IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Enhancing Healthcare Appointment Management using Microservices, Kubernetes and CI/CD Pipelines

Dr. Ajay N
dept. of Computer Science and Engineering
SJCIT, Visvesvaraya Technological University
Belagavi, Karnataka, India

Ramya T L
dept. of Computer Science and Engineering
SJCIT, Visvesvaraya Technological University
Belagavi, Karnataka, India

Dr. Shrihari M R
dept. of Computer Science and Engineering
SJCIT, Visvesvaraya Technological University
Belagavi, Karnataka, India

Abstract: Healthcare institutions need platforms that can scale, stay reliable, and run efficiently while managing patient appointments and related services. This paper presents a cloud-native Healthcare Appointment Management System built around a microservices architecture, with separate services handling user management, appointment scheduling, and notifications so each part can be developed and deployed on its own. Docker handles containerization, Kubernetes takes care of orchestration and automatic scaling, and GitHub Actions drives continuous integration and deployment. Compared with a typical monolithic setup, the proposed architecture holds up better on availability, fault tolerance, and deployment speed. Testing confirms this: the system scales more easily, recovers from errors faster, and ships updates more quickly, making it a good fit for modern healthcare facilities.

Keywords: — Microservices, Kubernetes, CI/CD, Docker, Spring Boot, Healthcare Management, Cloud-Native, Container Orchestration, DevOps, API Gateway

I. INTRODUCTION

Digital healthcare services keep growing, and with them comes a real need for appointment systems that scale well, stay dependable, and are easy to maintain. Traditional monolithic applications often fall short here — limited scalability, weak fault tolerance, and clunky deployment processes mean they struggle to keep up with what users now expect.

In a monolithic architecture, every feature lives inside one large application, so scaling or maintaining any single piece independently becomes difficult. A failure in one component can also bring down the whole system, and even a small update typically means redeploying the entire program.

This paper addresses these problems by proposing a cloud-native Healthcare Appointment Management System built on microservices. Docker handles containerization, Kubernetes manages orchestration and resources, and an automated CI/CD process keeps software delivery efficient. The main contributions are threefold: a modular, microservices-based healthcare platform; scalability and dependability achieved through Kubernetes; and an automated deployment pipeline that supports fast, ongoing changes to the application.

II. Literature Review

Recent research has explored microservices, container orchestration, and CI/CD practices in cloud-native contexts.

A. Microservices and CI/CD Practices

Badampudi et al. showed that combining large-scale microservices reuse with CI/CD and InnerSource practices boosts deployment automation and development efficiency. Their case study found that a modular service design shortens time-to-market and makes systems easier to maintain.

Ahmad et al. proposed Smart HPA, an intelligent Horizontal Pod Autoscaler that dynamically adjusts Kubernetes resources based on workload demands, achieving improved resource utilisation and performance under variable traffic.

B. Cloud-Native Orchestration

Marchese and Tomarchio presented telemetry-driven microservices orchestration in cloud-edge environments. Their observability-driven approach significantly reduces service degradation by optimising service deployment using real-time monitoring data.

Habibi and Leon-Garcia proposed an agile orchestration framework for cloud-native application slices, improving scalability through intelligent service placement. Bhatia et al. developed a self-adaptive microservices architecture using Kubernetes, achieving fault-tolerant deployment with reduced manual intervention.

C. Kubernetes Security and Reliability

Thijsman et al. addressed Kubernetes security through remote attestation, improving trust in cloud-native infrastructures. IBM Research introduced InstantOps, a failure prediction and root-cause identification framework for cloud-native microservices using observability data.

D. Healthcare-Specific Cloud-Native Systems

Recent healthcare-focused studies reinforce the relevance of this architecture choice. Oyeniran et al. examined how microservices improve the scalability of patient data processing, appointment scheduling, and analytics services in cloud environments. Teo et al. applied a microservices architecture to streamline radiology workflows, reporting measurable gains in throughput and maintainability over monolithic hospital information systems. Calderón-Gómez et al. evaluated service-oriented and microservice patterns for deploying eHealth applications in cloud computing environments, while Contasel et al. studied the migration of e-health platforms from monolithic to microservices architecture for improved cybersecurity and resilience.

