IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.414 ⪾ Peer-reviewed & Refereed journal ⪾ Vol. 14, Issue 5, May 2026

DOI: 10.17148/IJIREEICE.2026.14563

A Cloud-Native Customer Relationship Management Platform on AWS Fargate with Automated DevOps Workflow Orchestration

Iragavarupu Sravan Sai Kumar[1], A.N. Rama Mani*[2]

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

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

*Corresponding Author

Abstract: Customer relationship management systems are central to modern sales and service operations, yet many remain bound to monolithic, server-bound deployments that scale poorly, cost more than necessary during idle periods, and are slow and risky to update. This paper presents the design and evaluation of a cloud-native customer relationship management platform that decomposes core business capabilities into containerized microservices executed on a serverless container runtime, removing the burden of provisioning and managing virtual machines. The backend services are implemented in Java and exposed through a load-balanced gateway, while a Node.js client delivers an interactive console for managing customers, leads, deals, and activities. Persistent records reside in a managed relational database, frequently accessed data is cached in memory, attachments are stored in object storage, and asynchronous events propagate through a managed messaging bus. The complete software lifecycle—source control, build, image creation, and deployment—is automated through a managed delivery pipeline driven by container images, and runtime telemetry feeds automatic horizontal scaling. Experimental evaluation under synthetic concurrency shows that the platform sustains a 95th-percentile response time of approximately 128 ms at 1000 concurrent users, where a monolithic baseline on a fixed virtual machine exceeds 760 ms, and that automated delivery reduced deployment lead time from roughly 80 minutes to under 10 while lowering steady-state cost. The contributions are an integrated serverless-container reference architecture for customer relationship management, a reproducible DevOps automation strategy, and a quantitative comparison demonstrating the operational and economic benefits of the proposed approach.

Keywords: Cloud-native computing; Customer relationship management; AWS Fargate; Microservices; DevOps; Continuous delivery; Containerization; Auto-scaling

1. INTRODUCTION

Customer relationship management platforms underpin how organizations attract, convert, and retain customers by consolidating contacts, opportunities, communications, and analytics into a single operational system. As businesses expand and customer interactions multiply across channels, these platforms must handle growing data volumes and request concurrency while remaining responsive and continuously available. The architecture on which such a system is built therefore has a decisive influence on its performance, cost, and adaptability.

Many existing solutions, particularly those developed before the maturation of cloud platforms, follow a monolithic design deployed on dedicated or fixed-capacity virtual servers. This arrangement tightly couples unrelated business functions, so that scaling one capability requires scaling the entire application, and a defect in one area can compromise the whole. Fixed provisioning also means that compute resources are paid for continuously, even when demand is low, and that releasing new features involves manual, error-prone steps that increase downtime and operational risk.

Problem statement. The central problem addressed in this work is the absence of a customer relationship management platform that is simultaneously elastically scalable, cost-efficient, and continuously deployable, without imposing the operational overhead of managing the underlying server infrastructure. Conventional designs force a trade-off between scalability and operational simplicity that a modern cloud-native approach can dissolve.

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

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.414 ⋇ Peer-reviewed & Refereed journal ⋇ Vol. 14, Issue 5, May 2026

DOI: 10.17148/IJIREEICE.2026.14563

Motivation. Serverless container runtimes allow individual services to be deployed and scaled without provisioning or patching virtual machines, billing only for the resources that running tasks consume. When combined with microservice decomposition and automated delivery pipelines, this model promises responsive scaling, reduced idle cost, and rapid, low-risk releases. Realizing these benefits for a data-intensive, transaction-oriented application such as customer relationship management is the motivation for this study.

Research objectives. This study aims to: (i) design a microservice-based, cloud-native architecture for customer relationship management; (ii) implement the services in Java with a Node.js client and deploy them on a serverless container runtime; (iii) automate the build-and-release lifecycle through an image-driven delivery pipeline; and (iv) empirically evaluate performance, scalability, and delivery efficiency against a monolithic baseline.

