JURNAL TEKNOLOGI DAN OPEN SOURCE
Vol. 8, No. 1, June 2025, pp. 54~63
e-ISSN: 2622-1659, accredited Four Grade by Kemenristekdikti, Decree No: 152/E/KPT/2023
DOI: 10.36378/jtos.v8i1.4265

Analysis and Design of Monolithic System Architecture Migration to Microservices at PT. MALINDO Conceptual Approach

Ego Oktafanda[1], Nofri Wandi Al-Hafiz[2], Abdul Latif[3], Firman Santosa[4]
[1,4]Universitas Rokania, Rokan Hulu, Riau. Indonesia.
[2]Universitas Islam Kuantan Singingi, Riau, Indonesia.
[3]Universitas Bina Sarana Informatika, Jakarta. Indonesia

Article Info

Article history:

Received 04 08, 2025
Revised 04 15, 2025
Accepted 05 16, 2025

Keywords:

Monolith
Microservices
Scalability
Transition

ABSTRACT

This study analyzes the transition from a monolithic architecture to a microservices architecture within a corporate environment, specifically at PT.MALINDO. While the monolithic architecture was once effective, it now presents several challenges, including maintenance complexity, limited scalability, and frequent occurrences of deadlocks. The analysis highlights the growing need for a more adaptive and flexible system to respond to dynamic business requirements. In this context, a microservices architecture is proposed as a solution to overcome the limitations of the monolithic approach. Microservices break down applications into small, independent services, enabling easier and more efficient development, deployment, and maintenance. By using components such as an API Gateway and message brokers like Apache Kafka, inter-service communications can be managed more effectively, ensuring improved system performance and resilience. The findings indicate that transitioning to a microservices architecture can significantly enhance the system's performance, flexibility, and scalability. The resulting hypothesis suggests that adopting microservices will offer long-term benefits for PT.MALINDO, particularly in addressing system complexity and improving responsiveness to evolving business needs. Recommendations include implementation strategies and change management practices necessary to ensure a smooth and low-risk transit

Corresponding Author:

Ego Oktafanda
Department of Computer Science
Universitas Rokania
Riau, Indonesia
Email: egooktafanda1097@gmail.com

1. Introduction

In today's rapidly evolving digital era, technology adaptation has become a critical factor for the sustainability and success of organizations. Monolithic systems, which used to serve as the main foundation of software development, now face various challenges that can no longer be ignored. Recent research by Hatma Suryotrisongko (2017) emphasizes this issue and highlights the urgent need to explore alternative architectures such as microservices to overcome the complexity and limitations of monolithic systems [1].
Enterprise information systems are generally built using a monolithic approach, where the entire application is packaged into one large unit. As a result, any modification to one part of the code base can have a

significant impact on other components. However, this traditional approach is increasingly shifting towards a distributed model, where applications are divided into smaller, specialized modules (high cohesion) that operate independently (loose coupling). These services are usually developed and integrated using API (Application Programming Interface) endpoints [2]. PT.MALINDO, a company engaged in logistics, currently relies on a monolithic system for its operations and data management. One of the main problems faced in this arrangement is the frequent occurrence of deadlocks and timeouts, which hamper the overall system response. As systems continue to grow, the complexity of coordination and maintenance also increases, limiting the company's agility in responding to market changes efficiently. Furthermore, inadequate monitoring capabilities in many of PT.MALINDO's monolithic systems pose serious risks, as they make it difficult to detect and resolve problems quickly. In addition, the increasing demand for data consumption adds further complexity to PT.MALINDO's monolithic architecture, negatively impacting data processing performance and efficiency. Dependence on a particular technology and limited scalability are also important issues that need to be addressed. This study aims to explore potential solutions to these challenges through the adoption of a microservices architecture. By thoroughly understanding the limitations of monolithic systems and the advantages offered by microservices, PT.MALINDO can make informed decisions to improve its adaptability, scalability, and operational efficiency. It is hoped that this study will provide valuable insights and contribute significantly to the sustainable development of modern information systems within PT.MALINDO.

