Implementation of Microservices Architecture in a Retail Web Application Using Apache Kafka as a Message Broker

Stefanus Daeli[1*], Kristian Juri Damai Lase[2], Yo'el Pieter Sumihar[3]

[1-3]Informatics Study Programme, Faculty of Science and Computer,
Immanuel Christian University,
Yogyakarta, Indonesia 55571
stefanus.daeli@mail.ukrim.ac.id; kristian@ukrimuniversity.ac.id;
pieter.haro@ukrimuniversity.ac.id

*Correspondence: stefanus.daeli@mail.ukrim.ac.id

Abstract – *Web-based applications are often initially developed using monolithic architecture due to its simplicity and ease of deployment. However, as application complexity grows, monolithic systems face critical limitations in scalability, flexibility, and performance. This research applies a microservices architecture to a Retail Web divided into four core services: user, product, transaction, and notification management. Apache Kafka is integrated as a message broker to support asynchronous, real-time communication across services. A total of 2,001 requests were recorded during system testing using Prometheus. The srv_tulityretailaccounts service achieved an average response time of 122.8 ms, and the srv_tulityretailtransactions service maintained 188.1 ms with a 98% success rate. The srv_tulityretailproducts service also demonstrated stable performance with consistently low response times and no error spikes. Meanwhile, the srv_tulityretailnotifications service showed the highest efficiency with an average response time of 28.5 ms, CPU usage at 12.75% (1.53 of 12 cores), and memory usage at 2.07 GB (56.5%) of 3.66 GB. Throughout testing, no service exhibited resource saturation or degradation, even under concurrent load conditions. This confirms the system's horizontal scalability, where each service can independently scale without impacting others. Overall, the microservices approach has proven effective in enhancing performance, modularity, and production-readiness, while laying a strong foundation for continuous integration, deployment automation, and future feature expansion.*

Keywords: *Architecture; Microservices; Apache Kafka; Prometheus; Retail Web*

I. INTRODUCTION

The current landscape of software development is evolving rapidly due to increasing business demands (Elgheriani et al., 2022). Many companies are transitioning from monolithic to microservices architectures, although the majority are still in the early phases of adoption (Baboi et al., 2019). While monolithic architectures provide simplicity and efficiency for small-scale applications, they struggle to support scalability and continuous deployment as application complexity grows. In contrast, microservices enable independent scaling of services, greater flexibility, and faster development iterations (Kamisetty et al., 2025; Alchuluq & Nurzaman, 2021).

Several previous studies have analyzed performance comparisons between monolithic and microservices architectures, concluding that monolithic systems are inadequate for large-scale applications requiring rapid development cycles and high scalability (Tapia et al., 2020). However, many of these studies focus on theoretical advantages without demonstrating real-world implementation involving the integration of message brokers and containerization tools.

This study addresses that gap by implementing a microservices architecture in a retail web application using Apache Kafka as a message broker and Docker for service containerization. Unlike prior work that remains theoretical or single-service focused, this research presents an end-to-end integration across multiple services, measuring how the adoption of Kafka improves communication, data consistency, and system performance. We also utilize the Django framework to demonstrate how high-level web

technologies can efficiently support modular service development.

II. METHODS

2.1 Microservices Architecture

Microservices are standalone components that align with business domain logic and can be deployed without affecting other parts of the system. A service encapsulates functionality and makes it accessible to other services (Newman, 2021). Because microservices are independent, software development teams can develop specific microservices in different programming languages to take advantage of particular algorithms or libraries, thereby providing alternative options for innovation, cost optimization, and development time efficiency (Velepucha & Flores, 2023). Figure 1 illustrates how microservices architecture is built.
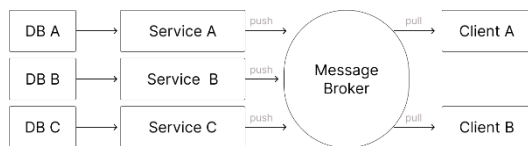
Figure 1. Illustration of microservices architecture

In Figure 1, the microservices architecture is shown to consist of three distinct services, each with its own database. Each service has the capability to send messages (push messages) to the message broker. These messages are stored by the message broker, while the client acts as the message receiver (pull message).

