Jurnal Teknik Informatika (JUTIF)
Vol. 5, No. 6, December 2024, pp. 1861-1872

DOI: https://doi.org/10.52436/1.jutif.2024.5.6.2617
p-ISSN: 2723-3863
e-ISSN: 2723-3871

ANALYSIS THE IMPACT OF REFACTORING FROM MONOLITHIC APPLICATIONS TO MICROSERVICES ON RESPONSE TIME USING THE MDA AND SCA APPROACHES

Shidqi Fadhlurrahman Yusri*[1], Dawam Dwi Jatmiko Suwawi[2], Monterico Adrian[3]

[1,2,3]Informatics, School of Computing, Telkom University, Indonesia
Email: [1]shidqify@student.telkomuniversity.ac.id, [2]dawamdjs@telkomuniversity.ac.id, [3]monterico@telkomuniversity.ac.id

(Article received: August 06, 2024; Revision: August 21, 2024; published: December 29, 2024)

Abstract

This study investigates the impact of refactoring from a monolithic to a microservices architecture on application response time. Monolithic architecture, initially chosen for ease of development, faces scalability challenges as the application grows. Microservices offer a solution by enabling independent service deployment and enhanced scalability. This research uses Meta-Data Aided (MDA) and Static Code Analysis (SCA) methodologies to facilitate the refactoring process, applying them to the inventory-application project from a collaborative software development platform (GitHub). The refactoring involves decomposing the monolithic application, containerizing it with Docker, and evaluating performance using JMeter. Results show that microservices significantly reduce response time, particularly in API interaction tasks. While microservices improve scalability and flexibility, they require careful management of service communication. This research enhances understanding of the benefits of microservices in terms of response time and offers practical guidance for developers considering refactoring.

Keywords: *Microservices, Meta-data Aided, Monolithic, Refactor, Response Time, Static Code Analysis.*

1. INTRODUCTION

Monolithic architecture is the first choice for software developers when building their applications. This choice is not without reason, as it offers ease in development, testing, and deployment, which can be managed through a single deployment [1], [2]. However, this convenience cannot be maintained as the application evolves, leading to increasing challenges in its maintenance. Issues such as scalability requirements, complex code structures, and the growing size of the application contribute to longer deployment times, which become significant obstacles in maintenance [1], [3], [4].

Given the above reasons, microservices present a solution to these issues. Microservices offer advantages such as better scalability, services focused on specific functionalities, and independent service deployments [5], [6], [7]. However, despite these benefits, the challenges remain, particularly in transforming an application from a monolithic structure to a microservices architecture. Refactoring is one of the approaches that can be used to address this challenge.

Refactoring is a common practice among software developers to improve the quality of their software. The focus is on enhancing the quality and maintainability of the software without altering its functionality [8], [9]. This approach is adopted because software continuously evolves to meet new requirements, enhance existing features, and address existing shortcomings [8]. Although refactoring is a practical method for structural changes and quality improvements in applications, it is a complex process for both developers and companies [5]. This complexity arises from the need to consider various aspects, particularly regarding software performance, such as response time.

In the ISO/IEC 25010:2011 standard [10], which outlines software quality, one of the measurement aspects is Time Behavior, with response time being a component of this aspect. Response time represents the wait time for results processed by the application. This wait time is the duration taken by data from when the client sends a request until the client receives the data back from the server [11], [12]. The longer the time required, the worse the user experience becomes. Furthermore, Khan R. [11] explains that response time is a parameter used to evaluate the performance and efficiency of an application.

To ensure that software performance remains optimal after the refactoring process, particularly regarding response time, developers need to select the appropriate approaches for refactoring. Several refactoring approaches are available, including Meta-data Aided (MDA) and Static Code Analysis (SCA) [1], [4]. MDA is an approach that analyzes applications based on available data sources such as diagrams, application descriptions, system specifications, and other documents [13]. In contrast, SCA is an approach that uses source code data from the application as input for analysis [1], [14].

Research on refactoring monolithic applications to microservices has been discussed in several journals [5], [6], [8], [15]. Goncalves N. et al. [15] highlight the challenges of refactoring and its impact on performance, showing variations in latency results between monolithic and microservices architectures. Traini L. et al. [8] examine the impact of refactoring on execution time, finding that approximately 55% of commits affect performance, with 75% showing no significant change. Ren Q. et al. [5] discuss the stages and challenges of refactoring as well as the benefits of microservices architecture in terms of resilience and management. This research emphasizes the importance of selecting the appropriate method for refactoring to improve application performance. Zaragoza [6], in his study, explores the migration process from monolithic software systems to microservices architecture (MSA). He emphasizes the benefits of MSA, such as improved maintainability, better scalability, and faster deployment. This migration consists of two main phases: first, building the microservices architecture from the existing monolithic source code, and second, transforming the code into microservices that adhere to MSA principles.

This study proposes a new method using transformation patterns to support this transition without compromising business logic and application performance. Zaragoza [6] also addresses the complexities that arise, particularly concerning class dependencies, which often disrupt effective microservices encapsulation. By utilizing an automated tool called MonoToMicro, this approach was tested in the context of monolithic Java system applications. The method aims to simplify the identification and grouping of classes into potential microservices, thus addressing challenges in code transformation and ensuring that the operational characteristics of MSA can be met.

Thus, this research will focus on a thorough exploration of the response time effects resulting from refactoring monolithic applications into microservices applications using the MDA and SCA approaches. Testing will be conducted on both architectures with the same test cases and environment, ensuring a fair comparison between them. This study aims to provide developers with greater insight into choosing the most appropriate architecture for their application development and the approaches to be used.

2. RESEARCH METHODS

The methodology used in this study generally includes project search, decomposition or separation of the monolithic application, construction of the microservices application, and testing as described in Figure 1.

