Linköpings universitet | Institutionen för datavetenskap
Examensarbete på grundnivå, 16hp | Datateknik
2020 | LIU-IDA/LITH-EX-G--2020/050--SE

En fallstudie på prestandajämförelse mellan monolitisk och mikrotjänst arkitektur baserat på ett kvalitetskontroll system

A case study of performance comparison between monolithic and microservice-based quality control system

Mats Eriksson

Handledare : Peter Dalenius
Examinator : Judy Foo

Extern handledare : Jonas Thellman

LINKÖPINGS UNIVERSITET

Linköpings universitet
SE–581 83 Linköping
013-28 10 00 , www.liu.se

A case study of performance comparison between monolithic and microservice-based quality control system

Mats Eriksson
mater307@student.liu.se
Linköping University
Linköping, Sweden

ABSTRACT

Microservice architecture has emerged as a new way to create large complex applications by removing some problems that exist for a monolithic counterpart. While this will asset agility, resilience, maintainability and scalability within the application, other problems will be predominant such as performance. This case study aims to provide more clarity on this matter by comparing a microservice architecture with a monolithic architecture. By conducting several experiment on two self-developed systems it could be found that microservice architecture will must likely show a lower performance in terms of throughput and latency on HTTP requests which use internal communication requests. On small intensive HTTP requests with minimum internal communication the difference between the architectures is so low it could almost be neglected. With microservice architecture comes other challenges that a company must keep into account such loadbalancing, caching and orchestration which are beneficial for the performance.

KEYWORDS

Containers, Microservice, Microservice Architecture, Performance

1 INTRODUCTION

The most common way to design and develop a software application is to use a single executable that is deployed on one machine. This is usually called a monolith. For such application resources and dependencies are shared within the application. This software development strategy works well as long as the system are contained as small application. Bad scalability, technology lock-in and slow deployment cycle are a few problems which the developers have to deal with for a monolithic architecture [4] [11].
However, the shift towards microservices and microservice architecture make an appealing new approach where each microservice typically handle one specific task by decouple from the rest of the application. This increases *maintainability* and *evolvability* within the microservice and generally make it easier to assign one development team the responsibility for a one microservice [2]. This does also increase scalability when a service could be deployed independently, due to firm module boundary around each service [11].
This new architecture does not always bring benefits. It is still quite unclear how performance is affected in comparison to monolithic architecture. There are studies which show almost no performance difference at all while other show as half the performance or more [15] [14]. One of the reasons for different end results is due to the configuration has been altered for each study. There is also an uncertainty how the internal communication is done within each architecture.
This case study will focus on how such microservice architecture could be implemented on an existing company, what benefits or drawbacks could be applied for such a system compared to a monolithic architecture, performance wise. The study will also evaluate the effect of caching within a microservice architecture.

2 BACKGROUND

Quality control is one of the main pillar in production to ensure that all delivered products from the factory maintain high customer value and low refund rate. By adding checkpoints along the manufacturing process, measured data can be checked so they are within the control limit. Usually this could be visualized in a control chart which is a very powerful tool [1]. Six Sigma and Lean Six Sigma (LSS) are for example well known quality disciplines to reduce variation in production using this principle [16].

Under normal circumstances measured values are stored in a database until responsible person collect the data and convert them into a valid *control chart* or diagram. This work could be automated by an application doing the necessary calculations and evaluation. Finished data could be fetched and presented to a frontend for easy conversion to a diagram. A typical development of such application, would be monolithic three-tier application where necessary data is collected from a datastorage and presented by a backend to a frontend.

3 RESEARCH QUESTIONS

The purpose of the report has been to evaluate if the quality control system can be utilized as microservice architecture and compare the performance on a monolithic counterpart. This has been done in three steps, beginning with a theoretical part checking previous research in this area. Then the report has extend into executing the theoretical part into a practical system, which will be used to compare the performance on both monolithic and microservice architecture. The report emphasise on answering the following research questions:

- Will a microservice architecture have any performance advantages or disadvantages versus a monolithic architecture in our case study?
- How important is caching for the case study and especially for a microservice architecture?

4 DELIMITATIONS

To narrowing the research question all experiments was executed under synchronous measurements. Under a more real scenario the microservice architecture has to be analysed under an asynchronous environment. This will probably lead to exchange of communication protocol such as AMQP[1] that allows the messages to queue up. This will increase reliability and performance on an asynchronous event driven system.

5 THEORY

Under normal circumstances a software application is developed under monolith architecture. However, in the last decades new ways to structure the code has evolved. Service-oriented architecture (SOA) and microservice architecture (MSA) has an increasing interest due to the benefits of decouple features and by that removing its dependency to other features or resources. Historically SOA started to be used and implemented in the 2000s, while for MSA in the 2010s.