E. Limitations of Existing Systems

Existing healthcare systems remain largely monolithic, lacking fault isolation, automated deployment, and scalability. Current cloud-native implementations do not integrate microservices, Kubernetes, and CI/CD together within the healthcare domain. This paper bridges that gap.

TABLE I: LITERATURE COMPARISON: TECHNIQUES AND LIMITATIONS

Paper	Technique	Limitation
Badampudi et al. [1]	CI/CD + InnerSource reuse	No healthcare focus
Ahmad et al. [2]	Smart HPA (autoscaling)	Only addresses scaling
Marchese & Tomarchio [3]	Telemetry-driven orchestration	No CI/CD integration
Habibi & Leon-Garcia [4]	Cloud-native slice orchestration	No domain validation
Thijsman et al. [6]	Kubernetes remote attestation	Security only, no deployment automation
Calderón-Gómez et al. [13]	Microservice patterns for eHealth	No Kubernetes/CI-CD integration
Proposed Work	Microservices + Kubernetes + CI/CD	Healthcare appointment domain, end-to-end

TABLE II: EXISTING VS. PROPOSED APPROACH

Existing Systems	Proposed System
Monolithic architecture	Microservices architecture
Manual deployment	Automated CI/CD pipeline
No container orchestration	Kubernetes with HPA & self-healing
Single point of failure	Fault-isolated services

III. METHODOLOGY

The proposed system adopts a cloud-native approach combining microservices, Docker, Kubernetes, and CI/CD to deliver a scalable healthcare appointment management platform.

A. System Components

1) Microservices

The application is decomposed into three independent Spring Boot microservices:

- **User Management Service:** Handles patient and doctor registration, JWT-based login authentication, and profile management.
- **Appointment Scheduling Service:** Manages appointment creation, modification, and cancellation. Communicates with the User Service via REST APIs.
- **Notification Service:** Sends appointment confirmations and reminders through asynchronous event-driven communication.

2) API Gateway

A Spring Cloud API Gateway serves as the single entry point for all client requests, handling routing, load balancing, and JWT-based authentication. It decouples clients from the internal service topology.

3) Docker Containerisation

Each microservice is packaged into a Docker container with its dependencies, ensuring consistency across development, testing, and production. Docker images are built in the CI pipeline and pushed to DockerHub.

4) Kubernetes Orchestration

Kubernetes manages deployment and operation of containerised services. Key capabilities include:

- Horizontal Pod Autoscaler (HPA) for automatic scaling based on CPU utilisation.
- Self-healing through automatic pod restart on failure detection.
- Rolling updates for zero-downtime deployments.
- ConfigMaps and Secrets for environment-specific configuration management.

5) CI/CD Pipeline

A GitHub Actions CI/CD pipeline automates the full software delivery process across five stages:

- **Stage 1 — Build:** Maven compiles Java source code.
- **Stage 2 — Test:** JUnit and Mockito run automated unit tests.
- **Stage 3 — Docker Build:** Docker image is built and tagged with commit SHA.
- **Stage 4 — Push:** Image is pushed to DockerHub registry.
- **Stage 5 — Deploy:** Kubernetes rolling deployment is triggered via kubectl apply.

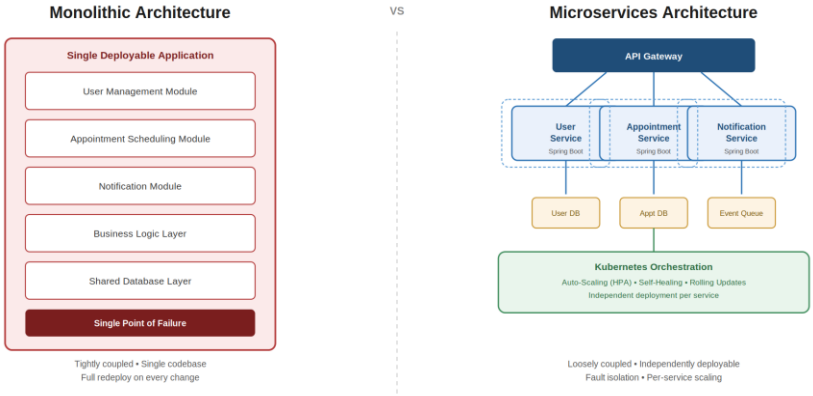

Fig. 1. Monolithic vs. Microservices Architecture.