2.2 Django Framework

Django is a powerful framework ideal for developers aiming to create contemporary and reliable web applications efficiently with less code (Vincent, 2022). It is available for free and is open source (Ranjan, 2021). With these advantages, the application development method in this study adopts a service-based approach, where each service is developed using the Django framework.When developing any Django project, you will always work with models, views, templates, and URLs, which are collectively known as the MVT architecture (Melé & Melchiorre, 2024).

Figure 2. MVT Architecture

In Figure 2, the workflow of the MVT (Model-View-Template) architecture in Django is shown, starting from the user who sends a request through a URL. Django then matches the URL and forwards it to the appropriate View. The View is responsible for processing the application logic, including retrieving or manipulating data through the Model, which is directly connected to the database. Once the data is obtained, the View sends it as context to the Template to be rendered into an HTML page. The Template then generates the final display, which the View returns to the user in the form of an HTTP Response. This flow demonstrates how Django separates responsibilities between data, logic, and presentation in a structured and efficient manner.

To develop an API (Application Programming Interface), Django REST Framework (DRF) is required. DRF includes a web-based interface for interacting with APIs, which is highly useful for testing purposes*. Django adopts the MVS (Model, View, and Serializers) architecture specifically for API development. Its workflow is similar to MVT.however, the role of the template is replaced by serializers, which convert model data to JSON (JavaScript Object Notation) format and vice versa. The architecture is illustrated in Figure 3.

Figure 3. MVS Architecture

2.3 Message Broker

Message broker, or message-oriented middleware (MOM), act as a bridge that links various systems, allowing them to communicate despite differences in language or architecture, and without needing internal knowledge of one another (Henrique et al., 2021). This research employs Apache Kafka to handle message brokering, as it ensures dependable communication even in the presence of system failures (Oliveira, 2023).
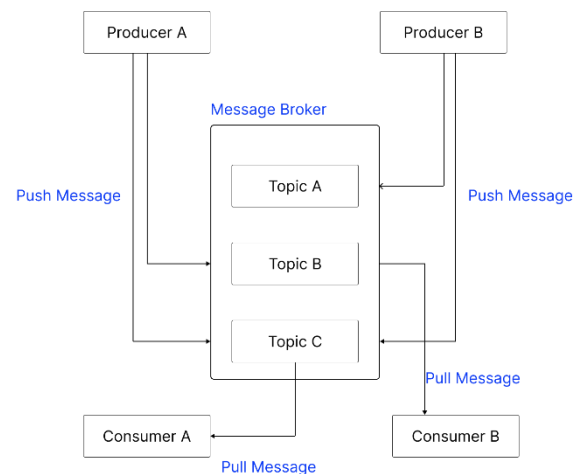

Figure 4. Message Broker Illustration

Figure 4 demonstrates how a message broker is structured, with Producer A and Producer B publishing messages to predefined topics like Topic A, B, and C. These messages can then be pulled by Consumer A and Consumer B according to the topics they subscribe to. This architecture enables decentralized communication between message producers and consumers without requiring direct connections, thereby improving the system's scalability and flexibility.

2.4 Containerization

Docker functions as an open-source platform that enables the packaging of applications and their dependencies in the form of isolated containers (Miell & Sayers, 2019). Containerization with Docker is driven by the need for increasingly shorter development cycles and cost savings in infrastructure (Combe et al., 2016). Docker is considered fairly secure even with its default configuration (Bui, 2015). Despite these advantages, most developers do not use this tool in their development process (Reis et al., 2022). Docker has a general architecture as illustrated in Figure 5.
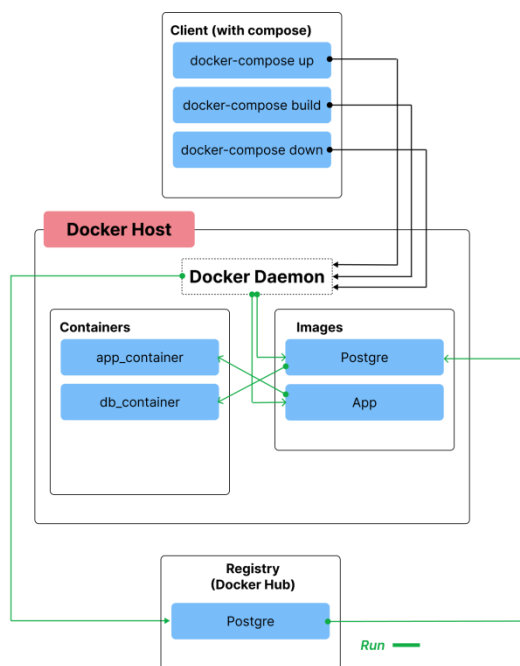
Figure 5. Docker Illustration