Monolithic Architecture (MA)

The main way to implement an application is to design a single executable program which share resources with the same machine. Here the resources could be anything from hardware related as memory, database etc to more software related resources as for example files and drivers. This makes the development easy and fast, but causes bigger issues when the application is growing. At some point it will become unmanageable. Any changes to the application will result into a new version which has to be deployed with a risk of shutting down the whole application. Also, when adding new features, the development team will be restricted to the same language which the application is based on.

Fowler and Lewis [11] describe a monolithic enterprise application usually based on three parts; client-side, server-side and database. The client-side work as an interface for the user and mostly built on javascript for a browser. The server-side application will handle HTTP requests from the client-side and execute relevant procedures to handle a response back to the client. Finally, there is a database management system to handle all queries from the server-side application. This can also be called *three-tier* application, where the client-side goes under the name presentation tier, server-side as application tier and the database as data tier. If the server-side application consist of a single logical executable it could be called a monolith.

Microservice Architecture (MSA)

Microservice Architecture (MSA) or microservices (MS) is a rather new software architecture which has received an increasing amount attention from many companies. Some companies which has embrace this architecture type are Amazon[2], Netflix[3], Uber[4]. Especially the benefits in the area of *scaling* and *maintainability* are attractive features, making it easy to change and redeploy each microservice. Because a microservice is totally decoupled from other services it is easy to develop a microservice for a cloud solution. Strangely enough there is no formal definition of what a microservice is, but it does have to fulfil the features of a small service which could be deployed, scaled and tested independently. Each microservice usually is assigned to a single task and communicate with other microservices under a HTTP mechanism. *Dragoni N.* have tried to define a microservice as the following [4]:

- **Definition of microservice**: A microservice is a cohesive, independent process interacting via messages.
- **Definition of a microservice architecture**: A microservice architecture is a distributed application where all its modules are microservices.

At first glance this seems very similar to what REST architecture try to achieve. In fact a RESTful API is very popular method to use as communication between microservices. The profit of using a microservice architecture will mainly be in the areas of agility, scalability, and resilience [3].

- **Agility**: During development of a microservice the team have easier to understand and overview the codebase when specification window is much smaller than for a monolithic application. The microservice can be

[1]Advanced Message Queuing Protocol

[2]https://www.amazon.com/

[3]https://www.netflix.com

[4]https://www.uber.com

deployed and tested independently, which make it easier to maintain.

- **Scalability**: Because the microservice is independent it can also be scaled at run time. Resources and workload could easily be modified or moved according to demand.
- **Resilience**: A microservice architecture is more resilient, if a component or service in the system fails. With decoupled microservices, synchronous dependencies could be avoided and small lightweight services can easily be taken down instead of risking large areas of functionality breakdown.

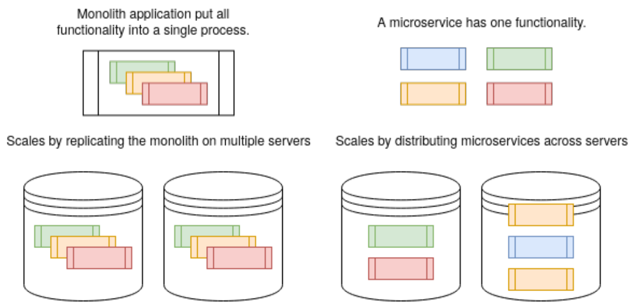
Monolith application put all functionality into a single process.
A microservice has one functionality.
Scales by replicating the monolith on multiple servers
Scales by distributing microservices across servers

Figure 1: Scalability of a microservice architecture

Microservice architecture does not only comes with benefits. Like any other architecture there will be areas that could cause problems. A few of them are as following :

- **Performance**: One percussive negative impact of a microservice architecture is the performance. The decomposition of the system into microservices will increase the number of requests across the network. For a monolithic architecture data could easily be fetched from a database and ones loaded into the memory it can be accessed anywhere within the application. However, for a microservice architecture same operation could involve a chain of synchronous call each contributing to the response time of the application [13][4].
- **Security**: As any distributed system with communication to another external system there could be a risk for security vulnerabilities. As many microservices uses REST protocol as the main interface the development team has to add additional security in the overhead or even encrypt the messages [4].
- **Load balancing**: As the traffic increase the need to distribute the messages between microservices will be more important. This is especially notable when a microservice is scaled up with several instances.
- **Reliability**: The microservice architecture is built around small simple components which in total will create a system. As the system grows it will become more complex and it's important that the messages between each service are reliable. For example adding asynchronicity will increase the complexity and shift the message passing mechanism to a more unreliable system [4].