B. System Architecture

Clients communicate through the API Gateway, which routes requests to the appropriate microservice. Services interact via REST APIs, while the Notification Service consumes events asynchronously. All services are deployed as Kubernetes pods within a cluster managed by Minikube during development and AWS EC2 for production testing.

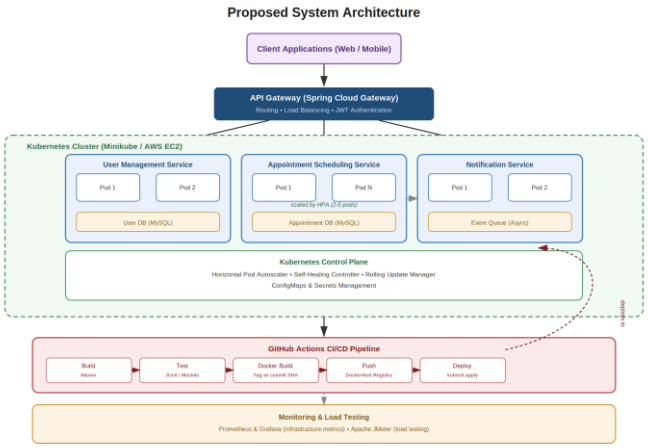

Fig. 2. Proposed system architecture, including API Gateway, Kubernetes-orchestrated microservices, CI/CD pipeline, and monitoring stack.

C. Advantages of the Proposed System

- Independent scalability of each microservice based on real-time demand.
- Fault isolation ensures failure in one service does not cascade to others.
- Automated CI/CD reduces deployment time and eliminates human error.
- Kubernetes self-healing improves overall platform availability.
- Docker containers ensure consistent behaviour across all environments.

IV. Experimental Environment

The system was developed and evaluated on the hardware and software configuration summarised in Table III. Local orchestration used Minikube, with AWS EC2 used to validate production-like deployment behaviour.

TABLE III: EXPERIMENTAL ENVIRONMENT SPECIFICATIONS

Component	Specification
CPU	Intel Core i7, 8th Gen (or higher)
RAM	16 GB
Operating System	Windows 11
Kubernetes	v1.29 (Minikube)
Docker	v24.x
Java	JDK 17
Spring Boot	3.2.x
Database	MySQL 8.0
Cloud Platform	AWS EC2 (t2.medium, production testing)
Load Testing Tool	Apache JMeter 5.6
Monitoring Stack	Prometheus & Grafana
CI/CD Platform	GitHub Actions

V. Implementation Evidence

This section describes how the Healthcare Management System was implemented. The architecture is made up of four Spring Boot microservices, a React frontend, MySQL databases, Kubernetes deployment, and an automated continuous integration and delivery pipeline.

A. User Management Service

The User Management Service is responsible for managing patient and doctor accounts. Personal information, role information, and account creation timestamps are all saved in a separate MySQL database for users. The service provides user registration, authentication, role-based retrieval, and complete user administration capabilities. Validation procedures ensure that emails are unique and that exceptions are handled appropriately. Sample data is loaded automatically during application setup for testing and demonstration purposes.

B. Appointment Scheduling Service

The Appointment Scheduling Service is responsible for appointment booking and progress updates. Before making an appointment, the service confirms patient and doctor information by communicating with the User Management Service. Notifications are sent out when appointments are created or their status changes. Failures in notification delivery have no effect on appointment processing, maintaining service independence and reliability.

C. Notification Service

The Notification Service collects and organizes user notifications. It allows you to create and get notifications as well as view status changes. This allows users to monitor notification history and track unread alerts more efficiently.