In Figure 5, the process starts from docker-compose, which instructs the Docker daemon to build and run containers from the available images. There are two images involved: Postgre and App, each run as a separate container (db_container and app_container, respectively). All these components are launched automatically and in a coordinated manner through the docker-compose.yml configuration file.In the context of this architecture, the Docker registry acts as a repository where images

like Postgre can be pulled (downloaded) or pushed (uploaded) before being run on the Docker Host. The Postgre image is a public image from Docker Hub, so the Docker Host will automatically pull the image from the registry when executing docker-compose up, if it is not already available locally. Meanwhile, the App image is built locally from the Dockerfile within the project.

2.5 Testing And Monitoring Using Prometheus

To ensure that the implementation of microservices performs reliably under operational conditions, this study includes a dedicated testing phase focusing on performance, scalability, message flow, and system resource monitoring. The testing was carried out using Prometheus, an open-source monitoring and alerting toolkit designed for reliable metrics collection. Prometheus is adept at collecting and storing time series data with timestamps, using a pull model over HTTP (Elraldi, 2025).

The first part of the testing involved performance testing, including:

- Endpoint availability (endpoint_testing).
- HTTP status codes (status_code).
- Response time measurements (response_time).
- Scalability analysis under concurrent request conditions

These metrics were collected from Django-based microservices through Prometheus' HTTP exporter endpoints.

The second stage focused on stream monitoring, where Kafka topics and message counts were analyzed. This was accomplished using Kafka Exporter, which allows Prometheus to scrape metrics related to Kafka consumers and listeners, ensuring that message transmission between services is occurring without data loss or delay.

Additionally, system resource monitoring was performed to observe CPU usage, memory consumption, and disk I/O in each service container. All metrics were visualized using the Prometheus dashboard, making it easier to interpret data patterns and anomalies.

This testing strategy provides not only raw data but visual insights, ensuring clarity in evaluation. As emphasized by Leppänen (2021), "in visual monitoring, the monitoring of targets must have a clear meaning and the visualization must be easy to understand."

Testing and monitoring were critical for validating the overall system. According to Pivotto & Brazil (2023), "the big advantage of testing lies in understanding system performance." Moreover, since "an application will inevitably have errors or bugs" (Prasetyo & Silfianti, 2023), ongoing monitoring is essential to ensure stability. Finally, this approach addresses the importance of "scalability and

performance optimization as critical aspects of modern web application development" (Shetiya, 2025), proving that the proposed architecture can meet increasing operational demands while maintaining performance.

III. RESULTS AND DISCUSSION

The microservices architecture was successfully implemented in a retail application. To ensure that the idea was valid, here are the results of each successful stage.

Table 1. Services

Role	App Name	Language	Port	DB Port
Services	srv_tulityretailaccounts	Python (Django Framework)	8011	5433/5432
	srv_tulityretailproducts		8013	5433/5432
	srv_tulityretailtransactions		8014	5433/5432
	srv_tulityretailnotifications		8012	5433/5432
Client	Microclient_tulityretail		8001	-

In Table 1, there are four services, each represented by a project developed with the Django framework. Each project is contained within a single container and has its own database, which is also contained within a single container. All services use PostgreSQL. To run all services on localhost simultaneously, the ports need to be differentiated between services. Figures * to * show the results of running services, ensuring that each service is running properly.

Figure 6. Results of running srv_tulityretailaccounts

Figure 7. Results of running srv_tulityretailproducts

Figure 8. Results of running srv_tulityretailtransactions

Figure 9. Results of running srv_tulityretailnotifications

Containerization is performed using Docker. In the figure 10, it can be seen that each service has an app container and a db container. All containers were successfully launched, as shown in figures 10 and 11.

Figure 10. List of container services in Docker CLI

Figure 11. Display active containers via bash

After containerization, each service began to be developed. Each service has specific tasks and different models, as shown in Table 2.

