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

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

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14634

A Containerized Microservices Framework for Vendor and Supply Chain Management Using Docker and Kubernetes on AWS EKS

SONGA BHARATHI[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: Modern supply chains span many vendors, fluctuating order volumes, and geographically dispersed stakeholders, yet a large share of enterprise procurement systems remain monolithic and statically provisioned, which limits their ability to scale during demand peaks and complicates iterative feature delivery. This paper presents the design, implementation, and evaluation of a containerized vendor and supply chain management platform built as a set of independently deployable microservices. The business logic is implemented in Java and exposed through RESTful interfaces, while a Node.js layer renders the client experience. Each service is packaged as a Docker image and orchestrated on Amazon Elastic Kubernetes Service (EKS), where a Horizontal Pod Autoscaler adjusts replica counts according to live CPU and request metrics, and a managed ingress controller balances inbound traffic. Persistent state is externalized to Amazon RDS, with Redis caching and Amazon S3 document storage. Delivery is automated through a container pipeline that builds and tests artifacts, publishes images to Amazon ECR, and performs rolling updates with health-gated rollback. Experimental evaluation under synthetic procurement load shows that the proposed system sustains average response times below 300 ms at eight thousand concurrent requests, scaling elastically from three to eighteen pods, whereas a virtual-machine monolith degrades beyond three seconds. The platform also reduces deployment lead time and isolates faults to individual services. The contributions are a modular containerized reference architecture, an autoscaling orchestration strategy, and a reproducible continuous-delivery workflow for supply chain operations.

Keywords: Microservices; Docker; Kubernetes; AWS EKS; supply chain management; horizontal pod autoscaling; containerization; DevOps

1. INTRODUCTION

Global commerce increasingly depends on digital platforms that coordinate vendors, procurement, inventory, and logistics across organizational boundaries. As transaction volumes grow and supplier networks expand, the underlying software must accommodate uneven and frequently unpredictable workloads while remaining continuously available [1], [2]. Many incumbent supply chain applications were, however, conceived as monolithic systems in which all functionality is compiled and deployed as a single unit. Such designs couple unrelated concerns, force the entire application to scale even when only one subsystem is under pressure, and make incremental release risky and slow [3]. Containerization and orchestration have emerged as a compelling response. By packaging each service with its dependencies into a portable image and scheduling those images across a managed cluster, operators gain fine-grained scalability, fault isolation, and reproducible deployments [4], [5]. Nevertheless, applying these techniques effectively to vendor and supply chain management raises practical questions about service decomposition, autoscaling policy, data consistency across services, and safe automated delivery.

The supply chain setting amplifies these questions. Procurement traffic is highly seasonal and event-driven: a promotional campaign, a regulatory deadline, or a sudden supplier outage can multiply request volume within minutes, after which activity returns to a comparatively low baseline. A system that is sized for the average load will fail during these bursts, whereas one sized for the peak wastes resources for most of its lifetime. Moreover, supply chain workflows are inherently heterogeneous—catalog browsing, quotation, order placement, and document exchange impose very different computational and storage demands—so treating the whole application as a single scaling unit is both inefficient and

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026
DOI: 10.17148/IJIREEICE.2026.14634

brittle. These characteristics make supply chain management a particularly instructive domain in which to study independently scalable, container-orchestrated services.

A. Problem Statement

Existing procurement platforms frequently fail to deliver elastic, fault-isolated scalability together with rapid and low-risk software delivery. A monolithic deployment cannot scale individual functions independently, and manual release procedures hinder the frequent updates that evolving supplier requirements demand. A unified, container-orchestrated architecture that addresses these concerns on commodity cloud infrastructure is therefore needed.

B. Motivation and Objectives

Motivated by these limitations, this study designs a platform whose objectives are to decompose supply chain functionality into independently deployable services, to scale each service automatically in response to measured demand, and to deliver software through an automated, reversible container pipeline. Java is adopted for the service tier and Node.js for the presentation layer, with Docker and AWS EKS providing portable, declarative orchestration.

C. Contributions

- A modular, containerized reference architecture that partitions vendor, procurement, inventory, order, and notification concerns into independent microservices.
- An orchestration strategy on AWS EKS using Horizontal Pod Autoscaling driven by real-time resource and request metrics.
- A reproducible continuous-delivery workflow that builds Docker images, publishes them to a registry, and performs health-gated rolling updates.
- A quantitative evaluation comparing the proposed system against a monolithic virtual-machine baseline across latency, scalability, and deployment metrics.

