Migration from Monolith to Microservices : Benchmarking a Case Study

Preprint · March 2020

DOI: 10.13140/RG.2.2.27715.14883

5 authors, including:

Antonio Bucchiarone
Fondazione Bruno Kessler
132 PUBLICATIONS **1,522** CITATIONS
SEE PROFILE

Manuel Mazzara
Innopolis University
297 PUBLICATIONS **2,300** CITATIONS
SEE PROFILE

Nicola Dragoni
Örebro universitet, Sweden, and Technical University of Denmark
138 PUBLICATIONS **2,302** CITATIONS
SEE PROFILE

Schahram Dustdar
TU Wien
78 PUBLICATIONS **1,900** CITATIONS
SEE PROFILE

Migration from Monolith to Microservices : Benchmarking a Case Study

Nichlas Bjørndal*, Manuel Mazzara†, Antonio Bucchiarone‡, Nicola Dragoni*, and Schahram Dustdar§
* DTU Compute, Technical University of Denmark, s173086@student.dtu.dk, ndra@dtu.dk
† Innopolis University, Russia, m.mazzara@innopolis.ru
‡ Fondazione Bruno Kessler, Italy, bucchiarone@fbk.eu
§ TU Wien, dustdar@dsg.tuwien.ac.at

***Abstract*—Migrating from a monolith architecture to a microservice architecture in order to modernize a system seems to have become popular in the recent years. However, the benefits of this migration does not appear to have been sufficiently verified. The goal of this article is to present a methodology able to verify whether or not a migration to microservices is beneficial. The methodology is derived starting from a systematic literature review useful to survey and analyze existing approaches in the field and to identify a set of metrics (i.e., latency, throughput, scalability, CPU usage, memory usage, and network usage) needed to benchmark the source and the target applications. We evaluate the solution proposed conducting benchmarking experiments on a reference system we have developed in two versions, monolith and microservices. We conduct small-scale experiments using consumer-grade hardware, and large-scale experiments using cloud services.**

I. Introduction

Microservices [6, 16, 34] is an architectural style originated from Service-Oriented Architectures (SOAs) [29] in order to bring *in the small* (within an application) those concepts that worked well *in the large*, i.e., for cross-organization business-to-business workflow. Monolithic architectures follow a modularization abstraction which relies on the sharing of resources of the same machine (memory, databases, files) where the components are not independently executable. A limitation of monoliths is *scalability*, and in general all the aspects related to change [27]. In the microservice paradigm, a system is structured by composing small independent building blocks, each with a dedicated persistence tool and communicating exclusively via message passing. In this kind of organization, the complexity is moved to the level of coordination of services. Each microservice is expected to implement a single *business capability*, which is delivered and updated independently. With this approach discovering bugs or adding minor improvements do not have any impact on other services and on their releases. In general it is also expected that a single service can be developed and managed by a single team [36, 16]. The idea to have a team working on a single microservice is rather appealing: to build a system with a modular and loosely coupled design, one should pay attention to the organization structure and its communication patterns as they, according to Conway's Law [13], directly impact the produced design. So if one creates an organization with each team working on a single service, such structure will make the communication more efficient not only on the team level, but within the whole organization, improving the resulting design in terms of modularity.

Shifting towards microservices is an important subject for scientific investigation at the moment, since several companies are involved in a major refactoring of their back-end systems in order to accommodate the advantages of the new paradigm. Our recent works ([11], [31]) describe the process of modernization of an old system, by migrating from a monolithic architecture to a microservice architecture based on the hypothesis that microservices performs better than a monolith. The migration turned out to be successful, but one of the major critique-points of the articles was the lack of verification of the migration being beneficial. The system was proprietary and thus the verification was infeasible both due to practical and legal reasons. To perform such an verification it would be beneficial to have a self-built and, preferably, open source, system. This means that in order to evaluate the migration two artifacts are needed: (1) *a methodology for comparing two systems* and, (2) *two reference systems for comparison*. The lack of migration evaluation represents a gap in this research. In this paper we aim at overcoming this problem benchmarking a system developed and migrated by us on which we have full access.

In this paper we propose a methodology organized over four steps that are summarized below. The article also follows the same structure:

1) A *Systematic Literature Review (SLR)* is used to evaluate software architectures via benchmarking and to identify a suitable metrics-suite (described from Section II to Section IV).
2) The identification of a *reference system* and its implementation both as monolithic and microservices-based systems in order to perform the benchmarking (presented in Section V and in Section VI).
3) The realization of two *benchmarking experiments*: one of small scale running on a local machine (described in Section), and one of large scale running in the cloud (described in Section).
4) An *analysis* of the results obtained in the executed experiments to determine whether or not the migration was beneficial (presented in Section VIII.

The article is concluded with some open issues raised by the approach in Section IX, while Section X concludes the

article with final considerations.

II. Benchmarking Methodologies Review

The goal of this section is to identify the benchmarking methodologies used to measure the performance of software architectures. We have used the guidelines presented in [37] to identify the methods and metrics to compare monolithic vs microservice architectures and to test the hypothesis of microservices performing better than monoliths. To better classify the existing benchmarking practices the following research questions have been identified:

- **Research Question 1**: What benchmarks exists within software engineering?
- **Research Question 2**: Which benchmarks metrics are currently used within software engineering?

Answering these questions will help determining a good approach for comparing system implemented using different software architectures.

In the follow we will describe the search strategy that has been used to construct the Systematic Literature Review.

The PICO [38] method was used to chose which search strings to query academic databases based the research questions mentioned above:

> **Population**: The population for this article is *benchmarking* which are used within the field of *software engineering*.
> **Intervention**: The intervention for this article is *methodologies* and *metrics* for benchmarking within *software engineering*.
> **Comparison**: The different studies are compared by their different benchmarking methodologies and their metrics.
> **Outcomes**: We present the different benchmarking methodologies and metrics, which allows us to discuss which kinds of metrics are best suited for comparing Monoliths and Microservices.

The usage of PICO resulted in the following *keywords* and their *aliases*:

- Benchmark (Benchmarking, Performance Measurement)
- Software Engineering (Software)
- Software Architecture (Service Oriented Architecture, Software Architecture, Microservice, Service Oriented Architecture, Service Oriented Computing)

These keywords were used to create the following academic database queries:

> **[Query 1]** TITLE ((”Benchmark” OR ”Benchmarking” OR ”Performance Measure”) AND (”Software Engineering” OR ”Software” OR “Computer Science”) AND (“Microservices” OR ”Service Oriented Architecture” OR “Service Oriented Computing”))
> **[Query 2]** (”Benchmark” OR ”Benchmarking” OR ”Performance Measure”) AND (”Software Architecture”)
> **[Query 3]** (”Benchmark” OR ”Benchmarking” OR ”Performance Measure”) AND (”Software Architecture” OR “Microservice”)
> **[Query 4]** (”Benchmark” OR ”Benchmarking” OR ”Performance Measure”) AND (”Software Engineering” OR ”Software”) AND (”Microservice” OR ”Software Architecture” OR ”Service Oriented Architecture” OR ”Service Oriented Computing”)

All the four queries focuses on benchmarking software systems with different scopes. Query 1 and 4 are focusing on more specific areas of software architectures, whereas query 2 and 3 are more wide-searching. After evaluating the four queries we decided that the query 4 was most suited to answer the research questions as it strikes a good balance between generality while including important keywords such as ”Benchmarking”, ”Microservices”, and ”Service Oriented Architecture”.
After the query was selected, we used it on three popular academic databases: Scopus[1], IEEE Xplore Digital Library[2], and ACM Digital Library[3]. The query yielded the results seen on Table I.