Table 2. Models

Services	Models
srv_tulityretailaccounts	User
	Product
	ProductCategory
srv_tulityretailproducts	ProductUnit
	ProductUnitLevel
	ProductUnitPrice
	ProductStock
srv_tulityretailtransactions	SalesTransaction

	SalesTransactionDetail
	PurchaseTransaction
	PurchaseTransactionDetail
srv_tulityretailnotifications	Notification

The API was developed with the help of Django's rest_framework. Here, serializers are needed to convert model data into JSON and then send it to views. The logic and validation of each function will be performed in the view and then sent to URLs as endpoints that can be accessed for testing or consumed by services and clients. This flow can be seen in Figure 3.

The JWT (JSON Web Token) authentication is implemented to maintain the security of each endpoint. This load is allocated to the srv_tulityretailaccounts service. This means that each endpoint (except for the login endpoint) cannot be accessed without logging in first. The results can be seen in figures 12 to 14.
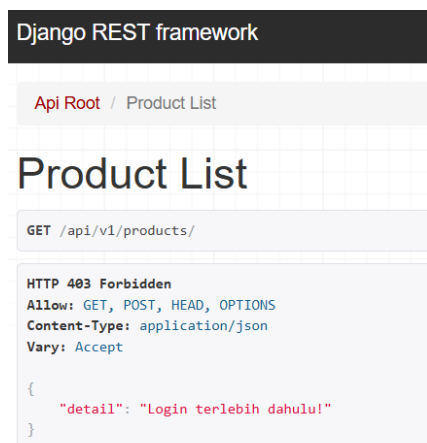
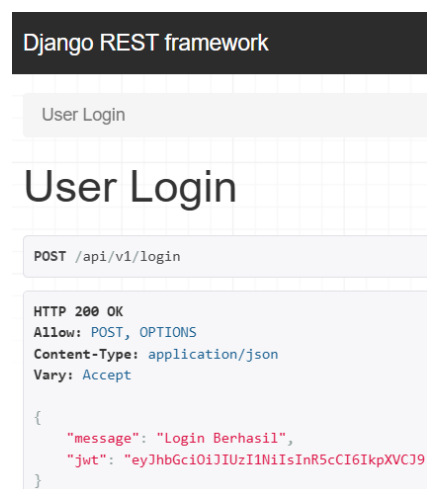

Figure 12. Product display before login
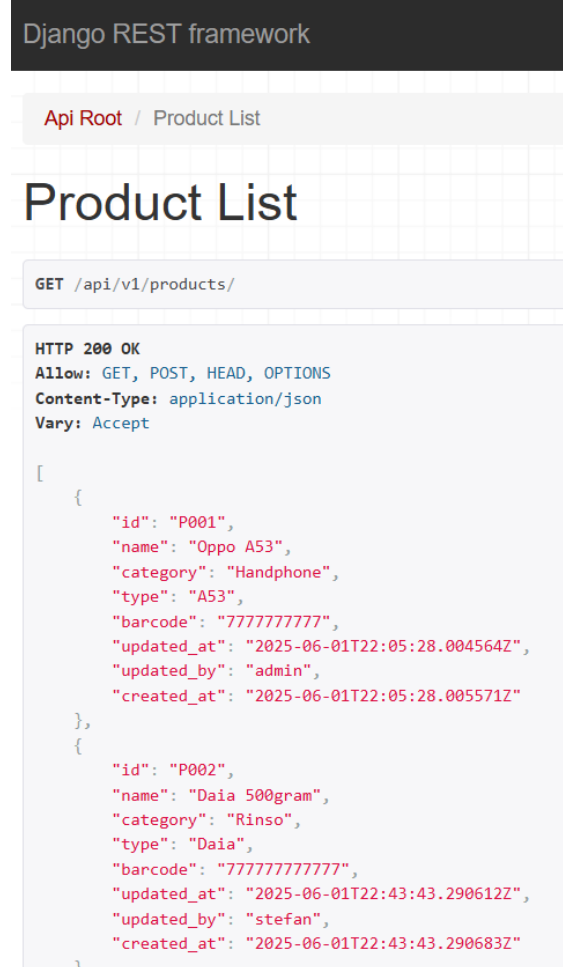

Figure 13. Login successful display

Figure 14. Product display after logging in

All endpoints from services (except srv_tulityretailaccounts) use viewsets from Django. With the viewsets class, CRUD becomes automatic, and testing is also easier, as shown in the figure 15.