Representational State Transfer (REST)

Representational State Transfer was defined by Roy Fielding in his PhD dissertion and is a software architecture which uses set of constraint for creating a service [6]. The principle is that the main communication is done with a client-stateless-server (CSS) architecture.

- **Client-Server**: By separating user (client) from the storage (server), portability is increased across multiple platforms. This will also improve scalability.
- **Stateless**: The communication must be stateless, meaning that a request must contain the necessary information to be handled.
- **Cache**: A request could be labelled cacheable so the client could reuse it later if the request is equivalent with the cacheable request.
- **Uniform Interface**: To simplify and improve the overview of the system the interface between components are uniformed. Any process, resource or data that are not used in the service has to be decoupled, so the service could work independent. This will come with the cost of decreasing efficiency.
- **Layered System**: Within a complex system, its necessary to isolate different services within layers. Adding a new services or components will be easier and it's possible to load balance the system. However, by default the overhead and latency will increase.

There is one additional architectural constraint "Code on demand" that is optional, but this is of no interest in this case study. All microservices and API are using REST in both architectures. This is because of its simplicity and it does not enforce any protocols or rules within the lower level.

Containers

Containers are a virtualization technology which has gained popularity for its easy way to build, ship and distribute an application. A container enables to isolate an environment which bundles one or more software feature(s) with its dependencies and resources into a single image for easy transfer. The image can later be executed by another host, which makes the deployment very convenient, fast and elastic. A container has less overhead compared to a virtual machine (VM) as the hypervisor layer is not present.
In general, a container will equals or exceeds a virtual machine in performance. Small intensive I/O request, where extra overhead has to be applied on each virtual machine

request has a significant effect on the performance, which benefits a container system. However, in large I/O request or streaming, the difference could be neglected [5].
Docker was selected as deployment platform of the case study due to its lightweight and popularity. Also, the monolithic architecture was set up in a container so the comparison would be equal in the scope of hardware and underlying dependencies and resources. Deployment of an application which are done directly on a single-tenant physical server is called *bare-metal*.

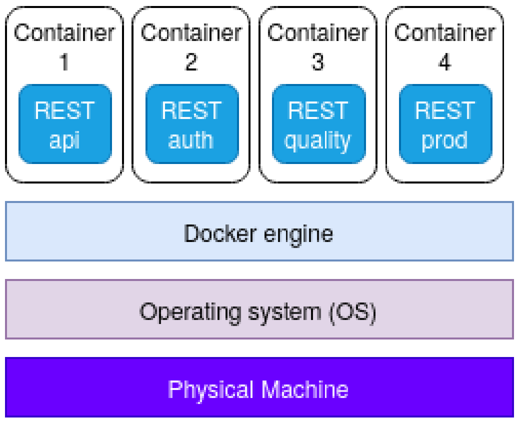

Figure 2: Typical structure of Docker containers

JMeter

Apache JMeter [8] is an apache project aimed to analyse performance of mostly web applications. The application can however handle several other protocols. The tool is developed in Java and used for so-called *load testing*. The procedure for a load test is to mimic a browser behaviour by submitting request and wait for a response by the server while measure the end-to-end *response time*. The aim in a performance test of a web application is to generate requests that are so similar to a real user as possible [12]. In general three different check is performed in practice [10].

- **Crash check**: This test is performed to check if the application has crashed, reload or another severe interrupt.
- **Perfromance check**: Performance check. This is usually performed to evaluate if there are any performance fluctuations in the system.
- **Basic error check**: A more in-depth analysis of the log files from the system.

Related studies in performance measurement for microservice architecture have used JMeter. This case study will also use JMeter as load test tool to receive comparable result.

Caching

While there exist many reports about cache in general, about how they should be designed for optimal performance in a CPU, operating system or a database. There are less information how cache would affect a microservice architecture. Sam Newman [13] has dedicated a whole chapter on *caching*. The benefit of using cashing is to optimize the performance by removing needless operation by storing previous result to be reused. This could for example be internal communication requests between two microservices or fetching data from a database. One problem with adding caching to a system is by exaggerate and adding too much. If the microservice architecture have multiple services, and they use large amount of internal requests, the freshness of the data will be hard to determine. This goes usually under the name *cache poisoning*. Caching could be summarized as client-side, proxy or server-side caching.

Related work

There are many publications discussing migration from existing monolith application to microservice architecture. However, most of them are just discussion and does not show any relevant information how to approach the refactoring process. This could be because it is a relatively new topic within software development (2020).
One recent study [9] compared different microservice frameworks, where two was chosen to be implemented into an existing monolithic system. The result shows positive effects in maintainability and scalability mostly due to the microservice architecture. The downside is that ones the framework was selected all new implemented microservices has to use the same framework. There was also an issue of load balancing on large datafiles.