Database	Results
Scopus	594
IEEE	281
ACM	172

TABLE I: Search results from the different digital libraries.

This query results in a total amount of articles of 1047 for further filtering and analysis. First step of the filtering was to remove duplicate entries, which reduced the dataset to 949.
After that, the articles were filtered based on the titles and abstracts with inclusion and exclusion criteria in mind. The following *inclusion and exclusion criteria* were used:

- Article is related to benchmarking or measuring a software system's performance.
- Articles that deal with actual measurement and not predictions.
- Article is published between 2010-2019.

To limit thee scope the following *exclusion criteria* was imposed:

- Article is not peer-reviewed.
- Article is not in English.
- Article's focus is CPU architecture.
- Article's focus is OS Kernels.
- Algorithm benchmarks.

The inclusion/exclusion criteria attempts to narrow down the scope to focus of the review towards user space applications such as web servers. This left 85 articles for further in-depth reading. After going the 85 articles, 33 were deemed applicable regarding *benchmarking of software architectures*. Data were then extracted from the 33 articles as data-items. One article can result in multiple data-items e.g. The same article both containing the metrics CPU usage and Throughput as seen in Table III.

The selected articles for the review can be seen on Table II. Note that articles which contains multiple types of metrics are mentioned in multiple rows.

[1]https://www.scopus.com/search/form.uri?display=basic
[2]https://ieeexplore.ieee.org/
[3]https://dl.acm.org/

Metric Type	Articles
Performance	[40] [19] [39] [5] [7] [8] [30] [42] [44] [21] [22] [25] [35] [20] [43] [23] [2] [10] [45] [41] [17]
Availability	[19] [7] [8] [32] [17]
Security	[18] [4] [14]
Scalability	[7] [43] [24] [45]
Consistency	[7] [45]
Quality	[9] [12] [33] [3] [15] [14]
Cost	[19]

TABLE II: Articles chosen in the Review.

The dataset could be expanded and possibly improved by performing reverse snowballing as recommended in [48]. However due to the limited scope of this article, the initial result of 33 articles were deemed a satisfactory sample-size to evaluate benchmarking techniques and trends. Another *threat to validity* is that the article evaluation and data extraction was performed by only one author. This means there is a certain bias when selection articles, and data being extracted from the chosen articles. However this bias can be somewhat mitigated by carefully following the review process by trying to be as objective as possible when choosing research questions, keywords, queries and inclusion/exclusion criteria.

Finally, data was extracted and structured into data items as seen in Table III. Data-items contains Metric Name, Metric Type and Benchmark Type, as these attributes were deemed most important for metric identification. These data items form the foundation of the benchmark design.

Metric Name	Metric Type	Benchmark Type
CPU	Performance	Measurement
Throughput	Performance	Measurement

TABLE III: Data items example.

a) Metrics Extraction.: The metric names are extracted primarily based on the vocabulary used in each article. In the case of similar concepts having been described by different terms e.g. ”requests per second” and ”throughput” the different terms will be merged into a single term, throughput, in order to simplify the data processing.

b) Metric Type.: The types of metrics are inspired by [7] to define the following types of metrics: Performance, Availability, Security, Scalability, Consistency, Cost, Quality. Through the SLR process the Quality metric type was identified, a type which represents metrics describing the quality of the code which software systems are built with.

c) Benchmark Type: The types of benchmarking was divided into Measurement(practical), Analytical (theoretical) and Analytical + Measurement (mixed). A fourth type, Simulation, could be added but it was excluded from the scope of this thesis, as it doesn't tell about the actual system, but rather a prediction.

d) Data analysis: The research questions now be attempted to be answered as the data has been gathered from the selected articles. This will be achieved by first investigating how often the different benchmarking types are used, and after that shift focus to the different benchmarking metrics. The raw article and metric data found at GibHub[4] [5].

A. Research Question #1

The frequency of the three different benchmarking types can be seen on Figure 1. The type ”Measurement” is, as expected, clearly the most popular by occurring 83.8%. This could be expected, as with experiments in many other scientific fields, getting data from an actual system is an desirable way to achieve a better understanding of the subject matter.

The type ”Analytical” is far less used at an percentage of 12.5%. This can possibly be explained by the fact that analytical/more theoretical benchmarking is abstract and thus more difficult to design and measure. However this type has some very interesting prospects, as it has the potential to by-pass the creation of a benchmark-suite, by looking straight at the source code. Some articles investigated benchmarking systems at design-time, meaning that certain performance metrics can be calculated before a single line of code is written. This is however very new area, with room for further investigation. The design-time analysis also borders to the aforementioned ”simulation” type.

The mix of Analytical + Measurement appears to be infrequently used at 3.8%, however it is applicability should not be underestimated as it has the possibility of drawing the best of both worlds by both looking at metrics for a running system's performance, and combining them with analytical metrics for the same system.

B. Research Question #2

To answer the second research question, a closer look must be taken at the metrics identified by the data items. First, a more general look at how often the different types of metrics are used must be taken, before diving down to look at each individual metric.

On Figure 2 it can be seen that the performance metrics are most widely used (62.5%). This is expected due to the dominant usage of Measurement benchmarking, as many of the performance metrics are also classified as measurement types. This could further indicate that in academia there is a strong preference for clearly measurable properties.

Availability appears 10.0% of time and is another important metric type. These type of metrics influences results in how often a user can expect to lose access to a system. This type of metric however shows a system characteristic over a longer period of time rather than a a view of the running system right this moment.

Quality metric types appears 10.0% of the time, and are primarily used to gauge the quality of the source code from which the system is built with. This metric can both indicate

[4]https://github.com/NichlasBjorndal/LibraryService-Appendices/tree/master/SLR/Articles

[5]https://github.com/NichlasBjorndal/LibraryService-Appendices/tree/master/SLR/Metrics

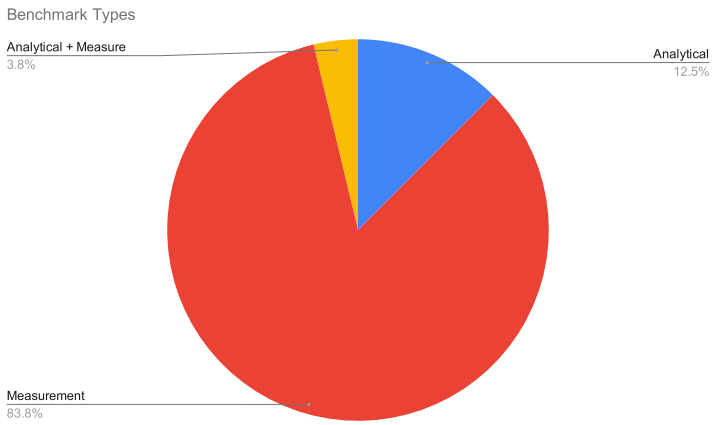

Fig. 1: Chart of the distribution of Benchmarking Types

the performance of architectural choices and the skill level of the developers writing it.

Scalability is an important type of metric for modern systems where thousands or even millions of concurrent users can attempt to use the system. This requires both hardware, software architecture and code to be designed in a way which supports that many users. Thus it was a bit of a surprise initially that the type of metric only appears 8.8% of the time, however this can be perhaps be explained by the complexity of the measurement process.

Security is an important metric but has a quite low usage rate of 3.8%. The low usage can possibly be explained by the high complexity of benchmarking and it not being prioritized stakeholders.

The complexity of security measurements can come from security evaluations still being partially an art. Some security scores can be extracted from looking a source code and interfaces as seen in some of the articles in this SLR, but it still might not give a complete view of a system's state of security.

Data Consistency was one of the least used metric type at 2.5% usage, however this might rise in the future as distributed systems and distributed databases become more used in the future.