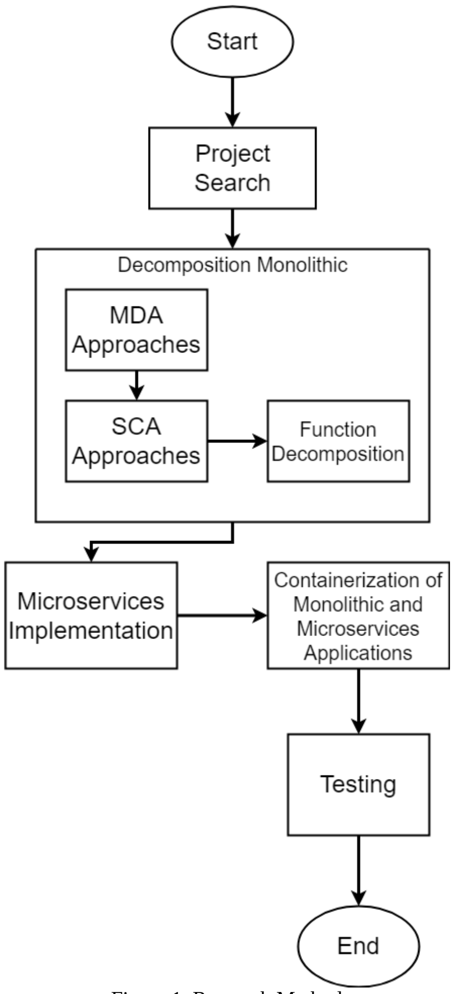

Figure 1. Research Method

The research begins with a project search on collaborative software development platforms such as GitHub. Once a suitable project is identified based on the research criteria, monolithic decomposition will be performed to separate modules that have no dependencies [15]. The monolithic decomposition process utilizes two approaches: Meta-data Aided (MDA) and Static Code Analysis (SCA). After applying and analyzing the MDA and SCA approaches, it is necessary to separate the related functions based on the results from MDA and SCA. Once the decomposition process is complete, the next step is the implementation of microservices according to the decomposition results. During the microservices implementation process, adjustments to the code will be made to facilitate communication between functions within the application and those in other APIs. After implementing the microservices, the application will be containerized to enable API communication. Containerization will also be applied to the monolithic application, ensuring that the development environment for the monolithic application is consistent with that of the microservices application. The final process to be conducted is testing, which will compare the response time performance results between the monolithic and microservices applications.

2.1. Project Search

The project used in this research will be selected through a search on GitHub. GitHub is a platform that hosts many open-source projects across various domains [16]. The chosen project will use the JavaScript programming language, as microservices are generally associated with lightweight programming languages like JavaScript and have proven effective in practice[17]. Additionally, the selected project must have related artifact documents within its GitHub repository. These documents are used for implementing the MDA approach, which requires artifacts such as diagrams and application specifications.

From the researcher's search for applications based on the mentioned criteria, one suitable repository was found: the inventory-application developed by Mohamed Eleshmawy [18]. This application is designed to track order status from the moment a customer places an order until the order is marked as shipped by the seller and then as received by the recipient. The project uses JavaScript as its programming language and includes an ERD (Entity Relationship Diagram) and Functional Requirements documents to explain the application's specifications.

2.2. Decomposition Monolithic

Monolithic decomposition is the stage of separating modules that have no dependencies [15]. In this phase, the researcher will use two approaches: Meta-data Aided (MDA) and Static Code Analysis (SCA). For the MDA approach, the researcher will use the artifact documents available in the GitHub repository of the selected project, including the Entity Relationship Diagram (ERD) and the application specifications. For the SCA approach, the researcher will identify functions within the source code and then scan the source code using SonarQube to identify code duplications.

2.2.1. MDA Approach

Meta-Data Aided (MDA) is an approach to refactoring applications that utilizes common sources such as diagrams, specifications, and application descriptions [3]. Documents such as the ERD and functional requirements are available for use in the selected application. In addition to these documents, the researcher also includes a Use Case diagram in accordance with the requirements found in the GitHub repository. The ERD and Use Case for this application are described in Figure 2 *Entity Relation Diagram* dan Figure 3 *Use Case* Diagram.

It can be concluded from the diagram above that the application can be divided based on user roles. The division of functions and services is based on the use cases performed by each type of user. For example, the seller service can only execute functions specific to sellers, and the shipper service can only perform functions relevant to shippers. Although the separation of functions and services has been formulated, there is still one more approach that needs to be undertaken before finalizing the service division.