In an industrial case study [2] where an application was migrated to a microservice architecture, showed that each service had *reduced feature overlapping* and are not technology dependent making the architecture multilingual. The development team was not dependent on any platform, but could choose any language for implementing the microservice. The application was a bank FX Core system (Danske bank) that was refactored as microservice architecture.

There have been several case studies which compare monolithic with microservice architecture in different environments. Villamizar et al. [15] has performed an evaluation comparing the average response time when deployed on the cloud. The performance analyse was executed with JMeter [8] which act as a frontend and was configured to send two types of requests. The first request has a high payload which was sent 30 times/min with an average response time of 3000ms. The second request represent a small but more used request which was send 1100 times/min and with an average response time of 300ms. The result shows a small performance degrade on the heavy payload request on the microservice architecture, while on the fast and more frequent request the performance was instead increased. However, the differences are so small that the architecture type will only be a small part of the impact on the overall performance.

Another case study made by Flygare and Holmqvist [7] compared three different architectures; *monolithic*, *microservice* and *bare-metal* with the performance aspects of RAM, CPU, latency and throughput collected by JMeter and Datadog. The analysis was performed locally on two computers where four stateless microservices (REST) was inspected. The same hardware was later used to measure the performance on monolithic and bare-metal system to have as equal conditions and circumstances as possible. The measurement was made on six different request, three INSERT and three GET, send to each microservice. The results shows that hardware resources such as CPU-usage and RAM-usage has a minor increase for microservice architecture system. Throughput and latency was also measured to a minor decreasing performance for the same system. However, the collected measurements show very high error rates which make the result very indefinite.

Ueda, Takanori and Nakaike [14] made a very detailed study comparing microservice architecture with monolithic architecture in a container environment. Docker was used as container platform. The two architecture was analysed with help of Acme Air, an open-source benchmark application that was configured to mimic an airline ticket reservation service. The experiment show that microservice architecture spent longer time on requests. Difference of twice the time or even more was measured. The study also concluded that *"The performance gap between the two service models is expected to increase as the granularity of services decreases"*. The study also recommend that optimizing the communication between services will lead to improved performance of the application.

6 METHOD

This chapter will outline the description of the quality control system and how it works. The system was the foundation for the performance experiments described in performance setup. This will be followed by information of the implementation.

The quality control system

The quality control system was developed with four major points in mind. The design of the quality control system can be viewed in figure 3. Many of the points listed will be solved in a microservice architecture where the overall behaviour of the system will correspond from the correlation between the microservices.

- **No technology lock-in**: The system must handle all request from different departments within or outside the facility. There should not be any limitations, when for example a second part application is using another operative system or software. Conflict could be reduced by using fixed interfaces between the system and second part application.
- **Enduser data**: The system should deliver data as raw measured values which could be converted into any control chart or diagram by the end user. Secondly it should deliver data in form of a control chart which could easily be visualized by any frontend application. This represents a fast way for any department within the facility to monitor any quality issue.
- **Independence**: Any calculation, conversion or changes made on the raw measured values into quality parameters has to be done independent within the system. If any process parameter goes out of scope it has to be reported or notified to responsible end-user.
- **Hook-in solution**: The quality control system should add a new feature to an already existing system without changing any important data or software. Most facilities have many ways to record quality parameters causing difficulties if the software or it's resources are altered.

Performance setup

From a performance perspective there are two different scenarios which are of interest. Small intensive HTTP request which interact by the quality control system by reading, adding or removing single object within either the quality or production database. This mean minimum internal communication within the architecture. This measurement will be referred to as S1. The second scenario (S2) generate a quality control chart which involves a large dataflow between REST_quality and REST_prod microservice. In the same request production data are calculated and checked according control limits. After generation the final control chart is responded back to the user.

JMeter was configuration to run as many requests as possible for 120 seconds on each thread. The number of threads were setup to vary from 1 to 120 in the step of 10 and represent the number of user on the system. Before each test

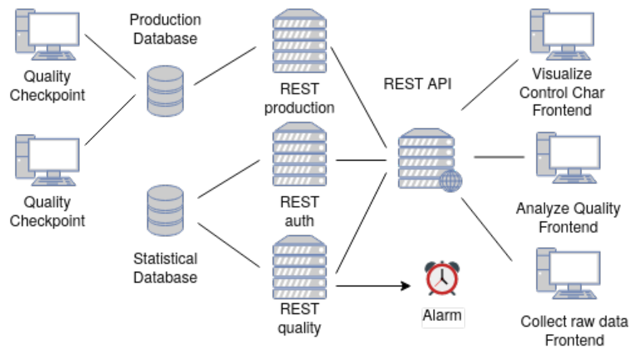

Figure 3: The design of the quality control system consist of four microservices. The REST_API work as a gateway between frontend and services. The REST_production microservice gathers production data from the existing production database. The REST_auth microservice handle all security related issues, such as user information and tokens. The last microservice REST_quality gather, calculate and evaluate statistical data. All data from this service will be stored in the statistical database. This service can also track a chosen process parameters which can give an alarm when going out of scope.