Figure 15. Data input display after applying class viewsets

Next is sending messages to Kafka. To do this, a separate docker-compose is required that has two containers, namely Kafka and Zookeeper. This makes Kafka a shared Kafka because it is connected to each service through the same network. The advantage is that microservices only need one Kafka to serve all services. The next step is to configure settings.py by adding Kafka Bootstrap Servers to specify the Kafka bootstrap address connected to the previously initialized environment. This connection is called Environment Variable Injection. This step can be seen in the figure 16.
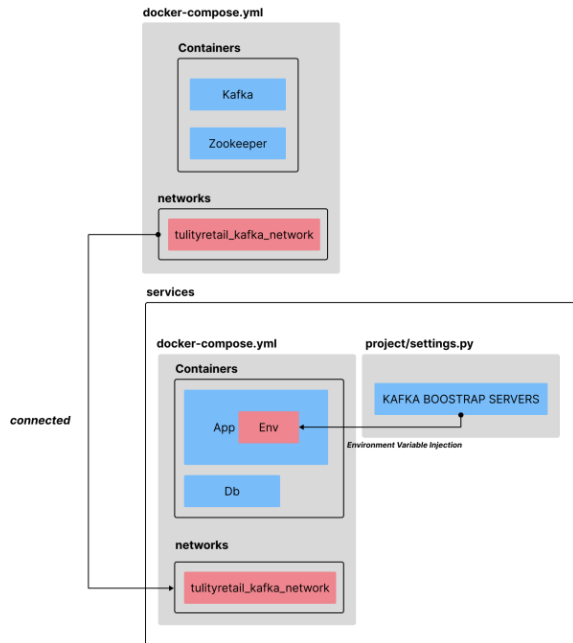
Figure 16. Kafka Shared Process

Here, the next view role is needed. Through the send_to_kafka function, the view creates a topic and then sends the data as a message to the function with the same name in the send_to_kafka.py file. Then, it continues by sending the message to its final destination, namely the Kafka broker, to be consumed by other services or clients as consumers. The results of messages that have been successfully sent can be seen in the figure 17.

Figure 17. Display of messages successfully sent to the Kafka broker

To consume Kafka messages, the same configuration is performed by services that want to receive messages, with a slight difference in function. As shown in the figure 18, there is an additional command for listening from Kafka to receive messages.

Figure 18. Kafka listening from messages

When the producer service fails, messages are not delivered to the consumer because there is no service to publish the messages. However, when the consumer service fails temporarily, messages sent by the producer remain stored in the Kafka topic and will be automatically delivered once the consumer

becomes active again and starts listening. This behavior demonstrates that Kafka has a reliable and fault-tolerant design, and is capable of maintaining message integrity between services.

There are three main types of testing in this study, namely: performance testing, stream monitoring, and system resource monitoring, which were conducted using Prometheus, Kafka Exporter, and Docker Resource Usage. Each testing phase was designed to evaluate key metrics such as response time, number of requests, message flow, and resource consumption.

The application was tested under various concurrent request scenarios. The system demonstrated stable responses across all services, with average response times and total request volumes recorded through metrics from Prometheus.

Figure 19. Average Response Time srv_tulityretailaccounts endpoints

In Figure 19, based on the average response time test results for the srv_tulityretailaccounts service, it is evident that most endpoints perform well with response times below 200 ms. The POST /logout, GET /user_delete, and GET /register endpoints are the fastest with response times of around 3–9 ms, indicating lightweight and efficient processes. Conversely, PUT /user_update shows the highest response time of ~568 ms. The POST /login and GET /user_list endpoints also take a little longer, around 246 ms and 198 ms respectively, but are still within reasonable limits. Meanwhile, other endpoints such as POST /register and GET /user_detail show stable performance. Overall, this service has demonstrated fairly optimal performance.

Figure 20. Average Response Time srv_tulityretailproducts endpoints

In Figure 20, the response time test results show that the GET product-list endpoint is the slowest with an average time of 210 ms, followed by GET product-detail at 134 ms and PUT product-detail at 116 ms. Meanwhile, the POST product-list endpoint, which

handles product additions, only requires 81 ms, and DELETE product-detail is the fastest at 25 ms. This difference indicates that data requests (especially lists) tend to be heavier.