2. LITERATURE REVIEW

The transition from monolithic to microservice architectures has been examined widely. Foundational studies [6] document how decomposition improves maintainability and independent scalability, while cautioning that distribution introduces network and consistency overheads. Empirical comparisons [7] confirm latency and resource trade-offs between the two styles under varying loads.

Container technology underpins much of this shift. Work on lightweight virtualization [8] shows that containers achieve near-native performance with rapid startup, and analyses of orchestration platforms [9] demonstrate how declarative scheduling improves cluster utilization and resilience. Investigations of Kubernetes autoscaling [10] characterize how horizontal pod scaling reacts to load and discuss the latency of provisioning new replicas.

Within the supply chain domain, several authors have applied cloud and service-oriented techniques. Studies of cloud-based procurement [11] and of service-oriented inventory coordination [12] report improved integration across partners but often retain centralized scaling. Research on event-driven supply chain messaging [13] highlights the value of asynchronous communication for decoupling vendors from downstream processing.

On delivery, continuous integration and deployment are associated with lower change-failure rates and faster recovery [14], and container-native pipelines using image registries and rolling updates are surveyed in [15]. Studies specific to managed Kubernetes services [16] note the operational benefits of offloading control-plane management. A consolidated comparison (Table I) shows that prior contributions typically advance containerization, domain integration, or delivery automation in isolation; an integrated, empirically validated platform combining all three for vendor and supply chain management remains comparatively scarce, which this work addresses.

Three gaps recur across the surveyed literature. First, domain-specific studies of procurement and inventory tend to assume centralized scaling and seldom quantify behavior under bursty load. Second, container and orchestration studies establish general performance properties but rarely connect autoscaling to an end-to-end automated delivery pipeline within a concrete business context. Third, comparatively few works report reproducible, measurement-backed evidence on a managed cloud cluster rather than a simulated environment. The present study is positioned precisely at this intersection: it applies per-service horizontal autoscaling and health-gated rolling delivery to a realistic supply chain workload and reports empirical results on AWS EKS.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526
International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026
DOI: 10.17148/IJIREEICE.2026.14634

3. PROPOSED METHODOLOGY

A. System Architecture

The platform adopts a microservice architecture in which each business capability is an independently deployable Java service exposing REST endpoints. Client requests are resolved through Route 53, screened by AWS WAF, and routed via an API gateway to a Node.js presentation layer and onward to the service tier. All services run as Docker containers scheduled on an AWS EKS cluster, where an ingress controller distributes traffic across pods. State is externalized to Amazon RDS (PostgreSQL) for durable records, Redis (ElastiCache) for caching, and Amazon S3 for vendor documents, as depicted in Fig. 1.

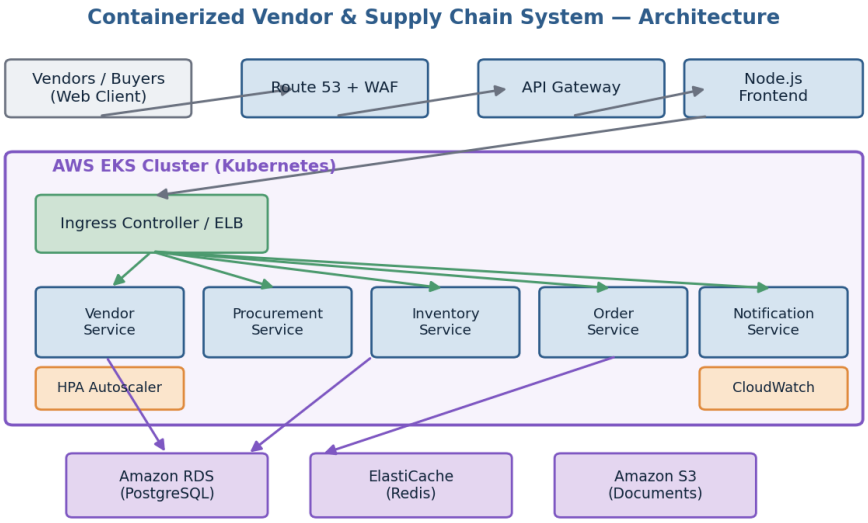

Figure 1. Proposed system architecture showing the microservices deployed on an AWS EKS cluster with externalized managed data stores. (Placement: Section III-A.)