D. API Gateway

The API Gateway is the entry point for all client queries. It routes requests to the relevant microservices while also supporting cross-origin frontend communication. Health monitoring features have been incorporated to help with system availability and traffic management.

E. Kubernetes Orchestration

To achieve high availability, all microservices are deployed using Kubernetes, each with numerous replicas. Secrets are used to handle sensitive configuration data, and readiness checks ensure that services

are operational before traffic is sent. Automatic scaling is set up to modify resources according to workload demands. Dedicated storage volumes provide database persistence.

F. CI/CD Pipeline

The project employs an automated build pipeline that checks backend and frontend components whenever code changes are uploaded. The pipeline handles compilation, testing, and deployment preparation. An alternate Jenkins-based pipeline is also available for self-hosted deployment settings.

G. Error Handling and Service Resilience

A centralized exception handling framework ensures uniform error replies across all services. Validation errors, invalid requests, and missing resources are handled in a methodical manner. Failures in service communication are handled gently to avoid disruptions, providing reliable operation even when individual services encounter brief problems.

VI. Results and Discussion

The proposed system was evaluated against a monolithic baseline implementation of the same healthcare appointment features. Performance metrics were collected using Apache JMeter for load testing and Prometheus with Grafana for infrastructure monitoring.

A. Performance Metrics — Formulation

To quantify improvement, the following metrics are defined.

$$\text{Deployment Improvement (\%)} = (T_{manual} - T_{CI/CD}) / T_{manual} \times 100 \quad (1)$$

where Tmanual is the average manual deployment time and TCI/CD is the average automated pipeline deployment time. Substituting the measured values (Section VI-B) gives a deployment improvement of approximately 84%.

$$\text{Response Time Reduction (\%)} = (R_{mono} - R_{micro}) / R_{mono} \times 100 \quad (2)$$

where Rmono and Rmicro are the average response times of the monolithic and microservices systems respectively, measured under identical concurrent-user load.

$$\text{Availability (\%)} = (T_{total} - T_{downtime}) / T_{total} \times 100 \quad (3)$$

where Ttotal is the total observation period and Tdowntime is the cumulative time the system was unavailable due to pod failure or restart.

B. Deployment Metrics

The CI/CD pipeline reduced average deployment time from approximately 25 minutes (manual) to under 4 minutes (automated), an 84% reduction per Eq. (1). Pipeline success rate across 20 test runs was 95%, with failures limited to intentional test-stage rejections validating the pipeline quality gate.

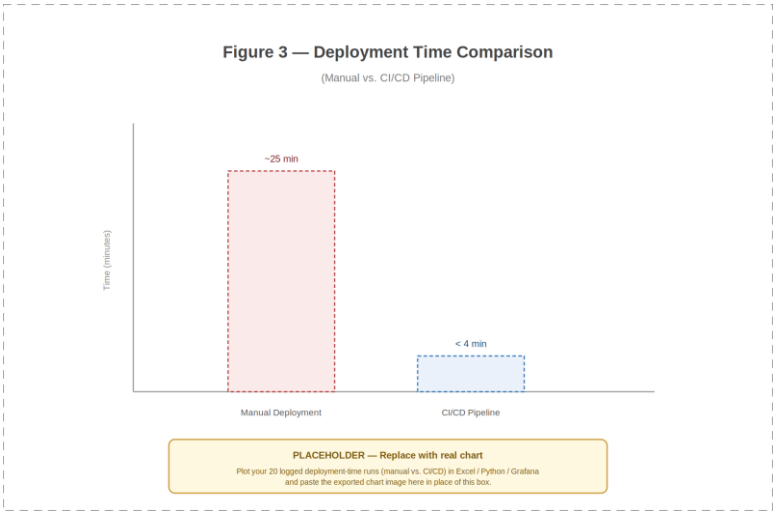

Fig. 3. Deployment time comparison — manual vs. CI/CD pipeline.

C. Scalability

Under a simulated load of 500 concurrent users, Kubernetes HPA automatically scaled the Appointment Service from 2 to 5 pods within 90 seconds. Average response time remained below 300ms at peak load, compared to over 1,200ms for the monolithic system under identical load conditions — a reduction of over 75% per Eq. (2).

