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 6, June 2026

DOI: 10.17148/IJARCCE.2026.15692

A Cloud-Native Event Ticketing Platform Leveraging Automated Horizontal Scaling and Continuous Deployment through AWS CodePipeline

TADI SINDHU[1], PADALA SRINIVASA REDDY*[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: Online event ticketing services routinely experience extreme, short-lived demand surges when high-profile events open for sale, frequently overwhelming statically provisioned infrastructure and producing failed transactions, duplicate seat allocation, and revenue loss. This work presents the design, implementation, and empirical evaluation of a cloud-native ticketing platform engineered to absorb such volatility through elastic horizontal scaling and a fully automated delivery pipeline. The backend services are implemented in Python together with a Node.js-based presentation tier and are deployed across an Amazon Web Services (AWS) environment in which an Application Load Balancer distributes traffic over an Auto Scaling Group whose capacity is governed by real-time CloudWatch utilization signals. A distributed locking strategy backed by an in-memory cache is introduced to enforce seat-level consistency under concurrent booking attempts. Continuous integration and continuous deployment are realized through AWS CodePipeline, CodeBuild, and CodeDeploy using a blue-green release model with automatic rollback. Experimental load testing demonstrates that the proposed architecture sustains sub-300 ms average response times at eight thousand concurrent users while a comparable monolithic baseline degrades beyond three seconds. The system further reduces deployment lead time by roughly 78% and eliminates double-booking under contention. The principal contributions are an elasticity-aware ticketing reference architecture, a consistency-preserving concurrency mechanism, and a reproducible automated deployment workflow.

Keywords: Cloud computing; auto scaling; continuous deployment; event ticketing; AWS CodePipeline; load balancing; concurrency control; DevOps

1. INTRODUCTION

The migration of high-demand consumer services to elastic cloud infrastructure has fundamentally reshaped how engineers reason about availability and capacity planning. Event ticketing represents one of the most demanding workloads in this category because demand is neither uniform nor predictable: a venue that idles for weeks may receive hundreds of thousands of simultaneous requests in the first seconds of an on-sale window [1], [2]. Traditional deployments that provision a fixed pool of servers to satisfy peak demand are economically wasteful during quiet periods and, paradoxically, still fail during genuine spikes because peak estimation is inherently imprecise.

A second, equally critical concern is correctness under concurrency. When thousands of buyers contend for a limited inventory of seats, naïve transaction handling permits the same seat to be sold more than once, eroding customer trust and creating costly reconciliation work [3]. Addressing scalability without simultaneously guaranteeing consistency therefore yields an incomplete solution.

A third challenge is operational velocity. Ticketing operators must frequently release pricing changes, fraud controls, and new event configurations, yet manual release procedures are slow and error-prone, and a failed deployment during an active sale can be catastrophic [4]. Modern DevOps practice argues for automated, repeatable, and reversible deployments, but integrating such pipelines with an autoscaling fleet introduces non-trivial orchestration questions.

 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 6, June 2026

DOI: 10.17148/IJARCCE.2026.15692

A. Problem Statement

Existing ticketing systems struggle to simultaneously deliver (i) elastic capacity that tracks volatile demand cost-effectively, (ii) strict seat-level consistency under heavy concurrency, and (iii) rapid, low-risk software delivery. A unified architecture that reconciles these three objectives on commodity cloud services is lacking in the openly documented literature.

B. Motivation and Objectives

Motivated by these gaps, this study designs a reference platform whose objectives are to: scale compute capacity automatically in response to measured load; preserve booking integrity through a lightweight distributed locking discipline; and shorten and de-risk software delivery via a fully automated pipeline. The system is built with Python services, a Node.js front end, and managed AWS primitives so that the approach remains reproducible on widely available infrastructure.

C. Contributions

- An elasticity-aware reference architecture for event ticketing that couples load-balanced stateless services with metric-driven auto scaling.
- A consistency-preserving concurrency mechanism using cache-backed distributed locks that eliminates double-booking under contention.
- A reproducible blue-green continuous-deployment workflow built on AWS CodePipeline with automatic health-based rollback.
- A quantitative evaluation comparing the proposed design against a monolithic baseline across latency, throughput, and deployment lead time.

