Investigating differences in response time and error rate between a monolithic and a microservice based architecture

GUSTAV JOHANSSON
SAMMANFATTNING

Med stora framsteg i molntjänster har microservice arkitekturen kommit att bli en lämplig kandidat för utveckling av företagsprogramvara. Denna typ av systemarkitektur har föreslagits att lösa de problem som den traditionella monolitiska arkitekturen medför; långsamma lanseringar, begränsad skalbarhet och låg produktivitet. Således fokuserar denna avhandling på att utforska de möjligheter samt utmaningar som följer vid adoptering av microservices samt skillnaden i prestanda jämfört med den monolitiska arkitekturen. Detta undersöktes på en av Sveriges största banker, SEB, den Skandinaviska Enskilda Banken.


Resultaten indikerar att microservice arkitekturen har en signifikant högre felfrekvens men en längsammare responstid än den monolitiska arkitekturen, vilket stärker resultaten av Ueda et. al. [47] och Villamizar et. al. [48]. Forskningsresultaten har diskuterats med hänsyn till den komplexitet och de utmaningar som följer vid implementering av distribuerade system. Från denna studie blir det tydligt att komplexiteten i en microservice arkitektur skiftar från inuti applikationen ut till infrastrukturen. Således bör microservices inte ses som en silverkula. Istället är valet av systemarkitektur strikt beroende på omfattningen av projektet samt storleken på organisationen i fråga.
Investigating differences in response time and error rate between a monolithic and a microservice based architecture

Gustav Johansson
Royal Institute of Technology
Stockholm, Sweden
gustjoh@kth.se

ABSTRACT
With great advancements in cloud computing, the microservice architecture has become a promising architectural style for enterprise software. It has been proposed to cope with problems of the traditional monolithic architecture which includes slow release cycles, limited scalability and low developer productivity. Therefore, this thesis aims to investigate the affordances and challenges of adopting microservices as well as the difference in performance compared to the monolithic approach at one of Sweden’s largest banks, SEB - the Scandinavian Individual Bank.

The investigation consisted of a literature study of research papers and official documentation of microservices. Moreover, two applications were developed and deployed using two different system architectures - a monolithic architecture and a microservice architecture. Performance tests were executed on both systems to gather quantitative data for analysis. The two metrics investigated in this study were response time and error rate.

The results indicate the microservice architecture has a significantly higher error rate but a slower response time than the monolithic approach, further strengthening the results of Ueda et. al. [47] and Villamizar et. al. [48]. The findings have then been discussed with regards to the challenges and complexity involved in implementing distributed systems. From this study, it becomes clear the complexity shifts from inside the application out towards infrastructure with a microservice architecture. Therefore, microservices should not be seen as a silver bullet. Rather, the type of architecture is highly dependent on the scope of the project and the size of the organization.

ACM Classification Keywords

INTRODUCTION
The microservice architecture has become a promising architectural style for enterprise software and have gained a lot of attention recently [23]. It is an evolution of the service-oriented architecture style [24] and have been proposed to cope with the problems of the traditional monolithic architecture. Instead of having one large-scale service that handles all business logic, a microservice architecture is a distributed system where all its modules are lightweight microservices, each dedicated to a single business capability [11]. This approach of building software brings a plethora of benefits, some of them being increased agility, resilience, scalability, and developer productivity [6] [46].

However, the benefits come with challenges [23]. With a microservice architecture, each microservice have to be deployed independently and have to communicate with other services over the network. To counter the increase in operational overhead, companies can deploy their applications on Infrastructure as a Service (IaaS) [50] or Platform as a Service (PaaS) [7] solutions which have evolved from the great advancements in cloud computing. OpenShift [36] is a PaaS built around Docker [10] and Kubernetes [22], allowing developers to independently host and scale their applications using container based virtualization as opposed to hardware virtualization. The containerization and container orchestration that PaaS solutions like OpenShift provides, allow companies to create resilient systems and set up automatic scaling for their services, both horizontal and vertical, to optimize resource usage and availability.
The microservice architecture have therefore increased the need of what is known as the DevOps culture [23]. DevOps is a set of techniques for streamlining and integrating the software development process with the deployment and operations. Some of the main responsibilities of DevOps include to set up pipelines for continuous integration and continuous delivery to shorten the systems development life cycle [17]. Therefore, in order to enable all benefits a microservice architecture allows for, getting DevOps correct is a crucial part for its success. When addressed appropriately, a microservice architecture allow teams to be more productive and build frequently more successful software products [46]. Netflix [28], Amazon [20], Spotify [15] and SoundCloud [8] are just a few examples of companies who have adopted the microservice architecture and gained many benefits from it. However, correctly adopting microservices is a journey which is far from easy [23].

