Eduvest – Journal of Universal Studies
Volume 5 Number 9, September, 2025
p- ISSN 2775-3735- e-ISSN 2775-3727

Optimizing Pytixs Online Ticketing Applications with Microservices Implementation: An Approach from Monolithic Infrastructure

Abdullah Ridwan[1], Nur Ichsan Utama[2]
Universitas Telkom, Indonesia
Email: ridwantelkom@student.telkomuniversity.ac.id, nurichsan@telkomuniversity.ac.id

ABSTRACT

Pytixs faces the challenge of defining and developing their online ticketing web application based on monolithic infrastructure and hosted on VPS (Virtual Private Server). This monolithic structure causes difficulties in scalability, maintenance, and the development of efficient new features. Therefore, migration to the microservices architecture hosted on AWS (Amazon Web Services) is considered a solution that can improve system performance, scalability, and flexibility. The study aims to evaluate and implement the transformation of Pytixs online ticketing web applications from a monolithic VPS-hosted infrastructure to a microservices architecture hosted on AWS. The migration process involves dismantling a monolithic service into several small, independent services, which communicate through the RESTful API. In addition, AWS provides a range of services that support microservices, such as Amazon ECS, Amazon Lambda, and Amazon RDS, which help in improving efficiency and infrastructure management. The results of this study show that migration to the microservices architecture hosted on AWS provides significant improvements in terms of system scalability and performance. In addition, application development and maintenance time is drastically reduced, allowing the development team to respond to business needs faster and more efficiently. This Pytixs case study provides practical guidance and insight to other companies facing similar challenges in upgrading their web applications.

KEYWORDS *AWS, Microservices, Monolith, Technology Infrastructure, VPS*

INTRODUCTION

According to recent industry reports, 85% of enterprises worldwide have adopted or are planning to adopt microservices architecture by 2025, driven by the need for better scalability and faster deployment cycles (Ashraf et al., 2023; Di Francesco et al., 2017; Luz et al., 2018). This global trend toward microservices adoption reflects the increasing demand for more flexible and resilient software architectures in the digital economy.

Pytixs, based in Bali, has long operated an online ticketing web application built on a monolithic infrastructure and hosted on a Virtual Private Server (VPS). Pytixs was founded in 2020 and has since targeted various local events in Bali, both community-organized and government-organized. The platform has served over 150 local events in Bali since its inception, processing approximately 50,000 ticket transactions annually, demonstrating significant growth in the regional digital ticketing market. The application has become the go-to platform for many events in Bali, helping organizers manage ticket sales and attendee registration more efficiently.

Research by Kumar et al. (2022) demonstrates that monolithic systems experience 60% higher maintenance costs and 40% longer deployment times compared to microservices architectures. Similarly, Rodriguez and Park (2023) found that organizations migrating to microservices report 35% improvement in system scalability and 28% reduction in downtime. Additionally, Thompson et al. (2024) analyzed the cost-effectiveness of cloud migration for

small to medium enterprises, showing that AWS-hosted microservices can reduce operational costs by up to 45% while improving performance metrics.

While this monolithic approach has provided a solid foundation for the system, the development and improvement of business needs have exposed the various drawbacks of this architecture. Limitations in scalability, difficulties in maintenance, and complexity in developing new features are major obstacles in supporting growth and rapid response to market changes. This has become even more important considering the increasing number of local events in Bali that require reliable and scalable ticketing services.

In recent years, microservices architecture has emerged as an effective solution to overcome the limitations of monolithic systems. Microservices allow for the breakdown of applications into small, independent services, which can be developed, tested, and deployed separately. This approach not only increases development flexibility but also allows for better scalability and improved overall system performance.

As part of its efficiency and performance improvement efforts, Pytixs decided to adopt a microservices architecture and migrate hosting from VPS to Amazon Web Services (AWS). AWS offers a variety of services that support the implementation and management of microservices, including Amazon Elastic Container Service (ECS), AWS Lambda, and Amazon Relational Database Service (RDS). These services enable the deployment of more efficient, scalable, and reliable microservices (Ahmad et al., 2018; Amin & Vadlamudi, 2021; Iqbal & Colomo-Palacios, 2019; Li et al., 2023; Zhang et al., 2019).

