Development and Deployment of Microservice-Based Architecture System for Efficient Application Loading in Cloud Environments

[1]Ikediego Henrietta Onyinyechi, [2]Akawuku I. Godspower, [3]Chekwube Georgina Nwankwo, [4]Adejumo O. Samuel

[1,2,4]Department of Computer Science, Nnamdi Azikiwe University, Awka, Anambra State

[3]Department of Computer Science, Chukwuemeka Odumegwu Ojukwu, Uli, Anambra State

DOI: https://doi.org/10.51584/IJRIAS.2026.11080095

Received: 20 August 2026; Accepted: 27 August 2026; Published: 10 September 2026

ABSTRACT

Cloud-based applications require architectures that can maintain efficient application loading and performance as workload increases. This study developed and experimentally evaluated three functionally equivalent e-commerce implementations: Monolithic, Baseline Microservices, and Proposed Microservices architectures. An experimental research methodology was adopted, with the same business logic, database, and workload scenarios maintained across the implementations. The Proposed architecture integrated Docker, Kubernetes on Amazon Elastic Kubernetes Service (EKS), horizontal pod autoscaling, load balancing, Redis caching, and CloudFront delivery. Controlled workloads were generated using k6, and performance was assessed using startup time, deployment availability, response time, throughput, autoscaling, resource utilisation, inter-service latency, full user-journey performance, cache effectiveness, and frontend loading metrics. Results showed that the Monolithic architecture had the shortest startup time (1.33 s), while the Proposed architecture required 25.36 s. Under 2,000 virtual users, however, the Proposed architecture achieved an average response time of 33.47 ms and throughput of 1,915.78 req/s, compared with 1,951.56 ms and 670.13 req/s for the Monolithic architecture. The Proposed architecture also recorded 0.00% deployment failure, scaled from 2 to 15 replicas, and reduced cache latency by 30.12% ($p = 0.0001$). Its frontend FCP (First Contentful Paint), LCP (Largest Contentful Paint) and TTI (Time to Interactive) were each 617 ms. The findings demonstrate that integrated cloud-native mechanisms can substantially improve application performance and scalability under increasing workloads.

Keywords: Cloud computing, microservices, monolithic architecture, Kubernetes, application loading, autoscaling, performance evaluation.

INTRODUCTION

Cloud computing has transformed the development and delivery of modern software applications by providing scalable computing resources, flexible deployment options, and on-demand access to infrastructure. However, the benefits of cloud computing do not automatically guarantee efficient application performance, as the underlying software architecture and deployment approach can significantly influence responsiveness, scalability, resource utilisation, and application loading efficiency. Traditional monolithic architectures package application functionalities as a single deployable unit, which can result in coarse-grained scaling, longer deployment cycles, and reduced flexibility when individual application components require modification or additional resources (Benavente et al., 2022).

Microservice Architecture (MSA) has emerged as an alternative approach in which an application is decomposed into smaller, independently deployable services, with each service responsible for a specific business function (Ileana et al., 2024). This approach supports independent deployment, scalability, fault isolation, and flexibility, and aligns closely with cloud-native practices such as containerisation, orchestration, and continuous integration and deployment. However, microservices also introduce distributed-system complexities, particularly inter-service communication, network latency, service coordination, and distributed resource management asserted

by Akawuku et al (2026). Consequently, decomposition into microservices alone does not necessarily guarantee improved application performance.

The deployment infrastructure supporting microservices therefore plays an important role in determining their operational performance. Containerisation technologies such as Docker provide portable and consistent execution environments, while Kubernetes supports automated deployment, service management, load balancing, and scaling of containerised applications (Waseem et al., 2020). Other cloud-native mechanisms, including continuous deployment, caching, and elastic scaling, can further influence application responsiveness and resource utilisation (Li et al., 2020). Nevertheless, the distributed nature of microservices means that these mechanisms must operate together effectively to realise their potential benefits (Aksakalli et al., 2021).

Despite the growing body of research on microservice architecture and cloud-native technologies, limited empirical attention has been given to application loading efficiency within complete, deployed microservice applications under controlled cloud-based workloads (Saucedo et al., 2024). Existing research has also provided comparatively limited investigation of how architectural decomposition and the combined use of orchestration, automated scaling, load balancing, caching, and related deployment mechanisms influence application performance (Alharthi et al., 2024). This creates a need for controlled experimental comparison in which equivalent application functionality is evaluated across different architectural implementations.

This study therefore develops and deploys three functionally equivalent implementations of an e-commerce application: a Monolithic architecture, a Baseline Microservices architecture, and a Proposed Microservices architecture incorporating cloud-native deployment mechanisms. The implementations are evaluated under controlled workloads using performance measures including response time, throughput, scalability, resource utilisation, inter-service communication latency, deployment availability, caching effectiveness, frontend loading performance, and full user-journey performance. The experimental design maintains the same business logic, database, and workload scenarios across the architectural implementations to support a controlled comparison.

The study aims to develop and deploy a microservice-based architecture system for efficient application loading in cloud environments and to empirically compare the performance of the three architectural implementations under varying workload conditions. The findings are intended to provide practical and empirical evidence on how architectural design and cloud-native deployment mechanisms influence the performance and scalability of cloud-based applications.

Summary of Related Works

Table 1: Summary of Related Works

S/N	Authors & Year	Method/Approach	Identified Strengths	Limitations
1	Chippagiri & Kassetty (2025)	Review of cloud-native technologies including microservices, Kubernetes and serverless computing	Provides an overview of technologies for scalable and resilient distributed systems	Conceptual study without experimental implementation or comparative performance evaluation
2	Beeraka (2025)	Enterprise microservices architecture in telecommunications	Reports improvements in scalability and availability	Does not provide controlled comparison of alternative architectures or isolate cloud-native enhancements

3	Alelyani et al. (2024)	Kubernetes scheduling strategy using Particle Swarm Optimization and Round Robin, evaluated with production-trace simulations	Improves latency, network traffic and resource balancing	Relies on simulation rather than a real Kubernetes deployment
4	Alharthi et al. (2024)	Study of reactive threshold-based autoscaling	Identifies limitations associated with delayed scaling responses	Limited evaluation of Kubernetes autoscaling within complete cloud-native applications
5	Ileana et al. (2024)	E-commerce microservice implementation using Docker, Django REST, PostgreSQL and JWT	Demonstrates modularity, scalability, flexibility and fault isolation	Focuses mainly on architecture and implementation without controlled performance evaluation under varying workloads
6	Barua & Kaiser (2024)	Cloud-enabled microservices using Docker, Kubernetes, API Gateway and autoscaling	Reports improved performance, scalability and reliability under concurrent workloads	Does not isolate the contribution of individual cloud-native mechanisms
7	Dave et al. (2024)	Review of scalable microservices in cloud-based distributed systems	Highlights Docker, Kubernetes, DevOps and independent deployment as scalability enablers	Literature-based; no empirical evaluation of deployment or application-loading performance
8	Egwom & Oladunjoye (2024)	Review of static and dynamic cloud load-balancing techniques	Highlights the importance of load distribution for response time and resource utilisation	Focuses on load-balancing algorithms rather than a complete deployed microservice system
9	Jun (2024)	Survey, interviews and theoretical analysis of microservice architecture	Highlights service decomposition, scalability, communication and security considerations	Limited empirical validation using a deployed system
10	Kaloudis (2024)	Literature and industry case-study analysis of monolithic-to-microservice transition	Identifies scalability, flexibility and fault-isolation benefits and recognises workload-dependent performance	Lacks controlled experimental comparison
11	Nogueira et al. (2024)	Mixed-method study involving software practitioners	Identifies maintainability, scalability, deployment flexibility and monitoring as adoption motivations	Focuses on migration experiences rather than application-loading performance
12	Raikar et al. (2024)	Microservices web application deployed on Google Cloud using Docker and Node.js	Demonstrates containerisation and deployment flexibility	Limited quantitative assessment of startup and application-loading performance