2. Literature review

Microservices Architecture is a software development approach that structures an application as a collection of isolated and independently deployable services. This approach addresses many of the challenges faced by traditional monolithic systems by offering improved flexibility, scalability, and resilience to change. Microservices architecture is characterized by a network of small, autonomous services that operate independently and communicate using lightweight protocols such as HTTP (Hypertext Transfer Protocol). Each service runs in its own process, is built around a distinct business capability, and can be deployed independently[3]. These services function independently, allowing for development, deployment, and modification without impacting other services. This modularity enables developers to focus on building small, well-defined services, thereby enhancing the overall system’s scalability and adaptability. Microservices have become a popular architectural style for data-driven applications due to their ability to decompose complex applications into smaller, autonomous services. This decomposition facilitates scalability, strong isolation, and the specialization of data systems tailored to specific workloads and data formats [4].

2.1. Advantages of Microservices

a. **Scalability**
   Systems can be scaled horizontally by increasing or decreasing the number of instances of specific services, allowing them to handle spikes in demand without affecting other parts of the system [5].
b. **Development Flexibility**
   Each service can be developed independently, enabling different development teams to work simultaneously without interfering with one another. This speeds up development and product release cycles [6].
c. **Resilience to Change**
   Service isolation allows changes to be made to one service without impacting the entire system. This supports faster and more efficient adaptation to evolving business requirements [7].
d. **Easier Maintenance**
   Maintenance and debugging are simplified due to each service’s limited scope and functionality. Enhancements or fixes can be made to individual services without disrupting the entire system [8].

2.2. Disadvantages of Microservices

a. **Management Complexity**
   Despite offering flexibility, managing a large number of distributed services introduces complexity. Coordinating inter-service communication, monitoring, and version control requires a mature and well-designed management infrastructure [9].
b. **Network Overhead**

Dividing an application into separate services increases the need for inter-service communication over the network. In large-scale distributed environments, this can lead to significant network overhead and impact overall system performance [6].

c. **Infrastructure Maintenance**
Migrating to a microservices architecture may increase infrastructure maintenance requirements. Managing distributed databases, configuration settings, and error handling becomes more complex and demands greater technical expertise [10].

d. **Cross-Service Transaction Consistency**
Handling transactions that span multiple services can be complex. Ensuring data consistency across these services requires careful design and robust implementation strategies [6].

e. **Error Handling Challenges**
Failures in one service can impact other parts of the system. Implementing effective error handling and recovery mechanisms becomes more challenging and requires a well-thought-out strategy [10].

f. **Not Suitable for All Applications**
Microservices architecture may not be ideal for all types of applications. Smaller or less complex applications may not benefit significantly from this approach, while the associated management overhead could outweigh the advantages [6].

3. Research Methodology

In the development of a Microservices Architecture, this study adopts the Software Development Life Cycle (SDLC) model as the primary research methodology. SDLC is a structured process used for creating and modifying software systems, encompassing several phases including planning, analysis, design, implementation, testing, and maintenance. By following the SDLC model, developers can ensure that system development is carried out in an organized and efficient manner, while meeting both business objectives and user requirements. For the system design phase, the author recommends the use of the prototyping method, based on the observation that frequent changes during software development can affect system stability and lead to significant delays. The prototyping approach is chosen because it involves the evaluation of prototypes prior to the actual coding stage. From these observations, the author emphasizes that the analysis phase should be thoroughly completed before proceeding to implementation. The main advantage of the prototyping method lies in this evaluation stage, where the suitability of the design can be assessed and validated by stakeholders. If any discrepancies or necessary adjustments are identified, a more focused and in-depth analysis can be conducted before entering the coding phase. This approach is expected to reduce the risk of unexpected changes during development. Additionally, the use of the prototyping method offers the benefit of **early user involvement**, enabling feedback and collaboration at an early stage of the process. Therefore, the author argues that this approach can accelerate development and enhance overall efficiency, while highlighting the importance of thorough analysis before the coding phase begins.