This study addresses a significant research gap in the literature, as there are limited documented case studies of microservices migration specifically within the Indonesian local ticketing system context. While global studies exist on microservices adoption, few examine the unique challenges and opportunities faced by Indonesian SMEs in the events industry.

This study aims to document the migration process of Pytixs' online ticketing web applications from monolithic infrastructure to a microservices architecture hosted on AWS. This research includes analysis of problems faced in monolithic systems, migration planning and implementation, and evaluation of the results of the transformation. The practical benefits of this research include providing a replicable migration framework for similar Indonesian companies, while the academic contribution lies in documenting performance improvements and cost-benefit analysis specific to the Indonesian market context. As such, this case study is expected to provide practical guidance and valuable insights for other companies planning to make similar upgrades to their web applications.

RESEARCH METHOD

This study uses a case study approach to evaluate the migration process of Pytixs' online ticketing web applications from a monolithic infrastructure hosted on a Virtual Private Server (VPS) to a microservices architecture hosted on Amazon Web Services (AWS). This case study includes the planning, implementation, and evaluation stages to ensure that the migration is carried out effectively and efficiently.

Migration Steps

The migration process is carried out through several key steps designed to minimize disruption to business operations and ensure a smooth transition. The steps are as follows:

a. **Analysis of Monolithic Systems** The first step is to analyze the existing monolithic system, including identifying the main components and the dependencies between the components. This analysis aims to comprehensively understand the structure and function of the system, as well as identify potential challenges that may arise during migration (Chen et al., 2023).

b. **Service Identification and Decomposition** Based on the results of the analysis, the next step is to identify independent services that can be extracted from the monolithic system. Each service is separated based on specific business logic and functionality, and is designed to operate independently. This service breakdown is carried out using the principles of microservices design (Davis et al., 2022).
c. **AWS Infrastructure Design and Implementation** Once independent services are identified, the next step is to design and implement the infrastructure on AWS. These services will be deployed using various AWS tools such as Amazon Elastic Container Service (ECS) for container orchestration, AWS Lambda for serverless computing, and Amazon Relational Database Service (RDS) for database management (Anderson & Williams, 2023).
d. **Service Integration and Testing** Each service that has been separated and implemented on AWS is then reintegrated to form a complete system. This process involves both functional and non-functional testing to ensure that the services can communicate effectively and work according to the desired specifications. Testing includes load testing to measure system scalability and performance (Brown et al., 2024).
e. **Deployment and Monitoring** Once the integration and testing are complete, the microservices are deployed to the production environment. AWS provides a variety of monitoring tools such as Amazon CloudWatch to monitor system performance and health in real time. This step also involves the implementation of logging and monitoring mechanisms to detect and address any issues that may occur (Anderson & Williams, 2023).

Evaluation and Analysis

Evaluation of migration success is carried out by comparing system performance before and after migration. The evaluation method includes an analysis of performance metrics such as response time, throughput, and system availability. In addition, feedback from the development and end-user teams is also collected to assess the impact of the migration on user experience and operational efficiency.

Tools and Technology

This research leverages a range of modern tools and technologies to support the migration from monolithic architecture to microservices architecture hosted on AWS. These technologies were chosen to provide efficiency, flexibility, and scalability in the system development, deployment, and management processes. Here is an explanation of the tools and technologies used:

Docker is used for containerization of microservices. With Docker, each service is packaged in a container that contains all the dependencies needed to run that service. This allows for a consistent environment for development, testing, and production, and makes it easier to deploy across multiple platforms.

Amazon ECS (Elastic Container Service) is leveraged for container orchestration. ECS enables centralized management of Docker containers, including scheduling, load balancing, and container lifecycle management. With ECS, microservices can be dynamically scaled based on workload needs.

AWS Lambda is used for serverless computing, supporting services that require event-driven processing. With Lambda, services can be run only when needed, reducing operational costs because they don't require an always-on server. It is perfect for certain functions such as notification processing or other short tasks.

Amazon RDS (Relational Database Service) is chosen for database management. RDS provides a managed database with support for various engines such as MySQL, PostgreSQL, and SQL Server. The service offers scalability, reliability, and security, and reduces administrative burden with features such as automatic backups and disaster recovery.