Cost, or rather hosting cost, is the least popular metric type with only occurring in one of the articles during the SLR.

After having looked at the different kind of metrics, the individual metrics found in the articles can be identified. The frequency the individual metrics are depicted on Figure 3.

1) Latency: Latency is the time delay measured between a response and request, this is particular important for communication between a client and server. This metric is primarily impacted by network speed and latency + the amount of time it takes for the server to process the request. This metric was the most popular in the examined articles, and with good cause as it relatively easy to measure, and it has a big impact on the end-user experience.

2) CPU: CPU Utilization is another popular metric, and understandably so. CPU usage is one of the indicators of how well the software system is able to utilize the host machine's resources. It can also be used to indicate code's efficiency, by comparing similar functionality different ways and comparing them, a lower CPU usage is better.

3) Throughput: The throughput of a system is how many actions/requests a system can process given a workload and hardware resources. E.g. a system that can process 100 requests per second performs worse than a system that can process a 1000 requests per second given the same resources. This metric is related closely related to *Latency* and *Scalability*.

4) Network: The Networking metric indicate how many networking resources a system uses. This becomes particular important when you have a set of Microservices which all communicate together over HTTP/HTTPS, and thus a high usage of network can impact the *Latency* of a system.

5) Scalability: When building a system to serve a few concurrent users, scalability might not be a concern. However, when a system must handle a plethora of users, it is important the that the system is built handle the extra work, that it can *scale-up*. The *Scalability* of a system is typically measured by how many requests can be processed while staying below a set *Latency* threshold e.g. how many requests can a system process before the latency rises above 200 ms. When it comes to Scalability in the cloud, the term *Elasticity* is often used when talking about how fast a cloud system can increase/decrease computing resources to allow a system to scale up or down to meet demand.

6) Memory: Memory is similar to *CPU* usage as it indicates well much of the systems RAM is being used, and thus the efficient of the system/source can be determined. This metric is becoming more important as virtualization such as virtual machines and containers becomes more prominent as these technologies are generally more memory demanding.

7) I/O: I/O (Input/Output) describes how much the system utilizes its host's input/output devices, e.g. read/write operations to disks or network cards.

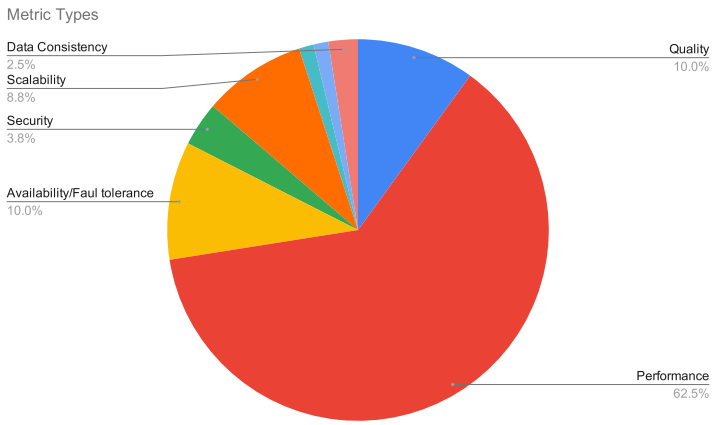

Fig. 2: Chart of the distribution of Metric Types

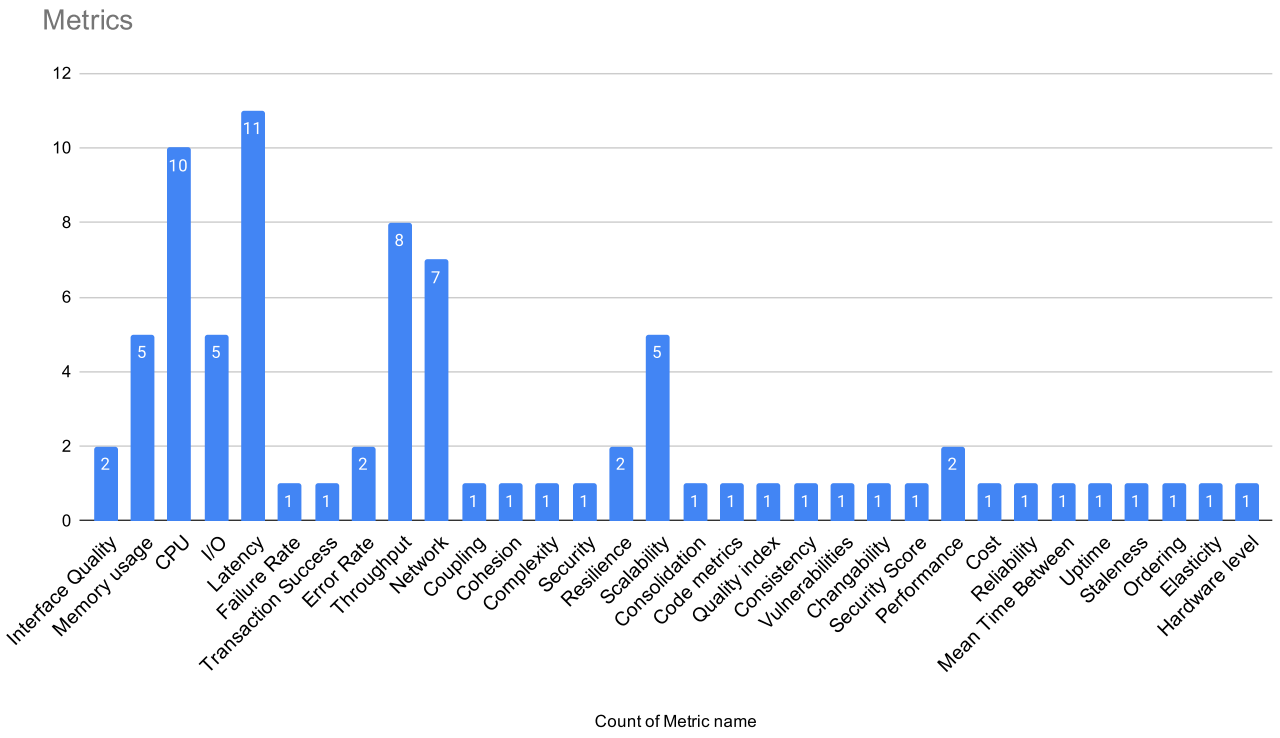

Fig. 3: Chart of the distribution of Metric Names.

8) Fault Tolerance: Fault Tolerance is used an umbrella term for the following metrics: Failure Rate / Error Rate, Transaction Success, Resilience/Reliability, Mean Time Between Failures, Uptime. In general these metrics how a system handles something going wrong. This is usually measured by investigating the amount errors, their frequency, and their impact on the system e.g. can a web server recover from an exception being thrown or is a reboot required?

9) Code Quality: Another way to analyze a system is by looking at the quality of the source code/system design. Through the SLR a bunch of different ways to analyze source code has been found: Interface Quality, Coupling, Cohesion, Complexity, Changeability, Quality Index.

10) Security: Evaluating security can take many forms. It is often done by security experts analysing the system or pen testers[6] who will actually try to break into the system. In the examined articles two security metrics were found, a Security Score and amount of vulnerabilities.

The *Security Score* is measured performing a static analysis on the entire application on build time. The score is then computing by a tool which compares the system with a repository of good architectural and coding practices. However

[6]https://www.doi.gov/ocio/customers/penetration-testing

the article does not compare the static score with a professional evaluation of the same system.

The amount of *Vulnerabilities* metric is a number calculated by a proposed service which analyzes metadata files and application byte code. This is achieved by simulating an application's life cycle and looking at entry points and callbacks, analyzing dependencies, and information flow analysis.