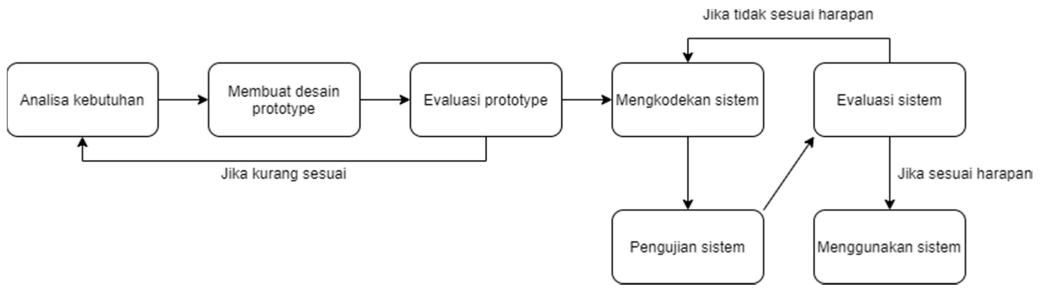

Figure 1 SDLC prototype

4. Result and Discussion

Functional Requirements Analysis is conducted to understand the capabilities of the system being designed and how it can address potential issues. This involves an initial phase of observing and elaborating on business processes, allowing system owners to anticipate application size, user volume, and expected traffic. In this context, choosing a microservices architecture is considered an appropriate decision. Microservices architecture decomposes an application into smaller, manageable services, making the system easier to understand and maintain. Unlike the monolithic approach, which results in a large, complex application package, microservices offer greater flexibility in development, maintenance, and deployment. When there is a need to adopt a new technology or programming language, microservices enable smoother transitions, as each service can be developed and managed independently. During the maintenance phase, a dedicated team is often required to manage changes originating from external systems, such as operating system updates or technology shifts. In applications that utilize microservices architecture, such changes are easier to implement due to the smaller size and modular nature of the services. This stands in contrast to monolithic systems, where such updates may require rebuilding the entire application to accommodate new technologies or programming languages.

4.1. Existing System Architecture

The current monolithic architecture is developed as a single large unit using the Serenity .NET framework. The server operates on a Windows-based operating system, and all data is stored in a single SQL Server database. Various components and application modules interact directly with this centralized database to perform operations such as storing, updating, and retrieving data. The main application functions as the central coordinator, managing system activities and ensuring data integrity across all modules. Other modules—such as user management, transaction processing, and reporting—are also directly connected to the same database, relying on the stored data to support operational tasks. This creates a high level of interdependency among components, where a change in one part of the system can have widespread impacts on others.

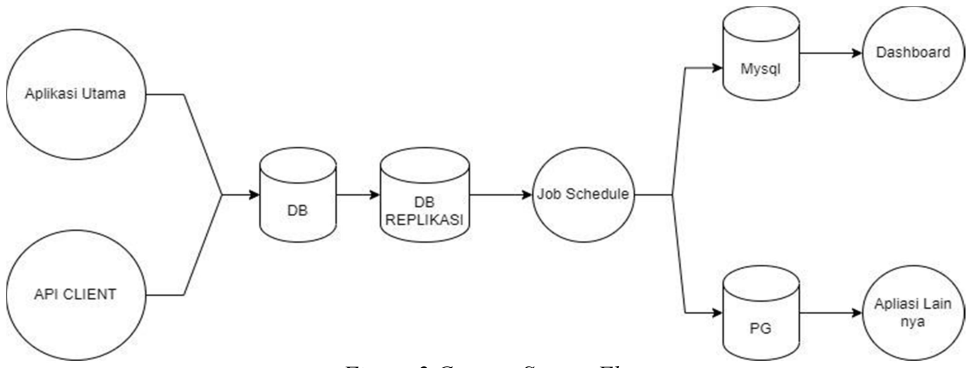

Figure 2 Current System Flow

As shown in Figure 2, the current monolithic system relies on a single central database shared by all applications. Both the main application and the API client interact directly with the primary database (DB), which then replicates data to a secondary database (DB Replication). A job scheduler manages the replication process to ensure the data in the primary DB is regularly updated in the replicated DB. From the replicated DB, data is distributed to two different databases: MySQL and PostgreSQL (PG), each serving different application needs. MySQL is used to provide data to a dashboard application, while PostgreSQL supports other auxiliary applications. This architecture places a substantial load on the primary database, as all data requests and replication tasks pass through a single centralized point. This setup highlights a significant limitation of monolithic architecture, particularly at scale: the reliance on a single database creates a performance bottleneck, which negatively impacts the overall efficiency and responsiveness of the system.