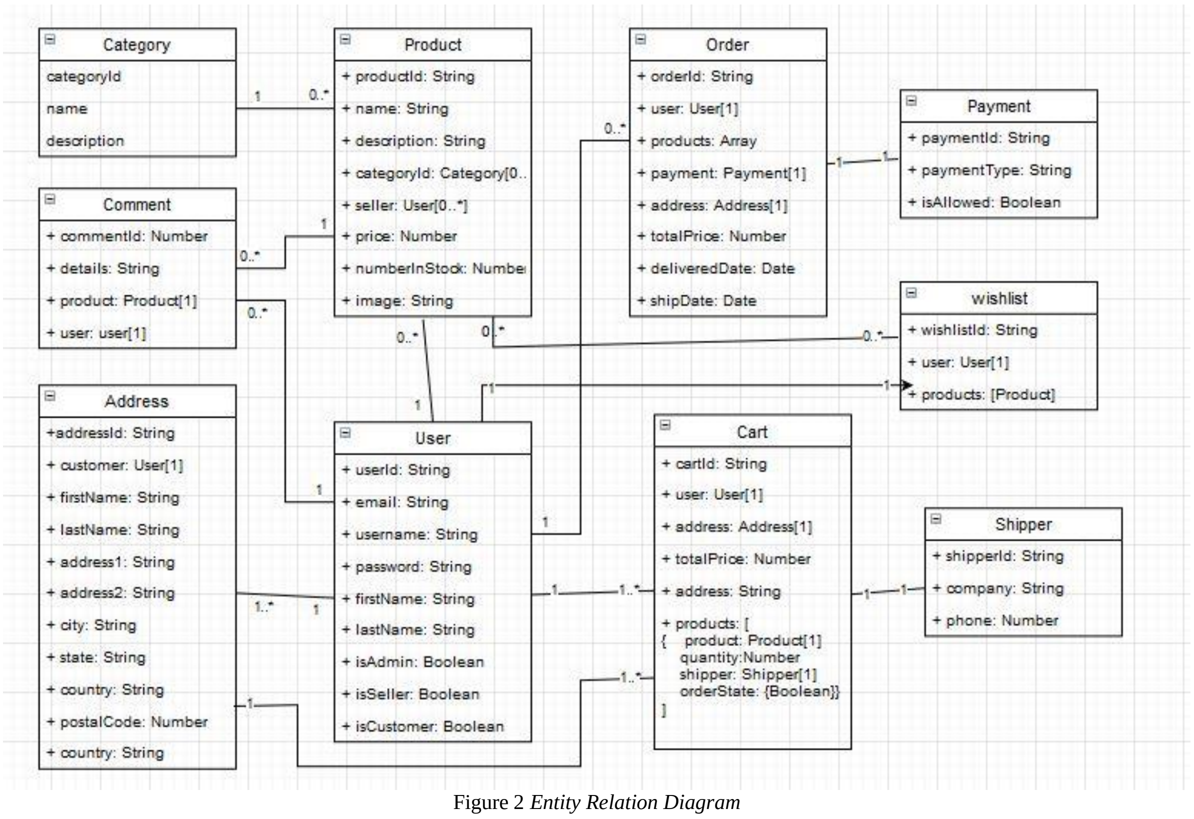

Figure 2 *Entity Relation Diagram*

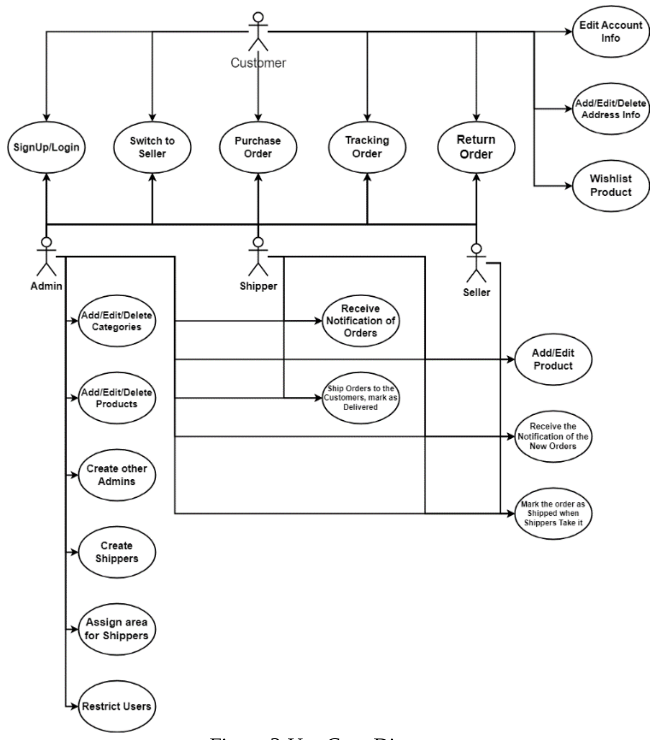

Figure 3 *Use Case* Diagram

2.2.2. SCA Approach

The Static Code Analysis (SCA) approach involves taking source code data from the application as input for analysis [1], [14]. Therefore, the researcher uses the source code from the inventory application for analysis. The analysis process involves scanning the source code with SonarQube. SonarQube is an application that helps developers analyze the source code of an application [19].

Based on the scan results, categories that need attention are bugs and duplication. Most of the detected bugs originate from CSS code used for styling, which does not significantly impact the application's performance. Regarding duplication, approximately 56.4% is found in the Validation folder, which contains files for data validation. This can be disregarded as the duplicated code is used repeatedly by design. Additionally, there is 14% code duplication in the controller folder, with 21.9% coming from cartController.js and 37% from userController.js. The duplicated code in cartController.js involves a block that is responsible for updating the total price in the cart. This can be resolved by creating a function updateTotalCartPrice(), which can be called by other functions that need to update the total price. Similarly, in userController.js, the duplicated code pertains to generating a hashed password. This can be addressed by encapsulating the block of code into a new function named genHashedPassword().

Table 1 List of Controllers from the Monolithic Application

Module	Method
Address	Post : AddAddress
	Put : EditAddress
	Delete : DeleteAddress
	Get : GetAddress
Cart	Put : addToCart
	Get : userCartInfo
	Get : removeFromCart
	Put : changeQuantityFromCart
	Put : chooseOrderAddress
Category	Get : categoryIndex
	Post : createCategory
	Get : categoryDetails
	Delete : deleteCategory
	Put : updateCategory
Order	Get : orderSuccess
	Get : userOrderHistory
	Get : ordersToShip
	Get : shippedOrder
	Get : markAsShipped
	Get : ordersToDeliver
	Get : markAsDelivered
Permission	Get : getAllUsers
	Get : getAllShipers
	Put : addShipper
	Put : addShiperInfo
	Put : addAdmin
	Put : restrictUser
Product	Get : allProducts
	Get : userProducts
	Post : createProduct
	Get : productDetails
	Delete : deleteProduct
	Post : updateProduct
User	Post : createUser
	Post : login
	Get : getUser
	Put : editUser
Wishlist	Get : addToWishlist
	Get : userWishlist
	Get : removeFromWishlist

In addition to scanning with SonarQube, the analysis process will also review the usage of each function within the application. This will assist in separating the dependent modules.

2.2.3. Function Decomposition

After the analysis process is conducted using the two previous approaches, the separation of functions can be carried out based on the analysis results obtained from the previous steps. The separation of functions and services is based on the analysis results using the MDA and SCA approaches, which have helped identify modules that can be separated and optimized. With this separation, it is hoped that each service can operate independently and efficiently, supporting further development of the application towards a microservices architecture. This process is crucial to ensure that each function operates with minimal load and reduces dependencies between services, ultimately improving the overall performance of the application.