11) Consistency: Data consistency was measured by two different ways in the examined articles: staleness and ordering. In a distributed database there is often multiple replicas of a piece of data for performance, backups etc. *Staleness* describes how far behind a piece of data from one database is compared to its replicas.

Ordering is a metric that describes whether there consistency in the order of events occurring, e.g. will the order of events happening client-side also reach the server/be processed by the server in the same order.

12) Hardware level: In one paper they investigated hardware level/CPU level metrics such how many CPU instructions it takes to handle a request, and the frequency of CPU data cache miss.

13) Cost: While the cost type did not occur a lot in the surveyed papers, its importance should not be underestimated as cost is often one of the primary factors when stakeholders are making business decisions, decisions which end up dictating how a system should built.

Now that data have been extracted through the SLR, it is time to take a closer look at the identified metrics and their applicability. The usage of the different metrics can be seen on Figure 3. The most popular metrics and their occurrences can be seen on Table IV.

Metric	Frequency
Latency	11
CPU	10
Throughput	8
Network	7
Scalability	5
Memory	5
I/O	5

TABLE IV: Frequency of the most popular metrics from the SLR.

This matches with the popularity of Measurement / Performance observed earlier, as 6 out of 7 metrics fall in both categories, indicating that generally performance related metrics are favored by the research community. Particular the metrics *Latency*, *CPU*, and *Throughput* seem to be quite suitable as they all rank highly on Table IV. Other notable metrics from the list includes *Scalability* and **Network** as those metrics as modern systems need to support more and more users in a global world, often provided over the internet.

III. Existing approaches

To further evaluate the initial set of metrics existing work is examined. Four possible reference systems for benchmarking is identified in [1]: Acme Air, Spring Cloud Demo Apps, Socks Shop, and MusicStore.

However only Acme Air[7] and MusicStore[8] has both a monolithic and microservice architecture version for benchmarking, which makes these two the only ones eligible for consideration. However, MusicStore is now archived on GitHub, indicating maintenance and further development of the project has been halted.

- **[Reference Systems A]** Workload Characterization for Microservices [43]
- **[Reference Systems B]** Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud [46]

1) Reference Systems A : In this article researchers from IBM Research - Tokyo uses the a monolithic and microservice version the sample application Acme Air. They use the benchmark to perform both a performance, runtime and hardware-level analysis comparing the two systems (implemented in a Java and Node.js[9] version): Throughput, Scalability, Path-Length[10], CPI[11], Data Cache Miss Cycles, Code execution time per transaction.

2) Reference Systems B: In this article the authors perform a case study together with a company to build a monolithic and microservice reference systems, primarily to support two services. The first service implements a CPU heavy service with a relatively long processing time. The second service is doing a read from a relational database with a relatively short response time.
The comparison of these two services consist of throughput, the average and maximum response time for a set amount of requests per minute, the cloud hosting cost defined as Cost per Million Requests.

IV. Survey

A small survey was another step taken to evaluate to choice of metrics. This survey was an attempt to gain inside from real-life stakeholders such as a customer and developers point of view.

A. Survey Design

The goal of the survey was to determine what kind of benchmarking metrics are important among different stakeholder roles. To determine this, an initial question was asked in free-text form about which metrics they deem important. Then as a follow up question they were asked to rank the importance of the most popular metrics from the SLR study on a scale from 1 to 5. The questions were:

1) What is your role in the company?
2) Which metrics/qualities of a software system are most important to you?
3) Some of the most popular metrics are listed below. Please rank them according to how important you find

[7]https://github.com/acmeair/acmeair
[8]https://github.com/aspnet/MusicStore
[9]https://nodejs.org/en/
[10]amount of CPU instructions
[11]average amount of CPU clock cycles to complete an instruction

them regarding software systems. *Latency, CPU utilization,Throughput, Network, Scalability, Memory usage, I/O, Hosting Cost, Security, Code Quality*

The survey was designed to be short and compact to motivate people to participate in the survey, while avoiding question fatigue which could reduce the answer quality which is a documented consequence of too long surveys [28].

The survey was kept to a small scale where the participants where handpicked from the author's current place of employment: NNIT A/S and one of its clients. This was to ensure a more personal touch and to maintain quality of the participants. however this comes at the cost of potential bias, a low sample size.

While the survey will not bring definitive answers it can further give an indication of what the software industry values through empirical software engineering. To get better results, a more thorough survey with a bigger sample size across projects and companies is needed in the future.

B. Survey Results

The survey were given to four carefully selected stakeholders: 1 Systems Architect, 1 Product Owner and 2 developers. The people interviewed are all from a project building a system which is serving the national prison system with a approximately 4000 registered users. While the survey asks about metrics in general it is assumed that survey participants carry some kind of a bias toward the project they are currently working on. The survey data can be seen at GitHub[12].

Which metrics/qualities of a software system are most important to you?

Looking at the free text answers to this question there's four general answers: Latency, Throughput, Code quality, and more "soft" metrics such as user experience/user needs. This reinforces the importance of Latency and Throughput, but also gives more important to Code Quality when the participants talk about maintainability and development speed. The survey participants also brings up metrics which are harder to quantify such as user experience and meeting user needs.

Some of the most popular metrics are listed below. Please rank them according to how important you find them regarding software systems.

Looking at the average score (with "don't know" being filtered out), it can be seen that again the *Latency*, *Throughput* and *Scalability* are ranking highly. However *Security* and *Code Quality* are also ranked very high, despite not being very popular in the SLR. This could indicate a gap in existing literature and potential new areas to research further. The average rating given to each metric in the survey can be seen on Table V.

[12]https://github.com/NichlasBjorndal/LibraryService-Appendices/tree/master/Survey

Metric	Average Score
Security	5.00
Latency	4.75
Code Quality	4.33
Throughput	4.00
Scalability	3.33
Network	3.00
Memory	3.00
CPU	2.67
I/O	2.67

TABLE V: The average ranking scores of the metrics in the survey

V. The Case Study

In order to compare two different software architectures, monolith and microservice, a benchmarking experiment has to be conducted according to specifically identified metrics. The experiment has to be performed on a suite of reference systems, each structured according to one of the architectures we want to test. In this article, we decided to design and implement our own references systems; first to design them as closely as possible to the current best practices, second to have full access and control over the software artifacts, problems that prevented us to perform full benchmarking in our previous migration experience [31]. We realized a *Library Management System* by building the *LibraryService* in both the monolithic and microservice versions. This system must be able to be used by regular library users, which should be able to sign-up and borrow books. The system's functionalities are described by the use case of Figure 4.

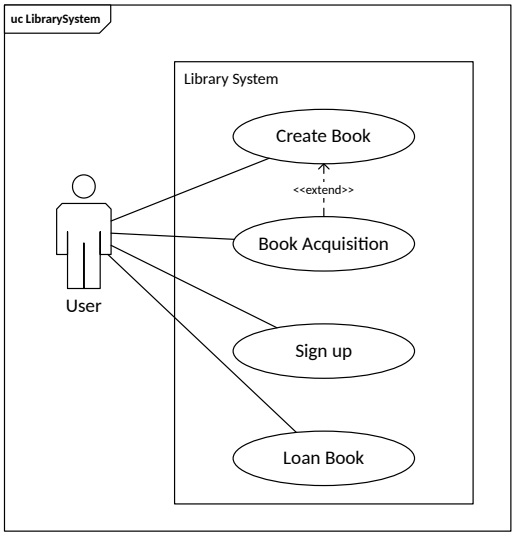

Fig. 4: Use Case Diagram for *LibraryService*

We decided to primarily focus on performance measurement metrics. **Latency** and **Throughput** appear in the popularity list, in the existing approaches and in the survey, and thus they are the most important to be included.