4.2. Proposed System Architecture

This architecture emphasizes a modular and isolated design, comprising four distinct layers that support the development of independent services. By adopting the microservices approach, the system is expected to be more responsive to change, offer greater scalability, and accelerate the development process. The design planning also encourages a thorough consideration of how each microservice interacts within the ecosystem, ensuring smooth and coherent integration across the entire system. The use of an API Gateway, service containers, and a message broker (Apache Kafka) plays a key role in facilitating efficient communication and data flow between components. With this design, the proposed architecture aims to maximize efficiency, enhance system resilience, and enable faster adaptation to evolving business requirements and technological advancements in the future.

The proposed system architecture is structured into several layers, as illustrated in Figure 3**.**

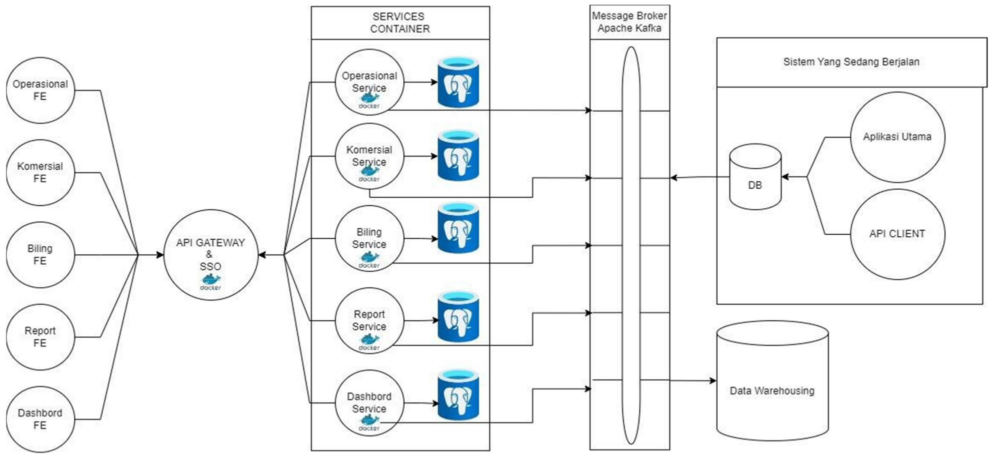

Figure 3 Proposed System Architecture

4.2.1. Frontend Layer

The separation between the user interface layer (frontend) and business logic has become an increasingly adopted architectural approach in modern system development. Various studies have shown that this separation not only improves software development efficiency but also supports the principles of modularity and scalability [11]. Within microservices-based architectures, the frontend plays a vital role as the visual interface for accessing distributed backend services, allowing users to interact with complex functionalities through a simplified and focused interface. From a technical standpoint, isolating the frontend layer enables developers to apply user-centered design principles without being burdened by the complexities of backend data flows. This separation is further supported by the use of API Gateways and Single Sign-On (SSO) mechanisms, which consolidate access control across multiple backend services through a unified entry point. Research in system complexity management suggests that this layered architecture not only simplifies debugging and testing but also enhances development agility by allowing teams to work in parallel across different layers [12]. Furthermore, the use of decentralized frontends aligned with service domains such as Operational FE, Commercial FE, and others reflects the adoption of domain-driven design principles. This approach is well-suited for large-scale, evolving systems, where the presentation layer can be tailored to the specific business logic of each domain. As such, the frontend becomes more than just a visual interface; it serves as a strategic component that supports digital transformation, promotes efficient development practices, and delivers a more intuitive user experience.

4.2.2. API Gateway and Single Sign-On (SSO) Layer

The implementation of Kong Gateway and a Single Sign-On (SSO) system plays a crucial role in establishing a secure, efficient, and integrated system architecture. These two components are specifically utilized to simplify user interactions across multiple services while enhancing access security. Kong Gateway, selected as the API gateway solution in this study, acts as the central controller for communication traffic among services. Its functionalities extend beyond simple request routing to include data aggregation, load management, and consistent application of authentication and authorization policies across the system. Meanwhile, the integration of an SSO system within the system design enables users to authenticate once and gain access to multiple services without repeated logins. This approach significantly improves user experience by reducing login friction, while also strengthening security through centralized credential management. It minimizes the risks associated with multiple logins and ensures policy consistency throughout the system. As a result, the system becomes both more secure and user-friendly two essential attributes for the development of complex and distributed digital services. Overall, the adoption of Kong Gateway and SSO within the scope of this research demonstrates a significant impact on service management efficiency, system security enhancement, and seamless service integration. These technologies contribute to the establishment of centralized access control, modular system architecture, and standardized workflows.