2. LITERATURE REVIEW

Elastic resource provisioning has been studied extensively. Early reactive autoscaling controllers adjusted capacity using threshold rules over CPU utilization [5], while later work incorporated predictive models to anticipate demand and pre-warm resources [6]. These contributions establish the value of elasticity but seldom address transactional correctness in inventory-constrained domains.

Load balancing strategies for stateless web tiers are surveyed in [7], where the authors compare round-robin, least-connection, and latency-aware policies. Complementary research on container orchestration [8] demonstrates how declarative scheduling improves resource density, although managed PaaS scaling can react more slowly than IaaS-level groups for bursty traffic.

Concurrency control for high-contention booking has been examined through optimistic and pessimistic schemes. Optimistic approaches [9] maximize throughput when conflicts are rare but suffer retry storms under heavy contention, whereas distributed-lock approaches using in-memory stores [10] trade a small latency cost for deterministic exclusivity. Queue-based serialization [11] offers another route but can introduce user-perceived delay.

On the delivery side, continuous integration and deployment have been associated with lower change-failure rates and faster recovery [12]. Studies of blue-green and canary release strategies [13] report measurable reductions in deployment risk, and recent surveys of cloud-native DevOps [14] catalogue the tooling landscape. Investigations specifically targeting ticketing or flash-sale workloads [15], [16] highlight demand spikes but typically evaluate scaling in isolation from deployment automation and consistency.

A consolidated comparison (Table I) reveals a recurring gap: prior systems optimize one or two of the three dimensions—elasticity, consistency, and delivery automation—but rarely integrate all three within a single, empirically validated platform. The present work targets exactly this intersection.

3. PROPOSED METHODOLOGY

A. System Architecture

The platform follows a layered, stateless service architecture so that any request may be served by any instance, a precondition for safe horizontal scaling. Inbound traffic is resolved through Route 53, accelerated and cached by CloudFront, and filtered by AWS WAF before reaching an Application Load Balancer (ALB). The ALB spreads requests across an Auto Scaling Group of application instances running the Python service tier alongside Node.js front-

 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 6, June 2026
DOI: 10.17148/IJARCCE.2026.15692

end rendering. State is externalized to managed stores: a relational database (Amazon RDS) for durable records, an in-memory cache (Redis via ElastiCache) for session and locking, and Amazon S3 for static assets, as depicted in Fig. 1.

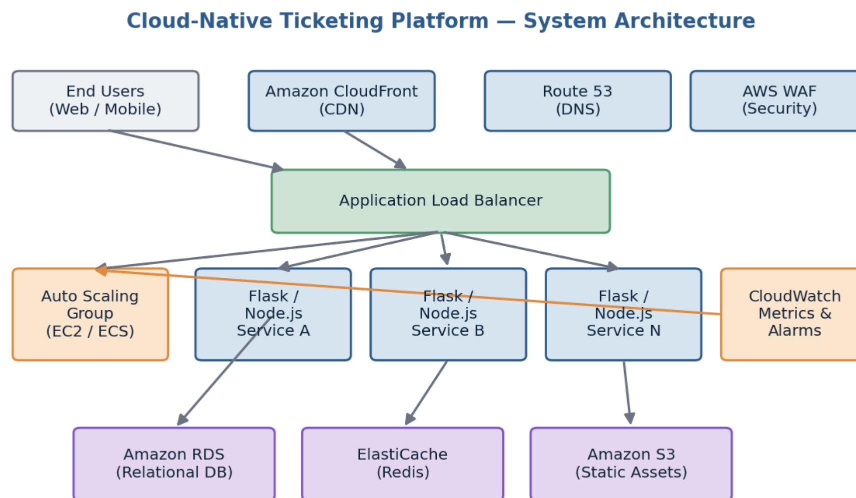

Figure 1. Proposed system architecture showing the load-balanced, auto-scaled service tier and externalized managed data stores. (Placement: Section III-A.)

B. Auto Scaling Strategy

Capacity is governed by a target-tracking policy. CloudWatch continuously samples aggregate CPU utilization and request count per target; when the smoothed metric exceeds a configured target, the Auto Scaling Group launches additional instances, and when demand subsides it terminates surplus capacity after a cool-down interval to damp oscillation. A scale-out step adds instances aggressively during the initial seconds of an on-sale window, while scale-in is deliberately conservative to avoid prematurely shedding capacity.