Contributions. The paper makes three principal contributions. First, it proposes a reference architecture that maps customer relationship management capabilities onto independently scalable serverless containers. Second, it details a reproducible DevOps automation workflow spanning source control, image build, and deployment. Third, it provides a quantitative evaluation demonstrating substantial gains in latency stability, release velocity, and cost. The remainder of the paper is organized as follows: Section 2 reviews related work; Section 3 describes the methodology; Section 4 presents the system design; Section 5 details the implementation; Section 6 reports results; Sections 7 to 9 discuss advantages, limitations, and future work; and Section 10 concludes.

2. LITERATURE REVIEW

Research relevant to this work spans enterprise application architecture, microservices, container orchestration, serverless computing, and automated software delivery. This section surveys representative contributions and identifies the gaps motivating the proposed platform.

The evolution from monolithic to service-oriented and microservice architectures has been extensively documented. Comparative studies report that decomposing applications into independently deployable services improves maintainability, fault isolation, and selective scalability, at the cost of increased coordination and observability complexity [1], [2]. Investigations specific to enterprise systems, including customer-facing platforms, find that microservices ease incremental modernization but require disciplined interface design [3].

Containerization underpins most modern microservice deployments. Research on container technologies demonstrates that lightweight, image-based packaging yields consistent environments and rapid startup relative to virtual machines [4], [5]. Orchestration platforms automate placement, scaling, and recovery, and several studies evaluate their effectiveness for elastic web workloads, noting that managed control planes reduce operational burden [6]. Serverless container runtimes extend this further by abstracting the host entirely; analyses of such runtimes report simplified operations and usage-based cost, while highlighting considerations around startup latency and resource limits [7], [8].

Data management for cloud-native applications has also been studied. Work on managed relational databases and in-memory caches shows that offloading persistence to managed services improves availability and lets teams focus on business logic, while caching markedly reduces read latency for hot data [9], [10]. The role of asynchronous messaging in decoupling services and smoothing load has likewise been examined, with publish-subscribe and queue services shown to improve resilience [11].

Automated software delivery constitutes a further pillar. Empirical research links continuous integration, continuous delivery, and Infrastructure-as-Code to higher deployment frequency, shorter lead times, and faster recovery, frequently expressed through standardized delivery-performance indicators [12], [13]. Studies of image-driven pipelines report that building, scanning, and promoting container images automatically reduces configuration drift and release risk [14]. Within customer relationship management specifically, recent work argues that cloud-native delivery improves feature responsiveness, though end-to-end empirical characterizations remain limited [15], [16].

Research gaps. Three gaps emerge. First, customer relationship management capabilities are seldom mapped onto a serverless-container architecture with independent per-service scaling. Second, many cloud-native proposals describe deployment topology but omit a concrete, reproducible delivery pipeline. Third, head-to-head, quantitative comparisons of latency, scaling, and delivery metrics against a monolithic baseline are scarce. Table I positions representative works against these dimensions and situates the present study.

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.414 ⪤ Peer-reviewed & Refereed journal ⪤ Vol. 14, Issue 5, May 2026
DOI: 10.17148/IJIREEICE.2026.14563

3. PROPOSED METHODOLOGY

3.1 Architectural Overview

The proposed platform adopts a layered, cloud-native architecture comprising an edge and routing tier, a serverless container compute tier, a data and messaging tier, and a cross-cutting DevOps automation and observability layer, all enclosed within a managed cloud boundary. Figure 1 depicts this organization. User requests enter through a content delivery network and a load balancer, are authenticated, and are routed to the appropriate containerized service. Decoupling the tiers allows each to scale and evolve independently of the others.