Objective
The Scandinavian Individual Bank, SEB, is one of Sweden’s largest banks. SEB is currently in an ongoing modernization process and are moving away from solutions that are “monoliths”, where upgrades and new versions are complex and where duplication of functionality exists. Instead, SEB aim for digital solutions such as API platforms, microservices and as-a-Service functionality to enable automated software delivery and hence reduce their time to market. Portfolio Management, a department at SEB which provides services in private banking, is in the middle of this process. The development team of Portfolio Management want to investigate the affordances and challenges of microservices as well as the difference in performance compared to the monolithic approach.

To do so, a new application was built, referred to as EBWeb. The application would be a minimal viable product, MVP, used by Portfolio Support, a first line support team that would use EBWeb to troubleshoot orders and manage instruments on the market. The application would also be used by the Broker Assistants at SEB, the assistants help principal brokers facilitate deals between buyers and sellers and would use EBWeb to manage customers and orders. EBWeb was built using two different system architectures - a monolithic architecture and a microservice architecture. The application would be a pilot study to gain insight

regarding to which extent SEB should adopt microservices.

Research Question
The main purpose for this study have been to investigate the affordances and challenges for a large-scale fintech company, such as SEB, to adopt a microservice architecture pattern and to what extent it should be adopted.

More specifically, the following research question have been investigated.

What is the difference in performance, specifically targeting response time and error rate, between a monolithic architecture and a microservice architecture?

Delimitation
The study was conducted with the following delimitation in mind.

- All database instances were deployed on high performance servers to eliminate the possibility of the database instances being potential bottlenecks during the performance tests.
- OpenShift was used as the PaaS solution for deployment as it is heavily integrated in the development culture and considered as standard use at SEB.

RELATED WORK
The chapter aims to provide the reader with a foundation in system architecture. More specifically, the characteristics, strengths and weaknesses of the two different system architectures that are being examined and investigated in this study; the monolithic architecture and the microservice architecture.

The Monolithic Architecture
The traditional way of building enterprise applications is to start off with a monolithic architecture [34]. A model based on a layered architecture, where each layer is tightly coupled and centrally integrated. It can be defined as a software application whose modules cannot be executed independently [11]. A monolithic architecture is generally composed of a user interface layer, a business logic layer and a data access layer. The client-side application makes requests to a centralized server application, which in turn handles the requests by
executing domain logic, retrieving and updating data from the database before sending back the result to the client [23].

However, as companies grow, the monoliths grow with them. The “monolith first” approach [33] appears to follow Gall’s Law, which states that “A complex system that works is invariably found to have evolved from a simple system that worked” [14]. With a monolithic architecture everything runs from a single executable, meaning their modularization abstractions rely on the sharing of resources of the same machine and interacts using in-memory function calls. The benefits of using a monolithic approach is therefore that all logic for handling a request runs in a single process, as the monolith holds all classes, functions and namespaces for the entire application. This makes certain tasks, such as testing, trivial for developers.

However, in today’s day and age with the recent advancements in cloud computing, in order to stay competitive, you need the ability to respond fast to inevitable change, have quick release cycles and short time to market. To do this, many companies follow an agile software development process, which advocates adaptive planning, continual improvement and encourages rapid and flexible response to change. To adopt this way of working when building monolithic applications can be challenging [21] as large-scale monoliths are difficult to maintain and evolve due to their complexity [43]. Any change in one module of a monolith requires a reboot of the whole application, making continuous deployment difficult [11]. For large-sized projects, restarting usually entails considerable downtimes, hindering development, testing, and the maintenance of the project [11]. As adopting an agile way of working is difficult when developing a monolithic application, monolithic software development instead tend to follow the waterfall development process [38]. The waterfall model is less iterative and flexible than the agile model. Once the development phase has begun, it is generally difficult, or in some cases impossible, to adapt to feedback [5].

Its massiveness also makes the code hard to understand, debug and modify [23]. Because of it being a monolith, technology lock-ins are introduced for developers who are bound to use the same language and frameworks of the original application [23] [34]. Lastly, scalability is greatly limited as a monolithic architecture can only scale in one dimension. With an increased transaction volume, it can scale up by running multiple copies of itself behind a load-balancer, known as horizontal scaling. However, different application components have different resource requirements - one might be CPU intensive while another might be memory intensive. With a monolithic architecture, each component cannot be scaled independently [23] [34].

The Microservice Architecture

The term microservice has gained a lot of attention recently [23]. It was introduced by Netflix [28] and has been adopted by several companies such as Amazon [20], Spotify [15] and SoundCloud [8]. The microservice architectural style is an approach to building distributed systems and have been proposed to cope with the problems of a monolithic architecture. It is an evolution of the traditional service-oriented architecture (SOA) [24] and puts emphasis on dividing a system into small, lightweight and loosely coupled services, each running on its own process space and built purposely to perform a very cohesive business function [23] [34].