13	Raja & Santhi (2024)	Burst-aware autoscaling framework for containerised microservices	Supports rapid traffic-surge detection and dynamic resource provisioning	Focuses on autoscaling effectiveness rather than application startup/loading metrics
14	Saucedo et al. (2024)	Systematic mapping study of 114 microservice migration studies	Identifies limited empirical evaluation and lack of standardised benchmarking	Does not itself provide a controlled implementation-based experiment
15	Thallapally (2024)	Review of scalable microservice architectures and real-world practices	Highlights Kubernetes, orchestration, load balancing and service management	Provides no direct quantitative comparison of application-loading performance
16	Velepucha & Flores (2023)	Survey of design principles for monolithic and microservice architectures	Identifies independent scaling and fault isolation as microservice benefits	Primarily theoretical; lacks empirical loading-performance evaluation
17	Weerasinghe & Perera (2023)	Experimental evaluation of asynchronous Redis Stream communication	Reduces inter-service communication latency and improves throughput	Results are tied to the Redis implementation and may have limited generalisability
18	Wyciślik et al. (2023)	Synthetic performance comparison of Spring Boot, Micronaut and Quarkus	Provides empirical evidence on cold-start and deployment trade-offs	Synthetic testing limits direct applicability to production environments
19	Xu et al. (2023)	Resource-provisioning algorithms evaluated on Alibaba Cloud microservice clusters	Improves resource utilisation by 10–15% while maintaining latency targets	Evaluation is specific to Alibaba Cloud and requires cross-platform validation
20	Benavente et al. (2022)	Comparative analysis of microservices and monoliths using AWS, JMeter and multiple case studies	Shows microservice advantages for response time and scalability in large/complex cloud systems	Limited examination of application-loading metrics beyond response time and throughput
21	Blinowski et al. (2022)	Controlled comparison of monolithic and microservice applications across local and Azure environments	Demonstrates context-dependent performance trade-offs	Performance varies by deployment environment and implementation
22	Dinh-Tuan et al. (2022)	Evaluation of four open-source frameworks for cloud microservices	Highlights framework efficiency and smaller Docker images for deployment	Synthetic environment may not fully represent production conditions
23	Marques et al. (2022)	AWS-based predictive monitoring and proactive resource management	Improves response time and reduces service degradation compared with reactive approaches	Does not directly measure initial application-loading time

24	Zaki et al. (2022)	Microservice healthcare framework evaluated on Microsoft Azure	Reports lower response times, fewer dropped requests and better resource utilisation than SOA	Internal mechanisms responsible for performance improvements are not fully detailed
25	Aksakalli et al. (2021)	Systematic literature review of microservice deployment approaches and communication patterns	Identifies automation and load balancing as important for efficient deployment and performance	Lacks fine-grained application-loading metrics

Knowledge Gap: Lapses And Shortcomings

The reviewed studies provide valuable insights into microservice architectures and cloud-native technologies; however, they show limited focus on application loading efficiency, limited empirical evaluation of complete microservice applications under controlled cloud conditions, and limited investigation of the combined effects of Kubernetes, autoscaling, load balancing and caching. There is also insufficient controlled comparison of monolithic and microservice architectures using equivalent applications and workloads. These limitations create a gap in understanding how architectural decomposition and integrated cloud-native mechanisms influence application loading and overall performance. This study therefore addresses the gap by implementing and experimentally evaluating Monolithic, Baseline Microservices and Proposed Microservices architectures under comparable conditions.

MATERIALS AND METHODS

The study adopted an Experimental Research Methodology to evaluate the effect of deployment architecture on application performance in a cloud computing environment. Three functionally equivalent implementations of the same e-commerce application were developed: Monolithic architecture, Baseline Microservices architecture, and Proposed Microservices architecture. The implementations maintained the same business logic, database and workload scenarios to ensure that observed performance differences were attributable primarily to architectural and deployment characteristics. Docker was used for application containerisation, while the Proposed Microservices architecture was deployed on Amazon Elastic Kubernetes Service (EKS) with Kubernetes orchestration, horizontal scaling and supporting cloud-native mechanisms. GitHub Actions was used for continuous integration and continuous deployment. Performance testing was conducted using k6 under controlled virtual-user workloads, and the architectures were evaluated using response time, throughput, resource utilisation, scalability, deployment availability, network latency, cache effectiveness and application loading metrics. The collected data were analysed quantitatively using statistical summaries, tables and graphical representations to compare the performance characteristics of the three architectural implementations.

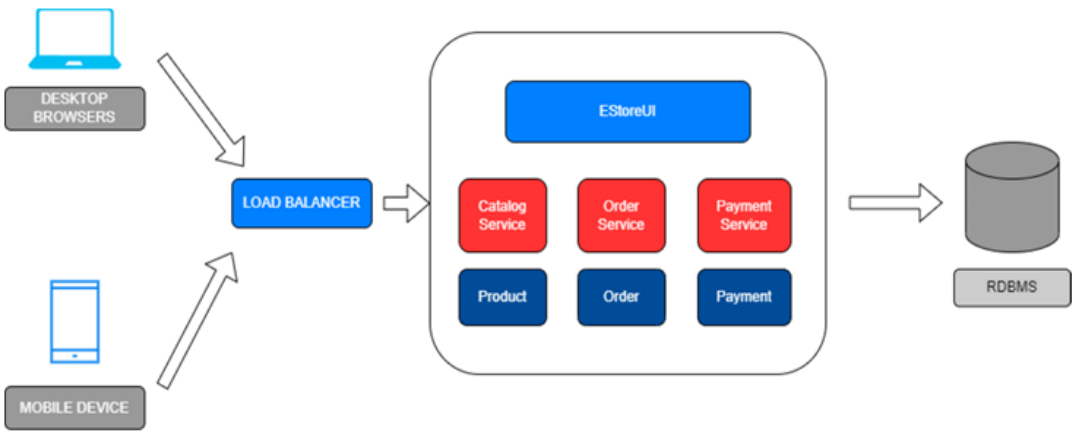

Figure 1: Architecture of the monolith system (Benavente *et al*., 2022)

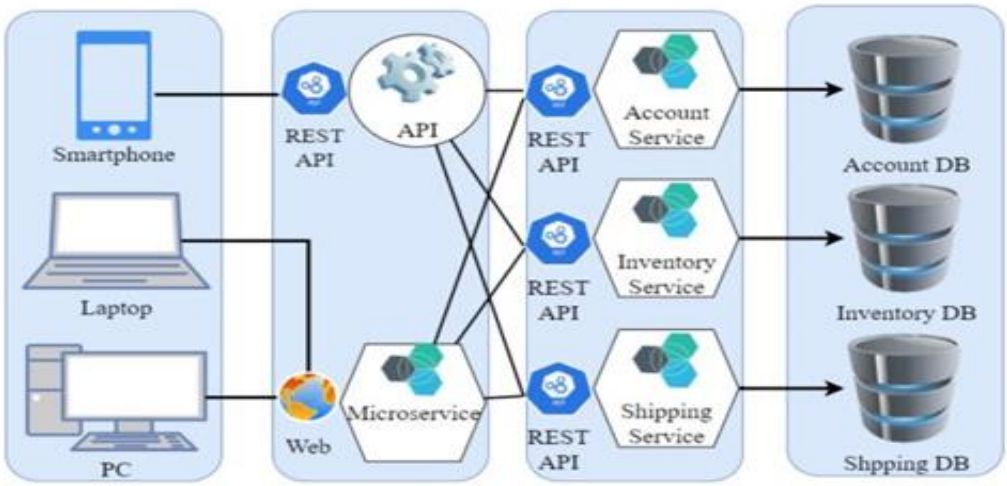

Figure 2: Architecture of the existing microservice-based system (Ileana *et al.*, 2024)

