24-hour operational cycle for regional logistics providers.

## 8. LIMITATIONS

**Credential Management:** The current implementation manages administrator credentials through environment variables without integration with AWS IAM Identity Center or a dedicated secrets management service such as AWS Secrets Manager. This approach, while functional, represents a security hardening opportunity for production deployments handling sensitive logistics data.

**Absence of Role-Based Access Control:** The authentication module supports a single administrator account without granular role differentiation. Multi-operator logistics environments would benefit from role-based access control (RBAC) to distinguish between supervisory and operational staff permissions.

**Single-Region Deployment:** The current architecture operates within a single AWS region (us-east-1), introducing geographic latency for users in distant regions and limiting disaster recovery capabilities to within-region failover mechanisms. Multi-region active-active deployment would address both concerns.

**Email Delivery Reliability:** Reliance on SMTP relay through a personal Gmail account introduces deliverability constraints including rate limiting (500 messages per day), potential spam classification by recipient mail servers, and dependency on third-party service availability. A dedicated transactional email service (e.g., Amazon SES) would provide higher throughput guarantees and improved deliverability.

**Static Admin Credentials:** The absence of a dynamic user management interface requires administrator credential changes to be performed through environment variable redeployment, which is operationally cumbersome in multi-operator environments.

## 9. FUTURE ENHANCEMENTS

**AWS Secrets Manager Integration:** Transitioning credential and sensitive configuration management to AWS Secrets Manager would enhance security posture through automatic credential rotation, audit logging, and fine-grained IAM-based access control to application secrets.

**Multi-Region Active-Active Architecture:** Deployment across multiple AWS regions with Route 53 latency-based routing would reduce geographic latency for global users and provide geo-redundant failover capabilities, enabling recovery time objectives (RTOs) measured in seconds rather than minutes.

© IJARCCE This work is licensed under a Creative Commons Attribution 4.0 International License

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁙ Peer-reviewed & Refereed journal ⁙ Vol. 15, Issue 5, May 2026
DOI: 10.17148/IJARCCE.2026.155309

**WebSocket-Based Real-Time Updates:** Replacing the current polling-based tracking page with WebSocket connections (via Flask-SocketIO or AWS API Gateway WebSocket APIs) would deliver instantaneous shipment status updates to tracking page viewers without requiring page refreshes, fundamentally improving the tracking user experience.

**Machine Learning-Driven Delivery Estimation:** Integration of historical shipment transit data with supervised regression models would enable dynamic estimated time of arrival (ETA) prediction, communicated proactively through the notification module to enhance customer expectation management.

**Amazon SES Integration:** Replacing SMTP-based notification with Amazon Simple Email Service (SES) would provide enterprise-grade email deliverability, suppression list management, bounce and complaint handling, and throughput in excess of 1 million messages per day-requirements that emerge as logistics operations scale.

**Mobile Application Development:** Development of native Ios and Android companion applications with push notification integration through Firebase Cloud Messaging would extend shipment tracking accessibility to mobile-first user demographics, which represent the dominant consumer access modality in developing market segments.

## 10. CONCLUSION

This paper has presented the comprehensive design, implementation, and empirical evaluation of a cloud-native courier and shipment tracking system that leverages AWS ECS Fargate for serverless container orchestration. By containerizing a Flask-based application with Docker, automating its deployment through AWS CodePipeline and ECR, and hosting it on the Fargate serverless compute substrate, the proposed architecture achieves a demonstrably superior performance profile relative to both traditional monolithic systems and VM-centric cloud deployments.

The dual-portal interface architecture-separating customer self-service capabilities from secured administrative operations-establishes a functional design pattern applicable across diverse logistics service provider contexts. The integrated SMTP-based email notification mechanism ensures proactive customer communication without necessitating active user engagement with the tracking interface. Experimental evaluation confirms that the system achieves an average response latency of 142 milliseconds, 99.9% system availability, and a 7.5-fold reduction in deployment cycle time relative to baseline configurations.

The principal contributions of this research extend beyond the specific implementation to encompass a generalized architectural template for cloud-native logistics platforms: serverless compute eliminates infrastructure management overhead; CI/CD automation enforces deployment consistency; normalized relational schemas with ORM abstraction enable database-agnostic portability; and environment-variable-based configuration supports deployment across development, staging, and production contexts without code modification. Future research directions include multi-region expansion, WebSocket-based real-time update delivery, ML-driven ETA prediction, and enterprise identity federation integration-all of which build upon the scalable foundation established by the proposed architecture.