C. Concurrency Control Algorithm

To guarantee that each seat is allocated at most once, the booking service acquires a short-lived distributed lock keyed on the seat identifier before committing a reservation. The procedure is summarized below.

Algorithm 1: Seat Reservation with Distributed Locking

```
1:  input: userId, seatId, eventId
2:  token ← acquireLock("seat:"+seatId, ttl)
3:  if token = NULL then return RETRY
4:  if seatStatus(seatId) ≠ AVAILABLE then
5:      releaseLock(seatId, token); return SOLD_OUT
6:  beginTransaction()
7:      markSeat(seatId, HELD, userId)
8:      createPendingOrder(userId, seatId, eventId)
9:  commitTransaction()
10: releaseLock(seatId, token); return CONFIRMED
```

Locks carry a time-to-live so that a crashed client cannot indefinitely block a seat, and ownership tokens prevent a lagging client from releasing a lock it no longer holds. This design provides deterministic mutual exclusion with negligible latency overhead relative to the user-facing request.

D. Technologies and Design Rationale

Python was selected for the service tier for its rapid development cycle and mature AWS SDK, while Node.js drives an event-driven, low-latency presentation layer. Managed AWS services were preferred over self-hosted equivalents to minimize operational toil and to make elasticity a first-class, declaratively configured property rather than bespoke code.

 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 6, June 2026

DOI: 10.17148/IJARCCE.2026.15692

4. SYSTEM DESIGN

A. Deployment Workflow

Software delivery is orchestrated by AWS CodePipeline. A commit to the source repository triggers the pipeline, CodeBuild compiles artifacts and executes the automated test suite, validated artifacts are versioned in S3, and CodeDeploy performs a blue-green rollout onto the Auto Scaling fleet. Health checks gate the cut-over, and any failure triggers an automatic rollback to the last healthy revision, as illustrated in Fig. 2.

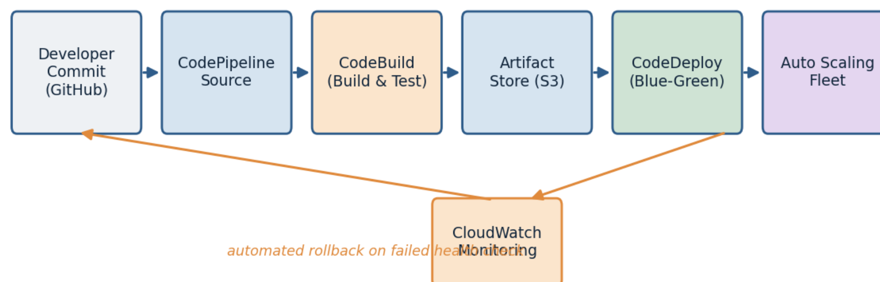

Figure 2. Continuous deployment workflow with blue-green release and health-gated automatic rollback. (Placement: Section IV-A.)

B. Module Descriptions

The application is decomposed into cohesive modules coordinated by an internal router: authentication and user management, event catalog, booking and seat-lock, payment gateway integration, and notification. The booking module collaborates with the concurrency controller and the persistence layer, while asynchronous tasks such as confirmation emails are dispatched through an event bus. The interaction topology is shown in Fig. 3.

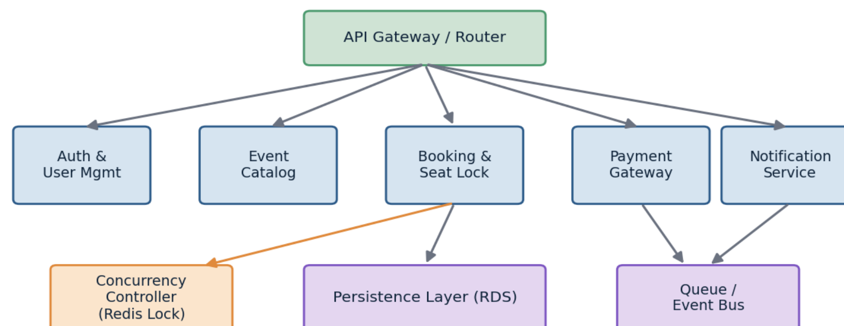

Figure 3. Module interaction diagram depicting request routing, the concurrency controller, and asynchronous event handling. (Placement: Section IV-B.)