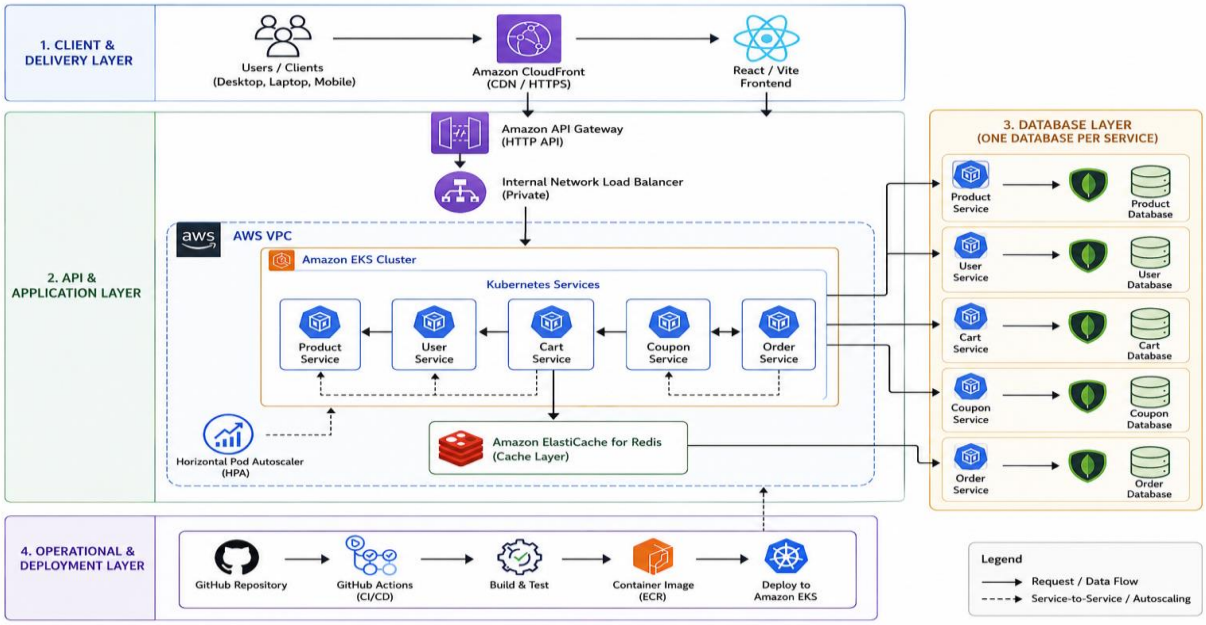

Figure 3: Architecture of the proposed microservice-based system

Functional Requirements

As shown in Table 2, the functional requirements define the core operations of the developed e-commerce system, including user management, product browsing, cart management, coupon application, order processing, payment, and administrative operations. These functions provide the common application functionality maintained across the three architectural implementations.

Table 2: Functional Requirements

S/N	Module	Functional Requirement
1	User Management	The system shall allow users to register, authenticate, and manage their accounts.
2	Product Management	The system shall allow users to browse and retrieve product information.
3	Cart Management	The system shall allow users to add, update, and remove products from their shopping carts.

4	Coupon Management	The system shall allow valid coupons and discounts to be created, managed, and applied to transactions.
5	Order Management	The system shall allow users to place orders and provide access to order information.
6	Payment	The system shall support payment processing as part of the checkout process.
7	Administration	The system shall allow authorised administrators to manage products, coupons, orders, and related system information.

Non-functional Requirements

Non-functional requirements specify the quality attributes and operational conditions expected of the system rather than its specific business functions. In this study, these requirements ensure that the developed e-commerce application operates efficiently and consistently across the different architectural implementations. The key requirements considered include performance, scalability, availability, security, reliability, maintainability, and portability, as presented in Table 3

Table 3: Non-functional Requirements

S/N	Requirement	Description
1	Performance	The system shall provide efficient response and application loading performance under varying workloads.
2	Scalability	The system shall accommodate increasing workloads through appropriate scaling mechanisms.
3	Availability	The system shall remain accessible during normal operation and deployment activities.
4	Security	The system shall provide authentication, authorisation, and secure communication between system components.
5	Reliability	The system shall process valid requests and transactions consistently without unnecessary failures.
6	Maintainability	The architecture shall support independent modification and deployment of application components.
7	Portability	The application shall support consistent deployment through containerisation.

Pseudo Code of the System

The pseudo code describes the authentication and authorisation process used by the microservices to verify incoming requests. Valid tokens allow requests to proceed, while missing, invalid, expired, or unauthorised credentials are rejected before access to protected operations is granted.

Step 1: Start

Step 2: Receive an incoming request.

Step 3: Retrieve the access token from the request cookie.

Step 4: If no access token is provided, return 401 – Unauthorized.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VIII August 2026

Step 5: Verify the access token using the access-token secret.

Step 6: If the token is invalid or expired, return 401 – Unauthorized.

Step 7: Extract the user's ID and role from the verified token.

Step 8: Attach the user ID and role to the request.

Step 9: If the requested operation requires administrator privileges, verify that the user's role is admin.

Step 10: If the user is not an administrator, return 403 – Access Denied.

Step 11: Allow the request to proceed to the appropriate service operation.

Step 12: End.

System Design

The system design illustrates the major processes, interactions, and structural components of the developed e-commerce application. Figures 4 and 5 present the Admin Operations and Customer Purchase flowcharts, respectively. Figures 6 and 7 show the customer purchase and administrative management activities, while Figures 8 and 9 illustrate the sequence of interactions during checkout and administrator order viewing. Finally, Figure 10 presents the component diagram, showing the organisation and relationships among the major components of the proposed microservice system. Together, these diagrams provide the behavioural and structural views of the developed system.