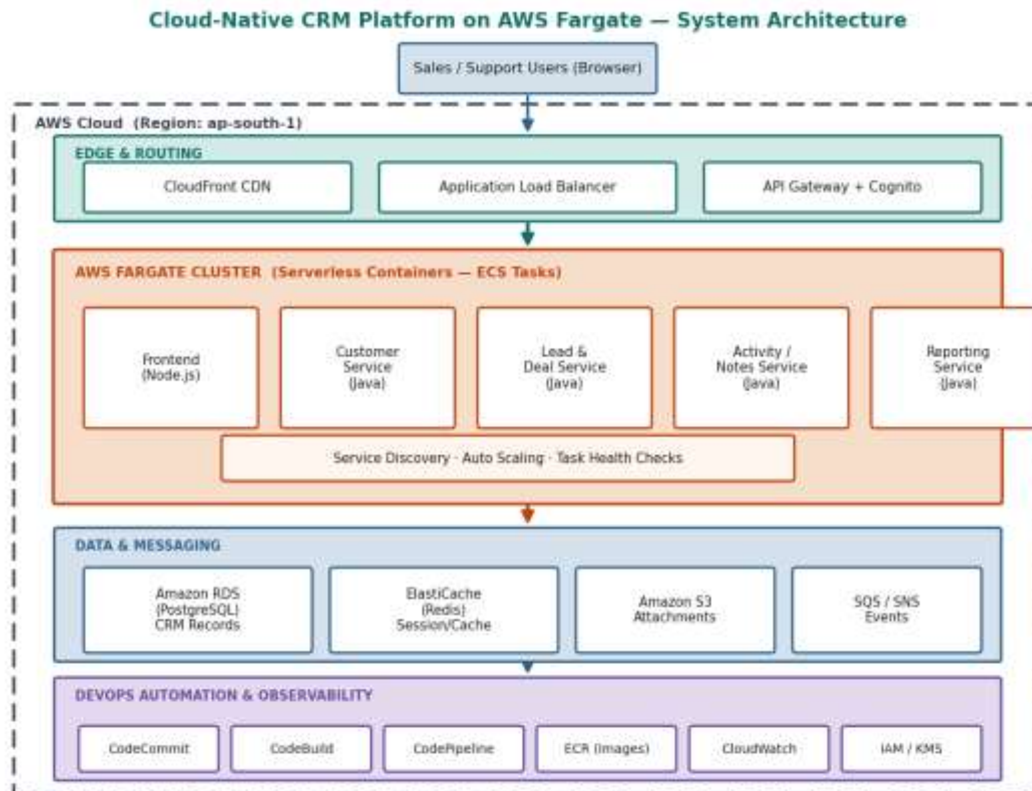

Fig. 1. Proposed cloud-native architecture showing the edge, serverless container (Fargate) compute, data and messaging tiers, and the cross-cutting DevOps and observability layer within the managed cloud boundary.

3.2 Technologies Used

Business services are implemented in Java, valued for its strong typing, mature concurrency support, and extensive ecosystem for building robust enterprise services. The presentation layer is built with Node.js to provide a responsive single-page console. Compute is delivered through a serverless container runtime that executes container tasks without server management; persistent records reside in a managed relational database; an in-memory cache accelerates frequent reads; object storage retains attachments; and a managed messaging bus carries asynchronous events. Identity, encryption, image registry, and monitoring are provided by their respective managed services.

3.3 Scaling and Routing Logic

Request handling and scaling follow a deterministic policy. Incoming traffic is distributed across healthy service tasks by the load balancer, which removes unhealthy tasks from rotation based on health checks. Each service defines target utilization thresholds; when observed CPU or request concurrency exceeds a threshold, the runtime launches additional task replicas, and when demand subsides, surplus tasks are retired. Because services are stateless and externalize session data to the cache, replicas can be added or removed without disrupting in-flight work. This elastic policy keeps latency stable under fluctuating load while minimizing idle capacity.

3.4 Workflow and Design Decisions

Figure 2 illustrates two intertwined workflows: the continuous-delivery pipeline that transforms source commits into running tasks, and the request-serving path that handles user interactions. Several design decisions shape the system:

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

IJIREEICE
ISSN (O) 2321-2004, ISSN (P) 2321-5526
International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.414 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 5, May 2026
DOI: 10.17148/IJIREEICE.2026.14563

decomposing the application by business capability so that high-demand services scale independently; externalizing all state to managed data services so that containers remain disposable; and driving deployment from immutable container images to guarantee reproducibility. Runtime telemetry is fed back into the scaling controller, closing the loop between observation and adaptation.