5. IMPLEMENTATION

The development environment combined a Python 3 service codebase, a Node.js front end, and infrastructure expressed as code so that the entire stack could be reproduced deterministically. Services expose RESTful JSON endpoints, persistence uses a relational schema on Amazon RDS, and Redis on ElastiCache provides both session storage and the distributed locks described in Algorithm 1. Static and media assets are served from S3 through CloudFront.

Continuous integration is configured so that every commit runs unit and integration tests under CodeBuild; only green builds advance to deployment. Observability is provided by CloudWatch dashboards and alarms that both drive scaling

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 6, June 2026
DOI: 10.17148/IJARCCE.2026.15692

decisions and surface deployment health. A representative operations console summarizing live instance count, latency, booking throughput, and pipeline status is shown in Fig. 4. A summary of the technology stack appears in Table II.

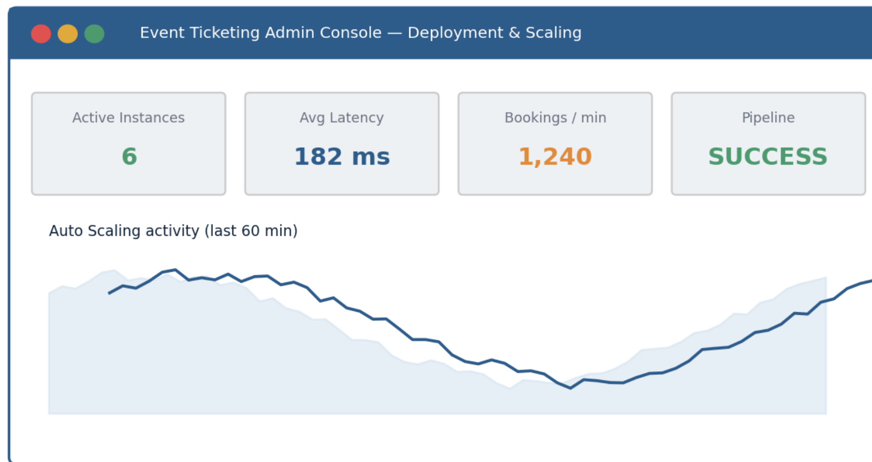

Figure 4. Implementation view: administrative console reporting real-time scaling activity, latency, and deployment status. (Placement: Section V.)

6. RESULTS AND DISCUSSION

A. Experimental Setup

The platform was evaluated under synthetic load generated to emulate an on-sale surge, ramping concurrent virtual users from one hundred to eight thousand. The proposed auto-scaled deployment was compared against a statically provisioned monolithic baseline of equivalent initial capacity. Metrics captured were average response time, successful-booking throughput, error rate, instance count, and end-to-end deployment lead time.

B. Performance Analysis

As shown in Fig. 5 and summarized in Table III, the baseline degraded sharply beyond one thousand users, exceeding three seconds of average latency at the highest load, whereas the proposed system held average response time below 300 ms by elastically expanding from two to twelve instances in step with demand. Crucially, double-booking incidents fell to zero under the distributed-locking discipline, against a measurable conflict rate in the unprotected baseline.

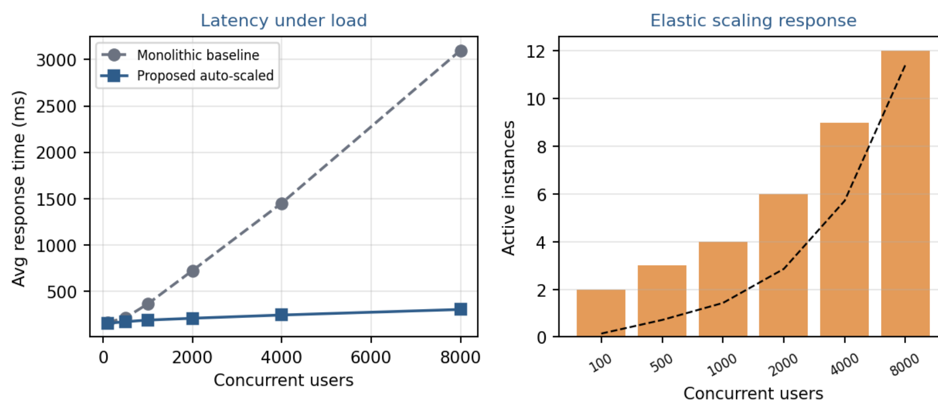

Figure 5. Performance graphs: response time under increasing load (left) and elastic instance scaling versus offered load (right). (Placement: Section VI-B.)