Scalability was identified as another important metric both in the literature and existing works, as it shows how well a system can handle fluctuations in number of users.

Hardware-near metrics such as **CPU**, **Memory** and **Network** were also deemed important enough to measure as they describes how the system resources are being spent.

Security and **Code Quality** metrics have also been considered notable. However, their use go beyond the scope of this article.

The aim of the reference system is to model and build a real-world-like system implemented according to best industry practices in two versions: the older approach of monolithic architecture and the newer microservice architecture version. Both versions expose the functionality described in Figure 4, so that on a user-level it is the same system, while their implementations differ. To better focus on the performance differences on an architectural level the amount of variables have been kept to a minimum, but by using the same technologies and tools as much as it made sense: to use the same programming language in both versions.

Based on the use case a simple domain model with the following entities has been identified: *User*, *Book*, *Loan*, *Author*, *PhysicalBook*, *Order*. These entities will serve as the foundation for the database design and the service endpoints.

A. Monolithic System

The monolithic system is modelled after older enterprise systems where the whole application runs in one process, tightly coupled to one tech stack and connected to a single database. This version of the *LibrarySystem* is built with ASP.NET Core[13] Web API using .NET Core 3[14]. It is connected to a SQL Server[15] database which contains tables for all the entities. The interaction between the database and the web API is done through the Entity Framework[16] - an Object-relational mapping (ORM) tool, which provides the Create, Read, Update, Delete (CRUD) operations.

To better model older systems it was decided against placing the monolith inside a container, even though it would be possible to do so. This was because it is often too expensive for the stakeholders to bring older systems up to date in order to *"Dockerize"* them.

B. Microservice System

The microservice version of the *LibraryService* is built on the same domain model as the monolithic version, however, instead of one application, it consists of four microservices. These services are called: *BookService*, *UserService*, *LoanService*, and *OrderService* and each of them is responsible for a part of the domain entities, their own bounded context.

The microservices are also built with ASP.NET Core Web APIs and Entity Framework, as it was intended to keep performance variables such as different programming languages out of the way. It was, however, intended that one of the microservices should be implemented in Java, a language very close to C# both performance and language-wise, but due to time constraints it was not implemented.

Each microservice has its own dedicated SQL Server database allowing each service to scale-up and down their different data stores as need arises.

The microservices run in Docker[17] containers, yielding a Docker image[18] per microservice which are stored in registries such as Docker Hub[19] and Azure Container Registry (ACR)[20]. This allows an easy distribution of the microservices services, by allowing them to be running in the same environment across different machines and hosts.

Docker containers can be run on their own without any orchestration tools, but it can quickly become a difficult task to manually keep the containers up and running in the right condition. To fix this, Kubernetes has been used to orchestrate the microservices allowing for easy horizontal scaling, load balancing, health checks, and much more.

Each service has its own deployment[21] which is responsible for meta data, pulling the Docker image from the registry, internal and external communication and replication(ensuring multiple pods[22] are running).

Each deployment is then exposed to external network through a Kubernetes service[23], which routes external traffic to Kubernetes pods, which can handle the request. Kubernetes offers load balancing[24] [26] which attempts to ensure the load is even spread out among the replicas of the microservices. Each deployment is also set up to perform a periodic liveliness check. If a replica fails a livenesness check Kubernetes will shut down the pod, and create a new one.

Locally the load balancing is achieved by using Ingress[25] which is an alternative way of handling load balancing by routing the external requests to inside the Kubernetes cluster.

To better decouple the microservices, an Azure Service Bus Message Queue[26] is used for asynchronous communication between *Book Service* and *Order Service*, and to illustrate the performance impacts of using message queues over HTTP(S) communication between services.

The typical structure of a *LibraryService* microservice is a ASP.NET Core Web API image containing all the dependencies needed to run it. Each service connect to its own SQL Server database which is only handling data related to that particular service.

This application is built into an image through a Docker file[27] and Docker Compose[28]. The image is then used by Kubernetes to run multiple instances of each microservices to handle redundancy, scalability etc.

This implementation consist of four microservices: *BookService*, *UserService*, *LoanService*, and *OrderService*.

[13]https://docs.microsoft.com/en-us/aspnet/
[14]https://docs.microsoft.com/en-us/dotnet/core/about
[15]https://www.microsoft.com/en-us/sql-server/sql-server-2019
[16]https://github.com/aspnet/EntityFrameworkCore
[17]https://www.docker.com/
[18]https://docs.docker.com
[19]hub.docker.com
[20]https://azure.microsoft.com/en-us/services/container-registry/
[21]https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[22]https://kubernetes.io/docs/concepts/workloads/pods/pod/
[23]https://kubernetes.io/docs/concepts/services-networking/service
[24]https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer
[25]https://kubernetes.io/docs/concepts/services-networking/ingress/
[26]https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions
[27]https://docs.docker.com/engine/reference/builder/
[28]https://docs.docker.com/compose/

VI. System Deployment

The *LibraryService* is deployed in two ways: a local version running on a desktop computer, and a cloud version running in Azure. In this section we will describe the technical details of both the deployments.

A. Local Version

The server machine is a desktop PC which consist of 8GB DDR3 1600 MHZ RAM, an Intel i5-4590 @3.3 GHZ quad-core CPU and a 250GB SSD. The server runs both the web API and the database server. The client machine is a laptop which consist of a 8GB DDR 3200MHZ RAM, an Intel i5-7300HQ @2.50 GHZ quad-core CPU, and a 250GB SSD. The two machines are connected by a WLAN.

The local version is deployed on a desktop computer running Docker Desktop[29] which also has a built-in version of Kubernetes.

All the images are hosted on Docker Hub registries which the local Kubernetes pulls down and runs. Kubernetes is configured through a series of deployment and service files: one deployment and service file for each service.

For the local load balancer an ingress solution by NGINX[30]. For this to work locally the kubernetes service is configured as NodePorts on port 80, and lets the ingress controller do the correct routing. The source code of the monolith and microservice system version deployed locally can be found at GitHub[31] [32].

B. Cloud Version

While the desktop deployment might work as a proof-of-concept, it does not have the hardware needed for supporting large user bases. For this a professional-grade hosting solution is needed such as a data-center or a cloud service. In this work Microsoft Azure[33] was chosen as the cloud provider as it is also made by Microsoft like other technologies used for this work such as C#, .NET Core and SQL Server allowing for an easy integration. Another important reason to chose Azure is the authors having previous experience with the platform, and Azure is offering $100 in cloud credits for students. However Amazon's cloud service[34] AWS or Google's cloud service Google Cloud[35] could also have been used.

C. Monolith Cloud Deployment

The monolith cloud version is built with Azure App Service[36] to host the web server. App Services generally allows to run 1 - 30 instances of the the web server depending on the virtual machine that is hosting it. The app service instances connects to a Azure SQL Database[37] which is setup as a serverless setup in such rather than running in its own fully dedicated virtual machine in order to reduce the spending of cloud credits. The database is setup up to run with six virtual cpu cores (vcores[38]), 32 GB of storage and a maximum of 18 GB memory. This version was hosted on two different hosting configurations[39]. The first configuration is with a P1V2 service plan with 1 CPU core, 3.50 GB memory and with 12 server instances. The second configuration is with a P3V2 service plan with 4 CPU core, 14 GB memory and with 30 instances (maximum amount of instances). An overview of this can be seen on Figure 5 which shows the monolithic app running on Azure's App Service in multiple instances. The source code of the monolith system deployed in the cloud can be found at GitHub[40].