This affirms that choosing the right infrastructure at the integration layer can greatly influence the system's performance, scalability, and reliability in the context of this research.

4.2.3. Service Layer

The service layer represents an architectural structure that segments applications into independent yet interconnected components, all linked through an API Gateway. In this approach, a single application is decomposed into multiple standalone modules, each managed by a separate development team or division. These independent services are designed with clear responsibilities and are capable of functioning autonomously, without requiring direct dependencies on other modules. The API Gateway plays a critical role in this architecture, serving as a centralized entry point that handles requests and responses across the distributed services. It acts as a unified interface that simplifies and streamlines communication between various services, enabling efficient and well-coordinated data exchange. By managing the integration layer through a standardized gateway, the system reduces complexity, enhances operational efficiency, and maintains better control over inter-service interactions. This architectural setup enhances scalability, flexibility, and maintainability, as each service can be developed, deployed, and maintained independently. It fosters an adaptive environment that allows development teams to respond quickly and effectively to changes within dynamic business contexts. While the first layer the frontend layer typically follows specific architectural recommendations, the service layer retains the flexibility to adopt the Model-View-Controller (MVC) paradigm when suitable. Even without strictly adhering to frontend-layer prescriptions, the service layer can still support MVC principles. This allows for a clear separation of concerns among data management (Model), presentation logic (View), and control flow (Controller). Ultimately, the service layer provides an open environment where developers can apply effective design patterns such as MVC based on the specific needs and characteristics of each service. This flexibility supports customized development approaches without relying on the frontend layer as a prerequisite.

4.2.4. Layer Message Broker

Apache Kafka serves as a critical communication intermediary within microservices architecture, enabling efficient and structured interaction among various services and systems. Its primary role revolves around facilitating service intercommunication—allowing distinct services such as Operational Service, Commercial Service, Billing Service, Reporting Service, and Dashboard Service to exchange messages without direct dependencies on one another. Kafka supports the principle of loose coupling, where services are designed to be independent, making it easier to manage, scale, and evolve the system. Each service can produce or consume data through Kafka without needing to know the internal workings of the other services it communicates with. This separation enhances modularity and simplifies maintenance. One of Kafka's most powerful capabilities is enabling real-time data streaming. It allows the seamless flow of data from various service sources to target destinations whether those be live operational systems or long-term storage like data warehouses. This real-time transmission ensures that insights and actions based on data can occur with minimal delay, increasing responsiveness and system efficiency. Kafka also contributes significantly to system scalability and resilience. It can handle high volumes of data while ensuring reliable message delivery, even in the event of service failures. This fault-tolerant design supports consistent and dependable system behavior, which is crucial for distributed architectures. Moreover, Kafka helps decouple data producers and consumers, eliminating the need for direct service-to-service connections. This abstraction layer not only simplifies communication but also makes the development and alignment of services more flexible and agile. As a message queue manager, Kafka guarantees that messages are delivered in an orderly and secure manner between services. It orchestrates the flow of data, ensuring that each message reaches its intended destination correctly and reliably. Finally, Kafka facilitates integration with legacy systems. It allows data from traditional applications and external clients to be routed through Kafka into modern, containerized services—without the need for major modifications to existing infrastructure. This ensures smooth interoperability between legacy and modern components. In summary, Apache Kafka is a cornerstone of microservices architecture. It empowers efficient communication, accelerates data processing, and enhances system modularity, flexibility, and scalability.