2.3. Microservices Implementation

The implementation of microservices is a key stage in refactoring a monolithic application. At this stage, the design patterns used will be the same as those used in the original application, ensuring that no design pattern factors will alter the application's performance. Below are the design patterns that will be employed in the microservices application.

```
Services
├── controllers/
│   └── ...
├── middlewares/
│   ├── validation/
│   └── ...
├── models/
│   └── ...
├── routes/
│   └── ...
├── index.js
└── package*.json
```

```
Views
├── public/
│   └── ...
├── src/
│   ├── assets/
│   │   └── ...
│   ├── Component/
│   │   ├── account-settings/
│   │   │   └── ...
│   │   ├── cart/
│   │   │   └── ...
│   │   ├── dashboard/
│   │   │   └── ...
│   │   ├── home-page/
│   │   │   └── ...
│   │   ├── login&signup/
│   │   │   └── ...
│   │   └── ...
│   ├── redux/
│   │   ├── actions/
│   │   │   └── ...
│   │   ├── reducers/
│   │   │   └── ...
│   │   └── ...
│   ├── style/
│   │   └── ...
│   ├── app.js
│   └── index.js
└── package*.json
```

The refactoring process begins with the researcher duplicating the code from the monolithic application into each repository according to the results of the previous decomposition. The researcher also prepares the database and sets up image storage services in Cloudinary to support separate file storage.

Next, the researcher makes adjustments to several functions that were identified as having duplicated code based on the SonarQube scan results. Functions that have dependencies on other functions outside their API domain are modified to call functions from other APIs, in accordance with the principle of loose coupling [4]. After these adjustments are completed, the researcher conducts light testing to ensure that all services are functioning properly and as expected. This testing includes checking each API endpoint to verify the integrity and overall functionality of the application.

2.4. Containerization of Monolithic and Microservices Applications

After the refactoring process is completed, the services need to be placed into Docker containers to facilitate communication between services. Containerization begins with the creation of images for both the application and the database used. The image creation process also takes into account the ports used by each API to avoid conflicts when the application is run. Containerization is not only applied to the microservices application but also to the monolithic application to place both applications in the same environment.

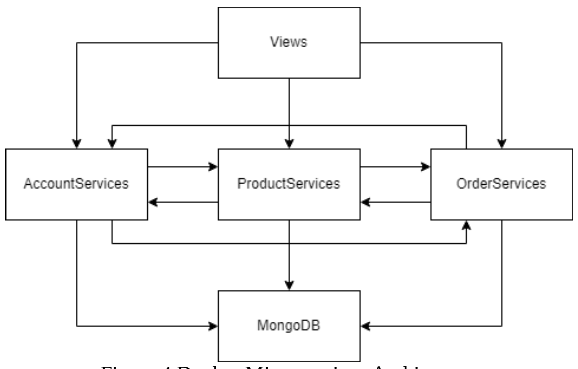

Figure 4 Docker Microservices Architecture

```
FROM node:16
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8081
CMD ["npm", "run", "start"]
```

Figure 5 Dockerfile Configuration for API Microservices

```
FROM node:16 as build
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
FROM nginx:alpine
COPY --from=build /usr/src/app/build /usr/share/nginx/html
COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 3000
CMD ["nginx", "-g", "daemon off;"]
```

Figure 6 Dockerfile Configuration for Views Microservices

Referring to Figure 4 , a Dockerfile is needed to create images for each API and Views for the Frontend. All APIs will run in a Node environment with version 16, and the code will be copied into the directory specified in WORKDIR. The duplication process starts with the package*.json files, as these files will be used first to install the library dependencies required by the API, followed by the RUN npm install command. Once this is completed, the code can be copied, with the port being exposed using EXPOSE command, and the image will be run using the command CMD ["npm", "run", "start"].

However, there is a difference between the Dockerfile for the API and the Views. For the Views, the application will be built first, and then the build output will be copied into a new environment running on nginx:alpine. The researcher also creates an Nginx configuration for server settings and for directing client requests to the backend API.

After all the Dockerfiles have been created, the next step is to build the Dockerfiles into containers. Building images can be done one by one, but this process can be quite time-consuming [5]. Therefore, a docker-compose file is needed to consolidate all the Dockerfiles into a single configuration and build them simultaneously. Below is an example of the docker-compose file used.

In docker-compose, each image is configured according to the requirements of each API, such as the source image to be built, container name, port, and environment variables. The docker-compose file also includes the MongoDB image to integrate it into the container. This way, the researcher can deploy all the created images into their respective containers with a single command.

```
version: '3.8'

services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db

  account-services:
    build:
      context: ./AccountService
    container_name: account-services
    ports:
      - "8080:8080"
    environment:
      -

  order-services:
    build:
      context: ./OrderService
    container_name: order-services
    ports:
      - "8082:8082"
    environment:
      -

  product-services:
    build:
      context: ./ProductService
    container_name: product-services
    ports:
      - "8081:8081"
    environment:
      -

  frontend:
    build:
      context: ./views
    container_name: frontend
    ports:
      - "3000:3000"
    environment:
      -
```

Figure 7 Docker-Compose Configuration for Microservices

```
FROM node:14 AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
COPY .env ./views
RUN npm install --prefix views
RUN npm run build --prefix views
FROM nginx:alpine
COPY --from=build /app/views/build /usr/share/nginx/html
COPY --from=build /app /app
COPY nginx.conf /etc/nginx/conf.d/default.conf
WORKDIR /app
RUN apk add --no-cache nodejs npm
COPY start.sh /start.sh
RUN chmod +x /start.sh
EXPOSE 2024
EXPOSE 5000
ENV PORT=5000
CMD ["/start.sh"]
```

Figure 8 Dockerfile Configuration for Monolithic Application

Unlike the microservices application, the monolithic application uses only one Dockerfile. The monolithic application is run in an environment using