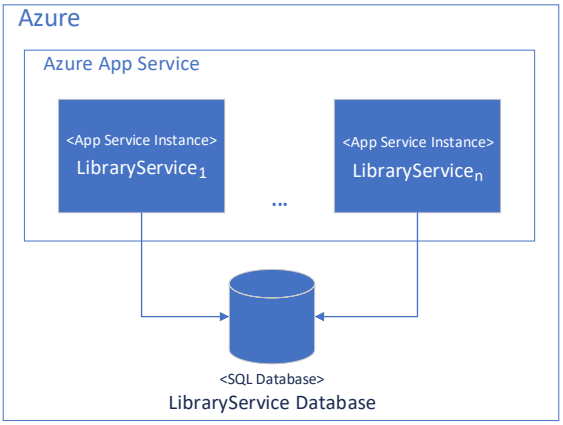

Fig. 5: System Diagram of the Monolithic Version deployed on the cloud.

D. Microservice Cloud Deployment

The microservice version of selected application is built with Azure Kubernetes Service[41] (AKS) to manage Kubernetes and its associated containers. The container images are this time hosted on Azure's own registry ACR, from which AKS can pull the containers from. The databases are configured with the same system resources as in Section VI-C. The microservice version was hosted on two different hosting configurations[42]. The first configuration is with 2 DSv2 virtual machines nodes which each has 2 CPU cores, 3.50 GB memory. The second configuration is with 5 DSv2 virtual machine nodes. Each service has its own external IP which is mapped to its own subdomain[43] e.g. http://libraryservice-bookservice.northeurope.cloudapp.azure.com/ where the load balancer then will spread out the incoming requests amongst the nodes and their pods. An overview of the microservice cloud version deployed in the cloud can be

[29] https://www.docker.com/products/docker-desktop
[30] https://kubernetes.github.io/ingress-nginx/
[31] https://github.com/NichlasBjorndal/LibraryService-Monolith
[32] https://github.com/NichlasBjorndal/LibraryService-Microservice-DotNet
[33] https://azure.microsoft.com/en-us/
[34] https://aws.amazon.com/
[35] https://cloud.google.com/
[36] https://azure.microsoft.com/en-us/services/app-service/
[37] https://azure.microsoft.com/en-us/services/sql-database/
[38] https://docs.microsoft.com/en-us/azure/sql-database/sql-database-service-tiers-vcore
[39] https://azure.microsoft.com/en-us/pricing/details/app-service/windows/
[40] https://github.com/NichlasBjorndal/LibraryService-Monolith-Cloud
[41] https://azure.microsoft.com/en-us/services/kubernetes-service/
[42] https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-general
[43] https://tools.ietf.org/html/rfc1034#section-3.1

seen on Figure 6 (Message queue not depicted). This figure shows the similarity to the locally deployed microservice version, however now there are multiple nodes, and now the Kubernetes services uses Azure's own load balancer to route traffic.

The source code of the microservice version deployed in the cloud can be found at GitHub[44].

VII. Benchmarking

In this section we use the previously selected metrics (see Section V) to compare the two versions of the *LibraryService* and to see if there are any notable differences. Two benchmarking experiments have been performed: one running locally and one in the cloud which is an environment closer to a more real-life system.

The data from the two experiments can be found at GitHub[45], while the metrics used during these experiments have been measured in the the following ways:

Throughput is measured by the amount of successful requests completed per second over a period of time. In this paper a successful request is defined as the HTTP response code 2xx[46] and with a latency lower than or equal to 200 ms. The higher throughput the better, the better the system.

Latency is measured by the difference in time between the client making the HTTP request and the client receiving the HTTP response. The lower the latency, the better the system.

Scalability is defined as the ratio of X% more throughput given Y% more resources [7]. The higher the scalability, the better the system.

CPU, Memory, Network These three metrics are all hardware-related metrics, which is measured by monitoring the host system and its CPU, Memory, and Network usage. In general the more efficient usage of hardware resources the better.

A. Experiment 1: Local Deployment

The first benchmarking experiment is a small scale experiment which consist of testing both versions of the *LibraryService* deployed on regular consumer grade hardware. The benchmark uses two synthetic workloads: a simple and a complex workload. A workload in this benchmark is defined as a series of HTTP requests at a steady rate, e.g. 100 requests per minute. The workloads are being generated and executed by Apache JMeter[47] (JMeter) on the client. JMeter uses a Constant Throughput Timer[48] to generate the approximate amount of desired requests over a minute. JMeter logs the sent requests and their data e.g. http response, latency, etc. which are used to determine the Throughput, Latency and Scalability metrics. The server uses Window's built-in monitoring app, Performance Monitor, to log CPU utilization, Memory usage and Network traffic.

The **simple workload** consists of two actions: get data about a specific book and create a new user. Getting the book information results in a HTTP GET request (resulting in a read operation in the database). Creating a new user results in a HTTP Post request (resulting in a write operation in the database).

The **complex workload** consists of getting information about a specific loan and creating an completed order (creating an order and a physical book). The retrieval of a loan results in a HTTP GET request requiring a join across multiple tables for the monolithic version, and multiple HTTP GET requests for the microservice version. The create order results in a HTTP Post request which results in write operations on multiple tables. For the microservice it requires communication between order service and book service through a message queue to achieve the creation of the order and physical book.

These workloads are applied to the server running the monolithic and microservice version, separately, meaning that only one version of the *LibraryService* is running at a time. Each version of the *LibraryService* is benchmarked at multiple amounts of requests per minute. The samples are in steps of 2500 requests per minute until the average latency rises above 200 ms indicating the system's maximum throughput (also refereed to as its break off point). The steps of 2500 requests per minute was chosen as a compromise between getting enough samples to find patterns, while not making the experiments too time consuming to perform. For each test iteration the Throughput, Latency, Memory Usage, CPU Utilization, and Network Usage is measured.

The monolith system is run in release mode through the *dotnet run*[49] command in PowerShell[50]. The microservice version is deployed on a local running instance of Kubernetes through Docker Desktop. Each microservice is replicated to three running instances, using a ReplicaSet[51] configured to run 4 CPU cores and 3GB RAM. The microservices which were being used during a workload were configured with nine pods, and unused services were configured with three pods.

Each table (except for the *CompletedOrders* and *PhysicalBooks*) are seeded with 15,000 entries through the Entity Framework. This value was chosen as the highest value possible for Entity Framework to seed the database doing database migration without causing a stack overflow exception.

1) Latency Results: The results for the latency metric can be seen on Figure 7. Note that the y-axes are displayed on a logarithmic scale due to the large range of latency values. The two simple workloads and complex monolithic workload performs fairly similar initially, however the simple microservice workload crosses the 200 ms threshold much earlier than its monolithic counterpart. The monolith workload does have a higher latency compared to the simple workload, but they

[44]https://github.com/NichlasBjorndal/LibraryService-Microservice-DotNet-Cloud

[45]https://github.com/NichlasBjorndal/LibraryService-Appendices/tree/master/Benchmarking%20Data

[46]https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

[47]https://jmeter.apache.org/

[48]https://jmeter.apache.org/api/org/apache/jmeter/timers/ConstantThroughputTimer.html

[49]https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-run?tabs=netcore30

