Berner Fachhochschule
Wirtschaft

Proof of Concept Microservices

Proof of Concept for Data Decomposition Using Microservice Architecture

Assignment 1

Course of study	**Bachelor of Science in Wirtschaftsinformatik**
Submitted by	**Marius Hägele, Nicola Bolt, Alper Simsek, Donjet Dzemaili**
Module	**Software Architecture - SDA2**
Experts	**Siddhartha Singh** **Prof. Dr. Sebastian Höhn**
Date of submission	**11.11.2024**

Contents

1. Introduction

The objective of this project is to implement a microservice-based architecture for a Point-of-Sale (PoS) system by transforming an existing monolithic structure. The project focuses on decomposing the monolithic database into multiple microservices, each responsible for a specific business domain and managing its data independently. This approach enhances scalability, fault tolerance, and maintainability. We selected five microservices to demonstrate this architecture.

Traditionally, enterprise systems, including PoS solutions, have been built as a single, unified codebase with a centralized database. While this approach is straightforward to start with, it makes modifying individual components challenging, as changes to one part of the system can impact the entire application. To address this, we are transitioning to a microservice architecture. In this model, each service is independent and dedicated to specific business functions and data management. In this paper, we also aim to analyze the trade-offs of adopting this architecture.

2. Microservices Selection

Ee have divided the system into five different core microservices. This chapter lists the individual microservices and describes their function and interactions.

2.1. Ticket System Microservice

Function: This microservice is designed to manage and track all sales transactions within the store. It manages customer invoices and records completed sales, ensuring that each sale is recorded and managed as an independent transaction.

Interactions: The service verifies customer data and links transactions to associated employees.

2.2. Customer Info Microservice

Function: This microservice manages customer records, including storing and updating customer profiles, contact information, and loyalty data.

Interactions: Provides customer data to other microservices when requested through secure API calls, ensuring data consistency across the system.

2.3. Employee Info Microservice

Function: This microservice manages all employee-related data and supports the business by providing staff details needed for various store operations.

Interactions: Interfaces with the Ticket System microservice to tag and verify employee involvement in transactions and other store-related processes.

2.4. Vendor Info Microservice

Function: This service manages the details and relationships with vendors who supply products to the store. It manages vendor profiles and facilitates order placements.

Interactions: Works closely with the Order Ticket microservice to provide vendor data needed for placing and tracking orders.

2.5. Order Ticket Microservice

Function: The Order Ticket microservice tracks orders placed with vendors for inventory replenishment. This includes generating order tickets, updating order statuses, and managing order details.

Interactions: Interacts with the Vendor Info microservice to validate vendor information and with the Product Inventory (if applicable) for product availability checks.

2.6. Justification of Microservice selection

We selected these microservices as they are essential for a Point-of-Sale (PoS) system, managing critical data that frequently requires access and updates. This selection demonstrates a balance between functionality and impact, concentrating on key services without adding excessive complexity. Each microservice operates independently, promoting scalability, maintainability, and data security. This design choice supports strong data ownership within each service and facilitates future system expansion. Overall, we believe this set effectively illustrates the practical advantages of a microservice architecture.

3. Architecture and Data Decomposition

3.1. Microservice Architecture

This PoS system has been designed with a microservice approach, splitting the traditional monolithic structure into modular services. This change to microservices offers benefits like better scalability, easier maintenance, and fault isolation.
Each microservice is an independent unit that manages its own business function and data. This setup lets teams work on different services without interfering with each other, speeding up development and reducing errors. Updating, scaling or replacing individual services is easier.

3.2. Comparison with Other Architectures

Monolithic: In monolithic architecture, the entire system is built as one unit. All components are used together. Monolithic architecture is easier to develop, but harder to scale, maintain, and update. One problem can crash the whole system. New changes require the redeploying of the whole system.

Service-Oriented Architecture (SOA): SOA is similar to microservices but has larger, more complex services that use an Enterprise Service Bus (ESB) for communication. SOA uses more middleware, while microservices use simpler communication patterns, making development more agile and lightweight.

3.3. Data Decomposition

When transitioning from a single database to multiple databases, data must be partitioned so that each service manages its own data independently. This approach enhances flexibility and simplifies management. In the PoS system, we have distributed data across our microservices accordingly.