there was a 20 seconds *ramp-up time*, which represent the timeframe where all threads have to be started. The first configuration (S1) run three different request methods, each aiming at one microservice avoiding any performance peeks of a certain type of request. Both requests and responses in S1 have a small payload. The second configuration (S2) use high payload request which requires several internal requests within the architecture. The microservice REST_quality calls for a large amount of data from REST_prod, which is calculated into chart control data. The final results was collected by saving the performance aspects into CSV files with the respect on mean latency time, error rate and throughput. Request methods and number of properties, together with affected microservice are according to table 1.

Table 1: Performance scenario S1 and S2

S1 - Requests	No. properties	Microservice
POST, GET and DELETE	8	REST_auth
POST, GET and DELETE	3	REST_quality
GET	5x8 (rows)	REST_prod
S2 - Requests	**No. properties**	**Microservice**
GET	5x3000 (rows)	REST_quality REST_prod

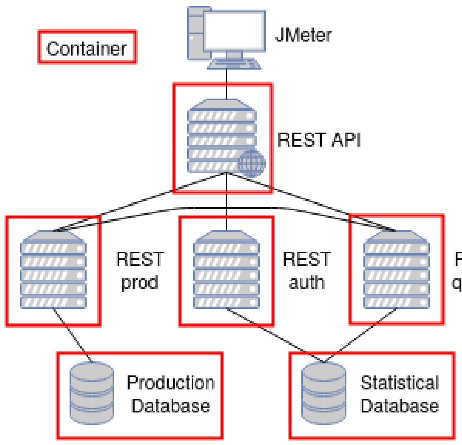

Figure 4: Setup A. Microservice Architecture setup, where each microservice is running in its own Docker container.

Experiments

The performance measurement consist of two setups, one for a microservice architecture (setup A) which is very similar to the quality control system in figure 3. The second setup consists of a monolithic three-tier architecture with API, backend and frontend (setup B). In both cases the frontend has been exchanged with JMeter [8] which execute and measure the request data. Both setups are illustrated in figure 4 and figure 5. In the aim of providing answer to the second research question, how a microservice architecture will respond to using cache, two more experiment was made by adding a cache feature to the same setup (A and B) giving total four experiments. All experiments were made on a wired local Gigabit Ethernet network to isolate the setup and remove any external performance latency to the measurement. Before executing the experiments unnecessary service and background tasks were shutdown to avoid interference with the result.

In related work [7][14][15] the performance aspects on CPU and RAM usage was measured as a part of their work. In this report both variables was removed due two reasons. Firstly, the tested measurement tools where not reliable enough, mostly because it doesn't isolate the measurement to a certain process. Secondly, related methods have no consistency of which type of measurement tool to use, which make the comparison of the result almost impossible.

Performance metrics

To compare the two different setup some metrics has to be defined before analysing the result. Following metrics was decided to be used.

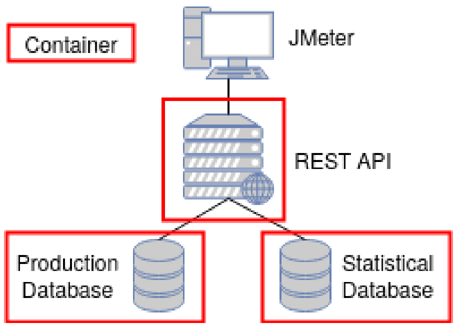

Figure 5: Setup B. Monolithic Arcitecture setup inside a Docker container.

- **Server throughput (requests/sec)**: One of the most important measurement is the number of request the system can handle under different situations. The throughput is measured in average of requests per seconds and is done with the tool JMeter [8].
- **Latency time (milliseconds)**: Represent the time between send request to start timestamp of received response in the respect of the enduser (JMeter). The latency time includes transfer of request to the API, process the request and return the data as JSON. However, it does not represent the time for a fully received response, which is called *response time*. The response time will always be higher than latency time. The latency time was measured as an average in milliseconds also recorded by JMeter [8].
- **Error rate (percent)**: The recording of this aspect has no other purpose than to ensure that the system does not validate any threats on the measurements. The end result for all measurement should be at or close to 0% for a successful experiment.

Implementation

Because the mainstream language on the existing platform was written in Python all microservices was decided to be done in the same language. Flask was used as web framework for the API and the selection was made mostly because of its lightweight features. While there are more popular and stronger web frameworks which could be used (example Django), Flask was used due to less built-in tools which give as more flexibility during the development phase, but also more work. To ensure security of the API and handle data to a database following extensions were used.