[50]https://docs.microsoft.com/en-us/powershell/

[51]https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

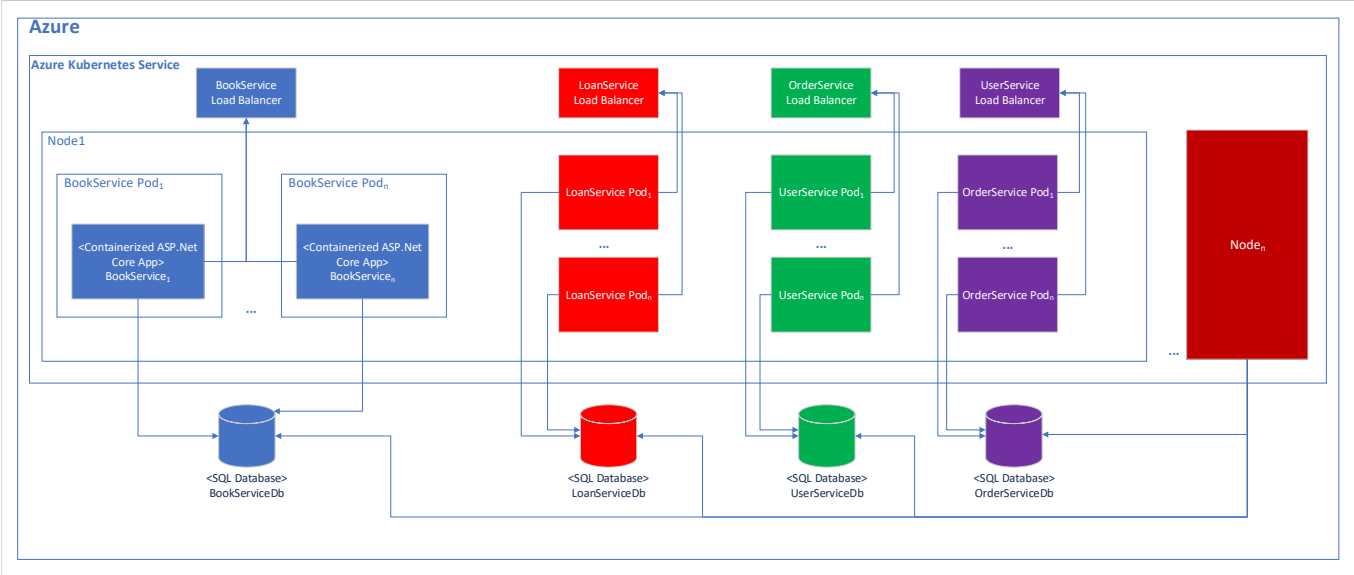

Fig. 6: Overview of the Microservice system deployed in the cloud

still follow each other until approximately 15,000 requests per minute. Lastly it might be noted that the microservice system handles the complex workload significantly worse as the average is many times higher than the three other workloads. The microservice system was also only able to handle the first two requests sizes. Comparing the two complex workloads it can be observed that it is not always favorable migrating to a microservice architecture if it is not done correctly. The general lower latency of of the monolithic system shows that the monolith performs the better for this metric.

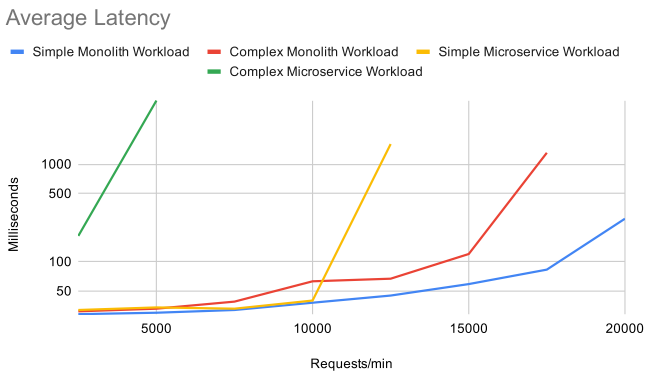

Fig. 7: Average Latency for the different workloads in experiment 1.

2) Throughput Results: The throughput can be seen on Figure 8. Interestingly, the throughput seems to inversely mirror the latency on Figure 7 with regards to the break off points. All workloads, except the complex microservice one, performs fairly similar until 10,000 requests per minute with the simple microservice workload reaching its break off point shortly after. Most notably, the monolith system performed more than 1.5 times more successful requests for the the simple workload, compared to its microservice counterpart. In stark contrast, the microservice system performed very poorly during the complex workload following the trend from earlier metrics. The higher throughput in both simple and complex workloads once again favors the monolith for the setup in this experiment.

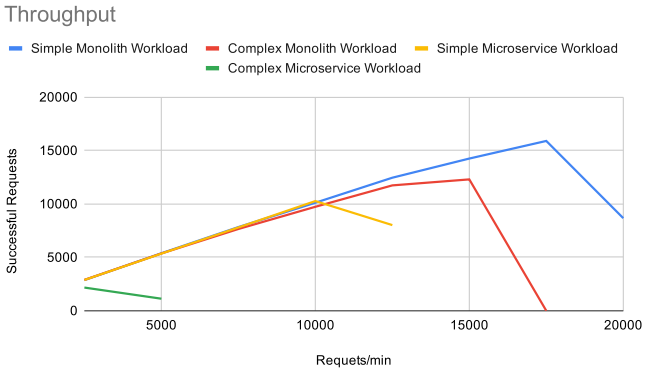

Fig. 8: Throughput for the different workloads.

3) Scalability Results: In this experiment, the increase in performance is measured by how many requests per minute a system can handle with an average latency of 200 ms. The two workloads were benchmarked a second time, this time only granting the two reference systems access to 2 CPU cores. Based on these results the scalability will be calculated as the ratio between the increase of requests per minute and the increase in hardware capacity. Note that the y-axes are displayed on a logarithmic scale due to the large range of latency values just as in Section VII-A1. Figure 9 shows the two workloads being run on a 2 and 4 CPU core on the monolith system. Here it can be seen that, the simple monolith workload with 4 cores performs the best and the complex workload with 2 cores performs the worst, as one could expect. An interesting observation is that the

simple monolith workload with 2 cores performs similar to the complex workload with 4 cores.

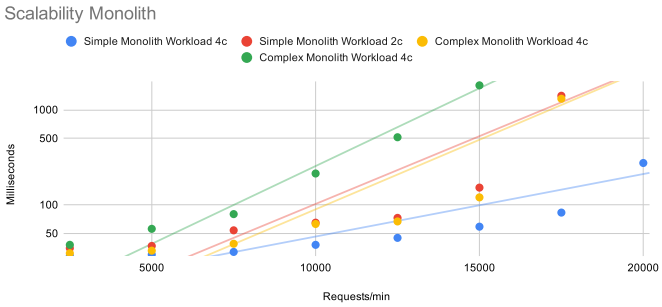

Fig. 9: Scalability for the monolith system in experiment 1.

Figure 10 shows the two workloads being run on a 2 and 4 CPU core on the microservice system. It can be seen that the simple workload with 4 cores, is the only one which the microservice system can handle at higher than 2500 requests per minute. The worst performance comes from the complex workload 2 cores, where the peak latency is 9011 ms at 5,000 requests per minute which clearly indicates that the microservice is not suited to handle a complex workload on a setup as the one used in this experiment.

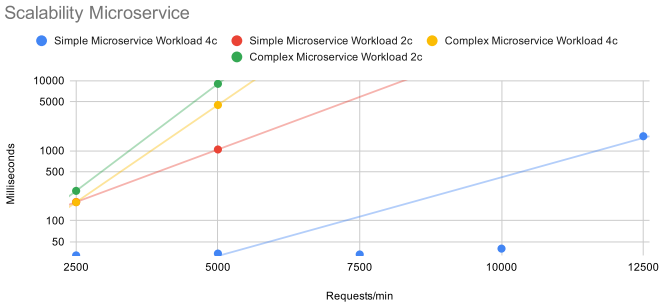

Fig. 10: Scalability for the microservice system in experiment 1.