Instead of having one large-scale service that handle all business logic, a microservice architecture is a distributed system where all its modules are microservices, each dedicated to a single business capability [23] [34]. This foster separation of concerns and allows each service to be independently replaceable, upgradeable and redeployable at any time. The independency also provides a decentralized service governance and a decentralized data management. Each service has its own, independent storage system, referred to as the database per service pattern [23], and can be built with the most optimal technology stack for the job to be performed. Because of its evolutionary design, a microservice architecture is flexible and creates opportunities for companies to respond faster to inevitable change [34].

Organizations are known to have a significant impact on the systems design. According to Conway’s law, "Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization’s communication structure" [9]. Microservice architecture motivates organization around business capabilities instead of the traditional way of building teams based on the technology layers [23]. This creates an agile working environment with cross-functional teams that can act autonomously to accelerate development [27]. This agility can be further amplified when combined with
PaaS solutions, e.g. OpenShift, that allow teams within companies to host, run and deploy their microservices using containers - without the concern that it might impact other teams or services.

A container is a lightweight operating system running inside the host system. It runs instructions native to the core CPU and eliminates the need for instruction level emulation. Containers are attractive as they provide good isolation with low overhead and fast start-up time leading to highly agile solutions [19] [12]. The isolation that containers provide, together with pipelines for continuous delivery (CD) available in PaaS solutions, creates fully autonomous teams that are responsible for the entire lifecycle of a service. This fosters continuous integration [13] and greatly eases software maintenance. By using automated CI/CD pipelines and modern container tools, it is possible to deploy an updated version of a service to production in a matter of seconds [29].

Another characteristic for microservice architecture is the use of smart endpoints and dumb pipes [23] [34]. Microservices use lightweight communication protocols, e.g. message bus or REST, for asynchronous communication, keeping all business logic in the endpoints - the services. This is the greatest differences when comparing microservices with SOA. In SOA, instead of having dumb pipes for communication, an Enterprise Service Bus (ESB) or similar is implemented to provide sophisticated message functionality such as message transformation, choreography and service orchestration [26].

If implemented correctly, a microservice architecture can unlock a plethora of benefits, such as: increase in agility, developer productivity, resilience, scalability, reliability, maintainability, separation of concerns, and ease of deployment [6] [46]. However, the benefits come with challenges. As microservices are deployed independently, services communicate over the network. This creates a need for service discovery, increased security management, communication optimization and load balancing [23] [34]. The network latency is also much greater than that of memory, meaning intercommunication within a microservice architecture is slower as compared to a monolithic architecture that makes use of in-memory calls [48][47]. Moreover, decomposing distributed systems into independent granular components brings complexity and operational overhead. Meaning a microservice architecture does not simply remove complexity, rather the complexity shifts from inside of the application out into infrastructure. As a reaction, this has given birth to what is known as the DevOps culture [23]. Some of the main responsibilities of DevOps include to set up pipelines for continuous integration and continuous delivery to shorten the systems development life cycle [17]. Therefore, in order to enable all benefits a microservice architecture allows for, getting DevOps correct is a crucial part for its success.

METHODOLOGY

Since the main focus for this thesis is system design, it follows the Design Science paradigm as described by Hevner, Ram, and College [16]. Therefore, the methodology used in this study is the Design Science Research Methodology (DSRM) as proposed by Peffers et al. [37]. The methodology is composed of six activities, all described shortly below.

1. **Problem identification and motivation:** Define the main research problem and the need for a solution.
2. **Define the objectives for a solution:** Define qualitative or quantitative objectives such that the solution can be compared to.
3. **Design and development:** Create the system in question.
4. **Demonstration:** Demonstrate how the system solves the problems through experimentation or other methods.
5. **Evaluation:** Determine to what extent the system does address the problems through measurement or other suitable means.
6. **Communication:** Communicate the problem, the systems design and conclusions to relevant audiences.

The problem identification and motivation, as well as the research question, is introduced in Chapter 1. The objectives for a solution, the design and development as well as the demonstration are provided in Chapter 3. The evaluation is described briefly in Chapter 2 whereas the results of the evaluation is presented in Chapter 4. In Chapter 5, the conclusions will be given.
Evaluation
To evaluate the differences between a microservice architecture and a monolithic architecture, two separate systems, S1 and S2, were developed and deployed. The two systems provide the same functionality but are built using two different system designs. S1 is a monolithic system whereas S2 is a microservice architecture. A set of performance tests were executed on both systems using Apache JMeter [4] to gather quantitative data for analysis. The performance tests measured response time as well as error rate under different workloads. To create realistic performance tests, the workloads were determined by use cases.

