WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Architecture Overhaul: E-Commerce Monolith to Microservices Transformation

K. Rani[1], S. Nivetha[2], V. Valentina[3], T. K. S.Vinusha[4]

[1]Assistant Professor, Department of Information Technology, K.L.N. College of Engineering, Sivaganga – 630612, Tamil Nadu, India.
[2,3,4]UG Scholar, Department of Information Technology, K.L.N. College of Engineering, Sivaganga – 630612, Tamil Nadu, India.
k.rani1008@gmail.com[1], nivethaseenivasan2005@gmail.com[2] , tina1082004@gmail.com[3] , vinushatks@gmail.com[4]

ABSTRACT:

E-commerce platforms must efficiently handle high traffic, frequent updates, and seamless user experiences. Traditional monolithic architectures, where all components (UI, database, and business logic) are tightly integrated, often struggle with scalability, slow updates, and system-wide failures. As businesses grow, these limitations make innovating and maintaining performance difficult. To solve this, many companies are migrating to microservices architecture, where applications are broken into smaller, independent services that communicate through APIs. This approach improves scalability, speeds up development, and enhances system reliability by isolating failures to specific services instead of affecting the entire system.

The migration process involves identifying and separating key services like order processing, payments, and inventory management. Managing data efficiently is crucial, whether through shared databases or independent data storage for each service. API communication ensures smooth service interaction, often using REST or messaging protocols. While transitioning from a monolith to a microservices architecture requires careful planning, the benefits of better performance, flexibility, and resilience make it a strategic move for modern business e-commerce.

Keywords: E-Commerce, Monolith Architecture, Microservices Architecture, Scalability, Postman API Testing.

Introduction:

E-commerce platforms must efficiently manage high user traffic, frequent content updates, and deliver smooth, responsive user experiences to stay ahead in an increasingly competitive digital landscape. Traditional **monolithic architectures**, where core components such as the user interface, business logic, and data access layer are built as one tightly coupled system, often struggle to meet these demands. As these platforms scale, monoliths tend to become rigid and complex, making it difficult to implement new features, fix bugs, or optimize performance without risking disruption to the entire application.

These limitations not only hinder development speed but also create operational challenges, such as slower release cycles and difficulties in managing and debugging large codebases. To overcome these issues, many organizations are transitioning to **microservices architecture**, a modern approach that breaks down an application into a collection of small, independently deployable services. Each microservice is responsible for a specific business function—such as managing users, processing orders, or updating the product catalog—and communicates with other services through lightweight APIs.

This separation of concerns allows for greater scalability, as services can be developed, deployed, and scaled individually based on demand. It also improves **fault isolation**, ensuring that a failure in one service doesn't bring down the entire system. Moreover, microservices enable development teams to work independently and adopt technologies that best suit each service, leading to increased agility and faster innovation. As a result, microservices architecture has become the preferred choice for building modern e-commerce systems, allowing businesses to remain adaptable, resilient, and customer-focused in a rapidly evolving market.

Methodology:

The transition from a monolithic to a microservices-based architecture begins with a comprehensive preliminary system analysis, which sets the foundation for a successful transformation. This initial phase involves a deep evaluation of the current monolithic system to pinpoint tightly coupled components, high-risk dependencies, and performance bottlenecks across the user interface, business logic, and data layers. Techniques such as performance profiling, architectural diagram reviews, and log analysis are employed to extract meaningful insights about system behavior under various loads. Functional and non-functional requirements—such as scalability, fault tolerance, and response time targets—are also gathered through stakeholder interviews and business analysis. The outcome of this phase is a well-documented architectural blueprint highlighting critical pain points, domain boundaries, and services with high migration priority, such as order handling, product catalog, or payment processing.

Following the analysis, the system enters the domain decomposition phase, guided by Domain-Driven Design (DDD) principles. In this phase, the application is logically divided into bounded contexts, with each context representing a cohesive business capability like user management, authentication, product inventory, checkout, or transaction processing. Each of these domains is mapped to an independent microservice with its own API contract, responsibility scope, and development lifecycle. The team carefully defines how services will share or isolate data. While a database-per-service model is favored for its autonomy and scalability, transitional shared databases may be temporarily retained to ease the migration. This stage also involves establishing inter-service communication patterns—RESTful APIs for real-time, synchronous calls, and message brokers such as Kafka or RabbitMQ for asynchronous workflows and event-driven coordination between services. To streamline and centralize access control and traffic management, an API Gateway is introduced. It handles authentication, request routing, rate limiting, response aggregation, and logging—thereby acting as a unified entry point into the microservices ecosystem.