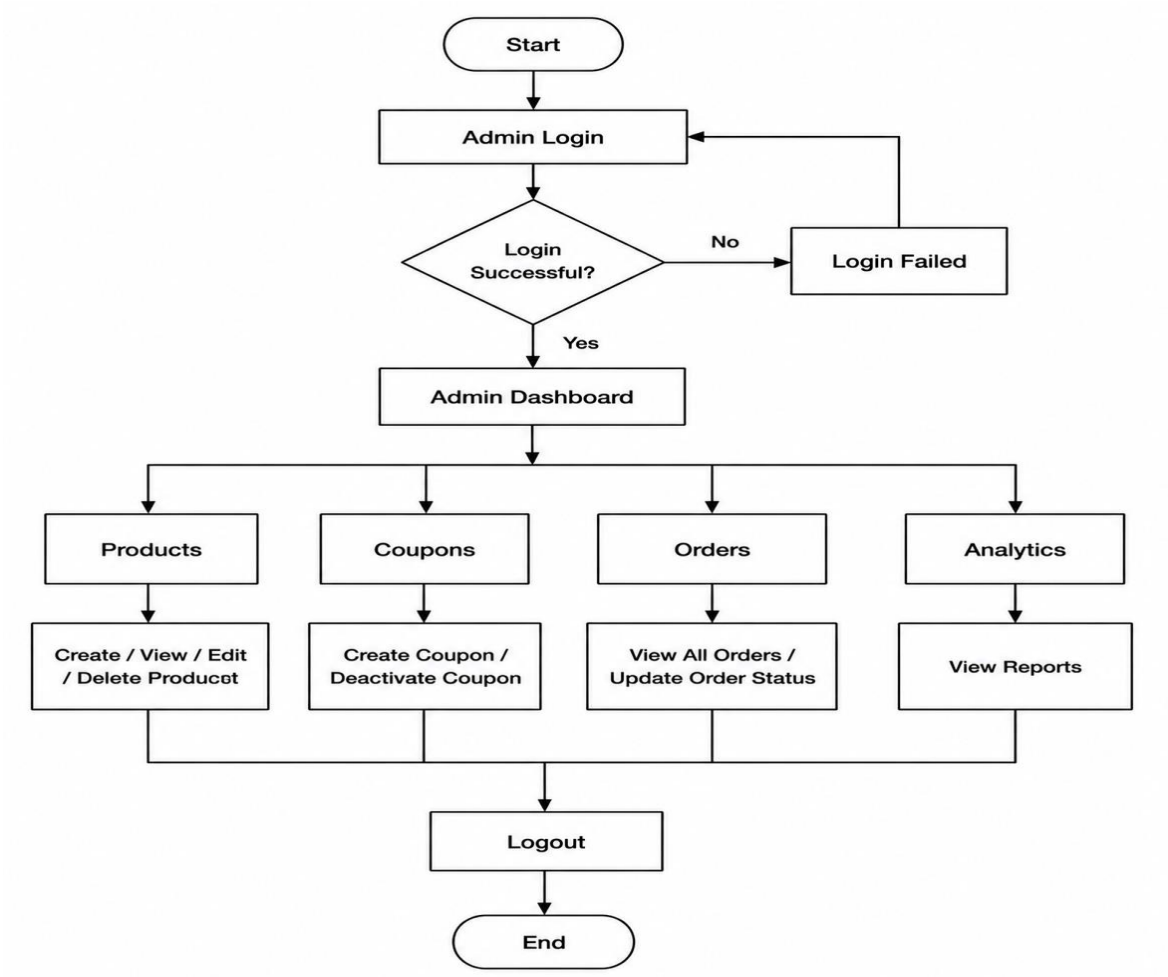

Figure 4: Admin Operations Flowchart

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VIII August 2026

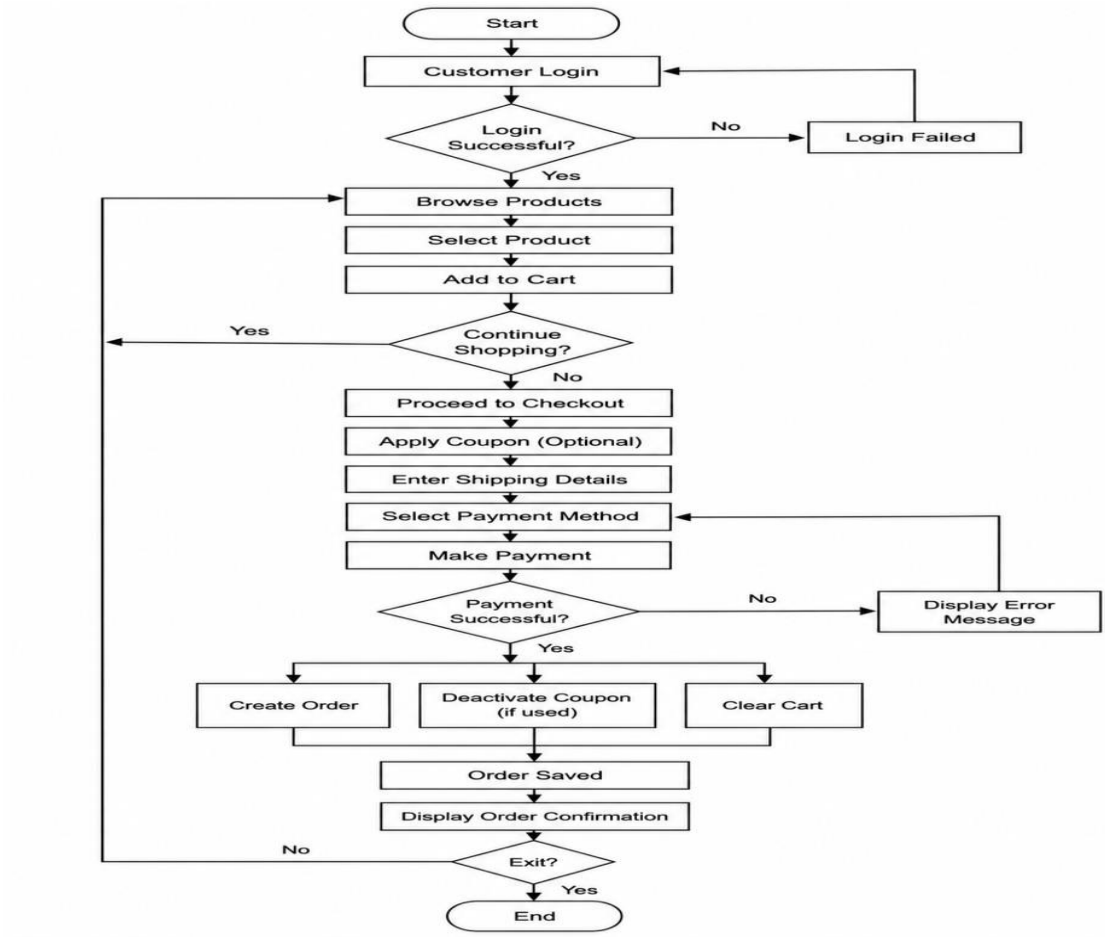

Figure 5: Customer purchase flowchart

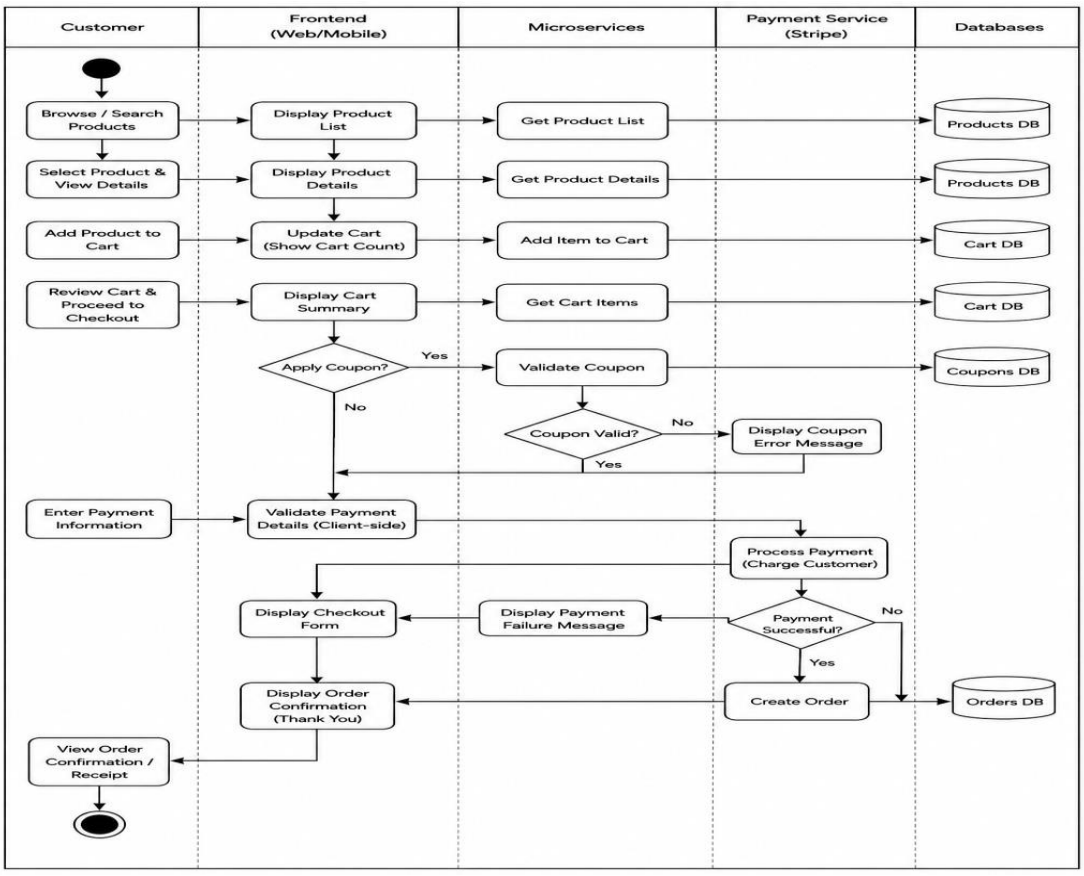

Figure 6: Customer Activity Diagram

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VIII August 2026

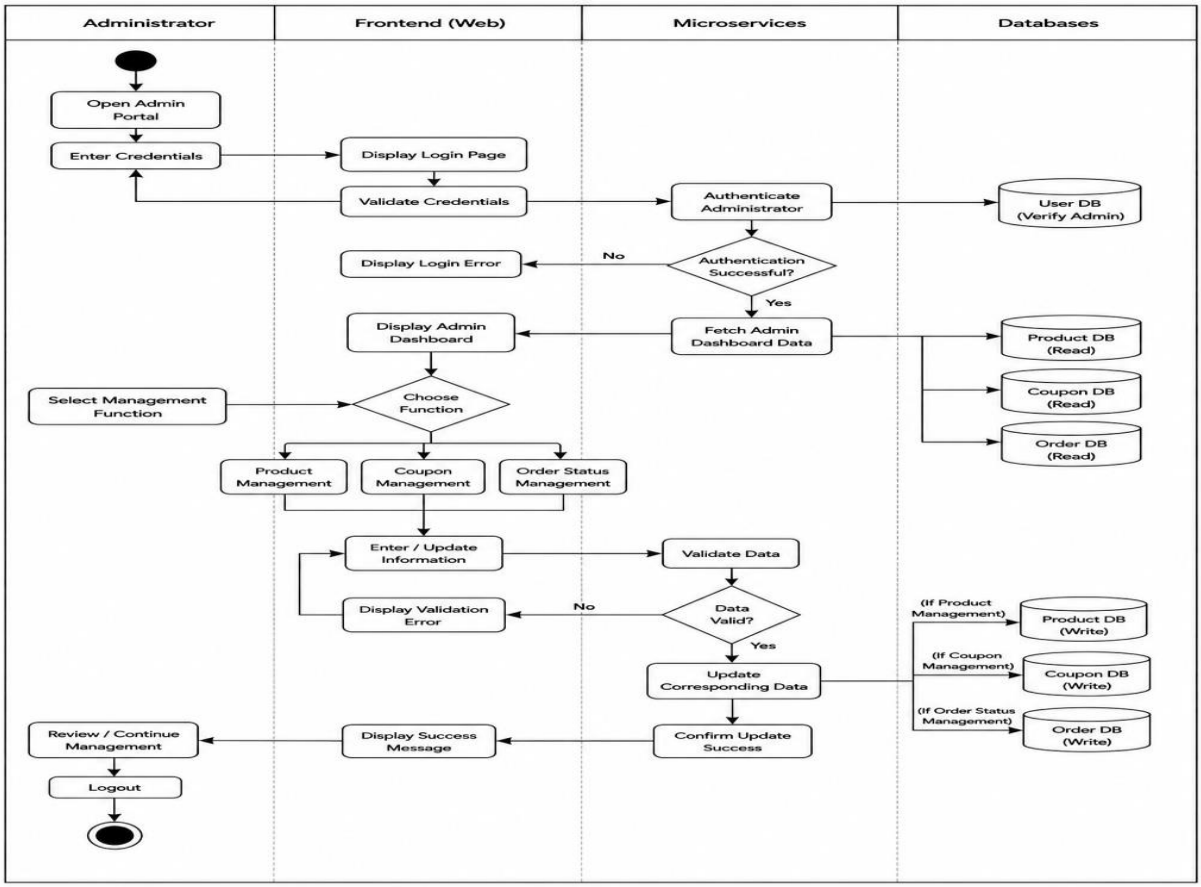

Figure 7: Administrative Activity Diagram

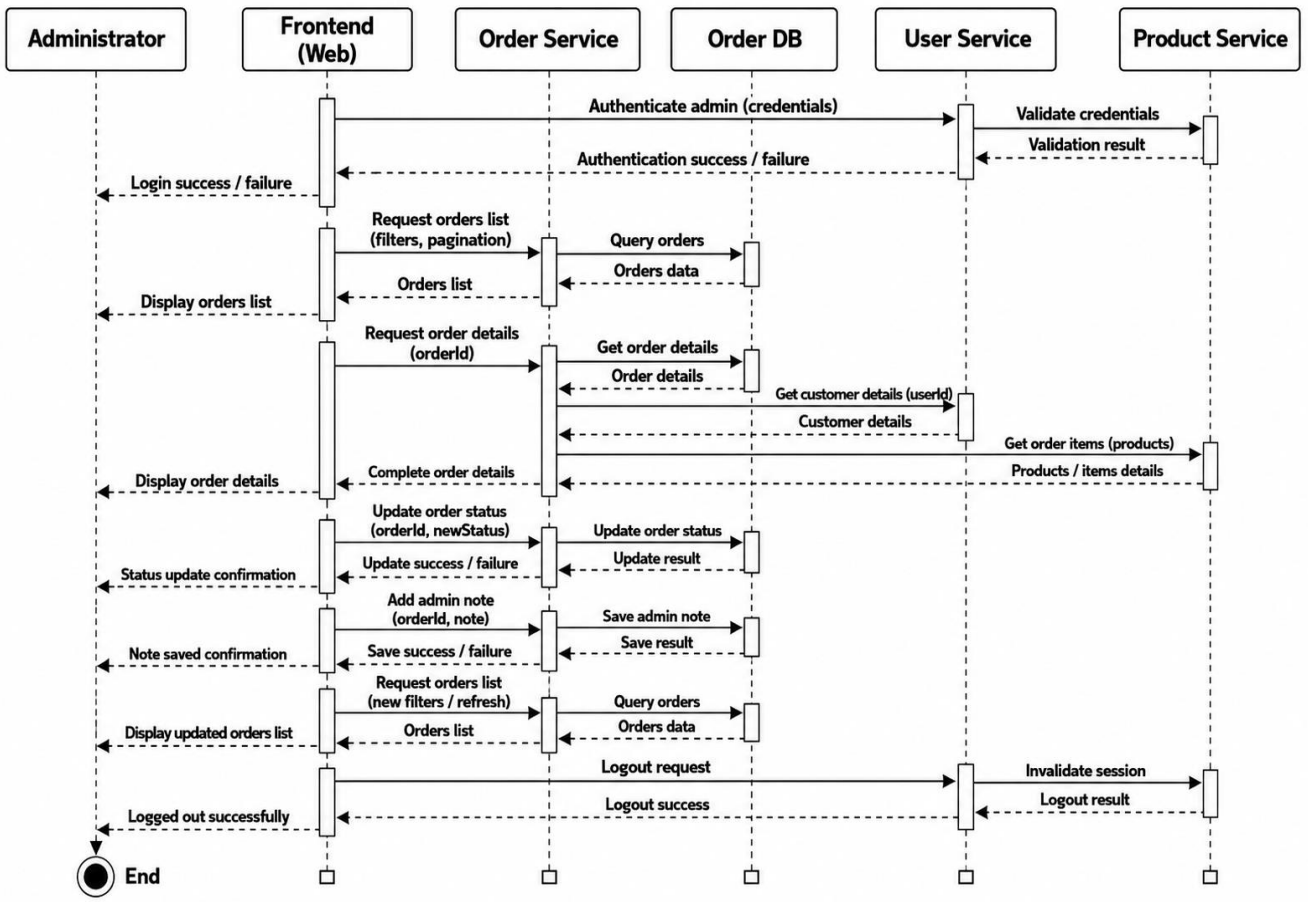

Figure 8: Administrator Sequence Diagram

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VIII August 2026

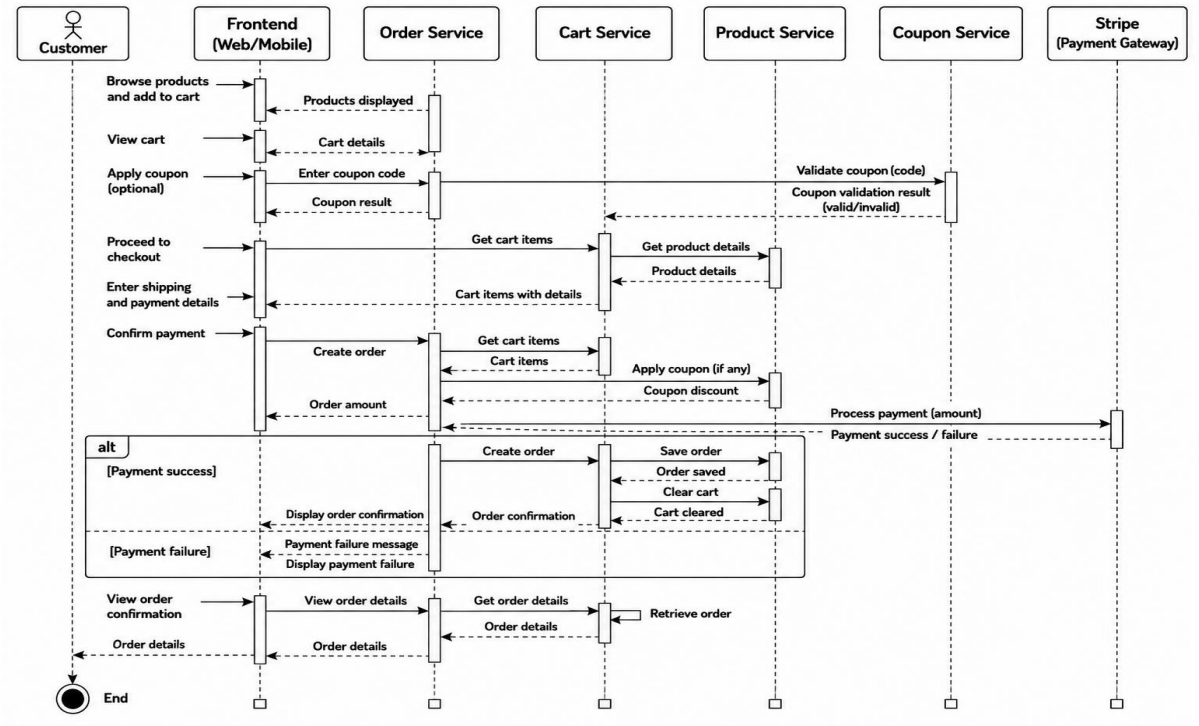

Figure 9: Customer Sequence Diagram

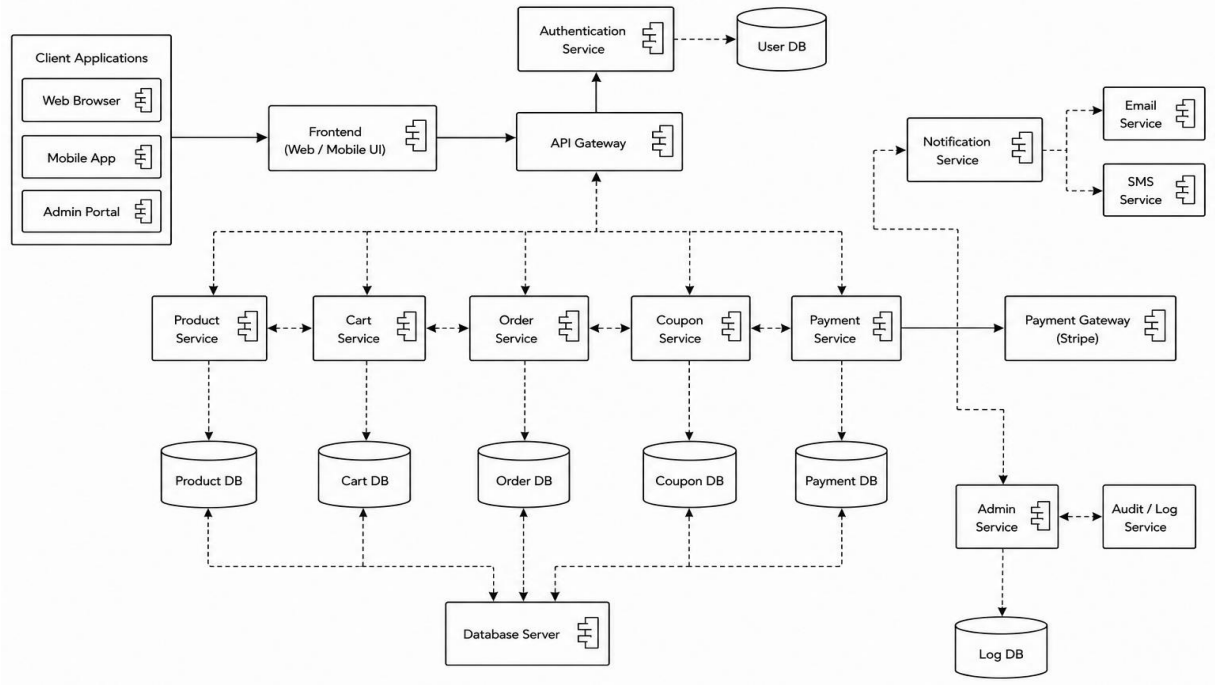

Figure 10: Component diagram of the proposed system

Performance Evaluation

The performance of the three architectural implementations was evaluated under controlled and comparable cloud-based conditions. The evaluation considered application startup time, deployment availability, response time, throughput, autoscaling responsiveness, resource utilisation, inter-service communication latency, full user-journey performance, Redis cache effectiveness, and