Node.js version 14. This is done to match the version of the library dependencies present in the application. The configuration of the Dockerfile for the monolithic application is identical to the Dockerfile used for the microservices application. However, in the Dockerfile for the monolithic application, it is necessary to add Node.js and npm to the nginx:alpine environment to run the backend program with the command RUN apk add --no-cache nodejs npm. The image is then run with the command CMD ["/start.sh"].

```
npm run start &
nginx -g 'daemon off;'
```

Figure 9 start.sh file configuration

```
version: '3.8'
services:
  app:
    build: .
    ports:
      - "2024:2024"
      - "5000:5000"
    environment:
      -
    volumes:
      - .:/app
    depends_on:
      - db

  db:
    image: mongo:latest
    ports:
      - "27019:27017"
    volumes:
      - mongo-data:/data/db
```

Figure 10 Docker-Compose Configuration for Monolithic

After creating the Dockerfile, the researcher then creates a docker-compose file to deploy the images into containers. The images configured in the docker-compose file include only the monolithic application and MongoDB.

2.5. Testing

Testing is required to assess the impact of the refactoring to understand the performance differences before and after refactoring. The testing will be conducted through load testing using JMeter to observe the response time of the application. This testing will involve two stages: the first is the creation of test functions, and the second is testing the application itself.

In the creation of the test plan, the researcher tests in two scenarios: Single Request and Group Request. Single Request involves testing only one request per test, using the Login and Add To Cart functions. Meanwhile, Group Request is conducted by recording user activities within the application, so the sequence of requests made during user activities will be recorded. There are two types of users tested in Group Request: customers and sellers. The flow of activities performed can be seen in Figure 11 and

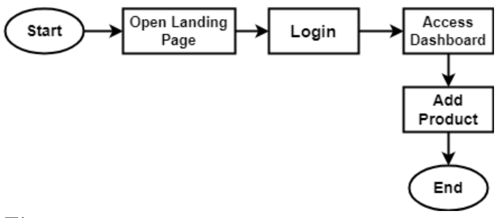

Figure 12 .

Table 2 Testing Cases

Testing Cases	Function
Single Request	Login
	Add To Cart
Group Request	Customer
	Seller

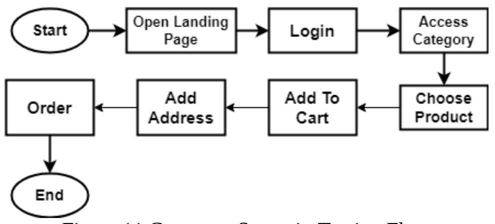

Figure 11 Customer Scenario Testing Flow

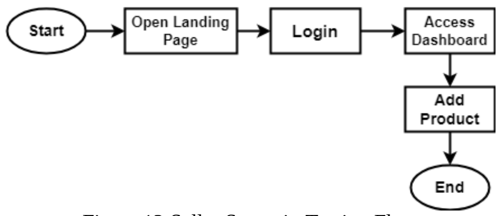

Figure 12 Seller Scenario Testing Flow

The configuration used in JMeter involves 300 thread users, a ramp-up period of 6 seconds, and a loop count of 1. This test aims to simulate the load of 300 users accessing the application simultaneously during a short ramp-up period to evaluate the application's capability to handle high loads. The results of this test provide insights into the differences in load experienced by the monolithic and microservices architectures, indicated by response time values. The response time data collected will only use the average response time for every 50 users and the overall test for each test case.

3. RESULTS

After conducting the research process, which included project search, monolithic decomposition, microservices implementation, application containerization, and testing, the following results were obtained:

3.1. Project Search Results

From the GitHub project search, the researchers successfully identified a repository that met the research criteria, namely the inventory-application developed by Mohamed Eleshmawy. This project uses JavaScript as its programming language and includes documentation artifacts such as the Entity Relationship Diagram (ERD) and Functional

Requirements. These documents serve as the foundation for the monolithic decomposition process to be carried out.

3.2. Results of Monolithic Decomposition

3.2.1. Results of the MDA Approach

The Meta-Data Aided (MDA) approach was conducted by analyzing the available artifact documents. The analysis of the Entity Relationship Diagram (ERD) and Use Case Diagram identified the separation of functions based on user roles, as shown in

Table 3 .

Table 3 The separation of functions and services based on MDA

Role	Use Case
Account	Login, SignUp, Edit Account, Add/Edit/Delete Address, Switch Roles
Admin	Add/Edit/Delete Categories, Add/Edit/Delete Products, Create Admin, Create Shipper, Assign Area for Shipper, Restrict User
Customer	Search Product, Wishlist Product, Purchase Order, Tracking Order, Return Order, Review
Seller	Add/Edit Products, Receive Notification from Customer, Change Status to Shipped
Shipper	Receive Notification from Seller, Change status to delivered

The separation of functions and services facilitates the development of services in the microservices architecture.

3.2.2. Results of the MDA Approach

The Static Code Analysis (SCA) approach was conducted using SonarQube, which resulted in the detection of issues as shown in

Table 4 .

Table 4 The results of the SonarQube scan

Category	Result
Bugs	4
Vulnerabilities	0
Security Hotspots	7
Code Smells	142
Duplication	12.2%

This analysis shows that code duplication is most prevalent in the Validation and controller folders. Significant code duplication was identified in the cartController.js and userController.js files. To address this issue, functions with duplicated code, such as updateTotalCartPrice() and genHashedPassword(), were developed to reduce redundancy and improve efficiency.

3.2.3. Function Decomposition

Based on the results of MDA and SCA, the function separation was carried out with separation as shown in **Error! Reference source not found.**.

Table 5 Function and Service Separation

Service Name	Method	Description	Controller	Table
AccountService	GET/PUT/POST	Account Management	User, Address, Wishlist, Permission	User, Address, Wishlist
OrderService	GET/PUT	Order Management	Order, Cart	Order, Cart, Payment, Shipper
ProductService	GET/PUT/POST/DELETE	Product Management	Product, Category	Product, Category, Comment
Views		Views FrontEnd		

This separation separate each function based on its domain and responsibility, supporting the principle of loose coupling and facilitating the development and maintenance of the application.