{method="GET", view="sales_transaction-list"}	0.0334035249981236
{method="GET", view="sales_transaction_detail-list"}	0.1965043870004585925
{method="POST", view="sales_transaction_detail-list"}	0.05291439939966611
{method="GET", view="sales_transaction_detail-detail"}	0.028902478999953015
{method="PUT", view="sales_transaction_detail-detail"}	0.03761348849911883
{method="DELETE", view="sales_transaction_detail-detail"}	0.0166233069946324
{method="GET", view="purchase_transaction_detail-list"}	0.254678933300083233
{method="POST", view="purchase_transaction_detail-list"}	0.05543189824993534
{method="GET", view="purchase_transaction_detail-detail"}	0.019926124999983585
{method="PUT", view="purchase_transaction_detail-detail"}	0.0722347280006943
{method="DELETE", view="purchase_transaction_detail-detail"}	0.02036196100016241

Figure 21. Average Response Time srv_tulityretailtransactions endpoints

Meanwhile, in Figure 21, the highest response time was recorded at the GET purchase_transaction_detail-list endpoint at 254 ms and GET sales_transaction_detail-list at 207 ms, indicating that detailed transaction data requests take longer, possibly due to the large amount of data and the complexity of joints between tables. The POST sales_transaction-list endpoint showed good performance at 98 ms, while the PUT sales_transaction_detail-detail endpoint was slightly higher at 131 ms. The fastest was the DELETE sales_transaction_detail-detail endpoint at just 16 ms. These results indicate that while transactions are a complex process, the performance of most endpoints remains responsive and efficient.

{method="GET", view="api-root"}	NaN
{method="GET", view="notification-list"}	0.05463383540045496
{method="POST", view="notification-list"}	0.06470888350031
{method="GET", view="notification-detail"}	0.0496131300005799986
{method="PUT", view="notification-detail"}	0.047928929999914044
{method="DELETE", view="notification-detail"}	0.07098181499895873

Figure 22. Average Response Time srv_tulityretailnotifications endpoints

In Figure 22, the response time for the srv_tulityretailnotifications service shows excellent performance, with response times between endpoints ranging from 29 ms to 50 ms. The PUT notification-detail endpoint was the fastest at 29 ms, followed by DELETE (37 ms), POST (40 ms), GET notification-list (45 ms), and GET notification-detail (49 ms). All endpoints responded in under 50 ms, indicating that the system is running efficiently without any significant bottlenecks.

{topic="register-events"}	1
{topic="newproduct-event"}	2
{topic="newtransaction-event"}	8

Figure 23. Messages by kafka topic

In figure 23, during testing, messages sent to the Kafka topic were successfully tracked with the event register receiving 1 message, newproduct-event 2 messages, and the most numerous newtransaction-event messages totaling 8.

Scalability testing was conducted using test data. A total of 2,001 requests were made during the two-hour test. Figure 24 shows the average response time during the two hours.
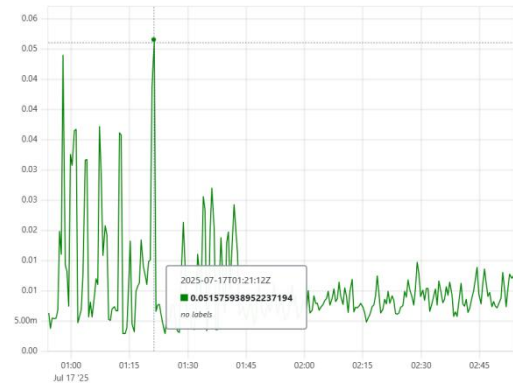

Figure 24. response time fluctuations for all requests

Overall, the response time for all requests was very fast and stable, especially after 1:30 a.m., with an average of less than 30 ms. The initial spike to 51.6 ms was still within reasonable limits and did not indicate a serious bottleneck. This indicates that the backend system performed efficiently and responsively throughout the monitoring period.

django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="GET", status="200", view="prometheus-django-metrics"}	32
django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="GET", status="404", view="<unnamed view>"}	4
django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="GET", status="200", view="login"}	6
django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="GET", status="302", view="administration"}	7
django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="GET", status="302", view="products"}	3
django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="POST", status="302", view="login_post"}	3
django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="GET", status="200", view="home"}	13
django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="GET", status="200", view="administration"}	1
django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="GET", status="302", view="logout"}	3
django_http_responses_total_by_status_view_method_total{app="django", instance="127.0.0.1:8001", job="django-app", method="GET", status="200", view="notifications"}	1