- Flask, version 1.1.2: To create the interface by routes
- Flask-Cors, version 3.0.8: To ensure that the client has a valid origin - CORS.
- Flask-Caching, version 1.8.0: Caching support for Flask
- PyJWT, version 1.7.1: To handle the authorization and JWT-token for the microservice.
- pymssql, version 2.1.4: To connect and handle data to/from MS-SQL database.
- smtplib, version : Used for sending an end user warning for triggered quality control parameter.
- python-dotenv, version 0.13.0: To handle all environmental variables.

Table 2: Hardware setup for Docker

Hardware	Value
OS	Linux Ubuntu 18.04.4 LTS
Memory	8 GB RAM
Processor	Intel©Core i5-4670K CPU@3.40GHzx4

It is important to highlight that the implementation of the application can be done on any other framework or language, which follow the features of REST and microservice. The route setup follow the defined interface which was specified by the bounded context. In an attempt to replicate the concept this will of course change according to that case and setup. The data was stored on a MS-SQL database. Before deployment each microservice was tested by using python unittest checking each request call.
The deployment was made on Docker 19.03.6 where each microservice has its own container located in a Docker bridge network. Gunicorn 20.0.4 was used as Web Server Gateway Interface for both the microservice architecture and monolithic system. The MS-SQL database was placed into a container, but not within the same Docker bridge network as the microservices. The hardware setup for Docker can be visualized in table 2.
Flask framework does not support caching by default, but could be added with the extension Flask-Caching. For this setup the cache type was set to filesystem which instantiate a file for each thread created by Gunicorn. A cache feature was applied to all methods that fetch data, hence all GET methods.

7 PERFORMANCE RESULTS

Experiment 1, S1 without cache

The result from the performance measurement of a none cached system can be shown in figure 6 and figure 7. In the experiment one (figure 6) both throughput and latency show almost the same performance with a small advantage for the monolithic architecture. The linear relation between latency and number of threads is caused by synchronised measurement where the bottleneck is the database. The latency show an average linear coefficient difference of 5,79 ms/thread

Table 3: Latency linear coefficient values

Test	Monolithic	Microservice	Difference
Without cache - Figure 6 & 7			
S1	22,8 ms/thread	28,6 ms/thread	5,8 ms/thread
S2	44,6 ms/thread	148,6 ms/thread	104,0 ms/thread
With cache - Figure 8 & 9			
S1	23,5 ms/thread	34,6 ms/thread	11,1 ms/thread
S2	-	67.4 ms/thread	-

Table 4: Mean throughput values

Test	Monolithic	Microservice	Difference
Without cache - Figure 6 & 7			
S1	39,6 requests/s	33,1 requests/s	6,5 requests/s
S2	19,0 requests/s	5,2 requests/s	13,8 requests/s
With cache - Figure 8 & 9			
S1	39.2 requests/s	27.4 requests/s	11.8 requests/s
S2	-	12.4 requests/s	-

and average throughput difference of 6,5 requests/s better performance for monolithic architecture.

Experiment 2, S2 without cache

Furthermore, the result from experiment two, S2 without caching, (figure 7) shows a very large difference between microservice architecture and monolithic architecture. The latency has a linear coefficient difference of 104 ms/thread which is almost 18 times worse than using small intensive HTTP requests as in experiment one. Because the system does not use any caching it is obvious that when a microservice needs data from another microservice the internal delay time will be added to the total request call. This could also be shown in the average throughput difference of 13.8 req/s which is about 2,1 times worse than from experiment 1 (S1).

Experiment 3, S1 with cache

When running the performance experiment three, S1 with caching implemented it could be shown that for small HTTP requests the cache capabilities does not add any major performance impact. In fact for the microservice architecture the latency actually increase making it worse, which was not expected.

Experiment 4, S2 with cache

The fourth experiment, executing S2 with caching, showed a performance increase on both latency and throughput for the microservice architecture. The latency coefficient was lowered from 148,6 ms/thread to 67,4 ms/thread which is almost 2,2 times better. Also, the throughput increase from 5,2 requests/s to 12,4 requests/s (2,4 times better).

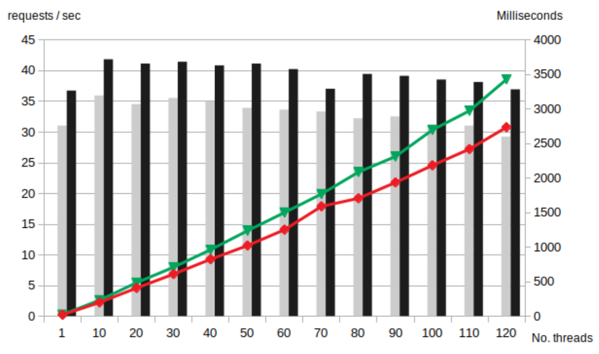

Figure 6: S1 - Throughput and latency without cache.