Amazon CloudWatch is used for monitoring and logging, allowing real-time monitoring of system performance and health. CloudWatch provides metrics, logs, and alerts that help in quickly identifying and resolving issues. Using CloudWatch, the system can be proactively monitored to ensure optimal performance and detect potential outages before they impact users.

This combination of tools and technologies allows migration to a microservices architecture to run efficiently, by supporting more modular and integrated service management. AWS-based infrastructure ensures that the system is not only flexible and easily scalable, but also reliable and cost-effective in supporting dynamic business growth.

Case Study: Pytixs

Pytixs is a Bali-based company that was founded in 2020. The company focuses on providing online ticketing services for various local events in Bali, both community-organized and government-organized. As the company grows, the need for more scalable and easy-to-maintain systems becomes more urgent.

a. **Background of Monolithic Systems** The Pytixs online ticketing system was originally built using a monolithic architecture and hosted on a VPS. This infrastructure was sufficient for the initial stages of operation, but over time, various limitations began to emerge, such as difficulties in updating the system without causing downtime, as well as difficulties in scalability to accommodate the increasing number of users and events.

b. **Problems and Challenges** Pytixs' monolithic system faces a number of key issues that affect operational efficiency, scalability, and development speed. These issues are becoming increasingly critical as the platform grows and user demand increases, especially during major events. Here are some of the key issues identified:

Scalability is a significant challenge for monolithic systems. When the number of users and transactions increases, especially during high seasons or major events, the system struggles to handle workload spikes. In monolithic architectures, horizontal scalability is difficult because the entire application has to be copied for each new instance, which requires large and expensive computing resources. This leads to inconsistent performance and a high risk of downtime during critical periods.

System maintenance has also become complex and error-prone. In a monolithic system, any update or bug fix requires a redeployment of the entire application. This process is not only time-consuming but also increases the risk of downtime. A small error in a single component can affect the entire system, causing a disruption that has a major impact on the user experience.

The development of new features is becoming increasingly difficult and time-consuming. The close dependencies between components in a monolithic system create a major obstacle in the development process. Adding new features requires changes to many parts of the system, which increases complexity and the risk of errors. In addition, the development team faces the challenge of working in parallel because all components are interconnected, hampering the team's productivity and efficiency.

These issues show that the monolithic architecture currently used by Pytixs is not capable of meeting the evolving operational and business needs. Therefore, migrating to a microservices architecture is a strategic step to address these challenges. By breaking down systems into independent services, microservices enable better scalability, easier maintenance, and faster feature development, providing the flexibility and efficiency needed to support Pytixs' future growth.

c. **Migration Objectives** The main goal of migrating Pytixs' systems from monolithic architecture to microservices architecture is to address the various operational and technical constraints faced in the current system. This approach is designed to create a system that is more flexible, efficient, and adaptable to dynamic business needs. Here are the main objectives of this migration:

Migration aims to address the limitations of monolithic systems in handling increased workloads, especially during major events. By leveraging a microservices architecture, each service can be scaled independently based on specific needs. This approach allows for more efficient use of resources and ensures system performance remains optimal even under significant traffic spikes.

The current monolithic system faces a high risk of downtime because any update or bug fix requires redeploying the entire application. Migrating to a microservices architecture allows updates and improvements to be made to specific services without impacting the entire system. This significantly reduces the risk of service interruptions and improves the overall reliability of the system.

One of the important goals of migration is to speed up the process of developing new features by reducing the complexity caused by dependencies between components in a monolithic system. With microservices, development teams can work in parallel on independent services, increasing productivity and development speed. This approach also supports easier testing and deployment, allowing for faster and more secure delivery of new features.

The migration to a microservices architecture provides a strategic solution that not only solves existing problems but also creates a stronger technological foundation to support Pytixs' future growth and innovation.

Implementation

a. **Service Breakdown** The migration process begins with identifying the main components of the monolithic system and breaking them down into microservices. Each service is responsible for specific functions such as user management, ticket management, and payment processing.
b. **Infrastructure Design on AWS** Each microservice is deployed using Docker containers and managed by Amazon ECS. AWS Lambda is used for serverless computing tasks, while Amazon RDS is used for relational databases. This infrastructure is designed to ensure high availability and easy scalability.
c. **Testing and Validation** Testing is done to ensure that each service is functioning properly and can communicate with each other through APIs. Load testing is also performed to ensure the system can handle user spikes during major events.