Use Cases
The application in question will be used internally by different teams and departments at SEB. Each team has different needs and will use the application differently. Therefore, to perform realistic performance tests, a use case was defined for each team to represent accurate workloads on the server.

IMPLEMENTATION
Two systems were developed, one being a monolithic service, S1, and the other one being a microservice architecture, S2. The microservice architecture, S2, were composed of three microservices, MS1, MS2 and MS3. The systems, S1 and S2, provide the same functionality. Moreover, a client, C1, was developed as well as an API Gateway, G1, for the microservice architecture.

Client
The client, C1, was developed using the JavaScript framework React.js [40], developed by Facebook, together with Redux [41], an open-source library for managing the application state. React.js is currently one of the most popular framework for developing single-page web applications [45] and was used together with standard JavaScript, HTML and CSS. The same client, C1, was used by both S1 and S2.

Services
The systems, S1 and S2, were developed using .NET Core 2.2 [30], an open-source, general-purpose development platform maintained by Microsoft. It supports cross-platform use and was used for development with the programming language C#. The .NET Core extension Entity Framework Core [31] is a lightweight and open-source type of data access technology. EF Core was used in development and served as an Object-Relational Mapper, ORM, which allows developers to work with databases using .NET objects. Even though one of the many benefits microservices provide is technological heterogeneity, the reason for implementing both systems using the same technology stack was to avoid unfair comparisons and misleading results when executing the performance tests.

Monolithic Architecture
The monolithic service, S1, was developed as a single codebase and executable. The system, S1, handles logic for managing orders, instrument and customers at the portfolio management department at SEB. The monolith uses a single relational SQL database to store its data. The composition of the monolith is presented in Figure 1.

![Figure 1: The composition of the monolith, developed as a single codebase and executable.](image)

Microservices Architecture
The microservice architecture, S2, provide the same functionality as the monolith. However, instead of being developed as a single executable, the microservice architecture consists of three independent microservices, MS1, MS2 and MS3. The microservices are each dedicated to a single business capability. MS1 is responsible for managing orders, MS2 is responsible for managing instruments and MS3 is responsible for managing customers. Each microservice is responsible for its own data, which is being stored in a relational SQL database, following the highly recommended database per service pattern [23]. The composition of all microservices are presented in Figure 2.
Figure 2: The composition of the microservices, developed as three independent executables, each dedicated to a single business capability.

Communication

The services in the microservice architecture, S2, as well as the monolith, S1, communicate over HTTP via REpresentational State Transfer Application Programming Interfaces, or REST APIs for short. REST [49] is a standard for how to build simple and easy to use APIs where an HTTP verb is specified for each request: GET, POST, PUT or DELETE. A RESTful API simply exposes methods inside a system accessible through URIs to allow external entities to perform CRUD (Create, Read, Update, Delete) operations on its data. The response to a request could use either eXtensible Markup Language, XML, or JavaScript Object Notation, JSON, as data format. For all of the APIs developed in this study, JSON was chosen as the main data-interchange format as it uses less overhead than XML [25].

While the monolith exposes only one large API for all of its functionality, each microservice exposes their own fine-grained API. The client, C1, can communicate directly with S1 by sending request to its API via HTTP, see Figure 3. When communicating with S2 however, the client, C1, has to communicate with the microservices by sending requests through an API gateway.

Figure 3: The flow of communication between client and the server in the monolithic architecture.

API Gateway

For the microservice architecture, S2, an API gateway, G1, was set up using API Connect [18], an API management solution developed by IBM. The API gateway is the single-entry point for the system and is similar to the Facade pattern in object-oriented programming [44]. The gateway, G1, receives requests from end-users via the client, C1. The gateway then either proxies or routes the request to the appropriate service or routes the request to several microservices, aggregates the results and then sends it back as a single response to the client, see Figure 4. Although it is possible to send requests to each microservice directly from the client, by implementing an API Gateway it reduces the number of concurrent requests to the server and assigns the heavy load on the internal infrastructure with high speed network, resulting in an increase in performance [42]. Moreover, API Connect provides features such as load balancing, service registry and security management.

Figure 4: The flow of communication between client and the server in the microservice architecture.
DEPLOYMENT

For both systems, S1 and S2, Azure DevOps Server [32], was used to provide source code management using Git. Azure DevOps Server is a Microsoft product that is standard for all development teams to use at SEB. Apart from providing source code management, Azure DevOps Server also provides requirements management, project management, release management, automated tests and pipelines for both CI and CD. Therefore, Azure DevOps Server covers the entire Application Lifecycle Management, ALM, and enable DevOps capabilities. The deployment for each service was handled by a PaaS solution known as OpenShift.