Grey columns:
Throughput for setup A in request per seconds.
Black columns:
Throughput for setup B in request per seconds.
Green line (triangle):
Latency for setup A in milliseconds.
Red line (square):
Latency for setup B in milliseconds.

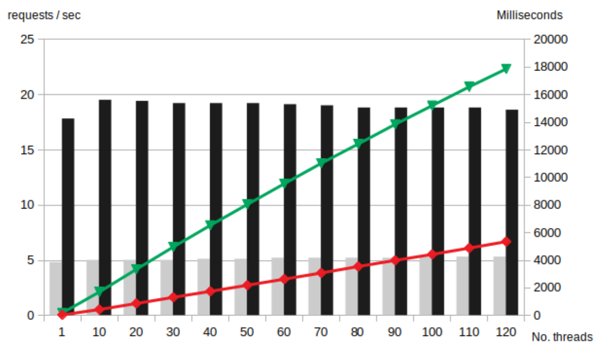

Figure 7: S2 - Throughput and latency without cache.

Grey columns:
Throughput for setup A in request per seconds.
Black columns:
Throughput for setup B in request per seconds.
Green line (triangle):
Latency for setup A in milliseconds.
Red line (square):
Latency for setup B in milliseconds.

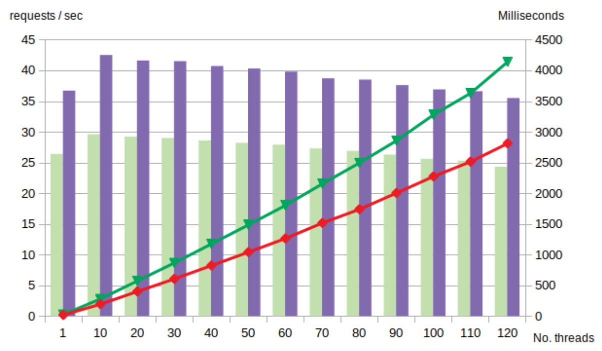

Figure 8: S1 - Throughput and latency with cache.

Grey columns:
Throughput for setup A in request per seconds.
Black columns:
Throughput for setup B in request per seconds.
Green line (triangle):
Latency for setup A in milliseconds.
Red line (square):
Latency for setup B in milliseconds.

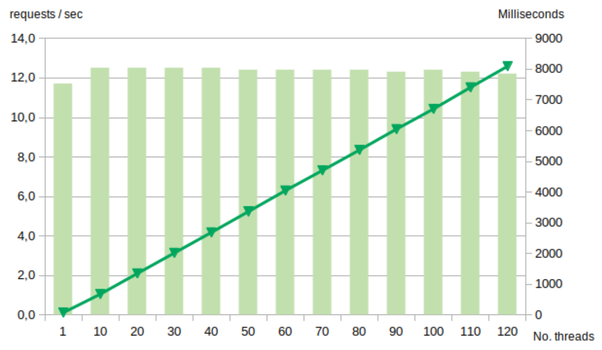

Figure 9: S2 - Throughput and latency with cache.

Grey columns:
Throughput for setup A in request per seconds.
Green line (triangle):
Latency for setup A in milliseconds.

8 DISCUSSION

Experiment 1, S1 without cache

The result from experiment one is similar to the result done by Villamizar et al.[15] and the thesis made by Flygare and Holmquist [7] which shows that microservice architecture is in general slower than a monolithic architecture. However, the performance difference is minor and should be compared against other features that a microservice architecture could provide.

Experiment 2, S2 without cache

If the API was added with a load balancing feature it will probably increase the throughput on the microservice architecture, especially if the system have more than three underlying microservices. This is suppoerted by the experience from Gouigoux and Tamzalit [9]. A microservice also has the option to scale up, by starting more microservices for increasing delivery on request answers. The complexity from handle large result set has moved outside of the microservice and challenges such as data exchange have to be balanced by the *orchestration* or *choregraphy*.

Experiment 3, S1 with cache

The strange result from microservice architecture with caching, was not investigated to find out this odd behaviour. But one speculation would be that the time for reaching the caching on filesystem is longer then fetching the data directly from the database. The throughput confirms the same pattern, where the number of executed requests/s dropped from 33,1 to 27,4. The results from the cached measurements can be visualised in figure 8 and figure 9.