frontend/CDN loading performance. k6 was used to generate controlled workloads, while the same application functionality, business logic, database and test scenarios were maintained across the Monolithic, Baseline Microservices and Proposed Microservices implementations to support a fair comparison. The evaluation was conducted using repeated performance tests, with the collected measurements analysed using descriptive statistics and comparative graphical representations.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VIII August 2026

Performance Metrics

Table 4 presents the performance metrics used to evaluate the three architectural implementations. These metrics collectively assess application loading, runtime performance, scalability, resource management, service communication, caching, availability, and frontend delivery.

Table 4: Performance Metric

S/N	Metric	Evaluation Focus
1	Application Startup Time	Time required for the application/services to become ready after restart or deployment
2	Deployment Availability	Ability of the system to remain accessible during deployment updates
3	Response Time and Throughput	Request latency and number of requests processed under increasing workloads
4	Autoscaling Responsiveness	Ability to provision additional service instances in response to increased workload
5	Resource Utilisation	CPU and memory utilisation under different workload levels
6	Inter-Service Latency	Communication time between microservices
7	Full User-Journey Performance	Time and success rate for completing the complete application workflow
8	Cache Effectiveness	Performance difference between cache misses and cache hits
9	Frontend/CDN Loading Performance	Frontend loading and content-delivery performance across the architectures

RESULTS AND DISCUSSION

The performance evaluation compared the Monolithic, Baseline Microservices and Proposed Microservices architectures under controlled workloads. The results are presented according to the performance metrics defined in the study.

Application Startup Time

Table 5 shows that the Monolithic architecture recorded the shortest startup time at 1.33 s, followed by Baseline Microservices at 6.43 s, while the Proposed Microservices architecture required 25.36 s. The higher startup time of the Proposed architecture reflects the additional provisioning and readiness checks associated with Kubernetes rolling deployment.