3.3. Results of Microservices Implementation

The implementation of microservices involved duplicating code from the monolithic application into individual service repositories. Code adjustments were made to address duplication issues detected by SonarQube. Functions that were interrelated were modified to utilize other APIs in accordance with the principle of loose coupling. Initial testing ensured that all services functioned correctly and met expectations.

3.4. Results of Containerization

Containerization was carried out for both application architectures microservices and monolithic.

.

3.4.1. Containerization of Microservices Applications

The containerization process for microservices applications involves creating Docker images for each API and frontend component. The Dockerfile for the APIs configures the application's ports and dependencies, while the Dockerfile for the frontend builds the application and sets up Nginx. Docker Compose is used to manage and build all images simultaneously, including MongoDB. This setup ensures that all components are containerized and can interact seamlessly within a unified environment.

3.4.2. Containerization of Monolithic Applications

For the monolithic application, the Dockerfile is tailored to match the Node.js version used in the previous application. Docker Compose manages the containers for both the monolithic application and

MongoDB, ensuring a consistent development environment that aligns with the microservices setup.

3.5. Results of Single Request Testing

The single request testing was conducted to evaluate whether the refactoring process significantly impacted individual processes. The requests tested were for the Login and Add to Cart functions. The Login function is frequently used by users when accessing the application but does not interact with other APIs during its process. In contrast, the Add to Cart function calls another API, namely ProductServices, to check the availability of items in the database. This setup was expected to reveal different results when the tests were executed. The results of the single request testing are detailed in the following figures and tables.

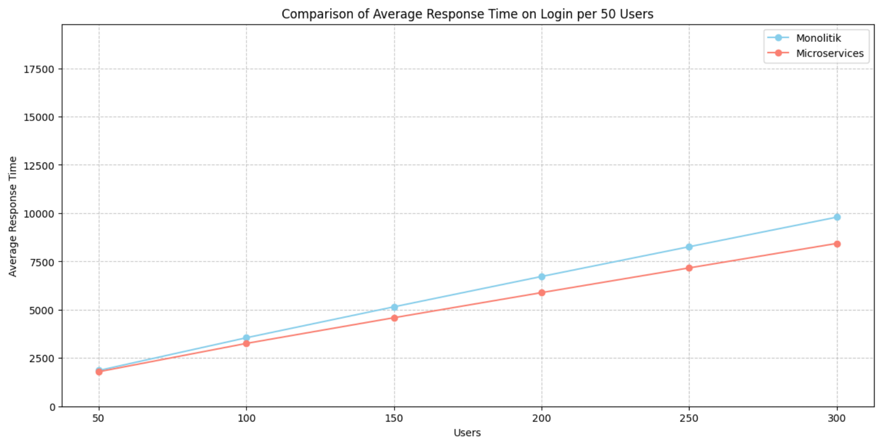

Figure 13 Comparison of Average Response Time for the Login Function

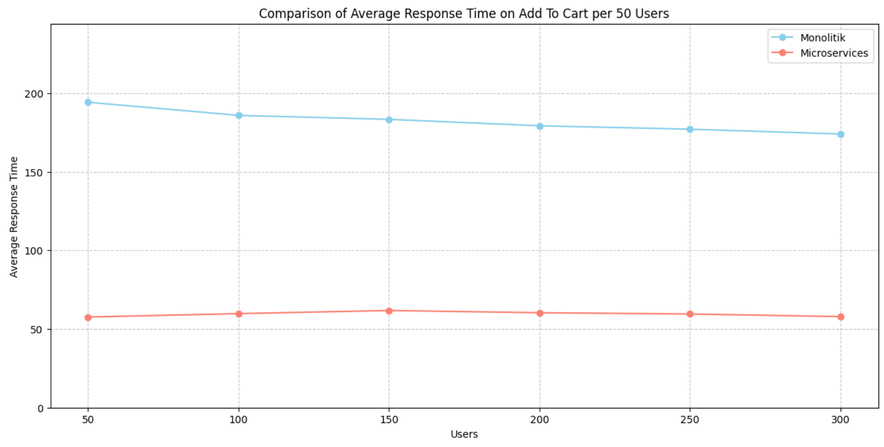

Figure 14 Comparison of Average Response Time for the Add to Cart Function

Referring to Figure 13 and Figure 14 , it is evident that the microservices architecture application has a lower average response time compared to the monolithic application. However, the difference in average response time between the login and add-to-cart scenarios varies. As previously explained, the add-to-cart function in the microservices architecture involves calling another API, which results in a smaller load on the API handling the add-to-cart function compared to the monolithic application. In contrast, the login function does not involve calls to other APIs. This results in a more significant difference in average response time for the add-to-cart scenario compared to the login scenario.

Table 6 Overall Average Data for Login Function

Request Login	Monolithic	Microservices
Average Response Time (ms)	9791.89	8432.576
Min. Response Time (ms)	81	78
Max. Response Time (ms)	18870	15843

Table 7 Overall Average Data for Add To Cart Function

Request Add To Cart	Monolithic	Microservices
Average Response Time (ms)	174.013	57.883
Min. Response Time (ms)	24	20
Max. Response Time (ms)	358	108

Table 8 Comparison of Average Response Time for Single Request

Function	Monolithic	Microservices	%
Login	9791.89	8432.576	16.12%
Add To Cart	174.013	57.883	200.63%

Data from Table 6 and Table 7 also show a significant difference between the monolithic and microservices applications, and Table 8 reinforces that inter-API communication greatly affects response time.

3.6. Results of Group Request Testing

Group request testing aims to evaluate the impact of application architecture differences on user experience during application use. Therefore, this test examines multiple requests simultaneously, representing user activities within the application. The results of the group request testing are detailed in the following figures and tables.