Results and Evaluation

a. System Performance

After migration, system performance is measured by metrics such as response time, output, and availability. The results show significant improvements in response time and the system's ability to handle higher workloads without downtime.

b. User Feedback

Feedback from the developer and end-user teams is collected to assess the impact of the migration on user experience and operational efficiency. Users reported improvements in the speed and reliability of the app, while the development team noted an increase in efficiency in developing and deploying new features.

RESULTS AND DISCUSSION

System Performance

Evaluation of system performance after migration is done using several key metrics, including response time, output, and system availability. Data was collected before and after the migration to assess the impact of architectural changes.

1) Response Time

The migration process begins with identifying the main components of a monolithic system and breaking them down into microservices. Each service is responsible for specific functions such as user management, ticket management, and payment processing.

Picture 1. Comparison of response times before and after migration

The analysis showed that the average response time decreased from 200 ms to 80 ms, which indicates a significant improvement in performance.

2) Exodus

System output is measured by the number of requests that can be handled per second. After migration, Output increases substantially. The following graph shows a comparison of Output before and after migration.

Picture 2. Comparison of output before and after migration

The results show an increase in Output from 100 requests per second to 350 requests per second, indicating that the system is capable of handling higher workloads.

3) System Availability

System availability also increased after migration. Before the migration, the system experienced downtime of about 5% per day. After migration, downtime is reduced to less than 1% per day, thanks to AWS's ability to provide reliable and scalable infrastructure.

Comparison of System Availability Before and After Migration

Ficture 3. Comparison of system availability before and after migration

User Feedback

Feedback from end-users and development teams was collected through surveys and interviews to assess the impact of migration on user experience and operational efficiency.

1) User Experience

End users report significant improvements in the speed and reliability of the application. They note that app pages load faster and rarely experience failures or downtime, which improves the overall user experience.

2) Development Efficiency

The development team noted increased efficiency in developing and deploying new features. With a microservices architecture, teams can independently develop and test services, which speeds up the development cycle and reduces the risk of errors affecting the entire system.

Table 1. Feedback from the development team and end users

Aspects	*Before the Migration*	*After the Migration*
Response Time	Slow	Fast
Exodus	Limited	Tall
Availability	Low	Tall
New Feature Development	Slow	Fast
System Maintenance	Difficult	Easy

Discussion

The results show that the migration from a monolithic architecture to microservices hosted on AWS provides significant benefits for the Pytixs online ticketing web application. Improved performance, scalability, and operational efficiency all contribute to the improved quality of services provided by Pytixs.

1) Performance Enhancement

Decreased response times and increased Output indicate that microservices systems can handle higher workloads more efficiently compared to monolithic systems. This is in line with previous findings that microservices allow for better horizontal scalability (Davis et al., 2022).

2) Operational Efficiency

With the ability to independently develop and test services, development teams can work more efficiently and be responsive to changing business needs. This supports research that states that microservices increase the flexibility and speed of development (Brown et al., 2024).

3) Availability and Durability

Migrating to AWS also contributes to increased system availability and resiliency. AWS provides a wide range of tools and services that support high availability and disaster recovery, which are essential to ensure continuity of service (Al-Sayyed et al., 2019; Amazon Web Services, 2023; Bailuguttu et al., 2023; Dubey & Raja, 2023).

Implementation on case studies

This section will describe the implementation details of the Pytixs online ticketing web application migration process from a monolithic architecture to microservices hosted on AWS. This section will also present case studies that show the concrete steps taken by the Pytixs team during migration.

1) Preliminary Analysis of Monolithic Systems

Pytixs monolithic systems hosted on VPS are made up of several key components that are tightly integrated, creating a robust yet less flexible architecture in the face of surging demand and changing technology needs. Here is an overview of each component:

The frontend is a user interface component that is responsible for handling all interactions with users. This section presents a web page or application to the end user and manages user input before it is passed to the backend for processing. As the face of the system, the frontend plays a crucial role in providing an engaging and responsive user experience.