Table 5: Application Startup Time

Architecture	Average Startup Time
Monolithic	1.33 s
Baseline Microservices	6.43 s
Proposed Microservices	25.36 s

Deployment Availability

As shown in Table 6, the Proposed architecture recorded zero failed requests during deployment, compared with failure rates of 1.67% and 3.33% for the Monolithic and Baseline architectures respectively. This demonstrates the availability benefit of Kubernetes rolling updates and load balancing during deployment.

Table 6: Deployment Availability

Architecture	Total Requests	Failed Requests	Failure Rate
Monolithic	180	3	1.67%
Baseline Microservices	180	6	3.33%
Proposed Microservices	450	0	0.00%

Response Time and Throughput Under Load

Table 7 shows that at 200 VUs, the Monolithic architecture recorded the lowest response time, while the Proposed architecture recorded the highest due to additional routing and service-communication overhead. However, as workload increased, the Proposed architecture performed substantially better. At 2,000 VUs, its average response time was 33.47 ms, compared with 1,951.56 ms for the Monolithic architecture and 2,305.35 ms for Baseline Microservices.

The throughput results show a similar pattern in table 8. While all three architectures performed similarly at 200 VUs, the Proposed architecture increased to 1,915.78 req/s at 2,000 VUs, whereas Monolithic and Baseline reached only 670.13 req/s and 598.91 req/s, respectively. The results indicate that the Proposed architecture was better able to accommodate increasing demand through distributed processing and dynamic scaling.

Table 7: Response Time Under Load

Architecture	200 VU Avg. (ms)	1,000 VU Avg. (ms)	2,000 VU Avg. (ms)
Monolithic	6.12	418.72	1,951.56
Baseline Microservices	7.90	589.30	2,305.35
Proposed Microservices	14.52	137.18	33.47

Table 8: Throughput Under Load

Architecture	200 VU (req/s)	1,000 VU (req/s)	2,000 VU (req/s)
Monolithic	198.37	700.21	670.13
Baseline Microservices	197.80	624.67	598.91
Proposed Microservices	196.04	872.42	1,915.78

Autoscaling Responsiveness

The Proposed architecture's Horizontal Pod Autoscaler increased the Product Service from 2 to 5 replicas when CPU utilisation exceeded the configured 70% target. A subsequent 2,000-VU test recorded scaling from 2 to 15 replicas. This demonstrates the ability of the Proposed architecture to dynamically provision service capacity in response to increased workload, a capability not present in the Monolithic and Baseline implementations.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VIII August 2026

Resource Utilisation

At 2,000 VUs, the Monolithic and Baseline architectures reached peak CPU utilisation of 93.9%, while the Proposed architecture maintained low per-pod utilisation as additional replicas were provisioned. This indicates that the Proposed architecture manages increasing demand by distributing workload across additional instances rather than allowing utilisation on a single instance to approach saturation.

Inter-Service Communication Latency

The Baseline architecture recorded lower median inter-service latency (6 ms) than the Proposed architecture (12 ms) as seen in table 9. The difference demonstrates the communication overhead introduced by Kubernetes service networking. Although this represents a performance cost, the Proposed architecture provides additional orchestration and scalability capabilities that are absent from the direct EC2-to-EC2 deployment.

Table 9: Inter-Service Communication Latency

Architecture	Communication Path	Median Latency
Baseline Microservices	Direct EC2-to-EC2	6 ms
Proposed Microservices	Internal Kubernetes Service	12 ms

Full User-Journey Performance

The Proposed architecture achieved the lowest p95 latency (358.87 ms) and highest throughput (52.5 req/s) during the complete user-journey test. However, its success rate was 97.81%, showing that the architecture did not eliminate all failures under authentication-heavy concurrent workloads, as shown in table 10.

Table 10: Full User-Journey Performance

Architecture	Total Requests	Success Rate	p95 Latency	Throughput
Monolithic	7,656	100.00%	4.87 s	41.3 req/s
Baseline Microservices	7,842	92.33%	2.27 s	42.3 req/s
Proposed Microservices	9,732	97.81%	358.87 ms	52.5 req/s

Redis Cache Effectiveness

The controlled cache experiment recorded a mean latency of 21.152 ms for cache misses and 14.782 ms for cache hits, representing a 30.12% mean reduction in latency. The paired Wilcoxon signed-rank test produced $p = 0.0001$, with an effect size of $dz = 0.752$, indicating a statistically significant difference between the two conditions.

CDN Performance

CloudFront cache hits recorded a median latency of 6.10 ms, compared with 92.00 ms for cache misses as shown in table 11. The result demonstrates the contribution of edge caching to reducing frontend content-delivery latency.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue VIII August 2026

Table 11: CDN Performance

Metric	Cache HIT	Cache MISS
Sample Size	495,921	25
Success Rate	100%	100%
Mean (ms)	7.02	94.84
Median (ms)	6.10	92.00
p95 (ms)	13.78	126.00

Frontend Page Load Performance

The Proposed architecture achieved the fastest FCP, LCP, Speed Index and TTI, while the Baseline architecture recorded substantially slower frontend loading as shown in table 12. Although the Monolithic and Proposed architectures had the same fully loaded time of 6.67 s, the Proposed architecture rendered meaningful content and became interactive considerably earlier.Overall, the results show that the Proposed Microservices architecture involved higher startup and communication overhead but provided stronger performance characteristics as workload increased. Its advantages were particularly evident in response time, throughput, deployment availability, autoscaling and frontend loading. At the same time, the results confirm that microservice decomposition alone does not guarantee better performance, as the Baseline architecture generally performed worse than the Monolithic architecture under several conditions. The findings therefore indicate that the performance benefits observed in the Proposed architecture is associated not simply with microservice decomposition, but with the integration of cloud-native mechanisms such as Kubernetes orchestration, autoscaling, load balancing, Redis caching and CloudFront delivery.

Table 12: Frontend Page Load Performance

Architecture	FCP (ms)	LCP (ms)	Speed Index (ms)	TTI (ms)	TTFB (ms)	Fully Loaded (s)
Monolithic	804	804	905	844	26	6.67
Baseline Microservices	3,770	3,770	3,770	3,830	3,000	10.40
Proposed Microservices	617	617	652	617	34	6.67

Performance and Resource-Overhead Trade-off

This study developed and evaluated three functionally equivalent e-commerce architectures: Monolithic, Baseline Microservices, and Proposed Microservices. Although the Monolithic architecture recorded the shortest startup time of 1.33 s, the Proposed architecture demonstrated stronger performance under increasing workloads. At 2,000 VUs, it achieved an average response time of 33.47 ms and throughput of 1,915.78 req/s, while also providing dynamic autoscaling and zero deployment failure. However, these benefits were accompanied by higher startup time, inter-service communication latency, infrastructure requirements, and operational complexity. The findings therefore show that microservices alone do not guarantee improved performance; rather, their benefits depend on appropriate cloud-native mechanisms and workload requirements. The Proposed architecture is particularly suitable for applications requiring high concurrency, scalability, and deployment availability, where its performance benefits can justify the additional resource and operational overhead.

CONCLUSION

The study developed and evaluated three functionally equivalent architectures for an e-commerce application: Monolithic, Baseline Microservices, and Proposed Microservices. The results showed that although the Monolithic architecture recorded the lowest startup time of 1.33s, the Proposed Microservices architecture achieved better performance under increasing workloads. At 2,000 VUs, the Proposed architecture recorded an average response time of 33.47 ms and throughput of 1,915.78 req/s, compared with 1,951.56 ms and 670.13 req/s for the Monolithic architecture. It also achieved 0% deployment failure, a 30.12% reduction in latency through Redis caching, and a frontend FCP of 617 ms. Overall, the study demonstrates that integrating Kubernetes orchestration, autoscaling, load balancing, caching, and CDN delivery with microservices can improve application loading efficiency, scalability, and performance in cloud environments. However, the higher startup time of the Proposed architecture indicates an area for further optimisation. Future implementations should consider optimising container images, pod readiness configurations, resource allocation, and Kubernetes deployment settings to reduce startup overhead while retaining the scalability and availability benefits of the architecture.