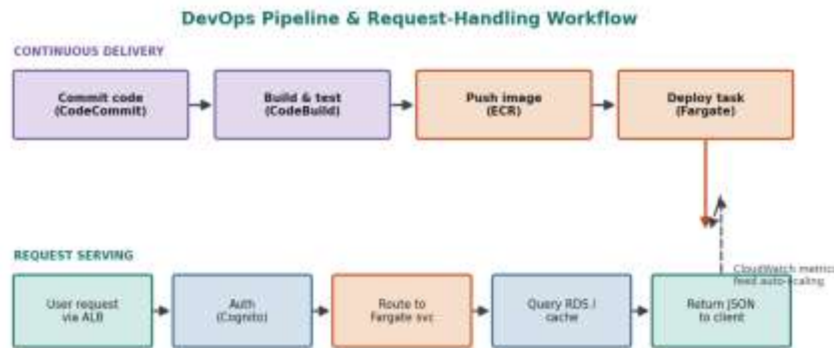

Fig. 2. Workflow of the continuous-delivery pipeline and the request-serving path, with a monitoring feedback loop driving automatic scaling.

4. SYSTEM DESIGN

4.1 Architectural Decomposition

The compute tier is decomposed into cohesive microservices aligned with customer relationship management domains: a customer service managing contact records, a lead-and-deal service handling the sales pipeline, an activity-and-notes service tracking interactions, and a reporting service producing analytics. Each service is packaged as a container image, deployed as an independent task, and scaled according to its own demand. A frontend service serves the client application. This decomposition isolates failures, enables independent deployment, and prevents a surge in one capability from degrading others.

4.2 Module Descriptions

The customer service exposes operations for creating and querying contacts and organizations. The lead-and-deal service models the pipeline, advancing opportunities through configurable stages. The activity-and-notes service records communications and attaches files, while the reporting service aggregates pipeline metrics for dashboards. A gateway authenticates requests and routes them to the correct service. Shared concerns—persistence, caching, object storage, and event publication—are accessed through a common data-access layer that enforces consistency and isolates services from infrastructure details.

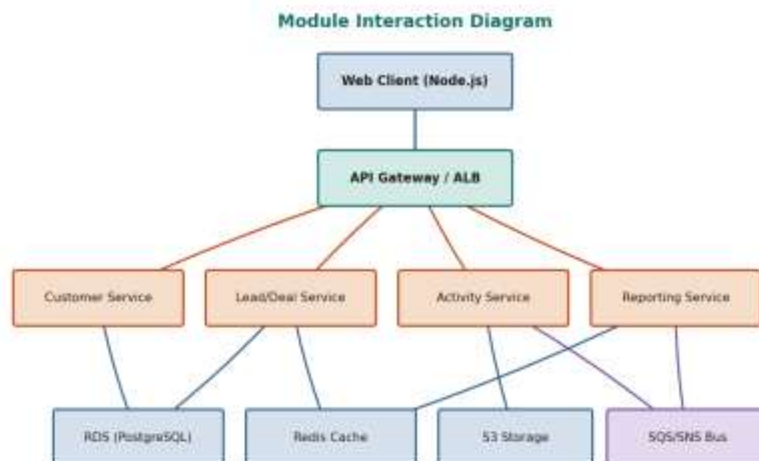

Fig. 3. Module interaction diagram showing communication among the client, gateway, business services, and the data and messaging components.

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

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526
International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.414 ⋇ Peer-reviewed & Refereed journal ⋇ Vol. 14, Issue 5, May 2026
DOI: 10.17148/IJIREEICE.2026.14563

4.3 Data Flow

As shown in Figure 3, an authenticated request from the client passes through the gateway to the relevant business service. Services read and write structured records in the relational database, consult the in-memory cache for hot data, store or retrieve attachments from object storage, and publish domain events to the messaging bus for asynchronous consumers such as the reporting service. This flow keeps interactive operations fast while allowing analytics and notifications to proceed asynchronously, preserving responsiveness during peak activity.