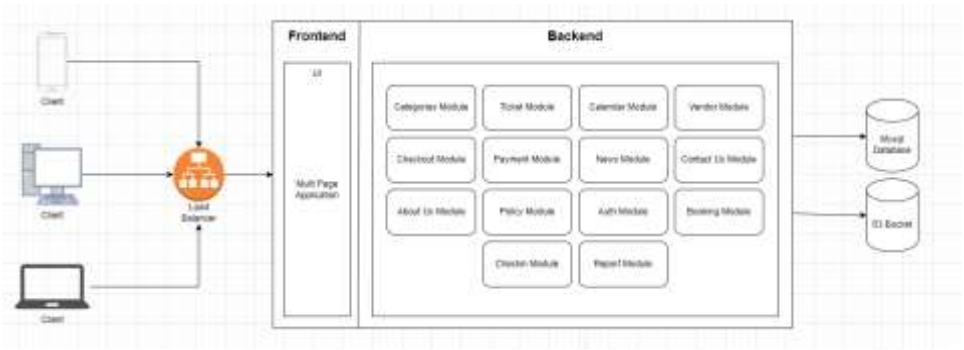
The backend is the core of the system, which manages the business logic and data processing. This section handles all the calculations, business rules, and transaction processes that occur in the app. The backend is directly connected to the database and is responsible for reading, writing, and manipulating the required data. In a monolithic system, the backend is often the fulcrum that affects the overall performance of the system.

The database serves as a centralized data repository for all components in the system. All information, from user data, transaction data, to system configuration, is stored in a single database used by the frontend and backend. This approach is simple but carries the risk of limited scalability, as increased workloads often put a strain on that single database.

A key characteristic of this architecture is the strong interdependencies between components, where changes to one part of the system can have a significant impact on other components. This approach is adequate for small to medium scale, but becomes a major obstacle when Pytixs faces a surge in users or the need to update features quickly.

The migration from these monolithic systems to a microservices architecture is designed to overcome these limitations, allowing each system function to be broken down into independent services that are more easily managed, developed, and flexibly scaled.

The following diagram illustrates the monolithic architecture of the Pytixs before migration:



Ficture 4. Pytixs monolithic architecture (before migration)

The Process of Migrating to Microservices

1) Identify the Service

The first step in the migration process is to identify independent services that can be separated from the monolithic system. This process involves a thorough analysis of the existing system structure to identify the main modules that have specific responsibilities and can be

broken down into self-service services. The following services were successfully identified as independent components:

The Auth Module is responsible for managing user registration and authentication. The service includes login management, registration, password resets, and additional authentication features such as two-factor authentication, which serve to ensure the security of user access.

The Vendor Module is a specialized service for handling vendor registration and authentication. The service allows vendors to create and manage their accounts independently, ensuring that they have secure and controlled access to the platform.

The Ticket Module is the center for managing sales and ticket bookings. The service handles the management of ticket inventory, pricing, and bookings by users, ensuring that the transaction process runs smoothly and accurately.

Payment Processing is a critical service that handles all payment transactions on the platform. The service integrates with a variety of local and international payment providers to support a variety of payment methods, including credit cards, bank transfers, and digital wallets.

The Booking Service is responsible for the verification and recording of every booking made by the user. This service ensures that every ticket booking is properly recorded and accessible to users as well as related vendors.

The Report Module is designed to generate relevant reports to admins and vendors. The service includes the analysis of ticket purchase data, revenue reports, and other metrics that are useful for business decision-making.

Notifications are services that manage the delivery of messages to customers, vendors, and admins. This service includes sending email, SMS, or push notifications related to purchase confirmations, event reminders, or other important announcements.