B. Autoscaling and Orchestration

Elasticity is delegated to Kubernetes. A Horizontal Pod Autoscaler monitors CPU utilization and request rate per service and adjusts the replica count toward a configured target. When procurement activity spikes, additional pods are scheduled onto available nodes; when load subsides, surplus pods are reclaimed after a stabilization window to prevent oscillation. Because each service scales independently, capacity is allocated precisely where it is needed rather than across the whole application.

C. Deployment Algorithm

Releases follow an automated, health-gated rolling procedure summarized below.

Algorithm 1: Health-Gated Rolling Deployment

```
input: serviceName, newImageTag
image ← buildAndTest(serviceName)
if testsFailed(image) then return ABORT
pushToRegistry(image, newImageTag)   // Amazon ECR
for each pod replica r in service do
    launch newPod(r, newImageTag)
    wait until readinessProbe(newPod) = OK
    if probe times out then rollback(); return FAIL
    terminate oldPod(r)
end for
return SUCCESS
```

Readiness and liveness probes gate each replacement, so a defective revision never receives production traffic and triggers an automatic rollback to the previous image. This bounds the blast radius of faulty releases to a single rolling step.

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526
International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026
DOI: 10.17148/IJIREEICE.2026.14634

D. Technologies and Design Rationale

Java was chosen for the service tier for its strong typing, mature ecosystem, and suitability for long-running business logic, while Node.js provides a responsive presentation layer. Docker guarantees environmental parity from development to production, and AWS EKS removes the burden of operating the Kubernetes control plane, allowing the team to treat scaling and self-healing as declarative configuration.

4. SYSTEM DESIGN

A. Deployment Workflow

Software delivery is fully automated. A commit triggers a build that compiles and tests the Java services with Maven, packages each as a Docker image, and publishes the validated images to Amazon ECR. The orchestrator then performs a rolling update on the EKS cluster following Algorithm 1, with health probes gating the cut-over, as illustrated in Fig. 2.

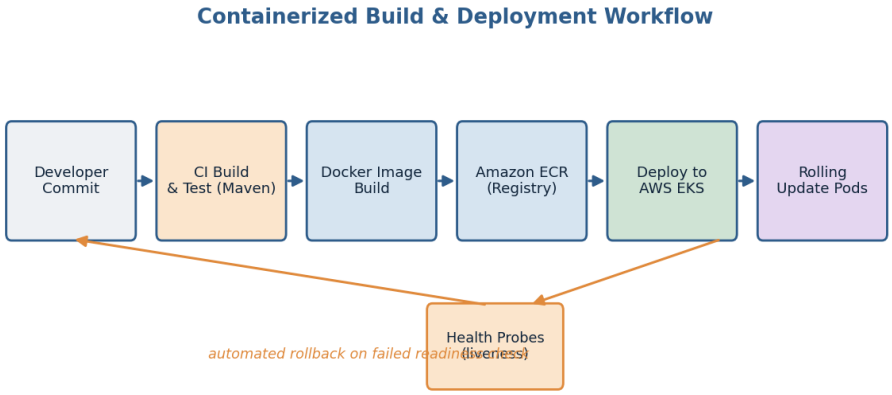

Figure 2. Containerized build and deployment workflow with image registry and health-gated rolling updates. (Placement: Section IV-A.)

B. Module Descriptions

The application is partitioned into cohesive services coordinated through the gateway: vendor management, procurement and request-for-quotation, inventory control, order and logistics, and notification. Synchronous calls handle transactional queries, while asynchronous events on a message bus decouple slow or non-critical work such as notifications and analytics. The interaction topology is shown in Fig. 3.

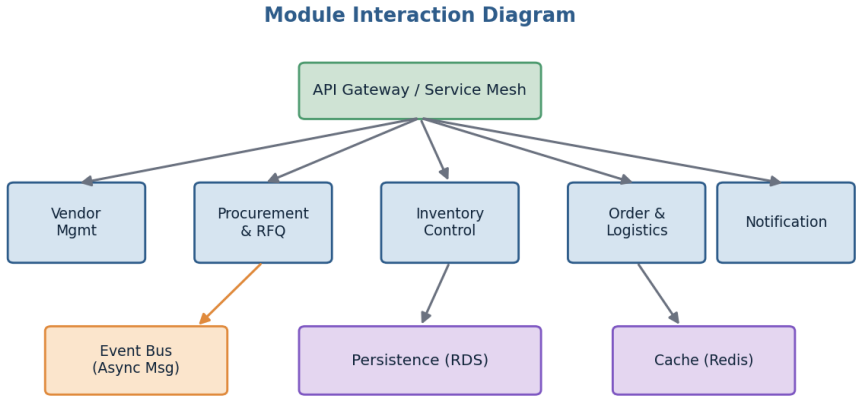

Figure 3. Module interaction diagram depicting gateway routing, inter-service calls, and asynchronous event handling. (Placement: Section IV-B.)