OpenShift

OpenShift [36] is a private platform as a service, PaaS, developed by Red Hat and provides containerization software. It is built around Docker [10] containers that are orchestrated and managed by Kubernetes [22]. Docker provides the abstraction for packaging and creating Linux-based, lightweight containers without the VM overhead. Kubernetes provides the cluster management and orchestrates Docker containers on multiple hosts. It provides mechanisms for deployment, maintenance, self-healing and application-scaling. The monolith system, S1, and the microservices, MS1, MS2, and MS3, were all deployed as individual applications on OpenShift. Each application runs within a pod, which is one or more containers deployed together on one host. OpenShift let developers manage resources by configuring the CPU and RAM allocation.

As each container running on a node consumes computing resources, OpenShift allows to set minimum and maximum values for how much CPU and RAM a container can consume. The minimum amount of resources that can be allocated a container is referred to as “request” in OpenShift and the maximum is referred to as “limit”. Meaning, when a container is being deployed, it will be instantiated with the requested amount of CPU and RAM. With an increase in traffic to the container, OpenShift will allocate more resources to it, up until it reaches the limit. If a container attempts to exceed the specified limit for CPU, the system will throttle the container. If the container exceeds the specified memory limit, it will be terminated and restarted.

Data Storage

Each microservice is responsible for its own data, following the database per service pattern [23]. All database instances for the microservice architecture as well as the monolithic architecture were relational SQL databases. Moreover, the databases instances were deployed on SEBs high performance servers to eliminate the concern of them being potential bottlenecks during the performance tests.

RESULTS

Performance tests were executed on both systems using JMeter, a performance testing tool used to distribute load across servers. The test plans were created using predefined use cases to mimic a more realistic test environment. The performance tests measured response time as well as error rate.

JMeter

The performance tests were run using Apache JMeter [4]. JMeter is used to distribute load across systems and analyze performance, such as response time and error rate. For the monolith, S1, requests were sent directly to its API via HTTP. For the microservice architecture, S2, the requests were sent to the API Gateway, G1, also via HTTP. The performance tests in JMeter was executed from a single computer running the Windows operating system. The result of each test was written to CSV files. A number of test plans were created based on use cases to create more realistic scenarios and workloads.

Use Cases

A total of two use cases was set up based on the different target groups at SEB that the application was built for: Portfolio Support and Broker Assistants. After having reviewed previous logs to determine the behavior of each target group, as well as being in contact and interviewing representatives from both groups during the development process, it became clear that the application was going to be used differently by the two. Portfolio Support were mainly interested in monitoring instruments and orders while the Broker Assistants were interested in customers and orders. With this in mind, two use cases were created with each use case having different weights on each microservice, see Table 1.
Table 1. Represents the workload distribution between the microservices for both use cases.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>MS1</th>
<th>MS2</th>
<th>MS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case 1</td>
<td>20%</td>
<td>0%</td>
<td>80%</td>
</tr>
<tr>
<td>Use Case 2</td>
<td>20%</td>
<td>80%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Resources Allocation

The CPU and memory allocation for each container was configured differently in both use cases. In Use Case 1, the monolith system, S1, runs with a requested amount of CPU of 1.5 cores and a limit of 3.0 cores. The memory requested for S1 is 1.5 GiB with a limit set to 3.0 GiB. The amount of resources consumed by the monolith was then distributed to the active microservices in Use Case 1. As the microservices handling orders, MS1, is both more memory and CPU intensive, it was allocated double the amount of CPU and RAM compared to the instrument service, MS2, see Table 2.

Table 2. The resources, CPU and memory (RAM), allocated for both systems, S1 and S2, for Use Case 1. Displaying both the requested amount of resources and the limits for each container.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>S1</th>
<th>MS1</th>
<th>MS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU, Requested</td>
<td>1.5 Cores</td>
<td>1.0 Cores</td>
<td>0.5 Cores</td>
</tr>
<tr>
<td>CPU, Limit</td>
<td>3.0 Cores</td>
<td>2.0 Cores</td>
<td>1.0 Cores</td>
</tr>
<tr>
<td>RAM, Requested</td>
<td>1.5 GiB</td>
<td>1.0 GiB</td>
<td>0.5 GiB</td>
</tr>
<tr>
<td>RAM, Limit</td>
<td>3.0 GiB</td>
<td>2.0 GiB</td>
<td>1.0 GiB</td>
</tr>
</tbody>
</table>

For Use Case 2, the same amount of CPU and RAM was allocated to the monolith as in Use Case 1. The order microservice, MS1, was allocated double the amount of CPU and RAM of the customer microservice, MS3, of the same reasons mentioned for Use Case 1 - MS1 being more memory and CPU intensive, see Table 3.