3.3.1. Ticket System

Data Decomposition: The Ticket System table, which was previously part of a larger monolithic database, is now exclusively managed by the Ticket System microservice. This table includes crucial fields such as:

- ticket_id (Primary Key)
- customer_id (Foreign key for customer verification)
- employee_id (Foreign key from employee info)
- time
- date
- quantity
- cash
- credit
- company_name
- cost

Reasoning: By isolating the transaction data within this microservice, it can handle all sales transactions efficiently and securely. The service is responsible for processing, storing, and updating customer purchase information without direct interference from other services.

Communication: For verifying customer details and employee details, this service sends API requests to the Customer Info microservice and Employee Info microservice, ensuring data consistency and maintaining a clear boundary between services.

3.3.2. Customer Info Microservice

Data Decomposition: The Customer Info table is managed solely by the Customer Info microservice. This table includes crucial fields such as:

- customer_id (Primary Key)
- name (first_name, last_name)
- email
- phone_number
- address

Reasoning: This microservice's role is to act as the single source of truth for customer data. By owning this data, the service ensures that customer details are securely stored, easily accessible, and up to date. This decomposition isolates customer management functions, preventing data inconsistencies across the system.

Communication: The Ticket System and the employee info access customer data via APIs, ensuring that only validated and controlled data is shared as needed.

3.3.3. Employee Info Microservice

Data Decomposition: The Employee Info table, now managed by the Employee Info microservice. This table includes crucial fields such as:

- employee_id (Primary Key)
- customer_id (foreign Key)
- name
- contact_information
- role
- store_location

Reasoning: Isolating employee data within this microservice provides clear control over staff records and activities. This service is crucial for managing access and tracking which employees handle specific transactions or tasks. This decomposition supports secure and efficient employee data management.

Communication: The Ticket System may query this microservice to log which employee processed a specific transaction, ensuring accurate records for accountability and performance tracking.

3.3.4. Vendor Info Microservice

Data Decomposition: The Vendor Info table is exclusively owned by the Vendor Info microservice. This table includes crucial fields such as:

- vendor_id (Primary Key)
- vendor_name
- contact_details
- product_associations

Reasoning: By decomposing the vendor data into its own service, the Vendor Info microservice manages supplier relationships independently. This structure allows seamless vendor updates and interactions without affecting the rest of the system. It ensures that data related to suppliers is stored and maintained securely and efficiently.

Communication: The Orders Ticket microservice interacts with the Vendor Info service to retrieve vendor details when placing or managing orders for restocking.

3.3.5. Order Ticket Microservice

Data Decomposition: The Order Ticket table is controlled by the Order Ticket microservice and contains:

- OTID (Primary Key)
- vendor_id (Foreign key linked to vendor Info)
- employee_id (Foreign key linked to employee Info)
- product_details
- order_date
- order_status

Reasoning: The Orders Ticket microservice handles the lifecycle of orders placed with vendors. By owning its specific data, it maintains accurate tracking of restocking activities without overlapping with other services. This decomposition ensures that the process of ordering and managing inventory is streamlined and isolated from customer-facing operations.

Communication: The service calls the Vendor Info microservice for vendor validation and uses the employee service when tracking orders (e.g., special order requests).

3.4. Benefits of Decomposition

This data strategy lets each microservice manage its own data, keeping it secure and reliable. Each service only handles its relevant data, so there are no slowdowns like in shared databases. Scalability is improved, allowing individual services to scale without impacting others. If one microservice fails, it won't affect the others. Security is improved by limiting data access to the own microservice, reducing the risk of unauthorized access or data breaches. Each service implements its own security protocols.

4. Project Workflow and Illustration

This section illustrates how a customer utilizes our PoS system to complete a transaction, demonstrating the interactions between the customer and various microservices throughout the transaction process.

4.1. Process Workflow

1. **Login**: The customer logs in through the Customer Info microservice, which checks their details and gets their profile.
2. **Employee involvement**: The Employee Info microservice logs which employee is assigned to this order.
3. **Starting a transaction:** The Ticket System microservice creates a transaction record with details like the ID and employee involved.
4. **Order placement:** If a product is needed for the transaction, the Order Ticket microservice creates an order ticket with product and vendor details for internal analysis purposes.
5. **Vendor confirmation:** The Vendor Info microservice provides information to the supplier about the changes in the stock.