4.2.5. Layer Datawarehouse

Although considered an optional layer, the Data Warehouse Layer plays a crucial role in enhancing data analysis processes and supporting data replication. This layer serves as a strategic component that helps minimize the risk of data loss when unexpected service disruptions occur. By integrating a Data Warehouse Layer, data can be accessed and analyzed more efficiently, enabling faster and more informed decision-making. Additionally, the data replication functionality provided by this layer contributes to system reliability. In the event of a service failure, the replicated data remains available, ensuring the continuity and integrity of critical information. Therefore, while it is not mandatory, the Data Warehouse Layer offers a strategic advantage in managing complex services by optimizing data workflows and providing a safety net for data protection.

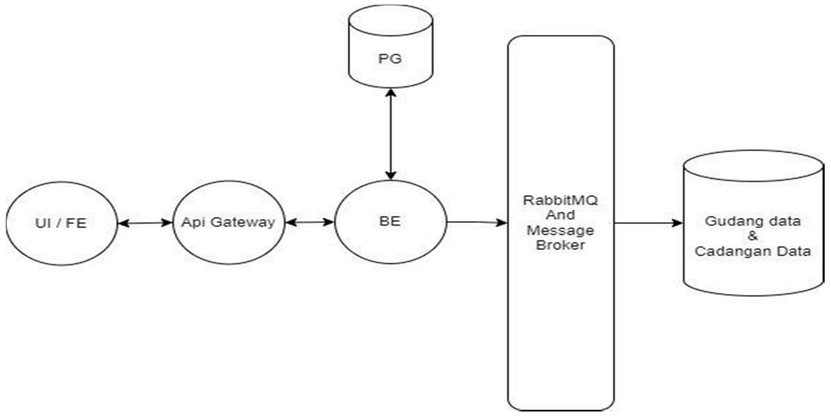

Figure 4 Architectural Fragment

Figure 4. illustrates a simplified breakdown of the proposed system architecture, showing the flow of processes beginning from the smallest components. In this diagram, the User Interface (UI) or Front-End (FE) interacts directly with an API Gateway, which serves as the entry point for managing incoming requests and routing them to the Back-End (BE). The back-end communicates with a PostgreSQL (PG) database to handle core data operations. In parallel, RabbitMQ, acting as a message broker, manages asynchronous communication across different system components. This ensures that the system remains scalable and responsive, even under high-load conditions. The data generated and processed through this architecture is then stored in the Data Warehouse. This centralized storage supports further analytics and acts as a reliable backup mechanism, contributing to the robustness and efficiency of the overall system.

4.3. Description of Technology Used

Table 1 description

Items	Description	Total
OS Server	Linux Ubuntu	3
Container management	kubernates	3
Container Isolasi	docker	21
Service	Aplication services	8
Frontend	ReactJs (Typescript)	
Backend	NodeJs	
Database	Postgresql & Mysql	
Message Broker	Apache Kafka	1
Api gateway	Kong Gateway	
Datawarehouse & Replikasi	Postgresql	
developer	2 frontend / 2 backend	4

This section presents a list of technologies implemented in this study, covering the core components that support the system architecture based on microservices. The selection of these technologies is guided by the system's requirements for scalability, efficient service management, as well as security and reliability in inter-component communication. The system is hosted on three Linux Ubuntu servers, which offer a stable and highly compatible environment for container-based development tools. For container orchestration, Kubernetes is employed, enabling centralized and automated deployment and management of services. Docker is used to isolate services, with a total of 21 active containers, each representing various microservice components within the system. The core services are implemented as part of the application services layer, consisting of eight units, including frontend, backend, and service containers. The frontend interface is developed using ReactJS with TypeScript, while the backend utilizes NodeJS, supporting high-performance and asynchronous processing. The database layer combines PostgreSQL and MySQL to enable replication and support data analytics requirements. To facilitate asynchronous communication between services, Apache Kafka is used as the message broker. Kong Gateway handles API request management and integrates seamlessly with the Single Sign-On (SSO) system for unified authentication. For long-term data storage and replication, PostgreSQL serves as the foundation of the data warehousing solution. The development team consists of two frontend and two backend developers, making a total of four active developers responsible for building and maintaining the system.