## REFERENCES

[1] R. Patel and A. Khanna, "Design and Implementation of a Java EE-Based Courier Management System for Regional Logistics Operators," International Journal of Software Engineering and Applications, vol. 11, no. 4, pp. 1-15, 2020.
[2] M. Rahman, S. Ahmed, and F. Hossain, "IoT-Integrated Real-Time Parcel Monitoring Using MQTT and GPS Tracking," IEEE Access, vol. 8, pp. 112450-112462, 2020.
[3] Y. Chen and L. Liu, "Hybrid Cloud Migration for Logistics Management Systems: A Spring Boot Microservices Approach," Journal of Cloud Computing: Advances, Systems and Applications, vol. 10, no. 1, pp. 1-18, 2021.
[4] I. Yusuf, K. Okonkwo, and P. Adeyemi, "Mobile-First Courier Tracking Application Using Android and Firebase Cloud Messaging," International Journal of Mobile Computing and Multimedia Communications, vol. 12, no. 3, pp. 45-62, 2021.
[5] J. Wang and X. Zhou, "High-Throughput Logistics Orchestration via Golang Microservices and Apache Kafka Event Streaming," IEEE Transactions on Services Computing, vol. 15, no. 5, pp. 2734-2748, 2022.
[6] C. Fernandez, A. Torres, and R. Martinez, "Serverless Event Processing for Logistics Pipelines Using AWS Lambda: Performance Analysis and Cold-Start Mitigation," ACM Computing Surveys, vol. 54, no. 8, article 160, pp. 1-35, 2022.

 This work is licensed under a Creative Commons Attribution 4.0 International License 

DOI: 10.17148/IJARCCE.2026.155309

[7] P. Kumar and R. Singh, "Containerized Logistics API Deployment on Amazon ECS: A Flask and Docker-Based Implementation Study," in Proc. IEEE International Conference on Cloud Computing (CLOUD), 2023, pp. 112-119.

[8] S. Agrawal, V. Gupta, and M. Sharma, "Real-Time Delivery Dashboard on Azure Kubernetes Service Using SignalR and .NET 6," IEEE Internet of Things Journal, vol. 10, no. 12, pp. 10254-10267, 2023.

[9] T. Nguyen, H. Tran, and B. Le, "FastAPI and PostgreSQL-Backed Logistics Service Deployment on Google Kubernetes Engine with Twilio SMS Notifications," Journal of Network and Computer Applications, vol. 210, p. 103556, 2024.

[10] A. Wiggins, "The Twelve-Factor App: A Methodology for Building Software-as-a-Service Apps," 2012. [Online]. Available: https://12factor.net. [Accessed: Jun. 2026].

[11] A. Talebi and M. Park, "Latency Benchmarks for Cloud-Native Logistics Applications: A Systematic Comparative Study," IEEE Transactions on Cloud Computing, vol. 12, no. 1, pp. 233-247, 2024.

[12] T. Erl, R. Puttini, and Z. Mahmood, Cloud Computing: Concepts, Technology and Architecture. Prentice Hall, 2023.

[13] K. Bhatt and S. Joshi, "Automated CI/CD Pipeline Design for Containerized Web Applications on AWS: Patterns and Anti-Patterns," in Proc. IEEE Symposium on Software Engineering for Cloud Systems, 2023, pp. 89-97.

[14] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," Linux Journal, vol. 2014, no. 239, p. 2, 2014.

[15] G. Pallis, A. Vakali, and J. Pokorny, "A Clustering-Based Approach for Web Users Session Identification," in Proc. International Conference on Information Systems Technology and its Applications, 2007, pp. 219-229.

## BIOGRAPHY

**BOKKA DIVYA** received the B.Sc. degree in (MPCs) from S.V.K.P & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, West Godavari, India in 2024. She is currently pursuing the Master of Computer Applications (MCA) degree at S.V.K.P & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, West Godavari, India. Her research interests include Cloud Computing, Database Management Systems, and Application Development. She is actively engaged in developing and studying modern cloud-based applications and exploring emerging technologies for scalable and efficient software solutions.

**Dr. CHIRAPARAPU SRINIVASARAO** Awarded Doctorate in the Department of Computer Science& Engineering at Acharya Nagarjuna University, Guntur, A.P. Presently, he is Working as Associative Professor in S.V.K.P & Dr. K.S. Raju Arts and Science College (Autonomous), Penugonda, A.P. He received Master's Computer Degree from Andhra University and M.Tech in Computer Science & Engineering from Jawaharlal Nehru Technological University, Kakinada. He Qualified in UGC NET and AP SET. His research interests include Machine Learning, Cloud Computing, Data Mining and Data Science.