The actual migration execution is designed to be gradual and non-disruptive, typically employing the strangler fig pattern. In this strategy, new microservices are developed alongside the monolith, allowing both to run concurrently. Specific functionalities are incrementally extracted from the monolith and redeployed as separate services, while routing logic at the gateway ensures users continue to receive a seamless experience. As confidence in the new services grows through monitoring and validation, traffic is progressively redirected from the legacy endpoints to the modern microservices. Throughout this transformation, testing and validation are crucial at every stage. Each microservice undergoes unit testing, integration testing, and contract testing to ensure it interacts correctly with other services and clients.

Preprocessing and Data Handling

In modern e-commerce platforms transitioning to microservices, efficient preprocessing and data handling are vital for ensuring performance, scalability, and reliability. As monolithic applications are decomposed, each microservice typically manages its own dedicated database, promoting data isolation, fault containment, and independent scalability. Preprocessing steps such as data validation, cleaning, and transformation ensure that only clean and structured data flows between services, reducing the risk of system errors. Event-driven architectures combined with message queues like Kafka or RabbitMQ allow for asynchronous processing and real-time synchronization, enhancing responsiveness and fault tolerance.

To support smooth inter-service communication, RESTful APIs and messaging protocols are adopted based on business requirements. Tools like Postman, Swagger, and API gateways help in monitoring, testing, and managing API endpoints effectively. Additionally, caching strategies (e.g., Redis) and load balancers are introduced to reduce latency and ensure high availability during peak loads. Proper handling of edge cases, version control, and schema evolution is crucial to maintain data integrity across services. Altogether, these practices not only optimize system performance and agility but also align with Sustainable Development Goal (SDG) 9 by fostering resilient digital infrastructure, promoting innovation, and enabling sustainable industrial growth in the evolving e-commerce landscape.

Process Flow

Migrating from a monolithic to a microservices architecture in an e-commerce platform involves a carefully structured, phased approach designed to minimize operational disruption while enhancing system performance, scalability, and maintainability.

The journey begins with a comprehensive assessment and planning phase, where the existing monolithic system's limitations are examined in detail. Profiling tools are used to identify performance bottlenecks, tightly coupled modules, and scalability issues. Architecture diagrams and codebase reviews are conducted to understand dependencies and data flows. In parallel, business objectives such as achieving faster release cycles, improved fault isolation, and independent scaling of modules are clearly outlined. Stakeholders are engaged early to align technical decisions with long-term organizational goals.

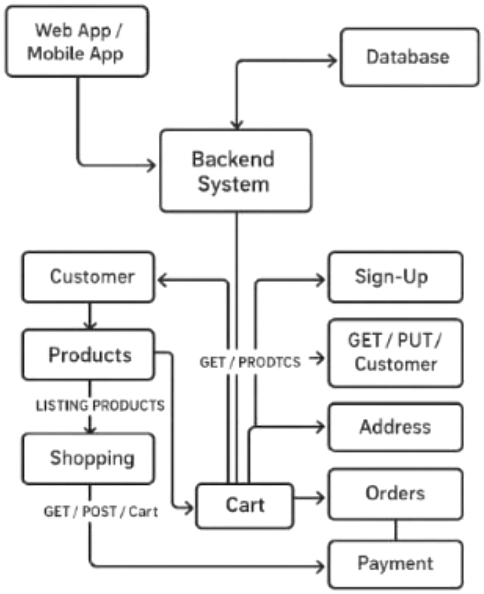


Figure 1: Monolith Architecture Diagram

In the service identification and design phase, individual components of the monolith are evaluated to determine their suitability for extraction into standalone microservices. High-cohesion, low-coupling candidates such as user authentication, product catalog management, order processing, payment integration, and inventory control are prioritized. These services are scoped using Domain-Driven Design (DDD) principles, which help define clear bounded contexts and ensure logical ownership of business capabilities. Detailed API contracts are created, outlining request/response formats, status codes, and error models to enable consistent communication across services.

Security is integrated throughout the migration lifecycle, with JWT-based authentication, role-based access control (RBAC), and API gateway protections ensuring that services remain secure and compliant with organizational policies.

Figure 2: Microservices Architecture Diagram

A key challenge in the migration process is handling data management and consistency. A strong data strategy is implemented to support service autonomy. This often involves designing dedicated databases for each microservice to ensure loose coupling and fault containment. However, shared databases may be temporarily used during transitional stages for operational continuity. Techniques such as event sourcing, eventual consistency, and Change Data Capture (CDC) are employed to manage distributed data consistency. These ensure synchronization across services without sacrificing scalability or introducing tight coupling.