Table 3. The resources, CPU and memory (RAM), allocated for both systems, S1 and S2, for Use Case 2. Displaying both the requested amount of resources and the limits for each container.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>S1</th>
<th>MS1</th>
<th>MS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU, Requested</td>
<td>1.5 Cores</td>
<td>1.0 Cores</td>
<td>0.5 Cores</td>
</tr>
<tr>
<td>CPU, Limit</td>
<td>3.0 Cores</td>
<td>2.0 Cores</td>
<td>1.0 Cores</td>
</tr>
<tr>
<td>RAM, Requested</td>
<td>1.5 GiB</td>
<td>1.0 GiB</td>
<td>0.5 GiB</td>
</tr>
<tr>
<td>RAM, Limit</td>
<td>3.0 GiB</td>
<td>2.0 GiB</td>
<td>1.0 GiB</td>
</tr>
</tbody>
</table>

To simplify the comparisons and analysis of the data gathered from the performance tests, auto-scaling was disabled. Therefore, each application was deployed as a single container, configured with the amount of CPU and memory presented in Table 2 and 3.

Response Time

Response time was measured for the systems, S1 and S2. The response time implies the elapsed time from just before sending the request to just after the response has been received. To measure response time, test plans were created to execute a constant workload of 1100 requests per minute during 5 minutes on each system, resulting in a total of 5500 requests. This created a steady workload on the systems without purposely making them crash. Test plans were created for both systems and both use cases. The test plans for each use case were identical for both S1 and S2, meaning the same requests were sent with the same parameters. In Use Case 1 and Use Case 2, for the monolithic system, S1, all 5500 requests were sent directly to the monolith. In Use Case 1 and Use Case 2, for the microservice architecture, the total amount of requests, 5500, was split between the microservices with respect to the use cases. In Use Case 1, a total of 1050 requests were sent to the order service, MS1, and 4500 requests were sent to the instrument service, MS2. In Use Case 2, a total of 1050 requests were sent to the order service, MS1, and 4500 requests were sent to the customer service, MS3. For the microservice architecture, S2, all requests bypassed the API Gateway, G1, which then distributed the requests to the respective microservice API.

The tests performed for Use Case 1 resulted in an average response time of 82 ms for the monolith architecture, S1, and 224 ms for the microservice architecture, S2. Meaning the monolith architecture had an average response time which were 64% faster than the microservice architecture. When inspecting the 90% line, meaning the response time of which 90% of the requests fall, the response time for the monolith architecture, S1, was 171 ms and for the microservice architecture, S2, it was 313 ms. Meaning 90% of the requests to the monolith architecture, S1, will have a 45% faster response time than the requests to the microservice architecture, S2. The results are presented in Table 4.
Table 4. Displays the data gathered from the performance test which measured response time for Use Case 1. The data displayed is sample size, the average response time and the 90% line response time.

<table>
<thead>
<tr>
<th>Use Case 1</th>
<th>Sample Size</th>
<th>Avg. Response (ms)</th>
<th>Response, 90% (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>5500</td>
<td>82</td>
<td>171</td>
</tr>
<tr>
<td>S2</td>
<td>5500</td>
<td>224</td>
<td>313</td>
</tr>
</tbody>
</table>

The tests performed for Use Case 2 resulted in an average response time of 60 ms for the monolith architecture, S1, and 172 ms for the microservice architecture, S2. Meaning the monolith architecture had an average response time which were 65% faster than the microservice architecture. When inspecting the 90% line, meaning the response time of which 90% of the requests fall, the response time for the monolith architecture, S1, was 158 ms and for the microservice architecture, S2, it was 314 ms. Meaning 90% of the requests to the monolith architecture, S1, will have a 49% faster response time than the requests to the microservice architecture, S2. The results are presented in Table 5.

Table 5. Displays the data gathered from the performance test which measured response time for Use Case 2. The data displayed is sample size, the average response time and the 90% line response time.

<table>
<thead>
<tr>
<th>Use Case 2</th>
<th>Sample Size</th>
<th>Avg. Response (ms)</th>
<th>Response, 90% (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>5500</td>
<td>60</td>
<td>158</td>
</tr>
<tr>
<td>S2</td>
<td>5500</td>
<td>172</td>
<td>314</td>
</tr>
</tbody>
</table>

For both systems and for both use cases, all requests returned with a status code 200, meaning all tests were successful and no errors occurred, which was sought after when measuring response time. The results show that the monolithic system, S1, had a significant lower response time in both use cases compared to the microservice architecture, S2 - being 64% faster in Use Case 1 and 65% faster in Use Case 2. This was an expected result as the request sent to the microservice architecture, S2, first has to be sent to the API Gateway, G1, before being routed to the specific microservice, therefore consuming more network than the monolith which is being hit directly with the requests.