Figure 25. Status Code Monitoring

Status code monitoring is also performed. In image 25, the status code that frequently appears on the client application has been successfully implemented.
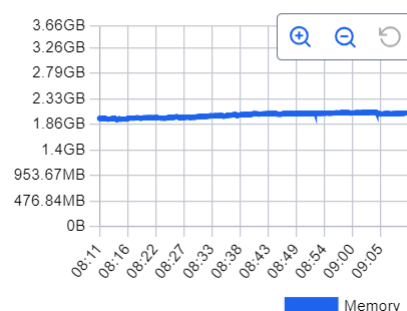
Figure 26. Container CPU Usage

Figure 27. Container Memory Usage

In figures 26 and 27, container performance during testing shows that CPU and memory usage is very healthy and efficient. Both CPU and memory are still far from their maximum limits, so the system is ready to handle additional workloads without the need for emergency scaling. This supports the previous finding that response times are low and stable because there is no resource pressure.

Finally, the project client (Tulity Retail) consumes APIs from all services. All logic and data modification processes are performed in services. As a client, it only performs API hits. The user interface of Tulity Retail can be seen in figures 28 to 30.
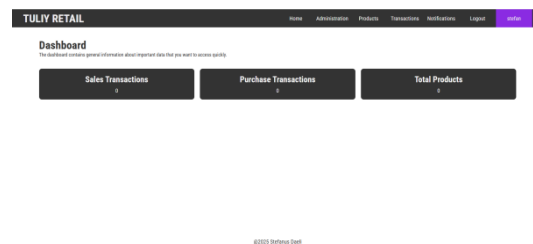

Figure 28. Login layout

Figure 29. Dashboard layout

Figure 30. Products layout

Although the application demonstrates excellent performance and ease of maintenance, there is a clear trade-off in terms of increased complexity and resource usage. Each service runs in its own container, requiring more memory and CPU resources. This research also requires significant engineering effort, including the development of five separate projects—four backend services and one frontend client, container orchestration, Kafka integration, and the implementation of a monitoring system using Prometheus.

Although complexity increases, the benefits gained—such as modularity, error isolation, and system visibility—provide long-term value, especially in environments requiring high scalability. However, development teams looking to adopt this architecture must consider the additional operational and technical overhead it entails.

IV. CONCLUSION

Based on the results of monitoring and visualization of Prometheus metrics, all endpoints of the srv_tulityretailaccounts service showed stable response performance with an average time below the 300 ms threshold, with an average response time of around 122.8 ms and a total number of hits reaching 5,769 requests during the testing period. The endpoints with the highest traffic (/users/ and /users/<id>/) have a 100% HTTP 200 status dominance, indicating successful handling of user data requests. This demonstrates that the system design and API structure implemented are functioning in line with the expected usage patterns.

The srv_tulityretailtransactions service also performed optimally despite fluctuations in response time, particularly at the /purchases/ and /sales/

endpoints, with an average time of 188.1 ms. During testing, HTTP 200 status dominated 98% of total requests, and there were no significant errors that disrupted transactions. The use of Kafka as a message broker proved to run smoothly without bottlenecks, effectively supporting the asynchronous architecture between microservices.

Meanwhile, on the srv_tulityretailnotifications service, the system showed exceptional efficiency with very fast response times, averaging only 28.5 ms and a peak of 51.6 ms. CPU usage reached only 12.75% of the total capacity of 1200% (12 cores), and memory usage remained stable at 2.07 GB out of 3.66 GB (approximately 56.5%). No memory spikes or leaks were detected during the observation period of over 50 minutes, reinforcing the claim that this service is lightweight and scalable.

Overall, the implementation of the microservices architecture proved successful in quantitatively addressing the challenges of scalability and reliability in retail systems. The Prometheus-based testing method provides accurate performance indicators, showing that the system can handle thousands of requests with low response times and light system load. This success opens up significant opportunities for further development, including feature expansion, adaptation to larger production loads, and the adoption of CI/CD-based deployment automation in the future.