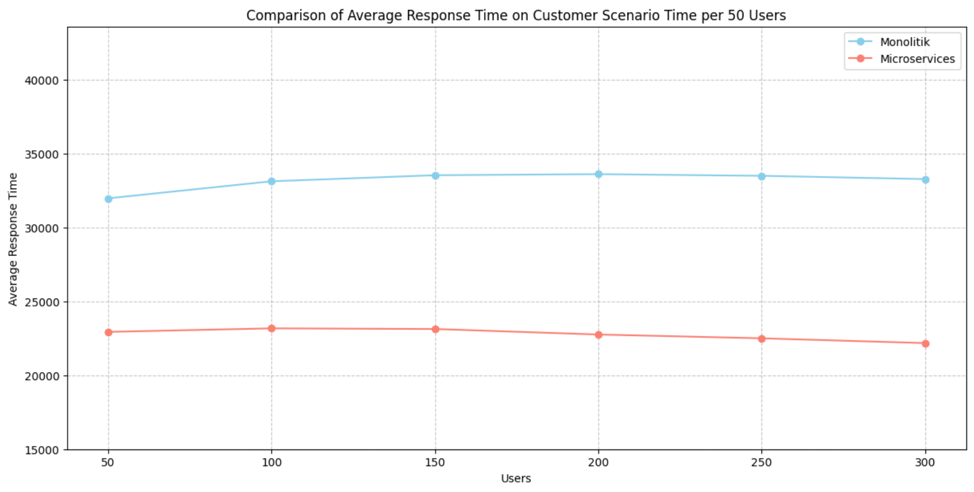

Figure 15 Comparison of Average Response Time for Customer Scenario

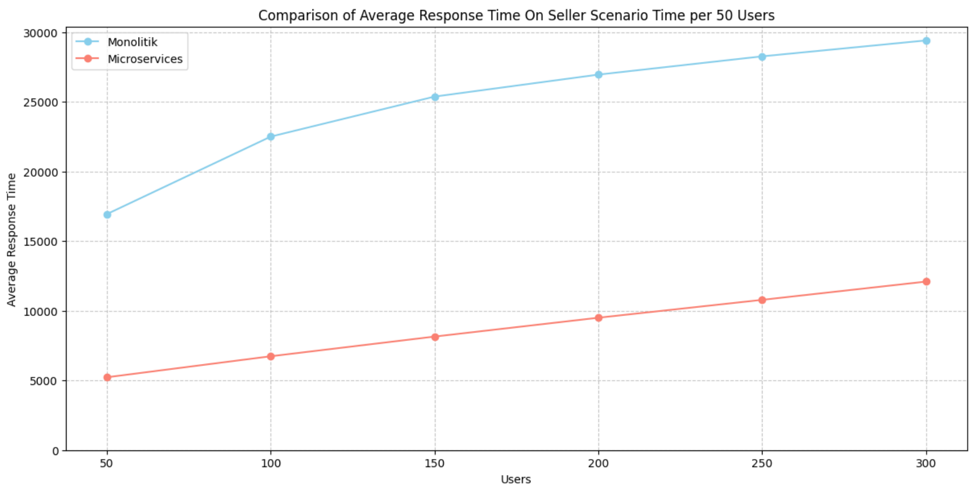

Figure 16 Comparison of Average Response Time for Seller Scenario

As seen in Figure 15 and Figure 16, the microservices architecture demonstrates a significant difference in response time. In the customer scenario, the average response time appears to remain relatively stable for both architectures. However, in the seller scenario, the average response time tends to increase as the number of users grows.

Table 9 Overall Average Data for the Customer Scenario

Request User Case	Monolithic	Microservices
Average Response Time (ms)	33285.577	22182.86
Min. Response Time (ms)	20257	6362
Max. Response Time (ms)	34492	24034

Table 10 Overall Average Data for the Seller Scenario

Request Add To Cart	Monolithic	Microservices
Average Response Time (ms)	29395.44	12091.293
Min. Response Time (ms)	10631	2290
Max. Response Time (ms)	46081	32150

Table 11 Comparison of Average Response Time for Group Requests

Scenario	Monolithic	Microservices	%
Customer	33285.577	22182.86	50.05 %
Seller	29395.44	12091.293	143.11%

The data in Table 9 and Table 10 show that the average response time for applications with a microservices architecture still has a significant advantage when running scenarios that are common to various types of users. Additionally, Table 11 reveals that the difference in response time for the seller scenario is higher, approximately 143.11%, between the monolithic and microservices architectures, while in the customer scenario, the difference is around 50.05%.

4. DISCUSSION

Testing of applications with both monolithic and microservices architectures shows that the microservices architecture consistently provides lower response times compared to the monolithic architecture, both in single request and group request scenarios. In single request testing, the login and add to cart functions reveal the tangible impact of inter-API calls on system performance. The add to cart function, which involves interaction between the OrderServices and ProductServices APIs, demonstrates improved efficiency in the microservices architecture compared to the monolithic architecture. This is due to the microservices' ability to distribute the workload across separate services.

In the group request testing, the results show that the microservices architecture is more effective at handling complex and large-scale loads, especially during spikes in demand. The more stable average response times indicate the architecture's ability to manage diverse user requests more efficiently. Previous research has shown that microservices architecture can optimize workflows by dividing tasks into smaller, interconnected services, resulting in improved performance [6]. Additionally, analysis of seller activities reveals that microservices are more efficient in processing computationally intensive operations. The methodology used in this research employs MDA and SCA approaches, which enable the identification of modules that can be decomposed and optimized. However, a limitation of these methods is that they do not allow developers to refactor with their own designed program logic, preventing optimization of execution time complexity through changes in program logic [20]. The implementation of microservices through containerization further enhances service isolation and management.

Overall, the results of this research indicate that transitioning from a monolithic to a microservices architecture has a significant positive impact on application performance. This study contributes to a deeper understanding of how microservices architecture can be applied to enhance system efficiency and responsiveness, and provides practical guidance for developers considering refactoring their applications. However, it is important to note that implementing microservices also requires careful consideration of the management of inter-service complexity and the maintenance of a more distributed infrastructure.