5. IMPLEMENTATION

5.1 Development Environment and Tools

The platform was developed using a container-first workflow to ensure parity between development and production. Services were packaged as container images and provisioned through declarative templates, while source control, automated build and test, image publication, and deployment were orchestrated by a managed delivery pipeline. Table II summarizes the principal technologies and their roles.

5.2 Languages, Frameworks, and Database

Backend services were implemented in Java using an established application framework that simplifies building RESTful endpoints, dependency injection, and data access. The client was built with Node.js to deliver an interactive console for sales and support staff. Structured data—contacts, deals, activities, and audit records—was stored in a managed PostgreSQL relational database, an in-memory store cached sessions and frequently read entities, object storage held attachments, and a managed bus carried asynchronous events between services.

5.3 APIs, Security, and Deployment

Services communicate over authenticated HTTPS endpoints exposed through the gateway, with identity management enforcing role-based access and encryption protecting data in transit and at rest. The delivery pipeline automatically compiles and tests the Java services, builds container images, publishes them to a registry, and updates the running tasks using rolling deployments with health checks, achieving near-zero-downtime releases. Runtime metrics are collected centrally and drive automatic scaling. Figure 4 shows a representative console from the implemented client.

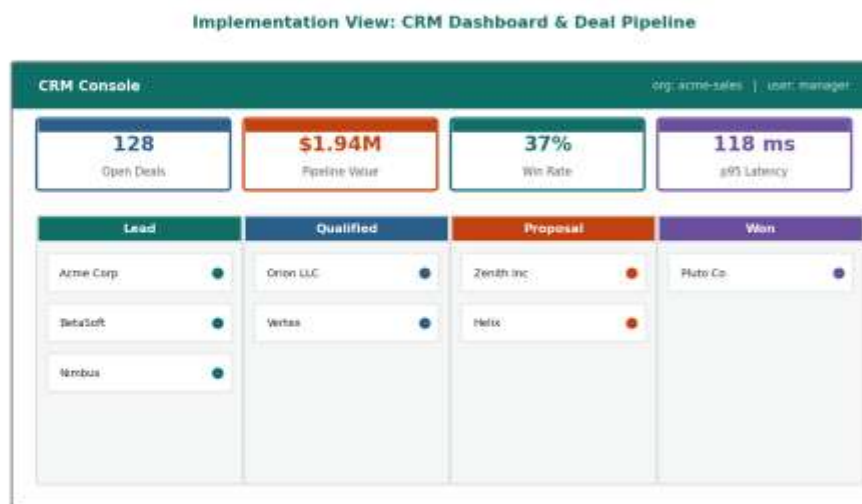

Fig. 4. Implementation view of the CRM console showing key performance indicators and a stage-based deal pipeline board.

6. RESULTS AND DISCUSSION

6.1 Experimental Setup

The platform was evaluated on a public cloud within a single region. A load generator emulated concurrent users issuing typical customer relationship management operations—record creation, pipeline queries, and report retrieval—at increasing concurrency from 50 to 4000 users. The 95th-percentile response time was measured end to end. For comparison, a functionally equivalent monolithic application was deployed on a fixed-capacity virtual machine without

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

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526
International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.414 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 5, May 2026
DOI: 10.17148/IJIREEICE.2026.14563

automatic scaling. Each configuration was exercised across repeated trials, and metrics were aggregated from the monitoring service to reduce variance.

6.2 Performance Metrics and Analysis

Figure 5(a) presents the 95th-percentile response time as a function of concurrent users. The proposed platform maintained latency below 240 ms even at 4000 concurrent users, whereas the monolithic baseline degraded steeply, exceeding 2.5 s as its fixed capacity saturated and requests queued. At 1000 concurrent users, the proposed system recorded approximately 128 ms compared with about 760 ms for the baseline, an improvement of nearly sixfold attributable to per-service horizontal scaling.