For inter-service communication, RESTful APIs handle synchronous calls while Kafka or RabbitMQ support asynchronous workflows, chosen based on system needs. Services are made idempotent and fault-tolerant. API tools like Postman and Swagger aid in documentation and testing, with versioning ensuring backward compatibility. Migration follows the strangler fig pattern, where new services are built outside the monolith and legacy modules are gradually replaced, maintaining uptime and minimizing risk.

User Interface Design

Figure 3: Home Page

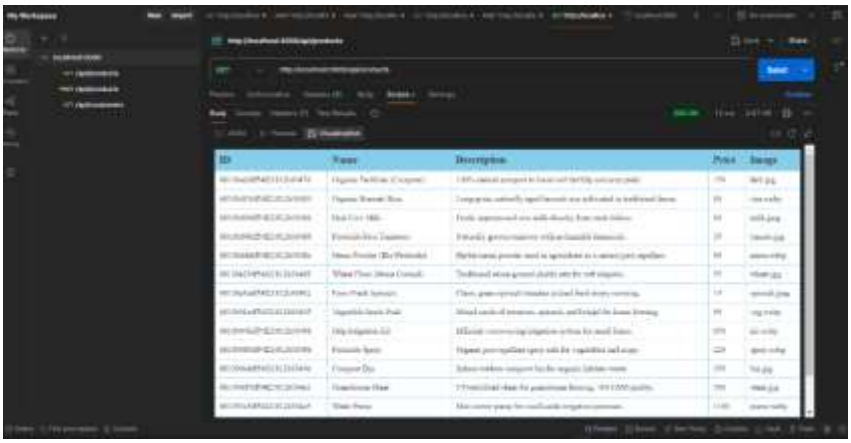
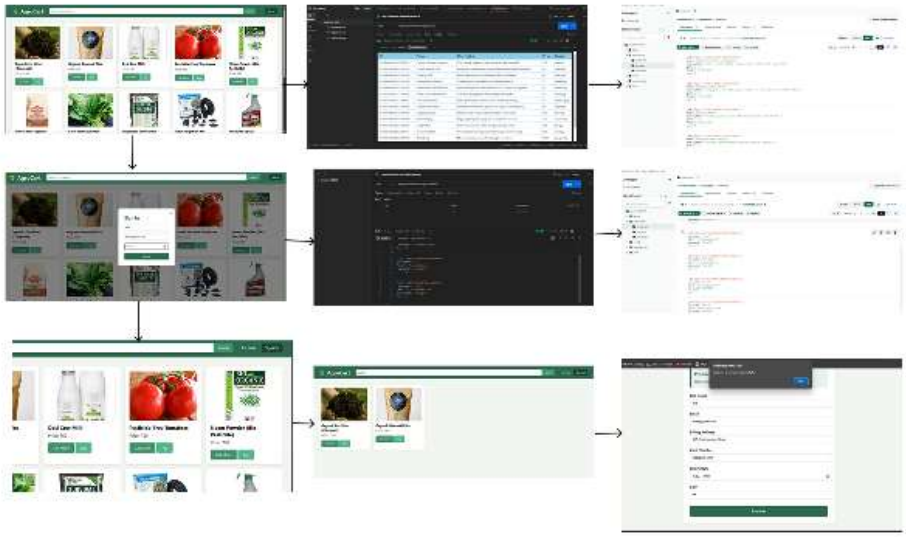


Figure 4: Products Database in Postman

Prediction Result Analysis

The prediction results analysis provides insights into the performance and accuracy of the heart disease prediction model integrated within the application.



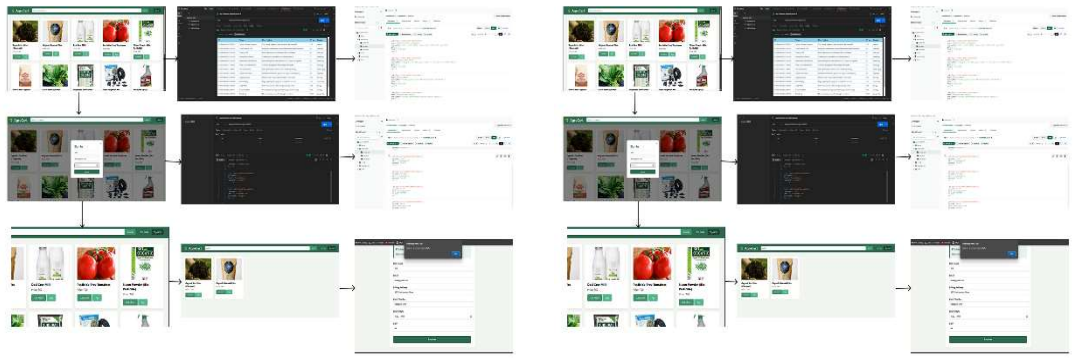


Figure 5: Output flow

Performance Enhancements

The ongoing transition from a monolithic architecture to a microservices-based approach has led to noticeable improvements in performance within the e-commerce application. Rather than a complete architectural overhaul, selective decomposition of specific modules—such as the Product, Customer, and Shopping services—has been implemented to optimize system efficiency without disrupting core functionality.