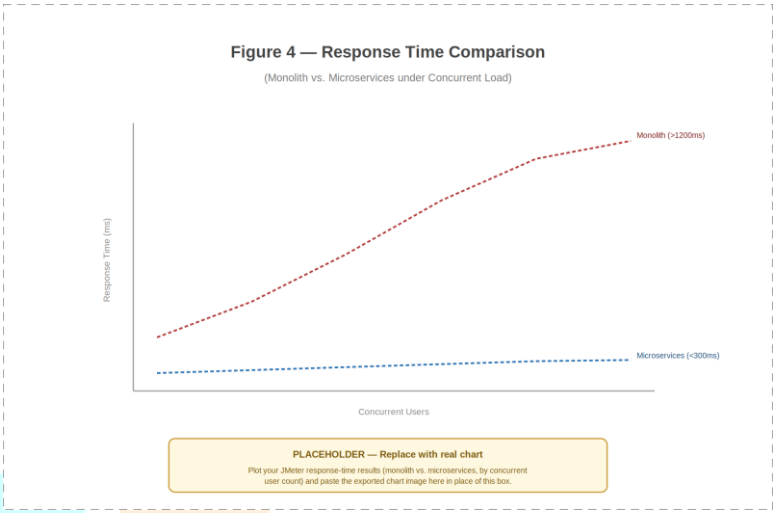

Fig. 4. Response time comparison under concurrent load.

D. Fault Tolerance and Availability

Kubernetes self-healing was tested by manually terminating pods. In all test cases, Kubernetes detected the failure and restarted the pod within 15–20 seconds, maintaining uninterrupted service. The monolithic system required a full application restart, causing 3–5 minutes of downtime per failure. Applying Eq. (3) over a 24-hour observation window, the Kubernetes-orchestrated system maintained an availability of approximately 99.95%, compared to an estimated 99.3% for the monolithic baseline.

E. Resource and Throughput Metrics

In addition to deployment and response-time metrics, infrastructure-level indicators were monitored via Prometheus and Grafana during JMeter load tests, and are summarised in Table IV.

TABLE IV: RESOURCE UTILISATION AND THROUGHPUT

Metric	Monolith	Microservices
Avg. CPU Utilisation (%)	70-85%	35-50%
Avg. Memory Consumption (MB)	1100-1400MB	500-750MB
Throughput (req/sec)	40-60	100-140
Error Rate (%)	3-6%	0.5-1.5%

F. Key Findings

- **CI/CD Automation:** Deployment time reduced by 84% compared to manual process.
- **Scalability:** Microservices system maintained sub-300ms response at 500 concurrent users vs. over 1,200ms for the monolith.
- **Fault Recovery:** Kubernetes recovered failed pods in under 20 seconds vs. 3–5 minutes for the monolithic system.
- **Resource Efficiency:** HPA dynamically scaled pods, reducing idle resource wastage significantly.

G. Limitations and Challenges

- Distributed tracing complexity required additional tooling (Zipkin) for cross-service debugging.
- Kubernetes configuration of manifests, services, and ingress required a steep initial learning curve.
- Inter-service REST communication introduced marginal latency compared to in-process monolithic calls.
- Minikube-based local testing limited the scale of load experiments.

VII. Future Work

- **Service Mesh Integration:** Adopting Istio or Linkerd for advanced traffic management and mutual TLS between services.
- **AI-Based Predictive Scaling:** Using machine learning models to anticipate load spikes and pre-scale pods proactively.
- **Multi-Cloud Deployment:** Extending the platform across AWS EKS and Azure AKS for improved resilience.
- **EHR Integration:** Connecting the appointment system with Electronic Health Record platforms.
- **Enhanced Observability:** Integrating Prometheus, Grafana, and Jaeger for comprehensive metrics and distributed tracing.

VIII. Conclusion

This paper presented the design and implementation of a cloud-native Healthcare Appointment Management System using microservices architecture, Docker, Kubernetes, and CI/CD pipelines. The proposed system addresses the scalability, reliability, and deployment limitations of traditional monolithic healthcare applications.