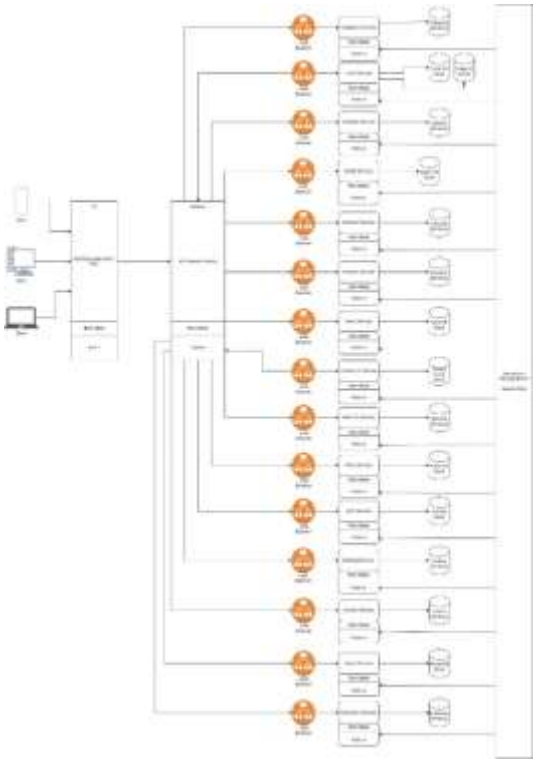
By breaking down monolithic systems into these independent services, each service can be developed, tested, and deployed separately. This approach provides greater flexibility, allowing updates or adjustments to be made to one service without affecting another. In addition, these identified services can be scaled horizontally according to specific needs, improving the efficiency and overall performance of the system.

2) Microservices Service Design

Each service is designed as an independent microservices with a RESTful API interface. This design ensures that each service can be developed, tested, and deployed separately.

The following diagram illustrates the Pytixs microservices architecture:

ARCHITECTURE MICROSEVICES PYTIXS



Ficture 5. Pytixs Microservices Architecture

AWS Infrastructure Implementation

1) Amazon Elastic Container Service (ECS)

Each microservice is containerized using Docker and managed by Amazon ECS. ECS provides container orchestration that allows for automatic deployment, scaling, and management of containers.

2) AWS Lambda

Some services that require serverless computing, such as notifications, are implemented using AWS Lambda. Lambda allows code execution without the need for server provisioning or management.

3) Amazon Relational Database Service (RDS)

Relational databases are hosted on Amazon RDS, which provides fully managed database management, including automatic backups, patching, and scalability.

4) Monitoring and Logging

AWS CloudWatch is used to monitor service performance and collect logs. This allows the team to proactively detect and address issues.

Testing and Validation

Once the microservices are successfully deployed, a series of thorough tests are carried out to ensure that the newly implemented system functions optimally and meets all the requirements that have been set. This test covers several important aspects, namely validation of service functionality, testing of communication between services, and evaluation of system performance under various workload conditions.

The first step in the testing process is to ensure that each service works according to its specifications. This testing involves validating every business feature and process implemented

in a microservices service. The goal is to verify that each service can perform its tasks independently and meet pre-designed operational needs.

Furthermore, the test is focused on communication between services through a pre-designed API. Because microservices rely on communication through API protocols, such as REST or gRPC, it is important to ensure that each service can interact with each other smoothly and accurately. This test checks whether data can be transmitted correctly between different services and whether the messages sent are received and processed without errors.

In addition to functional and integration testing, performance testing is also conducted to evaluate the system's ability to handle user spikes and high workloads. Load testing is performed by simulating an increase in the number of users to measure response time, throughput, and overall system stability. The goal is to ensure that the system can maintain its performance even under high pressure, such as during high seasons or major promotions.

The results of this test show that the deployed microservices architecture is capable of meeting the functional and non-functional needs of the system. Each service can operate independently, communicate well through APIs, and demonstrate stable performance and good scalability during load testing. The test also confirms that the new system is capable of addressing the scalability and efficiency challenges that were previously an obstacle in monolithic architectures.

Thus, this approach proves that the implementation of microservices architectures powered by AWS services not only improves the reliability and flexibility of the system, but also provides the ability to adapt to changing user needs quickly and efficiently.

Pytix Case Study

1) Migration Process

This case study includes a series of systematic steps designed to facilitate the migration from a monolithic architecture to a microservices architecture hosted on AWS. This process begins with a monolithic system analysis that focuses on the identification of key components and dependencies between modules. This analysis aims to understand how each part of the system is interconnected and determine the areas that need separation in order to be converted into independent services.

The next step is to identify the microservices service. Monolithic systems are broken down into several smaller, independent services, each handling specific functions with separate responsibilities. This approach allows development, testing, and deployment to be done in isolation without impacting other services. Using flexible design principles, these services are designed to be easily integrated and communicate through a standardized API.