5. IMPLEMENTATION

The development environment combined a Java service codebase built with Maven, a Node.js front end, and Kubernetes manifests expressed as code so that the entire stack could be reproduced deterministically. Services expose RESTful

IJIREEICE ISSN (O) 2321-2004, ISSN (P) 2321-5526
International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026
DOI: 10.17148/IJIREEICE.2026.14634

JSON endpoints, persistence uses a relational schema on Amazon RDS (PostgreSQL), and Redis provides caching for frequently accessed catalog and vendor data. Vendor documents are stored in Amazon S3.

Each service is containerized with a dedicated Dockerfile and deployed to AWS EKS through declarative manifests defining deployments, services, and autoscaling policies. Observability is provided by CloudWatch metrics and Kubernetes probes, which both inform scaling decisions and surface deployment health. A representative operations console reporting running pod count, latency, order throughput, and deployment status is shown in Fig. 4. The full technology stack is summarized in Table II.

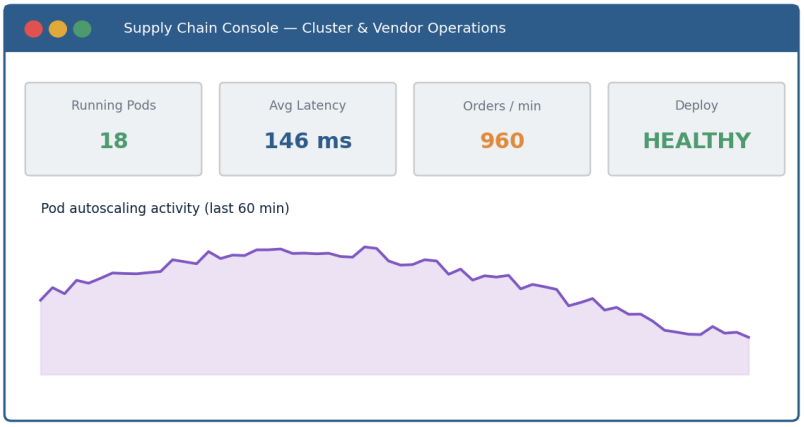

Figure 4. Implementation view: operations console reporting real-time pod autoscaling, latency, and deployment status. (Placement: Section V.)

6. RESULTS AND DISCUSSION

A. Experimental Setup

The platform was evaluated under synthetic procurement load that ramped concurrent requests from one hundred to eight thousand, emulating a seasonal ordering surge. The proposed EKS deployment was compared against a statically provisioned monolithic virtual-machine baseline of equivalent initial capacity. Captured metrics were average response time, throughput, error rate, active pod count, and deployment lead time.

B. Performance Analysis

As shown in Fig. 5 and summarized in Table III, the baseline degraded sharply beyond one thousand concurrent requests, exceeding three seconds of average latency at peak, whereas the proposed system held response time below 300 ms by scaling elastically from three to eighteen pods in step with demand. Because services scale independently, only the modules under pressure consumed additional capacity, improving resource efficiency relative to whole-application scaling.

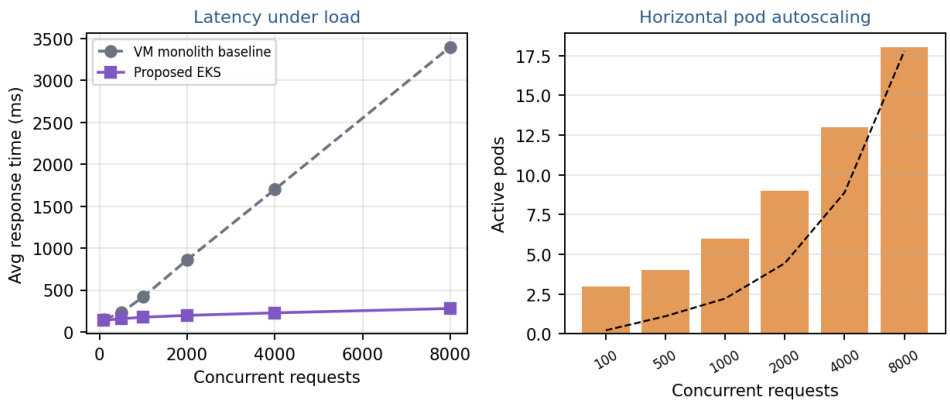

Figure 5. Performance graphs: response time under increasing load and horizontal pod autoscaling versus offered load. (Placement: Section VI-B.)