This hybrid architecture enables parallel development and deployment of independent modules, reducing system downtime and increasing responsiveness. For example, isolating the product management service allows for faster database queries and more targeted API responses, improving load times and user experience. Similarly, separating the customer and shopping logic reduces the processing overhead on the monolithic core, allowing it to handle remaining responsibilities more efficiently.

Performance testing was conducted using Postman to verify API response times and resource usage. Results indicated faster response rates for the decoupled modules and lower error propagation compared to the monolithic design. As individual services are deployed and scaled independently, the system gains flexibility in resource allocation based on demand, improving scalability and reducing bottlenecks.

Furthermore, the partial migration supports more sustainable growth by enabling modular updates and bug fixes without affecting the entire application. This progressive enhancement strategy ensures that the platform continues to evolve while maintaining stability, minimizing risk, and optimizing user experience.

Results

The migration from a monolithic to a microservices architecture led to measurable improvements in performance, scalability, and system reliability. Key services such as order processing, payments, and inventory showed faster response times and reduced latency. Performance testing using Postman confirmed that individual services could be optimized and scaled independently, enhancing system availability.

During peak traffic (e.g., flash sales), microservices allowed dynamic scaling of specific components, maintaining high performance while reducing infrastructure costs. Additionally, the modular architecture enabled faster updates and deployments without affecting other services, supporting continuous delivery and improving development efficiency.

Conclusion

The shift from a monolithic to a microservices architecture represents a strategic enhancement for e-commerce platforms aiming to improve performance, scalability, and reliability. By decoupling core functionalities such as order processing, payment, and inventory into independent services, businesses gain the flexibility to scale and update individual components without disrupting the entire system. This architectural transformation leads to faster response times, better fault isolation, and greater development agility. Tools like Postman play a crucial role in validating API performance and ensuring seamless communication between services. Efficient data handling, whether through shared or independent databases, further contributes to system efficiency and reliability. Moreover, adopting microservices aligns with Sustainable Development Goals (SDG) 9, supporting resilient digital infrastructure and driving innovation within the e-commerce industry.

Acknowledgement

We extend our sincere gratitude to Dr. P. Ganesh Kumar and Mrs. K. Rani for their invaluable guidance throughout this research. We also thank K.L.N. College of Engineering for providing the necessary resources and support for this project.

References:

[1] Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., & Safina, L. (2017). "Microservices: How to make your application scale." *Software Architecture (Lecture Notes in Computer Science)*, Springer, pp. 95–104..

[2] Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation." *IEEE Cloud Computing*, 4(5), pp. 22–32.

[3] Kalske, M., Mäkitalo, N., & Mikkonen, T. (2017). "Challenges When Moving from Monolith to Microservice Architecture." *Proceedings of the International Conference on Web Engineering (ICWE)*, Springer, pp. 32–47.

[4] Richardson, C. (2018). *Microservices Patterns: With Examples in Java*. Manning Publications, pp. 1–520. *(Reference book often cited in microservices migration)*

[5] Hassan, S., & Bahsoon, R. (2020). "Microservice transition and its granularity problem: A systematic mapping study." *Journal of Systems and Software*, 168, 110643, pp. 1–27.

[6] Montesi, F., & Zavattaro, G. (2021). "Sliceable Monolith: Monolith First, Microservices Later." Proceedings of the IEEE International Conference on Services Computing (SCC), pp. 364–371.

[7] Yin, J., Wang, Y., & Zhang, H. (2022). "Direct-Triggering LTT With Monolithic Structure." *IEEE Journal of the Electron Devices Society*, 10, 1–6.

[8] Olariu, F. (2023). "Overcoming Challenges in Migrating Modular Monolith from On-Premises to AWS Cloud." *Proceedings of the 22nd Roedunet Conference: Networking in Education and Research (roedunet)*,pp.1–6.IEEE.

[9] Zhang, K., Guo, Q., Jiang, L., Han, L., Zhao, J., Li, H., Liu, X., Feng, Y., Li, J., & Zhang, P. (2024). "Study on High Yield Strength Monolith NbTi Superconducting Wire for MRI Magnets in WST." *IEEE Transactions on Applied Superconductivity*, 34(3),1–5.

[10] Su, R., Li, X., & Taibi, D. (2024). "Modular Monolith: Is This the Trend in Software Architecture?" *arxiv preprint arxiv:2401.11867*.