This process ensures that each microservice contributes to a modular, secure, and efficient transaction flow within the PoS system.

4.2. Illustration

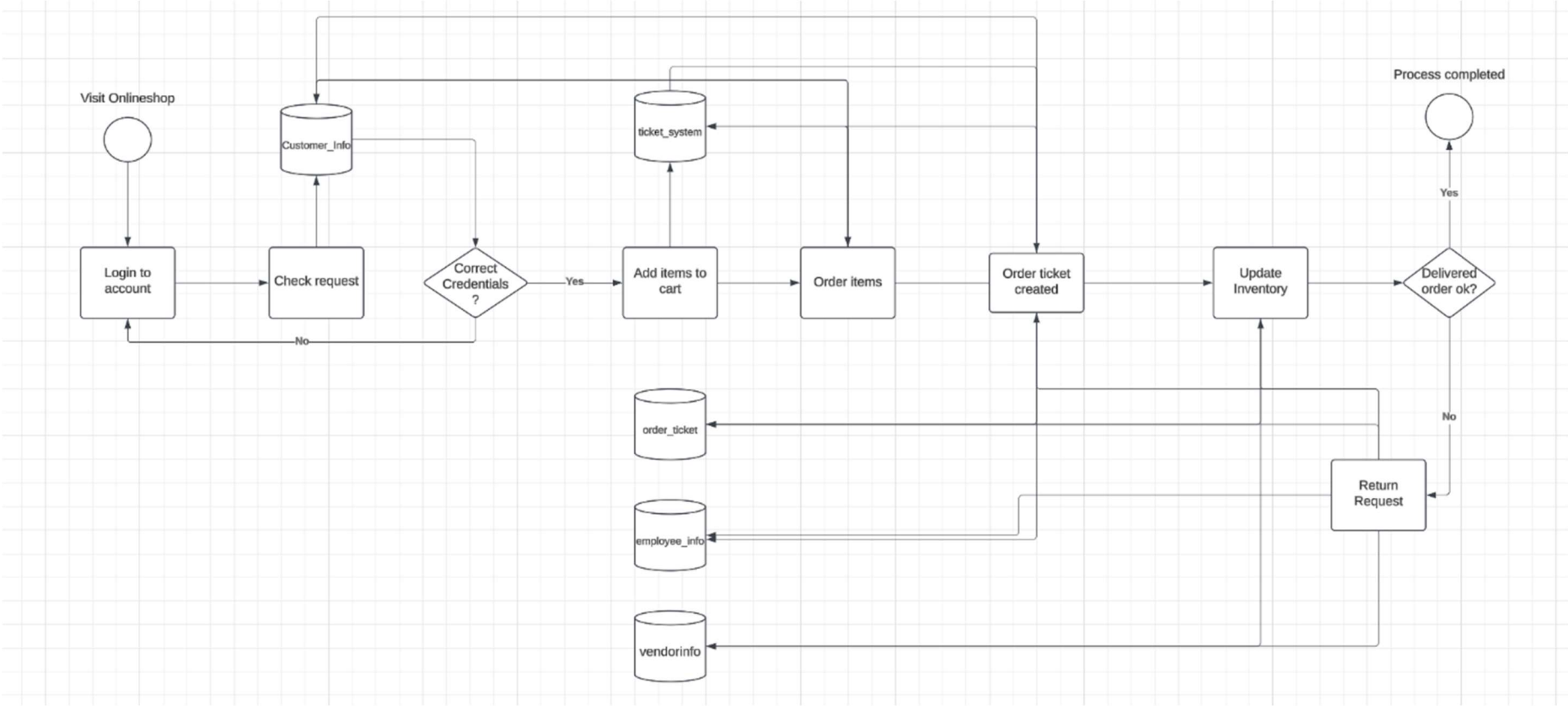


Figure 1: BPMN Process Flow

5. Benefits

A microservice architecture for the PoS system has many advantages and follows modern software development best practices.