Error Rate

Error rate was measured for the systems, S1 and S2. The error rate implies the amount of failed requests that returned a response with status code 500 (internal server error). To measure error rate, test plans were created to execute a constant workload of 4440 requests per minute during 5 minutes on each system, resulting in a total of 22200 requests. This created a heavy workload on the systems to purposely make the containers crash in order to see how the two different architectures behave when errors occur. Test plans were created for both systems and both use cases. The test plans for each use case were identical for both S1 and S2, meaning the same requests were sent with the same parameters. In Use Case 1 and Use Case 2, for the monolithic system, S1, all 22200 requests were sent directly to the monolith. In Use Case 1 and Use Case 2, for the microservice architecture, the total amount of requests, 22200, was split between the microservices with respect to the use cases. In Use Case 1, a total of 4200 requests were sent to the order service, MS1, and 18000 requests were sent to the instrument service, MS2. In Use Case 2, a total of 4200 requests were sent to the order service, MS1, and 18000 requests were sent to the customer service, MS3. For the microservice architecture, S2, all requests bypassed the API Gateway, G1, which then distributed the requests to the respective microservice API. The tests performed for Use Case 1 resulted in an error rate of 62% for the monolith architecture, S1, and 17% for the microservice architecture, S2. Meaning, out of the 22200 requests being sent to the monolith architecture, 38% responded with a status code 200, resulting in 8533 successful requests. While out of the 22200 requests being sent to the microservice architecture, 83% responded with a status code 200, resulting in 18461 successful requests. The results are presented in Table 6.

Table 6. Displays the data gathered from the performance test which measured failure tolerance for Use Case 1. The data displayed is the error rate for both systems.

<table>
<thead>
<tr>
<th>Use Case 1</th>
<th>Sample Size</th>
<th>Error Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>22200</td>
<td>61.70</td>
</tr>
<tr>
<td>S2</td>
<td>22200</td>
<td>16.84</td>
</tr>
</tbody>
</table>

The tests performed for Use Case 2 resulted in an error rate of 62% for the monolith architecture, S1, and 7% for the microservice architecture, S2. Meaning, out of the 22200 requests being sent to the monolith architecture, 38% responded with a status code 200, resulting in 8533 successful requests. While out of the 22200 requests being sent to the microservice architecture, 93%
responded with a status code 200, resulting in 20559 successful requests. The results are presented in Table 7.

Table 7. Displays the data gathered from the performance test which measured failure tolerance for Use Case 2. The data displayed is the error rate for both systems.

<table>
<thead>
<tr>
<th>Use Case 2</th>
<th>Sample Size</th>
<th>Error Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>22200</td>
<td>61.56</td>
</tr>
<tr>
<td>S2</td>
<td>22200</td>
<td>7.39</td>
</tr>
</tbody>
</table>

The results show that the microservice architecture, S2, has a significantly higher fault tolerance than the monolithic architecture, S1. One possible reason for the results are that the order service, MS1, is very CPU and memory intensive. When being hit with a high amount of requests, it exceeds the memory limit of the container in OpenShift which terminates the container and restarts it. For the microservice architecture, all requests sent to the order service will fail during the downtime of the container. However, all requests being sent to the instrument service, MS2, in Use Case 1 or the customer service, MS3, in Use Case 2, will not be affected by the restart of the order service container. This is illustrated in Figure 5 with Use Case 1.

![Image 1](image1.png)

Figure 5: Displays how requests being sent to the instrument service still succeeds, even though the order service is being terminated and restarted in Use Case 1.

However, in the monolithic architecture, when the memory exceeds the limit set for the container in OpenShift, not only the requests regarding orders will fail, but also all other requests being sent to the monolith, resulting in a greater error rate than the microservice architecture. This is illustrated in Figure 6.

![Image 2](image2.png)

Figure 6: Displays how all requests being sent to the monolith, no matter what type of request, fails as it is being terminated and restarted.

The results show that the response time for the microservice architecture is significantly slower when compared to the response time of the monolithic application. When evaluating error rate, the systems were exposed to a higher workload and the results show that the error rate is significantly lower in a microservice architecture than a monolithic architecture. The results were expected as it follows the theoretical implications mentioned by Newman [34] and Lewis and Fowler [23]. Moreover, the results also confirm and further strengthen the work of Ueda et. al. [47] and Villamizar et. al. [48], which reported that the performance of a microservice model is lower than that in a monolithic model.