Experiment 4, S2 with cache

The cache feature fulfills a much more important role for a system which uses data rich API calls or large payload HTTP request between services. The measurements for cached monolithic system was executed but values was removed from the analysis, because it basically shows the latency time for direct access to the cache. The extreme high throughput reveal that the system was reading a cachefile in a very time consuming way. The result from the error-rate was measured to 0% for all experiments, except for experiment four, cached, microservice setup. In this case JMeter was reporting an average error-rate of 0,10%. The log files from the microservice revealed that the connection was broken due to *ChunkedEncodingError* where the *content_chunk_size* was not correct. Another cache method such as *Memcached* [5] would have been more appealing for a microservice system with better memory management and easier to handle large data caches.

Development issues

It is rather appealing to have one isolated and decoupled process within the microservice. However, the decouple process could cause some problems if the internal structure of the database can not be changed. For example the use of serializers within the Flask framework was limited or in some cases not used at all. This has to be compensated with rather complicated SQL queries which is hard to read and understand. For some routes most of the response time consist

[5]https://www.memcached.org/

of two or three SQL queries to fulfil the response. Another problem was to execute SQL queries which use some sort of combination feature as for example *inner join* and all tables are not decoupled to the same microservice. This could of course be solved by decouple the database to each microservice and in an event of schema change it will only affect that microservice.

As the number of different microservices increases the management of the overall system gets more complex. While the development for each microservice is easy to handle with an isolated specific task, the *orchestration* take a much more important role in the design phase. In all experiments the external request was not load balanced at the API. To compensate the direct communication between each microservice the underlying microservices REST_prod and REST_quality had increased number of responsive *workers* to increase their performance.
The repetitive delivery process for each microservice to a container is a very time consuming task. It is essential that adoption to software tools which could automate this process is made early in the project to save time. Also, during the development, each microservice is tested in different environments making it essential to handle different configuration options, so the microservice can be deployed locally, as bare-metal or container.

9 CONCLUSIONS

Based on the development and measurements of the case study some conclusions could be experienced and identified in regards to microservice architecture. While the simplicity of making a microservice has increased with independent codebase and deployment, the overall management of the total system is more cumbersome. The design phase of the system must be more aware of what encapsulate the logic, resources and responsibility of each service. The complexity of using decoupled microservices creates challenges such data exchange, load balancing, strict interfaces and orchestration. Dispensation of any key features in the orchestration will punish the system with lower performance in both throughput and responsetime compared to a monolithic system. This is particularly prominent when dealing with HTTP requests with large amount of internal communication calls. Using cache will in general increase the performance of the application but depending on the behaviour of the application it maybe not necessary. Also here will HTTP requests which use many internal requests have must profit from using caching.

10 FUTURE WORK

This report have mostly focused around measuring performance for two architecture types that are under constant load from requests. Another interesting aspect would be to examine the patterns from zero load up to constant load under the same conditions and set up. This would be especially interesting with regards to scaling and monitoring of a system. It could also be that a company could accept a lower performance, but with a better behaviour before reaching maximum capacity.

REFERENCES

[1] Bergman, B., and Klefsjö, B. *Kvalitet från behov till användning*, 2nd. ed. Studentlitteratur, Lund, Sweden, 1995.

[2] Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S. T., and Mazzara, M. From monolithic to microservices: An experience report from the banking domain. *Ieee Software 35*, 3 (2018), 50–55.

[3] Clark, K. Microservices, soa, and apis: Friends or enemies?, 2016.

[4] et al., D. N. *Microservices: yesterday, today, and tomorrow*, 1st ed. Springer International Publishing, Cham, Sept. 2017.

[5] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)* (2015), IEEE, pp. 171–172.

[6] Fielding, R., and Taylor, R. *Architectural styles and the design of network-based software architectures*, vol. 7. University of California, Irvine, 2000.

[7] Flygare, R., and Holmqvist, A. Performance characteristics between monolithic and microservice-based systems. Master's thesis, , Department of Software Engineering, 2017.

[8] Foundation, A. S. Apache jmeter - load test applications and measure performance, 2004.

[9] Gouigoux, J., and Tamzalit, D. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (April 2017), pp. 62–65.

[10] Jiang, Z. M. Automated analysis of load testing results. In *Proceedings of the 19th international symposium on Software testing and analysis* (2010), pp. 143–146.

[11] Lewis, J., and Fowler, M. Microservices - a definition of this new architectural term, 2014.

[12] Menascé, D. A. Load testing of web sites. *IEEE internet computing 6*, 4 (2002), 70–74.

[13] Newman, S. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[14] Ueda, T., Nakaike, T., and Ohara, M. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)* (2016), IEEE, pp. 1–10.

[15] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., and Gil, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (2015), IEEE, pp. 583–590.

[16] Young Hoon Kwak, F. T. A. Benefits, obstacles, and future of six sigma approach. *Technovation 5*, 6 (May 2006).