Table 1: Benefits of microservice architecture

Benefit	Description	Our example
Scalability	Each microservice can be scaled independently to handle varying loads, ensuring efficient use of resources.	The Ticket System and Order Ticket can handle high transaction volume without affecting other services.
Modularity	Self-contained microservices make it easier to maintain and update the system, reducing the risk of issues.	Updating Customer Info doesn't affect Vendor or Employee Info.
Fault Isolation	Failures are limited to individual services, so the system stays up.	If the Employee Info service fails, other services like the Ticket System remain unaffected.
Independent Deployment	Services can be deployed or updated separately, which speeds up development and reduces risks.	Vendor info can be updated without affecting order tickets.
Optimized Performance	Each service only handles its own data, leading to faster processing and reduced latency.	Customer Info Service handles data quickly and efficiently even if vendor info has reached capacity.

This microservice architecture makes the PoS system flexible, reliable, and easy to maintain while supporting business growth. This setup lets each service evolve and perform well without affecting the system's stability.

6. Challenges and Trade-offs

Every architecture has its benefits but those come also with trade-offs. A microservice architecture for the PoS system has also that.

Table 2: Challenge of microservice architecture

Challenge	Description	Trade-off
Complexity in Design	Managing multiple microservices makes systems more complex. They must be deployed and managed separately.	It costs more to start, but it's easier to scale and develop features faster because the service is modular.
Data Consistency	Ensuring data consistency across services (for us the **Ticket System** and **Customer Info**) is complex due to distributed data ownership.	The implementation of distributed or eventual consistency enables the optimization of data handling, with a particular focus on the specific requirements of the service in question.
Inter-Service Communication	API communication between services adds latency and failure points.	Enables loose coupling and independent scaling, improving overall system resilience.
Deployment and Monitoring	Coordinating deployments and monitoring of microservices can be complex.	Advanced DevOps practices ensure better control, observability, and fault isolation.
Fault Tolerance	Handling errors in services like Order Ticket or Vendor Info requires good error management.	It makes the system more reliable by fixing problems one at a time.
Security Concerns	More entry points mean more ways for hackers to attack, so security needs to be stronger.	Service-specific security measures strengthen overall system protection. But it costs more.
Data Redundancy	Duplicating data across services can cause storage and consistency issues.	Improves service performance and response times.

Microservice architecture makes things more scalable, modular, and fault-tolerant, but it can be difficult to manage and cost more to begin with. By using best practices, the PoS system can use microservices effectively while addressing their challenges.

7. Docker Implementation

To implement the microservices with examples, we used Docker. Docker is well-suited for this task because it provides applications in isolated containers that can operate independently. Each microservice can run in its own container, which simplifies management, scalability, and maintenance.

All services are defined in the docker-compose.yml file to streamline the startup process. By using the command **docker-compose up --build**, five images and containers are created and launched for the different microservices. Once the images and containers are running, the microservices are ready to process data through CRUD requests.

The following is a list of all available CRUD requests that can be made within each microservice to demonstrate their functionality.

7.1. Ticket System Microservice

Ticket System Microservice illustrates inter-service communication. When a ticket is created, the customer's and employee's first and last names are provided. The system then queries the respective customer_id or employee_id from the relevant microservice and uses this response in the generated ticket.

7.1.1. Create Ticket

Methode: POST

URL: http://localhost:8080/tickets

Body:

```
{
    "company_name": "Neues Unternehmen AG",
    "quantity": 3,
    "subtotal": 150.0,
    "total": 165.0,
    "cost": 135.0,
    "discount": 15.0,
    "tax": 15.0,
    "tax_rate": 0.1,
    "cash": 80.0,
    "credit": 85.0,
    "cart_purchase": 1,
    "customer_first_name": "Marius",
    "customer_last_name": "Hägele",
    "employee_first_name": "Julia",
    "employee_last_name": "Meier"
}
```

7.1.2. Retrieve Ticket Information

Methode: GET

URL: http://localhost:8080/tickets/2

7.1.3. Update Ticket

Methode: PUT

URL: http://localhost:8080/tickets/2

Body:

```
{
    "quantity": 4,
    "total": 180.0,
    "discount": 20.0
}
```

7.1.4. Delete Ticket

Methode: DELETE

URL: http://localhost:8080/tickets/2

7.2. Customer Info Microservice

7.2.1. Create Customer

Methode: POST

URL: http://localhost:8081/customers

Body:

```
{
    "email": "neuer.kunde@example.com",
    "password": "NeuesPasswort123",
    "first_name": "Max",
    "last_name": "Mustermann",
    "phone_number": "0799999999",
    "rewards": 2.5,
    "street_address": "NeueStrasse 10",
    "city": "Zürich",
    "state": "Zürich",
    "zip_code": 8000
}
```