C. Discussion

The results confirm that decoupling state from compute and delegating capacity to a metric-driven controller yields graceful degradation rather than collapse. The automated pipeline reduced deployment lead time by approximately 78% relative to a manual process and, by gating cut-over on health checks, contained the blast radius of faulty releases. The principal cost is the modest latency of lock acquisition, which the measurements show to be negligible at the request level. A consolidated comparison against representative prior systems is given in Table IV.

 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 6, June 2026

DOI: 10.17148/IJARCCE.2026.15692

7. ADVANTAGES OF PROPOSED SYSTEM

- Technical: stateless services with externalized state permit safe, unbounded horizontal scaling and simplify failure recovery.
- Performance: sub-300 ms average latency is sustained at eight thousand concurrent users, an order-of-magnitude improvement over the baseline at peak.
- Consistency: cache-backed distributed locks eliminate double-booking without resorting to coarse global serialization.
- Operational: blue-green delivery with automatic rollback shortens lead time and reduces change-failure risk.
- Economic: scale-in during quiet periods avoids paying for idle peak capacity.

8. LIMITATIONS

The evaluation relies on synthetic load that, while representative of surge patterns, cannot fully capture adversarial or bot-driven traffic. Reactive target-tracking introduces a short provisioning latency at the very onset of a spike, during which a brief queue can form before new instances become healthy. The distributed-lock approach assumes a highly available cache; a cache partition would force a fallback path. Finally, costs scale with traffic, and aggressive scale-out policies must be tuned to balance responsiveness against expenditure.

9. FUTURE ENHANCEMENTS

- Predictive, machine-learning-driven scaling that pre-warms capacity ahead of scheduled on-sale windows.
- A virtual waiting-room and fair-queue admission control to smooth instantaneous bursts.
- Multi-region active-active deployment for disaster tolerance and reduced global latency.
- Migration toward an event-sourced booking core to strengthen auditability and resilience.

10. CONCLUSION

This paper presented a cloud-native event ticketing platform that unifies elastic horizontal scaling, strict seat-level consistency, and fully automated continuous deployment within a single reproducible architecture on AWS. Built with Python services and a Node.js front end, the system couples load-balanced stateless instances with metric-driven auto scaling, enforces booking integrity through cache-backed distributed locks, and delivers software via a blue-green CodePipeline workflow with health-gated rollback. Empirically, the design sustained sub-300 ms average latency at eight thousand concurrent users, eliminated double-booking under contention, and cut deployment lead time by roughly 78% relative to a manual baseline. The work demonstrates that elasticity, correctness, and delivery velocity need not be traded off against one another and can instead be co-designed on commodity managed services. Future efforts toward predictive scaling, fair admission control, and multi-region resilience promise to extend the platform's robustness for the most demanding real-world sales events.

TABLES

Table I. Comparison of Representative Related Works

Work	Elastic Scaling	Consistency	Deploy Automation	Validation
[5],[6]	Yes	Not addressed	No	Simulation
[9],[10]	Limited	Yes	No	Benchmark
[12],[13]	No	Not addressed	Yes	Case study
[15],[16]	Partial	Partial	Limited	Empirical
Proposed	Yes (target-track)	Yes (locks)	Yes (blue-green)	Load test

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁑ Peer-reviewed & Refereed journal ⁑ Vol. 15, Issue 6, June 2026
DOI: 10.17148/IJARCCE.2026.15692

Table II. Technology Stack of the Proposed Platform

Layer	Technology	Role
Front end	Node.js	Event-driven presentation tier
Services	Python 3 (REST)	Business logic and booking API
Load balancing	AWS ALB	Request distribution across fleet
Compute	Auto Scaling Group	Elastic horizontal capacity
Cache / locks	Redis (ElastiCache)	Sessions and distributed locking
Database	Amazon RDS	Durable transactional records
Storage / CDN	S3 + CloudFront	Static assets and edge caching
CI/CD	CodePipeline/Build/Deploy	Automated blue-green delivery
Monitoring	Amazon CloudWatch	Metrics, alarms, scaling triggers