5. Results, Conclusion, and Recommendation

The results of the research conducted are based on the processes of design, implementation, and evaluation of a microservices architecture system in the case study of PT. MALINDO. The analysis was carried out to test the previously established hypotheses, evaluate the impact on system performance and efficiency, and assess the success of the architectural transformation from a monolithic structure to microservices. Conclusions were drawn from the findings obtained, and recommendations are provided as a guide for the next steps in system development.

5.1. Hypothesis

This study is based on the hypothesis that adopting a microservices architecture can address the limitations of the current monolithic system used by PT.MALINDO, particularly in terms of performance, scalability, and development flexibility. The key hypotheses tested include:

a. That by implementing containerization using Docker, each service can be isolated and configured with dedicated resources, thus preventing system timeouts and performance degradation across services even when hosted on the same physical server.
b. That the use of open-source tools and platforms including ReactJS, NodeJS, Kong Gateway, Apache Kafka, PostgreSQL, and Kubernetes—not only promotes flexibility and modularity but also offers economic benefits due to the availability of extensive libraries, plugins, and strong community support.
c. That the separation of the frontend, backend, and service logic layers enables more focused and structured development and maintenance processes, which in turn simplifies debugging and scaling efforts.

5.2. Findings and Conclusion

Based on the architectural design, simulation outcomes, and supporting literature, this research concludes that transitioning to a microservices architecture offers substantial advantages over a monolithic approach, especially in the context of PT.MALINDO's operational needs. Key findings include:

a. The decomposition of services into independent units successfully mitigated previous bottlenecks and potential deadlocks encountered in the monolithic system.
b. With container-based deployment and orchestration, individual services can now be developed, tested, and deployed independently without disrupting overall system stability.
c. The integration of Kong Gateway enables centralized and secure API request management, while the inclusion of a Single Sign-On (SSO) system enhances the user experience by providing seamless and consistent access control.
d. The adoption of Apache Kafka significantly improves asynchronous service communication, particularly in handling transactional data and replication workflows.

Development processes benefited from specialized, domain-focused teams, allowing for more effective division of tasks and clearer alignment with business logic across services.

Overall, the microservices approach has proven to enhance system resilience, development efficiency, integration ease, and service scalability all of which are essential in supporting the digital transformation goals of the organization.

5.3. Recommendations

Based on the findings of this research, it is recommended that the migration to microservices be carried out gradually and strategically, taking into consideration the current infrastructure and the readiness of internal resources. The following strategic suggestions are proposed:

a. Conduct a comprehensive evaluation of existing infrastructure and align it with microservices requirements, particularly in terms of container orchestration, message brokering, and data synchronization.
b. Establish domain-specific development teams and invest in upskilling them with technical training on tools such as Docker, Kubernetes, and Kong Gateway.
c. Implement a DevSecOps strategy to ensure security, automation, and long-term sustainability across development and deployment pipelines.
d. Adopt a robust observability framework, including real-time monitoring, centralized logging, and distributed tracing, to gain deeper operational insights.
e. Ensure active stakeholder involvement and foster cross-functional team collaboration, as successful migration depends not only on technology but also on organizational culture and change management.

With a comprehensive and consistent approach, the migration to a microservices architecture can serve as a strong foundation for the digital continuity and long-term success of PT.MALINDO in today’s fast-evolving business environment.

References

[1] H. Suryotrisongko, “Arsitektur microservice untuk resiliensi sistem informasi,” *SISFO*, vol. 6, no. 2, pp. 6, 2017.

[2] G. Munawar and A. Hodijah, “Analisis model arsitektur microservice pada sistem informasi DPLK,” *Sinkron: Jurnal dan Penelitian Teknik Informatika*, vol. 3, no. 1, pp. 232–238, 2018.

[3] M. G. de Almeida and E. D. Canedo, “Authentication and authorization in microservices architecture: A systematic literature review,” *Applied Sciences*, vol. 12, no. 6, pp. 3023, 2022.

[4] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, “Data management in microservices: State of the practice, challenges, and research directions,” *arXiv preprint arXiv:2103.00170*, 2021.

[5] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, O’Reilly Media, 2017.

[6] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed., O’Reilly Media, 2021.

[7] M. Nygard, *Release It!: Design and Deploy Production-Ready Software*, 2nd ed., Pragmatic Bookshelf, 2018.