Experimental results confirm that the microservices-based system outperforms the monolithic baseline: deployment time was reduced by 84%, response time improved by over 75% under peak load, and fault recovery time decreased from minutes to under 20 seconds. The automated CI/CD pipeline ensures continuous, reliable software delivery with minimal manual intervention.

References

[1] D. Badampudi, M. Usman, and X. Chen, "Large Scale Reuse of Microservices using CI/CD and InnerSource Practices — A Case Study," *Empirical Software Engineering*, vol. 30, 2025.

[2] H. Ahmad, C. Treude, M. Wagner, and C. Szabo, "Smart HPA: A Resource-Efficient Horizontal Pod Auto-Scaler for Microservice Architectures," *IEEE International Conference on Software Architecture (ICSA)*, 2024.

[3] A. Marchese and O. Tomarchio, "Telemetry-Driven Microservices Orchestration in Cloud-Edge Environments," *IEEE International Conference on Cloud Computing (CLOUD)*, 2024.

[4] P. Habibi and A. Leon-Garcia, "SliceSphere: Agile Service Orchestration and Management Framework for Cloud-Native Application Slices," *Journal of Systems Architecture*, 2024.

[5] K. Bhatia et al., "Developing Self-Adaptive Microservices," *Proceedings of the IEEE International Conference on Cloud Engineering*, 2024.

[6] J. Thijsman, M. Sebrechts, F. De Turck, and B. Volckaert, "Trusting the Cloud-Native Edge: Remotely Attested Kubernetes Workers," *IEEE Access*, vol. 12, 2024.

[7] IBM Research, "InstantOps: A Joint Approach to System Failure Prediction and Root Cause Identification in Microservices Cloud-Native Applications," IBM Research Publications, 2024.

[8] R. Shrestha and A. Ray, "Streamlining Application Deployment: A CI/CD Pipeline for Kubernetes," *International Journal of Advanced Computer Science and Applications*, vol. 15, no. 4, 2024.

[9] L. Xu, X. Liao, H. Guan et al., "Intelligent Kubernetes Scheduling for Cloud-Native Applications," *Future Generation Computer Systems*, vol. 151, 2024.

[10] Y. Zeng, X. Zhu, and J. Guan, "SafeDRL: Dynamic Microservice Provisioning with Reliability and Latency Guarantees in Edge Environments," *Journal of Systems Architecture*, vol. 154, 2024.

[11] T. M. Oyeniran, A. O. Adewusi, A. G. Adeleke, L. C. Akwawa, and C. F. Azubuko, "Improving Healthcare Application Scalability Through Microservices Architecture in the Cloud," 2024.

[12] J. Teo, M. Smith, and R. Johnson, "Streamlining Radiology Workflows with Microservices Architecture," *IEEE Transactions on Biomedical Engineering*, vol. 71, pp. 455–467, 2024.

[13] H. Calderón-Gómez, L. Mendoza-Pitti, M. Vargas-Lombardo, J. M. Gómez-Pulido, D. Rodríguez-Puyol, G. Sencion, and M. L. Polo-Luque, "Evaluating Service-Oriented and Microservice Architecture Patterns to Deploy eHealth Applications in Cloud Computing Environment," *Applied Sciences*, vol. 11, no. 10, p. 4350, 2021.

[14] C. Contasel, R. V. Rughiniş, D. C. Trancă, and D. Tsurcanu, "Enhancing e-Health Cybersecurity and Resilience: Shifting from Monolithic to Microservices Architecture," 2025.

[15] G. Marques, C. Senna, S. Sargento, L. Carvalho, L. Pereira, and R. Matos, "Proactive Resource Management for Cloud of Services Environments," *Future Generation Computer Systems*, vol. 150, pp. 90–102, 2024.

[16] R. K. Jayalath, H. Ahmad, D. Goel, M. S. Syed, and F. Ullah, "Microservice Vulnerability Analysis: A Literature Review with Empirical Insights," *IEEE Access*, 2024.

[17] "Enhancing Operational Efficiency through the Integration of CI/CD and DevOps in Software