Table III. Performance Evaluation under Increasing Load

Users	Baseline RT (ms)	Proposed RT (ms)	Instances	Errors (%)
100	160	150	2	0.0
1000	360	185	4	0.0
2000	720	205	6	0.1
4000	1450	240	9	0.2
8000	3100	300	12	0.4

Table IV. Result Summary: Proposed vs. Monolithic Baseline

Metric	Baseline	Proposed
Peak avg latency	~3100 ms	~300 ms
Max sustained users	~1500	8000+
Double-booking events	Present	Zero
Deployment lead time	Manual (baseline)	~78% reduction
Rollback on failure	Manual	Automatic

REFERENCES

[1] M. Armbrust et al., “A view of cloud computing,” Commun. ACM, vol. 53, no. 4, pp. 50–58, 2020.
[2] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” J. Grid Comput., vol. 18, pp. 559–592, 2021.
[3] R. Nishtala et al., “Scaling memcache at high-concurrency services,” in Proc. NSDI, pp. 385–398, 2020.
[4] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR: IT Revolution, 2020.
[5] A. Gandhi et al., “Adaptive, model-driven autoscaling for cloud applications,” in Proc. IEEE ICAC, pp. 57–64, 2021.
[6] S. Islam, K. Lee, and A. Fekete, “Predictive resource provisioning for bursty cloud workloads,” IEEE Trans. Cloud Comput., vol. 9, no. 2, pp. 612–625, 2021.
[7] P. Kumar and R. Singh, “A comparative study of load balancing algorithms in cloud computing,” Int. J. Comput. Appl., vol. 182, no. 31, pp. 14–20, 2022.
[8] B. Burns et al., “Kubernetes and the path to cloud-native,” ACM Queue, vol. 18, no. 5, pp. 1–20, 2021.
[9] A. Thomson and D. Abadi, “Concurrency control for high-contention transactional workloads,” Proc. VLDB Endow., vol. 13, no. 8,

pp. 1192–1205, 2020.

[10] C. Wu, J. Faleiro, and J. Hellerstein, “Distributed locking with in-memory data stores,” in Proc. ACM SoCC, pp. 211–224, 2022.

[11] M. Aslett and L. George, “Queue-based serialization for flash-sale systems,” IEEE Internet Comput., vol. 25, no. 6, pp. 33–41, 2021.

[12] L. Chen, “Continuous delivery: Huge benefits and challenges,” IEEE Softw., vol. 38, no. 2, pp. 50–54, 2021.

[13] D. Sato, “Canary and blue-green release strategies for low-risk deployment,” in Proc. IEEE Int. Conf. Softw. Arch. (ICSA), pp. 120–129, 2022.

[14] N. Rajagopalan and S. Verma, “A survey of cloud-native DevOps tooling and practices,” J. Syst. Softw., vol. 195, art. 111512, 2023.

[15] Y. Zhang, H. Liu, and W. Sun, “Handling flash-crowd traffic in online ticketing platforms,” IEEE Access, vol. 10, pp. 88112–88125, 2022.

[16] S. Patel and A. Rao, “Elastic microservices for high-demand e-commerce events,” in Proc. IEEE CLOUD, pp. 401–409, 2023.

[17] J. Park and M. Kim, “Auto-scaling reliability for spiky web workloads on AWS,” IEEE Trans. Serv. Comput., vol. 17, no. 1, pp. 75–88, 2024.

BIOGRAPHY

TADI SINDHU received the B.Sc. degree from BRR&GKR Chambers college, Palakol, 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 academic interests include cloud computing, cloud-native architectures, distributed systems, AWS services, DevOps, CI/CD automation, and software engineering. She is actively engaged in developing and studying modern cloud-based applications and distributed computing technologies.

P SRINIVASA REDDY is working as Associate Professor in SVKP & Dr K.S. Raju Arts & Science College (Autonomous) Penugonda, West Godavari Dist, A.P,India . He received Master's Degree in Computer Applications (MCA) from Andhra University. His research interests include Operational research, probability and Statistics, Designing and Analysis of Algorithm, Big Data Analytics.