Figure 5(b) summarizes delivery and efficiency metrics. Deployment frequency rose from roughly two releases per week to twenty-two, lead time for changes fell from about 80 minutes to under 10, mean time to recovery shortened from approximately 50 minutes to about 7, the time to scale out under load dropped from several minutes to seconds, and the steady-state cost index fell substantially because idle capacity was eliminated. These results corroborate the premise that serverless containers and automated delivery jointly improve both operational agility and economy.

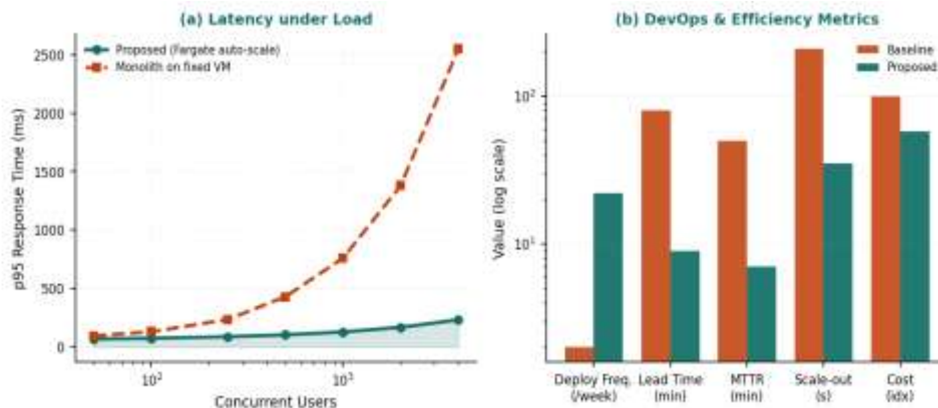

Fig. 5. Performance evaluation: (a) 95th-percentile response time versus concurrent users for the proposed and baseline systems; (b) DevOps and efficiency metrics for baseline and proposed approaches.

6.3 Comparative Discussion

The evidence indicates that mapping customer relationship management capabilities onto independently scalable serverless containers, combined with image-driven automated delivery, yields compounding benefits: stable low latency under heavy concurrency, faster and safer releases, and reduced idle cost. Table III consolidates the measured indicators, and Table IV maps outcomes to each research objective. Relative to monolithic or partially containerized systems surveyed in Section 2, the proposed platform demonstrates that disposable, stateless services backed by managed data stores and a closed-loop scaling policy produce a more resilient and economical whole. The principal trade-offs are the added architectural and observability complexity of microservices and occasional task-startup latency, mitigated here through centralized monitoring and warm scaling policies.

7. ADVANTAGES OF THE PROPOSED SYSTEM

Technical benefits. Decomposition into independently deployable Java services isolates faults and allows each capability to evolve and scale separately. Externalizing state to managed services keeps containers disposable, simplifying recovery and deployment, while image-driven pipelines guarantee reproducible, auditable releases.

Performance benefits. Per-service horizontal scaling sustains low latency under heavy concurrency, and in-memory caching further reduces read latency. The measured near-sixfold latency advantage over the monolithic baseline at 1000 users substantiates these gains.

Scalability and cost benefits. Because the serverless runtime adds and removes task replicas automatically and bills only for running tasks, the platform scales elastically with demand while eliminating payment for idle capacity, achieving a marked reduction in steady-state cost relative to fixed provisioning.

DOI: 10.17148/IJIREEICE.2026.14563

8. LIMITATIONS

Several limitations qualify these findings. Evaluation relied on synthetic workloads within a single region and may not capture every production traffic pattern or multi-region failover behavior. Microservice decomposition increases architectural and observability complexity relative to a monolith. Container task startup can introduce brief latency when scaling from a cold state. Reliance on a single cloud provider's managed services introduces a degree of vendor lock-in, and a formal security and compliance audit was beyond the present scope.