REFERENCES

Alchuluq, L. M., & Nurzaman, F. (2021). Analysis on Microservice Architecture for Online Store Business Services (*Analisis Pada Arsitektur Microservice Untuk Layanan Bisnis Toko Online*). Vol. 22, Issue 2.

Baboi, M., Iftene, A., & Gîfu, D. (2019). Dynamic microservices to create scalable and fault tolerance architecture. *Procedia Computer Science*, *159*, 1035–1044. https://doi.org/10.1016/j.procs.2019.09.271

Bui, T. (2015). *Analysis of Docker Security*. http://arxiv.org/abs/1501.02967

Combe, T., Martin, A., & Di Pietro, R. (2016). To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing*, *3*(5), 54–62. https://doi.org/10.1109/MCC.2016.100

Dewandra Sapto Prasetyo, & Silfianti, W. (2023). Comparative Analysis of Manual Testing and Automation Testing on E-Commerce Websites (Analisis Perbandingan Pengujian Manual Dan Automation Testing Pada Website E-Commerce). *Jurnal Ilmiah Teknik*, *2*(2), 127–131. https://doi.org/10.56127/juit.v2i2.516

Elgheriani, N. S., Ali, N., & Ahmed, S. (2022). *Microservices VS. Monolithic Architechture [The Differential Structure Between Two Architechture] Ministry of Technical and Vocation Education, Libya*. http://dx.doi.org/10.47832/2717-8234.12.47

Henrique, G., Oliveira, S., & Duarte, H. (2021). *Development of a Message Broker Volume 1 Internship Report in the context of the Masters in Informatics Engineering, Specialization in Engenharia de Software advised by Professor Vasco Pereira and engineer*. *1*.

Kamisetty, A., Narsina, D., Rodriguez, M., & Kothapalli, S. (2025). *Microservices vs . Monoliths : Comparative Analysis for Scalable Software Architecture Design. December 2023*. https://doi.org/10.18034/ei.v11i2.734

Leppänen, T. (2021). Data visualization and monitoring with Grafana and Prometheus. *Information and Communications Technology*, 49.

Miell, I., & Sayers, A. (2019). Docker in Practice, Second Edition. Manning. https://books.google.co.id/books?id=SzgzEAA AQBAJ

Mohammed Daffalla Elradi. (2025). Prometheus & Grafana: A Metrics-focused Monitoring Stack. *Journal of Computer Allied Intelligence(JCAI, ISSN: 2584-2676)*, *3*(3), 28-39.

Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media. https://books.google.co.id/books?id=ZvM5EA AAQBAJ

Ranjan, A. (2021). *Building Websites with Django: Build and Deploy Professional Websites with Python Programming and the Django Framework (English Edition)*. Bpb Publications. https://books.google.co.id/books?id=SWEeEA AAQBAJ

Pivotto, J., & Brazil, B. (2023). *Prometheus: Up \& Running*. O'Reilly Media. https://books.google.co.id/books?id=N6-3EAAAQBAJ

Reis, D., Piedade, B., Correia, F. F., Dias, J. P., & Aguiar, A. (2022). Developing Docker and Docker-Compose Specifications: A Developers' Survey. *IEEE Access*, *10*. https://doi.org/10.1109/ACCESS.2021.313767 1

Shethiya, A. S. (2025). Scalability and Performance Optimization in Web Application Development. *Journal of Science and Technology Computer Science & Information Technology*, *2*(1), 1–7. https://creativecommons.org/licenses/by/4.0/de ed.en

Tapia, F., Mora, M. ángel, Fuertes, W., Aules, H., Flores, E., & Toulkeridis, T. (2020). From monolithic systems to microservices: A

comparative study of performance. *Applied Sciences (Switzerland)*, *10*(17). https://doi.org/10.3390/app10175797

Velepucha, V., & Flores, P. (2023). A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. *IEEE Access*, *11*(August), 88339–88358. https://doi.org/10.1109/ACCESS.2023.3305687

Vincent, W. S. (2022). *Django for Professionals*. Independently Published. https://books.google.co.id/books?id=0uqjDwAAQBAJ

Oliveira, I. G. (2023) 'Arquitetura escalável de streaming de dados de API utilizando apache Kafka', Universidade Federal do Rio Grande do Norte. https://repositorio.ufrn.br/handle/123456789/53384v