**DISCUSSION**

Within this paper, the affordances and challenges of adopting a microservice architecture have been investigated. The specific metrics studied in this thesis was response time and error rate. Regarding response time, previous research shows that 40% of users will wait up to a maximum of 3 seconds for a page to load before abandoning it [1]. Therefore, the response time for both system architectures investigated in this study can be considered very fast, even though the response time for the microservice architecture was almost double the response time of the monolithic application. Although, as the systems investigated in this study was aimed towards internal use at SEB and not directly towards customers, high availability is prioritized over low response time. With this in mind, combined with the theoretical knowledge and previous research in the field, the microservice architecture is a more suitable choice.
for an organization like SEB. Not only will the systems be highly available, it will also provide an increase in developer productivity, system scalability and code reusability. These are all important aspects within an organization which provides a large amount of applications and services, both for internal use and towards customers.

However, the complexity of a distributed system brings security issues, an aspect not very well researched [2] but highly important in industry, especially in fintech. The complex network interaction model between components [3] makes it challenging to get a global view of the entire application, hence compromising tasks such as debugging, monitoring and auditing the network traffic. This creates a greater surface attack area which allows attackers to more easily exploit and attack the system. Also, microservices are often designed to completely trust each other. Meaning, if a malicious adversary attacks the system and gains control of one microservice, several other services or in the worst case, the entire application, could be compromised and brought down. It is therefore vital for systems that process sensitive data to identify users in all components within a service communication chain.

Lastly, it is important to note that the microservice architecture is not a silver bullet, rather the type of architecture is highly dependent on the scope of the project and the size of the organization. From this study alone, it becomes clear that the deployment and infrastructure for the microservice architecture is more complex compared to the monolithic solution. Hence, the complexity is not removed, rather it shifts from inside the application out towards infrastructure. Pipelines for CI and CD had to be set up not just for one service, but for every individual microservice within the ecosystem. Luckily, with PaaS solutions like OpenShift used in this study, the operational overhead can be heavily reduced. With just a small amount of DevOps engineers, pipelines for CI and CD can be setup for several hundreds of microservices, allowing updates in a microservice to go online in just a matter of seconds.

**Method Criticism**

When measuring error rate, the performance tests ran for a period of 5 minutes. A longer sample time could have made an impact on the results and would have been interesting to investigate. Moreover, the system developed in this study, EBWeb, was an MVP that would serve as a pilot study and aimed to investigate the affordances and challenges of a microservice architecture. Therefore, the complexity of the system was fairly low with a total of only three microservices. Each microservice was also fairly independent, meaning the need for intercommunication between microservices was close to non-existent. This opens up to debate if the system can be considered a distributed system at all. Because of the low level of complexity, circuit breakers and an event-driven architecture for data consistency, both considered key components in a successful microservice architecture, were not implemented. With time, EBWeb will continue to grow and become increasingly more complex which will increase the need for these key components to be implemented in a later phase.

**FUTURE WORK**

The findings of this investigative study in system design have implications for both researchers within this area of expertise as well as for practitioners who are working in web development companies and plan on adopting a microservice architecture. This study heavily focus on differences in performance metrics, such as response time and error rate, for systems developed with modern frameworks and tools. For future work, it would be interesting to develop a more complex system, consisting of a greater amount of microservices with a higher level of granularity. This would create a higher level of dependency between microservices and would therefore be increasingly susceptible to partial failure. Circuit breakers [35] would have to be implemented to cope with unavailable dependencies and event-driven architectures, such as RabbitMQ [39], would have to be implemented for data consistency. Such a system would be more similar to the enterprise software application in today’s industry and could therefore provide insightful information. Moreover, tracing requests within a microservice ecosystem, or any distributed system, is complicated and compromises security, an aspect not very well researched till this day [2].

**CONCLUSION**

In this paper, a minimal viable product in the form of a web application was developed using two different system architectures. One as a microservice architecture and one as a traditional monolithic architecture. The systems were developed and later deployed on the cloud.
using the PaaS solution OpenShift. Performance tests were executed on both systems using JMeter to measure the differences in response time and error rate. The results show that the microservice architecture is slower than the monolithic architecture but has a significantly greater error rate. Therefore, the results follow the theoretical implications mentioned by Newman [34] as well as Lewis and Fowler [23]. While a majority of studies in this area of research shows that a well implemented microservice architecture is superior to the traditional monolithic approach, it is of utter importance to understand the underlying complexity. With a microservice architecture, the complexity is not removed, rather it shifts from inside the application out towards infrastructure, creating an increased need for DevOps practices. A microservice architecture can therefore be considered more suitable for larger organizations, such as SEB, with an already established customer base and expected growth.

REFERENCES