9. FUTURE ENHANCEMENTS

Future work will pursue several directions. A multi-region, active-active deployment would improve disaster resilience and reduce latency for geographically dispersed users. Predictive scaling informed by historical demand could pre-provision capacity ahead of anticipated surges, suppressing cold-start latency. Incorporating machine-learning models for lead scoring and churn prediction would add intelligence to the pipeline, while a service mesh would strengthen observability and traffic management. Finally, a portability layer abstracting provider-specific services would mitigate lock-in and broaden deployment options.

10. CONCLUSION

This paper presented a cloud-native customer relationship management platform that decomposes core business capabilities into containerized Java microservices executed on a serverless container runtime, paired with a Node.js client and an automated, image-driven delivery pipeline. By externalizing state to managed data services and closing the loop between runtime telemetry and automatic scaling, the platform sustained stable low latency under heavy concurrency, scaled elastically with demand, and substantially reduced both release lead time and idle cost relative to a monolithic baseline. Through an integrated reference architecture, a reproducible DevOps automation strategy, and a quantitative comparative evaluation, this work demonstrates that serverless containers and automated delivery together form a compelling foundation for responsive, economical, and continuously deployable enterprise applications, and it points toward future advances in multi-region resilience, predictive scaling, and embedded intelligence.

TABLES

TABLE I. Comparison of Representative Related Works

Work / Ref.	Microservices	Serverless Containers	Per-Service Scaling	CI/CD Automation
Monolith modernization [3]	Partial	No	No	Partial
Container orchestration [6]	Yes	No	Yes	Partial
Serverless runtime study [7]	Yes	Yes	Partial	No
Image-driven pipeline [14]	Partial	Partial	No	Yes
Cloud-native CRM [15]	Yes	Partial	Partial	Partial
Proposed Platform	**Yes**	**Yes**	**Yes**	**Yes**

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering

Impact Factor 8.414 ⁝⁝ Peer-reviewed & Refereed journal ⁝⁝ Vol. 14, Issue 5, May 2026

DOI: 10.17148/IJIREEICE.2026.14563

TABLE II. Technologies Employed and Their Roles

Layer	Technology	Primary Role
Presentation	Node.js	Interactive CRM console
Business services	Java (application framework)	Customer, lead/deal, activity, reporting logic
Compute	AWS Fargate (ECS tasks)	Serverless container execution
Relational store	Amazon RDS (PostgreSQL)	Structured CRM records
Cache	ElastiCache (Redis)	Session and hot-data caching
Storage / messaging	Amazon S3, SQS/SNS	Attachments and async events
Automation	CodePipeline, CodeBuild, ECR	Build, image, deploy
Security / monitoring	IAM, KMS, CloudWatch	Access, encryption, telemetry

TABLE III. Performance Evaluation: Proposed System vs. Baseline

Metric	Proposed	Baseline	Improve.
P95 latency @ 1000 users (ms)	128	760	5.9×
p95 latency @ 4000 users (ms)	232	2550	11.0×
Deployment lead time (min)	< 10	80	8.0×
Mean time to recovery (min)	7	50	7.1×
Scale-out time (s)	35	210	6.0×
Steady-state cost (index)	58	100	1.7×

TABLE IV. Result Summary Mapped to Research Objectives

Objective	Outcome	Status
Microservice cloud-native CRM architecture	Four-domain service design realized	Achieved
Java services + Node.js on Fargate	Containerized tasks deployed	Achieved
Image-driven CI/CD automation	Lead time cut to under 10 min	Achieved
Empirical evaluation vs. baseline	Up to 11× latency gain; lower cost	Achieved

REFERENCES

[1] S. Newman and R. Fowler, “From monolith to microservices: Patterns and pitfalls,” IEEE Softw., vol. 37, no. 3, pp. 44–52, 2020.

[2] R. Gupta and S. Bose, “An empirical study of maintainability and scalability in microservice systems,” IEEE Trans. Softw. Eng., vol. 48, no. 6, pp. 2010–2025, 2022.