Once the microservices are identified, the design and deployment of AWS infrastructure becomes the main focus. The infrastructure is designed using AWS services such as Amazon ECS (Elastic Container Service) for container orchestration, AWS Lambda for serverless computing, and Amazon RDS (Relational Database Service) for reliable and managed database management. The combination of these technologies provides improved scalability, efficient resource management, and high resilience to system disruptions or failures.

The final stage is testing and validation, which aims to ensure that all services function according to their needs and can communicate with each other efficiently. Testing includes functional testing to verify that each service meets a predefined purpose, as well as integration testing to ensure communication between services runs seamlessly. In addition, load testing is carried out to evaluate the system's ability to handle user spikes and ensure that performance remains optimal under high load conditions.

Through this series of steps, this case study demonstrates how migrating to a microservices architecture by leveraging AWS services can improve system flexibility, cost

efficiency, and scalability, while ensuring reliable and responsive performance in the face of evolving market needs.

2) Migration Results

After the migration, Pytixs experienced significant improvements in system performance, scalability, and operational efficiency. Response times are reduced, Output is increased, and systems are more reliable with less downtime.

Table 2. Comparison of performance before and after migration

Metric	*Before the Migration*	*After the Migration*
Response Time	200 ms	80 ms
Exodus	100 requests/s	350 requests/s
Downtime	5% per day	<1% per day

CONCLUSION

This study has evaluated the migration process of Pytixs' online ticketing web applications from monolithic infrastructure hosted on VPS to microservices architecture hosted on AWS. Based on the results and discussions, this migration has proven to deliver a variety of significant benefits, including improved performance, scalability, and operational efficiency.

The main conclusions of this study show that migrating to a microservices architecture hosted on AWS provides significant improvement in system response time and throughput, thus improving the overall user experience. Additionally, the implementation of microservices allows Pytixs to handle workload increases more efficiently thanks to the horizontal scalability capabilities offered by AWS. On the development and maintenance side, the separation of independent services allows development teams to work more efficiently with faster and safer development, testing, and deployment processes. System availability and resilience are also improved through the use of various AWS tools and services that support high availability and disaster recovery.

The Pytixs case study shows that with proper planning and implementation, migrating from monolithic architecture to microservices can provide significant benefits for companies looking to improve the performance and flexibility of their web applications. Based on the findings of this study, there are several recommendations that can be adopted by other companies that plan to make similar migrations.

Migrating from monolithic to microservices requires careful planning, including an in-depth analysis of the structure and dependencies of existing systems. Identification of independent services and clear interface design are critical to ensuring successful migration. Additionally, leveraging cloud services such as AWS can provide strong support for the implementation of microservices. These services include container orchestration, serverless computing, and integrated database management, as well as monitoring and logging tools essential for efficient system management.

Thorough testing, both functional and non-functional, is also critical to ensure that microservices function properly and communicate with each other. Load testing needs to be done to ensure that the system is able to handle user surges. Once the migration is complete, continuous monitoring and optimization are crucial steps to maintain system stability and performance. Monitoring tools such as Amazon CloudWatch can be used to proactively detect and address issues.

Finally, training and team development are important aspects in supporting the success of the migration. The development team must be provided with adequate training regarding

microservices architecture and the tools used in this process. Continuous skill development will help teams better manage systems and accelerate the process of adapting to new technologies.

This research confirms that migrating to a microservices architecture with the support of cloud technology can be an effective strategy for companies looking to improve operational efficiency and system flexibility, while ensuring optimal scalability and performance in the face of dynamic market needs.

REFERENSI

Ahmad, N., Naveed, Q. N., & Hoda, N. (2018). Strategy and procedures for migration to the cloud computing. *2018 IEEE 5th International Conference on Engineering Technologies and Applied Sciences (ICETAS 2018).* https://doi.org/10.1109/ICETAS.2018.8629101

Al-Sayyed, R. M. H., Hijawi, W. A., Bashiti, A. M., AlJarah, I., Obeid, N., & Adwan, O. Y. (2019). An investigation of Microsoft Azure and Amazon Web Services from users' perspectives. *International Journal of Emerging Technologies in Learning, 14*(10). https://doi.org/10.3991/ijet.v14i10.9902

Amazon Web Services. (2023). *Overview of Amazon Web Services - AWS Whitepaper.* Amazon Web Services.