7.2.2. Retrieve Customer Information

Methode: GET

URL: http://localhost:8081/customers/2

7.2.3. Update Customer

Methode: PUT

URL: http://localhost:8081/customers/2

Body:

```
{
    "email": "aktualisierte.email@example.com",
    "first_name": "AktualisierterName",
    "rewards": 20.0
}
```

7.2.4. Delete Kunde

Methode: DELETE

URL: http://localhost:8081/customers/2

7.3. Employee Info Microservice

7.3.1. Create Employee

Methode: POST

URL: http://localhost:8082/employees

Body:

```
{
    "email": "neuer.mitarbeiter@example.com",
    "password": "NeuesPasswort123",
    "pin_number": 5678,
    "first_name": "Anna",
    "last_name": "Schneider",
    "user_id": 9456,
    "phone_number": "0799123456",
    "SSN": 987654321,
    "street_address": "Musterstrasse 12",
    "city": "Bern",
    "state": "Bern",
    "zip_code": 3000,
    "start_date": "2024-11-11",
    "company_name": "Galaxus",
    "number_of_stores": "50",
    "user_type": 1,
    "customer_id": 2002
}
```

7.3.2. Retrieve Employee Information

Methode: GET

URL: http://localhost:8082/employees/2

7.3.3. Update Employee

Methode: PUT

URL: http://localhost:8082/employees/2

Body:

```
{
    "email": "aktualisierte.email@example.com",
    "first_name": "Aktualisiert",
    "pin_number": 1111,
    "company_name": "Digitec"
}
```

7.3.4. Delete Employee

Methode: DELETE

URL: http://localhost:8082/employees/2

7.4. Orders Ticket Microservice

7.4.1. Create Order

Methode: POST

URL: http://localhost:8083/orders

Body:

```
{
    "date": "2024-11-11",
    "time": "15:30:00",
    "quantity": 3,
    "subtotal": 300.0,
    "total": 330.0,
    "discount": 15.0,
    "tax": 30.0,
    "tax_rate": 0.1,
    "cash": 100.0,
    "credit": 230.0,
    "status": 1,
    "employee_id": 2002,
    "vendor_id": 3
}
```

7.4.2. Retrieve Order Information

Methode: GET

URL: http://localhost:8083/orders/2

7.4.3. Update Order

Methode: PUT

URL: http://localhost:8083/orders/2

Body:

```
{
    "quantity": 4,
    "total": 440.0,
    "status": 3
}
```

7.4.4. Delete Order

Methode: DELETE

URL: http://localhost:8083/orders/2

7.5. Vendor Info Microservice

7.5.1. Create Vendor

Methode: POST

URL: http://localhost:8084/vendors

Body:

```
{
    "company_name": "Neuer Lieferant GmbH",
    "department": "Logistik",
    "street_address": "NeueStrasse 5",
    "city": "Neustadt",
    "zip_code": 67890,
    "phone_number": 1231231234.0,
    "fax_number": 1231231235.0,
    "email": "kontakt@neuerlieferant.de"
}
```

7.5.2. Retrieve Vendor Information

Methode: GET

URL: http://localhost:8084/vendors/1

7.5.3. Update Vendor

Methode: PUT

URL: http://localhost:8084/vendors/2

Body:

```
{
    "company_name": "Aktualisierter Lieferant GmbH",
    "city": "Aktualisierte Stadt",
    "phone_number": 1234567899.0
}
```