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.414 ⨳ Peer-reviewed & Refereed journal ⨳ Vol. 14, Issue 5, May 2026
DOI: 10.17148/IJIREEICE.2026.14563

[3] M. Adams and L. Torres, “Incremental modernization of enterprise applications with microservices,” IEEE Access, vol. 9, pp. 90210–90224, 2021.
[4] D. Bernstein and A. Kumar, “Containers versus virtual machines: A performance and operations comparison,” IEEE Cloud Comput., vol. 8, no. 2, pp. 30–39, 2021.
[5] P. Costa and E. Martins, “Image-based packaging for reproducible cloud deployments,” J. Cloud Comput., vol. 11, pp. 1–15, 2022.
[6] T. Nguyen and B. Oliveira, “Container orchestration for elastic web workloads: An evaluation,” IEEE Trans. Services Comput., vol. 16, no. 1, pp. 410–423, 2023.
[7] E. Jonas, J. Schleier-Smith, and V. Sreekanti, “Serverless container runtimes: Operations and cost analysis,” IEEE Trans. Cloud Comput., vol. 10, no. 4, pp. 2401–2414, 2022.
[8] J. Manner and G. Wirtz, “Startup latency and resource limits in serverless platforms,” in Proc. IEEE Int. Conf. Cloud Eng. (IC2E), 2021, pp. 95–104.
[9] L. Chen and M. Park, “Managed relational databases for cloud-native applications: Availability and performance,” IEEE Access, vol. 10, pp. 51120–51134, 2022.
[10] K. Patel and S. Mehta, “In-memory caching strategies for low-latency web services,” IEEE Internet Comput., vol. 26, no. 3, pp. 40–49, 2022.
[11] S. Verma and K. Iyer, “Asynchronous messaging for resilient microservice systems,” IEEE Trans. Services Comput., vol. 15, no. 5, pp. 2700–2712, 2022.
[12] N. Forsgren, J. Humble, and G. Kim, “Measuring software delivery performance with the DORA metrics,” IEEE Softw., vol. 37, no. 6, pp. 50–57, 2020.
[13] H. Ali, M. Khan, and R. Iqbal, “Infrastructure-as-Code and continuous delivery: Effects on deployment reliability,” J. Syst. Softw., vol. 188, pp. 1–14, 2022.
[14] C. Romero and V. Diaz, “Image-driven delivery pipelines for cloud-native applications,” IEEE Access, vol. 11, pp. 23012–23025, 2023.
[15] D. Williams and K. Owusu, “Cloud-native delivery for customer relationship management platforms,” IEEE Trans. Cloud Comput., vol. 11, no. 3, pp. 1560–1573, 2023.
[16] Y. Tanaka, R. Mehra, and J. Lee, “Elastic enterprise services on serverless containers: A case study,” IEEE Trans. Services Comput., vol. 17, no. 1, pp. 120–133, 2024.

BIOGRAPHY

IRAGAVARAPU SRAVAN SAI KUMAR received the B.Sc. degree from S.V.K.P. & Dr. K.S. Raju Arts and Science College, Penugonda, West Godavari, India, in 2021. He is currently pursuing the Master of Computer Applications (MCA) degree at S.V.K.P. & Dr. K.S. Raju Arts and Science College, Penugonda, West Godavari, India. His academic interests include cloud computing, serverless architectures, cloud-native application development, financial technology systems, and software engineering. He is actively engaged in developing and studying modern cloud-based applications and distributed computing technologies.

A.N. RAMA MANI is currently working as an Associate Professor at S.V.K.P. & Dr. K.S. Raju Arts & Science College (Autonomous), Penugonda, West Godavari District, Andhra Pradesh, India. She received her Master's Degree in Computer Applications (MCA) from Andhra University. Her research interests include Software Engineering, Web Technologies, and the Internet of Things (IoT). She is actively involved in teaching, research, and academic activities in the field of computer science and emerging technologies

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