[8] O. Greevy, S. Ducasse, and T. Girba, “Analyzing software evolution through feature views,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 6, pp. 425–456, 2005.

[9] C. Richardson, *Microservices Patterns: With Examples in Java*, Simon and Schuster, 2018.

[10] V. F. Pacheco, *Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices*, Packt Publishing, 2018.

[11] M. Fowler, “Microservices,” *MartinFowler.com*, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[12] S. Newman, *Building Microservices*, O’Reilly Media, 2021.

[13] Nofri Wandi Al-Hafiz, Helpi Nopriandi, and Harianja, “Design of Rainfall Intensity Measuring Instrument Using IoT-Based Microcontroller”, *JTOS*, vol. 7, no. 2, pp. 202 - 211, Dec. 2024.

[14] N. W. Al-Hafiz and H. Harianja, “Design of an Internet of Things-Based automatic cat feeding control device (IoT)”, *Mandiri*, vol. 13, no. 1, pp. 161–169, Jul. 2024.

[15] PutriD. and Al-HafizN., “SISTEM INFORMASI SURAT KETERANGAN GANTI RUGI TANAH PADA KECAMATAN KUANTAN TENGAH MENGGUNAKAN WEBGIS”, *Biner : Jurnal Ilmiah Informatika dan Komputer*, vol. 2, no. 2, pp. 112-121, Jul. 2023.

[16] H. Harianja, N. W. Al-Hafiz, and J. Jasri, “Data Analysis of Informatics Engineering Students of Islamic University of Kuantan Singingi”, *JTOS*, vol. 6, no. 1, pp. 23 - 30, Jan. 2023.

[17] Siregar, M., and N. Al-Hafiz. "Design of Cloud Computer to Support Independent Information System Servers Universitas Islam Kuantan Singingi." *Journal of Information System Research (JOSH)* 3.2 (2022)

[18] A. Apri Denta, H. Nopriandi, and E. Erlinda, “Information System Design Realization and Performance Achievements of the Manpower Office of Kuantan Singingi District”, *JTOS*, vol. 7, no. 1, pp. 01 - 09, Nov. 2024.

[19] A. AP Putra, E. Erlinda, and M. Yusfahmi, “Sales System in the Endocell Mobile Phone Business Using the CRM (Customer Relationship Management) Method) in Kompe Berangin Village, Cerenti District”, *JTOS*, vol. 7, no. 1, pp. 10 - 21, Nov. 2024.

[20] D. Setiawan, F. Haswan, and J. Jasri, “Design of School Bell Scheduling Application Based on Arduino Uno on MTs Babussalam Simandolak”, *JTOS*, vol. 7, no. 1, pp. 22 - 30, Jul. 2024.

[21] D. Juniarti, A. Aprizal, and S. Chairani, “Information System for Analyzing Disease Trends in the Region Cerenti Health UPTD”, *JTOS*, vol. 7, no. 1, pp. 31 - 43, Jun. 2024.

[22] F. Restuadi, H. Nopriandi, and A. Aprizal, “ANALISIS QOS JARINGAN INTERNET FAKULTAS TEKNIK UNIVERSITAS ISLAM KUANTAN SINGINGI MENGGUNAKAN WIRESHARK 4.0.3”, *JTOS*, vol. 7, no. 1, pp. 44 - 54, Jun. 2024.

[23] J. Jasri and N. W. Al-hafiz, “Designing a mobile-based infaq application markazul quran wassunnah foundation (MQS)Kuantan Singingi”, *J. Teknik Informatika CIT Medicom*, vol. 15, no. 5, pp. 247–254, Nov. 2023.

[24] R. Nazli, A. Amrizal, H. Hendra, and S. Syukriadi, “Modeling User Interface Design E-Business Applications for Marketing Umkm Products in Payakumbuh City Using Pieces Framework”, *JTOS*, vol. 7, no. 2, pp. 55 - 66, Nov. 2024.

[25] Siti Saniah and Mhd. Furqan, “Classification Of Rice Plant Diseases Using K-Nearest Neighbor Algorithm Based On Hue Saturation Value Color Extraction And Gray Level Co-Occurrence Matrix Features”, *JTOS*, vol. 7, no. 2, pp. 212 - 223, Dec. 2024.