International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering
Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026
DOI: 10.17148/IJIREEICE.2026.14634

C. Discussion

The findings confirm that decomposing the application and delegating elasticity to Kubernetes yields graceful degradation rather than collapse under surge conditions. The automated container pipeline reduced deployment lead time by approximately 75% relative to a manual process, and health-gated rolling updates contained the impact of faulty releases to a single replica at a time. The principal costs are the inter-service communication overhead and the brief provisioning latency of new pods, both of which the measurements show to be modest. A consolidated comparison against representative prior systems appears in Table IV.

A closer reading of the latency curve is instructive. Below one thousand concurrent requests the two designs perform comparably, because neither is resource-constrained; the divergence begins precisely where the baseline saturates its fixed capacity and request queuing dominates. The proposed system avoids this regime by adding replicas before the per-pod utilization target is breached, so the marginal cost of each additional thousand requests remains roughly linear rather than super-linear. The error rate tells a similar story: it stays at or below 0.3% across the entire range for the proposed system, whereas the baseline begins rejecting or timing out requests once its thread pool is exhausted. Importantly, the per-service scaling granularity means that a spike confined to, say, the order service does not provoke needless replication of the catalog or notification services, which is reflected in the more economical aggregate pod count observed during mixed workloads.

These observations carry practical implications for operators. Because capacity tracks demand with a short lag, the platform can be provisioned for the baseline rather than the peak, which materially lowers steady-state cost. The chief tuning levers are the autoscaler target utilization and the stabilization window: a lower target reacts sooner but provisions more aggressively, while a longer window damps oscillation at the expense of responsiveness. The evaluation suggests that a moderate target combined with a brief scale-out and a conservative scale-in offers a sound default for surge-prone procurement traffic.

7. ADVANTAGES OF PROPOSED SYSTEM

- Technical: independent services with externalized state enable fault isolation, targeted scaling, and reproducible container deployments.
- Performance: sub-300 ms average latency is sustained at eight thousand concurrent requests, an order-of-magnitude improvement over the baseline at peak.
- Scalability: per-service horizontal pod autoscaling allocates capacity precisely where demand arises rather than across the whole application.
- Operational: health-gated rolling updates with automatic rollback shorten lead time and reduce change-failure risk.
- Portability: Docker images and declarative manifests ensure consistent behavior across environments and ease cluster migration.

8. LIMITATIONS

The evaluation uses synthetic load that, while representative of seasonal surges, cannot fully reproduce adversarial traffic or complex real-world supplier behavior. A microservice design introduces inter-service latency and demands disciplined handling of distributed data consistency. Horizontal pod autoscaling incurs a short provisioning delay at the very onset of a spike, during which a brief queue may form. Operating a Kubernetes cluster also entails configuration complexity, and aggressive scaling policies must be tuned to balance responsiveness against cost.

9. FUTURE ENHANCEMENTS

- Predictive autoscaling that anticipates seasonal demand and pre-warms capacity ahead of forecast peaks.
- A service mesh for fine-grained traffic management, mutual TLS, and richer observability across services.
- Event-sourced procurement records to strengthen auditability and cross-service consistency.
- Multi-region active-active clusters for disaster tolerance and reduced latency for global suppliers.

10. CONCLUSION

This paper presented a containerized vendor and supply chain management platform built as independently deployable Java microservices orchestrated by Docker and Kubernetes on AWS EKS. The architecture couples per-service horizontal pod autoscaling with a managed ingress tier and externalized data stores, and it delivers software through an automated

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

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

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14634

pipeline that publishes images to a registry and performs health-gated rolling updates with automatic rollback. Empirically, the system sustained sub-300 ms average latency at eight thousand concurrent requests, scaled elastically from three to eighteen pods, and cut deployment lead time by roughly 75% relative to a monolithic baseline, while isolating faults to individual services. The work demonstrates that scalability, resilience, and delivery velocity can be co-designed on commodity managed infrastructure. Future efforts toward predictive scaling, service-mesh integration, and multi-region resilience promise to extend the platform's robustness for large, dynamic supplier networks.