REFERENCES

1. Akawuku I. G, Ikediego H. O., Nwankwo C, Joshua J. (2026), "Intense technologies: microservices in enterprise solutions", Journal of Contemporary Issues in Accountings, Vol. 6, (Issue 1), pp 15-32. https://journals.unizik.edu.ng/jocia/article/view/7999/6740
2. Aksakalli, I. K., Celik, T., Can, A. B., & Tekinerdogan, B. (2021). Systematic approach for generation of feasible deployment alternatives for microservices. IEEE Access, 9, 29505–29529. https://doi.org/10.1109/access.2021.3057582
3. Alelyani, A., Datta, A., & Hassan, G. M. (2024). Optimizing cloud performance: A microservice scheduling strategy for enhanced fault-tolerance, reduced network traffic, and lower latency. IEEE Access, 12, 35135–35153. https://doi.org/10.1109/access.2024.3373316
4. Alharthi, S., Alshamsi, A., Alseiari, A., & Alwarafy, A. (2024). Auto-scaling techniques in cloud computing: Issues and research directions. Sensors, 24(17), 5551. https://doi.org/10.3390/s24175551
5. Barua, B., & Kaiser, M. S. (2024). Cloud-enabled microservices architecture for next-generation online airlines reservation systems. Research Square. https://doi.org/10.21203/rs.3.rs-5182678/v1
6. Beeraka, N. B. S. (2025). Enterprise microservices: Revolutionizing telecom network architecture. International Journal of Scientific Research in Computer Science Engineering and Information Technology, 11(1), 290–297. https://doi.org/10.32628/cseit25111231
7. Benavente, V., Yantas, L., Moscol, I., Rodriguez, C., Inquilla, R., & Pomachagua, Y. (2022). Comparative analysis of microservices and monolithic architecture. 2022 14th International Conference on Computational Intelligence and Communication Networks (CICN), 177–184. https://doi.org/10.1109/cicn56167.2022.10008275
8. Blinowski, G., Ojdowska, A., & Przybylek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. IEEE Access, 10, 20357–20374. https://doi.org/10.1109/access.2022.3152803
9. Bjørndal, N., Mazzara, M., Bucchiarone, A., Dragoni, N., & Dustdar, S. (2021). Migration from Monolith to Microservices: Benchmarking a Case Study. The Journal of Object Technology, 20(2), 3:1. https://doi.org/10.5381/jot.2021.20.2.a3
10. Chippagiri, S., & Kassetty, N. (2025). Beyond the monolith: Comprehensive strategies for architecting, scaling, and sustaining resilient distributed systems. International Journal of Research in Computer Applications and Information Technology (IJRCAIT), 8(1), 152–168. https://doi.org/10.34218/ijrcait_08_01_016
11. Dave, N. S. A., Nadukuru, N. S., Singiri, N. S., Goel, N. O., Tharan, N. O., & Jain, N. A. (2024). Scalable microservices for cloud based distributed systems. Deleted Journal, 12(3), 776–809. https://doi.org/10.36676/dira.v12.i3.132
12. Dinh-Tuan, H., Mora-Martinez, M., Beierle, F., & Garzon, S. R. (2022). Development frameworks for microservice-based applications: Evaluation and comparison. arXiv. https://doi.org/10.48550/arxiv.2203.07267

13. Egwom, O. J., & Oladunjoye, J. A. (2024). A comparative assessment of load balancing techniques in cloud computing: A review. Elsevier. https://doi.org/10.2139/ssrn.4869408
14. Ileana, M., Oproiu, M. I., & Marian, C. V. (2024). E-commerce solutions using distributed web systems with microservices-based architecture for high-performance online stores. 2024 47th MIPRO ICT and Electronics Convention (MIPRO), 994–999. https://doi.org/10.1109/mipro60963.2024.10569273
15. Jun, C. (2024). A comprehensive study and design of microservices architecture. ResearchGate.
16. Kaloudis, M. (2024). Evolving software architectures from monolithic systems to resilient microservices: Best practices, challenges and future trends. International Journal of Advanced Computer Science and Applications, 15(9). https://doi.org/10.14569/ijacsa.2024.0150901
17. Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., & Babar, M. A. (2020). Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. Information and Software Technology, 131, 106449. https://doi.org/10.1016/j.infsof.2020.106449
18. Marques, G., Senna, C., Sargento, S., Carvalho, L., Pereira, L., & Matos, R. (2022). Proactive resource management for cloud of services environments. Research Square. https://doi.org/10.21203/rs.3.rs-2165603/v1
19. Nogueira, V. L., Felizardo, F. S., Amaral, A. M. M. M., Assuncao, W. K. G., & Colanzi, T. E. (2024). Insights on microservice architecture through the eyes of industry practitioners. arXiv. https://doi.org/10.48550/arxiv.2408.10434
20. Raikar, M. M., Patil, P., Guggari, S., Shavi, P., Mudavi, S., Patil, N., & Rangannavar, V. (2024). Leveraging Docker containers for deployment of web applications in microservices architecture. ResearchGate, 1–6. https://doi.org/10.1109/incowoco64194.2024.10863683
21. Raja, R. V., & Santhi, G. (2024). Enhancing microservices efficiency: Integrating adaptive learning for automated scaling in cloud environments. 2024 3rd International Conference for Innovation in Technology (INOCON), 1–6. https://doi.org/10.1109/inocon60754.2024.10512047
22. Saucedo, A. M., Rodríguez, G., Rocha, F. G., & Santos, R. P. D. (2024). Migration of monolithic systems to microservices: A systematic mapping study. Information and Software Technology, 107590. https://doi.org/10.1016/j.infsof.2024.107590
23. Thallapally, N. N. (2024). Designing scalable and robust microservice architectures for modern applications. International Journal of Science and Research Archive, 13(2), 4140–4145. https://doi.org/10.30574/ijsra.2024.13.2.2232
24. Velepucha, V., & Flores, P. (2023). A survey on microservices architecture: Principles, patterns and migration challenges. IEEE Access, 11, 88339–88358. https://doi.org/10.1109/access.2023.3305687
25. Waseem, M., Liang, P., & Shahin, M. (2020). A Systematic Mapping study on Microservices Architecture in DevOps. Journal of Systems and Software, 170, 110798. https://doi.org/10.1016/j.jss.2020.110798
26. Weerasinghe, S., & Perera, I. (2023). Optimized strategy for inter-service communication in microservices. International Journal of Advanced Computer Science and Applications, 14(2). https://doi.org/10.14569/ijacsa.2023.0140233
27. Wyciślik, Ł., Latusik, Ł., & Kamińska, A. M. (2023). A comparative assessment of JVM frameworks to develop microservices. Applied Sciences, 13(3), 1343. https://doi.org/10.3390/app13031343
28. Xu, M., Yang, L., Wang, Y., Gao, C., Wen, L., Xu, G., Zhang, L., Ye, K., & Xu, C. (2023). Practice of Alibaba Cloud on elastic resource provisioning for large-scale microservices cluster. Software Practice and Experience, 54(1), 39–57. https://doi.org/10.1002/spe.3271
29. Zaki, J., Islam, S. M. R., Alghamdi, N. S., Abdullah-Al-Wadud, M., & Kwak, K. (2022). Introducing cloud-assisted micro-service-based software development framework for healthcare systems. IEEE Access, 10, 33332–33348. https://doi.org/10.1109/access.2022.3161455