5. CONCLUSION

This study successfully demonstrates that transitioning from a monolithic architecture to microservices has a significant positive impact on application performance, particularly in terms of response time. The testing results indicate that the microservices architecture consistently provides lower response times compared to the monolithic architecture, both in single request and group request scenarios.

The advantages of microservices in distributing workload and managing inter-service communication enable it to address bottlenecks that are often encountered in monolithic systems. These findings align with existing literature, which indicates that microservices can enhance system scalability and flexibility, as well as provide benefits in terms of software management and maintenance over the long term [5], [15].

The implementation of refactoring using the Meta-Data Aided (MDA) and Static Code Analysis (SCA) approaches enables the identification and separation of appropriate modules, which in practice enhances operational efficiency and application performance. However, these methods do not address the optimization of execution time complexity within their approach. Containerizing services with Docker further support better isolation and management of services, aligning with the loose coupling design principles underlying microservices architecture.

Although microservices offer many advantages, their implementation requires careful consideration of service coordination and the maintenance of more complex infrastructure. Therefore, it is crucial for developers to consider the specific needs and context of their application before deciding to transition to this architecture.

Overall, this research provides valuable insights for developers considering a transition to a microservices architecture and offers practical guidance for effective refactoring. These results are expected to encourage further research in the future to explore the best strategies for addressing the challenges associated with implementing microservices and to enhance system performance and efficiency.

REFERENCES

[1] Y. Wei, Y. Yu, M. Pan, and T. Zhang, “A Feature Table approach to decomposing monolithic applications into microservices,” in *ACM International Conference Proceeding Series*, Association for Computing Machinery, Nov. 2020, pp. 21–30. doi: 10.1145/3457913.3457939.

[2] F. Ponce, G. Márquez, and H. Astudillo, “Migrating from monolithic architecture to microservices: A Rapid Review,” *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pp. 1–7, 2019, doi: 10.1109/SCCC49216.2019.8966423.

[3] B. J. Široký, “From Monolith to Microservices: Refactoring Patterns,” 2021.

[4] J. Fritzsch, J. Bogner, A. Zimmermann, and

S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, 2019, pp. 128–141. doi: 10.1007/978-3-030-06019-0_10.

[5] Q. Ren and S. Li, "Method of Refactoring a Monolith into Micro-services," *Journal of Software*, vol. 13, no. 12, pp. 646–653, Dec. 2018, doi: 10.17706/jsw.13.12.646-653.

[6] P. Zaragoza, A. D. Seriai, A. Seriai, H. L. Bouziane, A. Shatnawi, and M. Derras, "Refactoring monolithic object-oriented source code to materialize microservice-oriented architecture," in *Proceedings of the 16th International Conference on Software Technologies*, *ICSOFT 2021*, SciTePress, 2021, pp. 78–89. doi: 10.5220/0010557800780089.

[7] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges When Moving from Monolith to Microservice Architecture," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, 2018, pp. 32–47. doi: 10.1007/978-3-319-74433-9_3.

[8] L. Traini *et al.*, "How Software Refactoring Impacts Execution Time," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, Apr. 2022, doi: 10.1145/3485136.

[9] A. Almogahed, M. Omar, and N. H. Zakaria, "Impact of Software Refactoring on Software Quality in the Industrial Environment: A Review of Empirical Studies," 2018. [Online]. Available: http://www.kmice.cms.net.my/

[10] ISO/IEC 25010, "ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.," 2011

[11] R. B. Khan, "Comparative Study of Performance Testing Tools: Apache JMeter and HP LoadRunner," 2016, [Online]. Available: www.bth.se

[12] D. Al Fansha, M. Yusril, H. Setyawan, and M. N. Fauzan, "Load Test pada Microservice yang menerapkan CQRS dan Event Sourcing," *Jurnal Buana Informatika*, vol. 12, no. 2, pp. 126–134, 2021, doi: https://doi.org/10.24002/jbi.v12i2.4749.

[13] Q. Gu, S. Wagner, and J. Fritzsch, "A Meta-Approach to Guide Architectural Refactoring from Monolithic Applications to Microservices," 2020, doi: http://dx.doi.org/10.18419/opus-11501.

[14] J. Zhao and K. Zhao, "Applying Microservice Refactoring to Object-2riented Legacy System," in *Proceedings - 2021 8th International Conference on Dependable Systems and Their Applications, DSA 2021*, Institute of Electrical and Electronics Engineers Inc., 2021, pp. 467–473. doi: 10.1109/DSA52907.2021.00070.

[15] N. Goncalves, D. Faustino, A. R. Silva, and M. Portela, "Monolith Modularization towards Microservices: Refactoring and Performance Trade-offs," in *Proceedings - 2021 IEEE 18th International Conference on Software Architecture Companion*, *ICSA-C 2021*, Institute of Electrical and Electronics Engineers Inc., Mar. 2021, pp. 54–61. doi: 10.1109/ICSA-C52384.2021.00015.

[16] J. Han, S. Deng, X. Xia, D. Wang, and J. Yin, "Characterization and prediction of popular projects on gitHub," in *Proceedings - International Computer Software and Applications Conference*, IEEE Computer Society, Jul. 2019, pp. 21–26. doi: 10.1109/COMPSAC.2019.00013.

[17] L. Baresi and M. Garriga, "Microservices: The evolution and extinction of web services?," in *Microservices: Science and Engineering,* Springer International Publishing, 2019, pp. 3–28. doi: 10.1007/978-3-030-31646-4_1.

[18] Mohamed Eleshmawy, "inventory-application." Accessed: Mar. 19, 2024. [Online]. Available: https://github.com/moelashmawy/inventory-application

[19] D. Marcilio, R. Bonifacio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *IEEE International Conference on Program Comprehension*, IEEE Computer Society, May 2019, pp. 209–219. doi: 10.1109/ICPC.2019.00040.

[20] Dr. D. De Silva, P. Samarasekera, and H. Ridmi, "A Comparative Analysis of Static and Dynamic Code Analysis Techniques Techniques," 2023, doi: 10.36227/techrxiv.22810664.v1.