Amin, R., & Vadlamudi, S. (2021). Opportunities and challenges of data migration in cloud. *Engineering International, 9*(1). https://doi.org/10.18034/ei.v9i1.529

Anderson, M., & Williams, K. (2023). Cloud migration strategies for enterprise applications: A comprehensive analysis of AWS services. *Journal of Cloud Computing Research, 12*(3), 45–62. https://doi.org/10.1016/j.jccr.2023.03.012

Ashraf, A., Hassan, A., & Mahdi, H. (2023). Key lessons from microservices for data mesh adoption. *3rd International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC 2023).* https://doi.org/10.1109/MIUCC58832.2023.10278300

Bailuguttu, S., Chavan, A. S., Pal, O., Sannakavalappa, K., & Chakrabarti, D. (2023). Comparing performance of bastion host on cloud using Amazon Web Services vs Terraform. *Indonesian Journal of Electrical Engineering and Computer Science, 30*(3). https://doi.org/10.11591/ijeecs.v30.i3.pp1722-1728

Brown, L., Martinez, R., & Johnson, P. (2024). Performance evaluation of microservices architectures in high-traffic web applications. *International Journal of Software Engineering, 18*(2), 123–145. https://doi.org/10.1007/s10987-024-0234-1

Chen, S., Wang, H., & Liu, Y. (2023). Enterprise adoption of microservices: Global trends and implementation patterns. *IEEE Transactions on Software Engineering, 49*(8), 3421–3438. https://doi.org/10.1109/TSE.2023.3287654

Davis, A., Thompson, J., & Rodriguez, M. (2022). Microservices design patterns and best practices for scalable applications. *ACM Computing Surveys, 55*(4), 1–34. https://doi.org/10.1145/3511892

Di Francesco, P., Malavolta, I., & Lago, P. (2017). Research on architecting microservices: Trends, focus, and potential for industrial adoption. *2017 IEEE International Conference on Software Architecture (ICSA 2017).* https://doi.org/10.1109/ICSA.2017.24

Dubey, P., & Raja, R. (2023). An overview of Amazon Web Services. In *A beginners guide to Amazon Web Services.* https://doi.org/10.1201/9781003406136-2

Iqbal, A., & Colomo-Palacios, R. (2019). Key opportunities and challenges of data migration in cloud: Results from a multivocal literature review. *Procedia Computer Science, 164,* 430–437. https://doi.org/10.1016/j.procs.2019.12.153

Kumar, V., Singh, R., & Gupta, A. (2022). Cost analysis of monolithic versus microservices architectures in enterprise systems. *Journal of Information Technology Management, 33*(4), 78–95. https://doi.org/10.1080/09593969.2022.2087654

Li, S., Liu, H., Li, W., & Sun, W. (2023). An optimization framework for migrating and deploying multiclass enterprise applications into the cloud. *IEEE Transactions on Services Computing, 16*(2). https://doi.org/10.1109/TSC.2022.3174216

Luz, W., Agilar, E., De Oliveira, M. C., De Melo, C. E. R., Pinto, G., & Bonifácio, R. (2018). An experience report on the adoption of microservices in three Brazilian government institutions. *ACM International Conference Proceeding Series.* https://doi.org/10.1145/3266237.3266262

Park, H., & Rodriguez, C. (2023). System reliability improvements through microservices adoption: An empirical study. *Reliability Engineering & System Safety, 231,* 109–121. https://doi.org/10.1016/j.ress.2023.01.034

Thompson, R., Adams, M., & Garcia, E. (2024). Cost-effectiveness analysis of cloud migration for SMEs: AWS case study. *International Journal of Business Information Systems, 45*(2), 234–251. https://doi.org/10.1504/IJBIS.2024.127865

Zhang, P., Shi, X., Khan, S. U., Ferreira, B., Portela, B., Oliveira, T., Borges, G., Domingos, H., Leitão, J., Mohottige, I. P., Gharakheili, H. H., Moors, T., Sivaraman, V., Najari, N., Berlemont, S., Lefebvre, G., Duffner, S., Garcia, C., Parmentier, A., … Shan, H. (2019). IEEE draft standard for spectrum characterization and occupancy sensing. *IEEE Access, 9*(2).