7.5.4. Delete Vendor

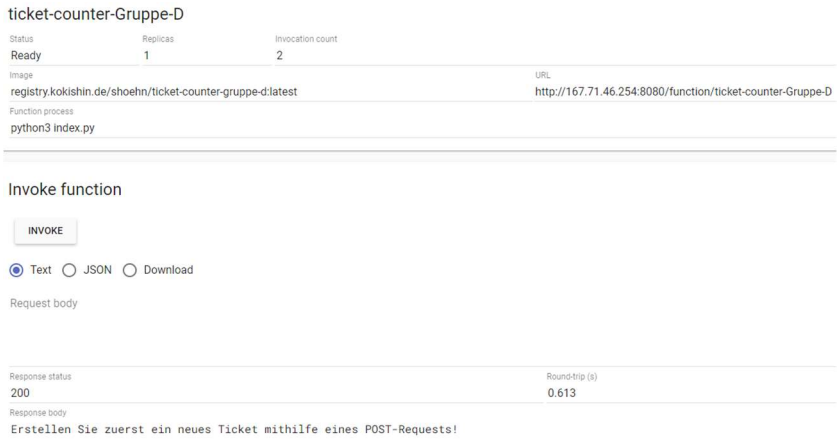
Methode: DELETE

URL: http://localhost:8084/vendors/2

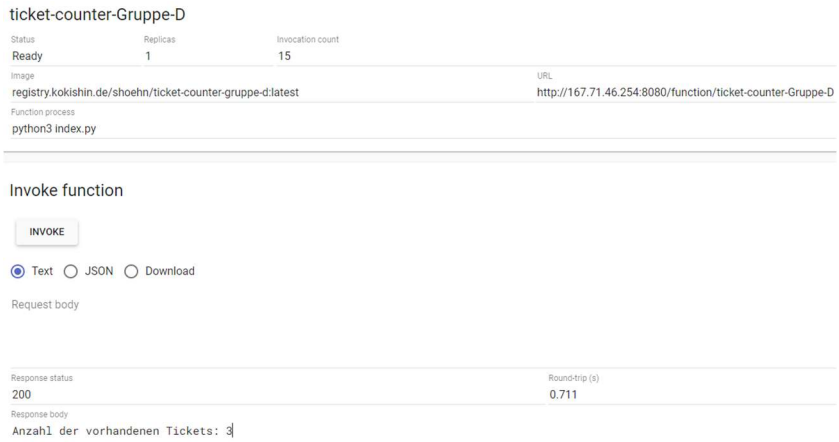
8. FaaS Implementation

To incorporate a Function as a Service into our work, we needed to choose a suitable platform. Since we reviewed the OpenFaaS platform in class and found it easy to use with server-based functionality, we decided to implement it. On OpenFaaS, we created the function "ticket-counter-Group-D," which performs the following function:

1. By clicking "Invoke" on OpenFaaS, a response is received. If there are no tickets yet in the Ticket System Microservice, the function responds with " Erstellen Sie zuerst ein neues Ticket mithilfe eines POST-Requests!"



2. If there are already tickets in the Ticket System Microservice, clicking "Invoke" will return the number of tickets that have been created. An example response might be: " Anzahl der vorhandenen Tickets: 4"



When creating a ticket in the Ticket System Microservice, a POST request is sent to OpenFaaS. The function receives this request and updates the ticket count in a file. When "Invoke" is clicked on the OpenFaaS interface, the function retrieves the ticket count from the stored file and displays it on the interface.

9. Summary

This project aimed to create proof of concept for a PoS system using microservices. The system was designed to split a traditional structure into smaller, independent services, each handling specific business functions and data. The chosen microservices for the PoS system are Ticket System, Customer Info, Employee Info, Vendor Info, and Order Ticket. These microservices were chosen to cover essential business operations. The PoS system achieved modularity, scalability, and fault tolerance by adopting a microservice architecture. The data decomposition strategy ensured each microservice had clear data ownership, optimized performance, and secure data management. The result was a flexible, efficient, and reliable system that can grow with the business.

9.1. Our Experiences

Starting this project was hard. We had difficulties to understand the goal of this project in the beginning. We were a little overwhelmed at first when we started. The project required a lot of research to learn how to do each step. We spent a lot of time looking for help and getting things clear, including asking ChatGPT about a thousand questions. This process helped us understand microservice development and best practices. We overcame the initial challenges and successfully implemented a microservice-based PoS system. This project showed the value of persistence, adaptability, and using what's available to overcome software development challenges.

10. Appendices

10.1. List of Abbreviations

PoS	Point-of-Sale
FaaS	Funktion-as-a-Service
SOA	Service-Oriented Architecture
BPMN	Business Process Model and Notation
API	Application Programming Interface
ESB	Enterprise Service Bus

10.2. List of Tables

10.3. List of Illustrations