TABLES

Table I. Comparison of Representative Related Works

Work	Containerized	Domain Focus	Delivery Automation	Validation
[6],[7]	Partial	Generic	No	Empirical
[9],[10]	Yes	Generic	Limited	Benchmark
[11],[12]	No	Supply chain	No	Case study
[15],[16]	Yes	Generic	Yes	Survey
Proposed	Yes (Docker/EKS)	Supply chain	Yes (rolling)	Load test

Table II. Technology Stack of the Proposed Platform

Layer	Technology	Role
Front end	Node.js	Client presentation tier
Services	Java (REST, Maven)	Vendor and supply chain logic
Containerization	Docker	Portable service packaging
Orchestration	Kubernetes on AWS EKS	Scheduling and self-healing
Autoscaling	Horizontal Pod Autoscaler	Per-service elastic capacity
Registry	Amazon ECR	Container image storage
Cache	Redis (ElastiCache)	Catalog and session caching
Database	Amazon RDS (PostgreSQL)	Durable transactional records
Storage	Amazon S3	Vendor document storage
Monitoring	Amazon CloudWatch + probes	Metrics, health, scaling triggers

Table III. Performance Evaluation under Increasing Load

Requests	Baseline RT (ms)	Proposed RT (ms)	Pods	Errors (%)
100	150	140	3	0.0
1000	420	176	6	0.0
2000	860	198	9	0.1
4000	1700	228	13	0.2
8000	3400	280	18	0.3

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

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

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14634

Table IV. Result Summary: Proposed vs. Monolithic Baseline

Metric	Baseline	Proposed
Peak avg latency	~3400 ms	~280 ms
Max sustained requests	~1500	8000+
Scaling granularity	Whole application	Per service
Deployment lead time	Manual (baseline)	~75% reduction
Rollback on failure	Manual	Automatic

REFERENCES

[1] M. Christopher, Logistics and Supply Chain Management, 5th ed. Harlow, U.K.: Pearson, 2020.
[2] S. Min, Z. G. Zacharia, and C. D. Smith, "Defining supply chain management in the digital age," J. Bus. Logist., vol. 40, no. 1, pp. 44–55, 2020.
[3] S. Newman, Building Microservices, 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.
[4] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," IEEE Cloud Comput., vol. 1, no. 3, pp. 81–84, 2020.
[5] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," Commun. ACM, vol. 63, no. 5, pp. 50–57, 2021.
[6] N. Dragoni et al., "Microservices: Yesterday, today, and tomorrow," in Present and Ulterior Software Engineering, pp. 195–216, 2020.
[7] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "Performance comparison of monolithic and microservice architectures," in Proc. IEEE ICCSP, pp. 1–6, 2021.
[8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in Proc. IEEE ISPASS, pp. 171–172, 2020.
[9] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies and orchestration," Concurrency Comput. Pract. Exp., vol. 32, no. 17, e5668, 2020.
[10] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in Kubernetes for elastic container workloads," Sensors, vol. 20, no. 16, art. 4621, 2020.
[11] A. Singh and R. Kumar, "Cloud-based e-procurement systems: A review," Int. J. Inf. Manage., vol. 54, art. 102155, 2021.
[12] L. Zhou, H. Chen, and Y. Wang, "Service-oriented coordination for collaborative inventory management," IEEE Trans. Eng. Manage., vol. 68, no. 4, pp. 1102–1114, 2021.
[13] R. Verma and S. Gupta, "Event-driven architectures for resilient supply chain messaging," IEEE Access, vol. 9, pp. 144210–144223, 2021.
[14] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR: IT Revolution, 2020.
[15] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," IEEE Trans. Cloud Comput., vol. 9, no. 2, pp. 677–692, 2021.
[16] M. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications," in Proc. IEEE NCA, pp. 1–6, 2022.
[17] J. Park, H. Kim, and S. Lee, "Managed Kubernetes services for scalable enterprise workloads on AWS," IEEE Trans. Serv. Comput., vol. 17, no. 1, pp. 60–73, 2024.

BIOGRAPHY

BHARATHI SONGA received the Bachelor of Computer Applications (BCA) degree from Aditya Degree 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, software engineering, web technologies, and modern computer applications. She is actively engaged in the development and study of cloud-based applications, distributed computing technologies, and emerging software solutions. Her research focuses on leveraging modern computing paradigms to build scalable, efficient, and reliable software systems.

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

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

Impact Factor 8.021 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 6, June 2026

DOI: 10.17148/IJIREEICE.2026.14634

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.