Exponential regression has been applied to generate trend-lines, which will be used to calculate the amount of requests that can be handled given the maximum latency of 200 ms. However there is a limitation to this exponential growth, at some point the server will completely unable to process the load causing a crash. Google Sheets was used to generate the equations for calculating the requests for each workload.

The amount of requests per minute a system can handle with a max average latency of 200 ms, with a given hardware configuration and workload, can then be calculated. The results can be seen in Table VI and Table VII.

Workload	Requests/min
Simple Workload Monolith 4 cores	19,708
Simple Workload Monolith 2 cores	12,059
Complex Workload Monolith 4 cores	12,383
Complex Workload Monolith 4 cores	9,341

TABLE VI: Amount of requests per minute the monolith system can support with 200 ms latency

Workload	Requests/min
Simple Workload Microservice 4 cores	8,579
Simple Workload Microservice 2 cores	2,611
Complex Workload Microservice 4 cores	2,561
Complex Workload Microservice 4 cores	2,290

TABLE VII: Amount of requests per minute the microservice system can support with 200 ms latency

The scalability can now be calculated as the ratio between increase in requests per minute and increase in CPU cores:

$$scalability = \frac{IncreaseRequestsPerMinute}{AmountOfCPUCores} \quad (1)$$

The results from using Equation 1 can be seen in Table VIII. The complex workloads do not tend to scale well if it all. However for the microservice complex workload the system was so overworked that the results are likely somewhat misleading.

For the simple workload, the microservice scales approximately three times better than the monolith, indicating that the microservice system performs better than the monolith, as expected by the hypothesis in the Section I. Though in general the scalability ratio is expected to be a number between 0 and 1 [7] which could indicate that Kubernetes/Docker only having access to two desktop CPU cores is not sufficient and results in under-performance, skewing the number of requests per minute it can handle.

System Type	Request Increase	CPU Cores Increase	Scalability
Monolith Simple Workload	7649	2x	0.32
Monolith Complex Workload	3042	2x	0.16
Microservice Simple Workload	5968	2x	1.14
Microservice Complex Workload	271	2x	0.06

TABLE VIII: Scalability ratio for different systems and workloads.

4) CPU Results: The CPU usage of the different workloads can be observed on Figure 11. It can be seen that generally the monolithic version uses much less CPU time while handling the same amount of requests per minute compared to the microservice version, which is a preferable characteristic.
The CPU usage of simple monolith, complex monolith and simple microservice all rise proportionally until they reach their maximum throughput. The simple microservice workload can be seen falling after 10,000 requests per minute which is where it reaches its maximum throughput, however it is still far from 100% meaning that the CPU is not the only limiting factor of the system's performance. It must be noted that it was attempted with 12 pods rather than 9 to further improve CPU utilization, but this brought the system to a halt with 100% usage at a relatively low to medium amount of requests per minute (around 7,500-10,000 requests). Conversely it can be observed that the Simple Microservice Workload drops

in CPU usage once it reaches its breaking point rather than increasing like the 3 other workloads. A similar trend can be observed for network usage on Figure 13. This could indicate that the microservice system, possibly the load balancer or Kubernetes/Docker network service, is overloaded and thus stops being able to handle more network requests which results in the CPU having less computations to perform. It can also be noted that the complex workload approaches nearly 80% CPU usage, which could likely be the reason that it cannot handle more than 5,000 requests per minute. Generally the CPU usage of the monolithic system is lower which is another metric in favor of the monolith.

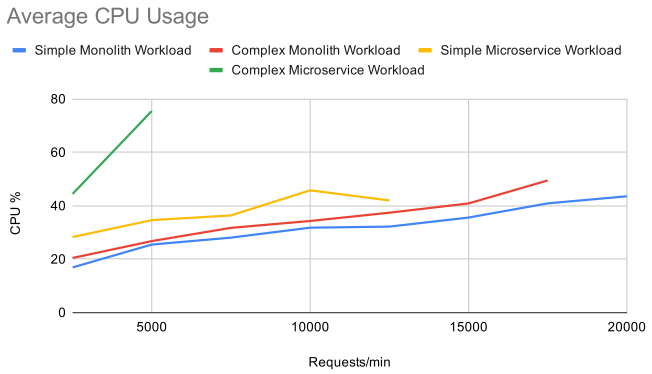

Fig. 11: CPU Usage for the different workloads in experiment 1.

5) Memory Results: The memory usage of the different workloads can be seen on Figure 12. Both systems uses a fairly constant amount of memory, with the microservice version using approximately 2 GB more of memory. This is one of the places where the overhead from virtualization really shows its impact on resource usage, as Containerization/Kubernetes uses more resources. However interestingly, the memory consumption remains fairly constant throughout the workload, except for the dip seen on the complex monolith workload which can at least partially be explained by what is described in Section VII-B.

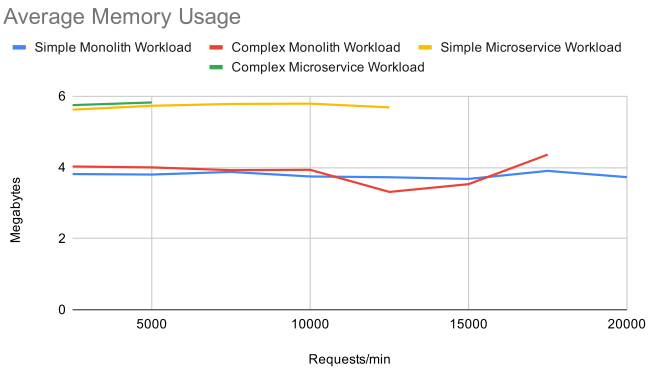

Fig. 12: Memory Usage for the different workloads in experiment 1.

6) Network Results: The network usage of the different workloads can be seen on Figure 13. The network usage grows close to linearly up until 10,000 requests per minute, except for the complex microservice workload. The complex workload clearly shows the network impact of a "chatty"[52] microservice as the network usage is much higher before the server dies off. The drop in the network usage for the complex microservice workload at 5,000 requests per minute is another indicator that the system is overloaded and cannot process all the requests.

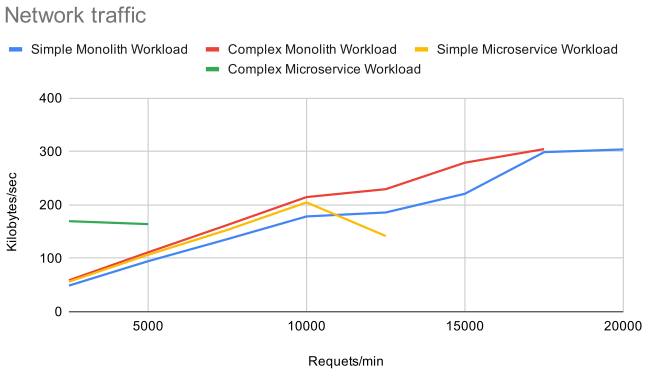

Fig. 13: Network Usage for the different workloads in experiment 1.

B. Threat to validity

The desktop PC did not run the two LibraryService systems in complete isolation, as other background processes did run in the background hence the high CPU and memory baseline for experiment 1. All the demanding (more than 5% CPU and 100 MB memory) non-Library services were shut down, but many low demand processes still ran which could have an impact on performance numbers, particularly CPU and memory usage.

Another threat to validity is the fact that the network communication was over a local 2.4 GHz WiFi network which results in a higher latency and higher risk of packet loss than a wired connection. This can potentially impact the average latency and throughput.

C. Experiment 2: Cloud Deployment

The second benchmarking experiment consists of testing the two versions of LibraryService running on production grade hardware - a medium to large scale experiment. In this experiment Azure's Performance Testing tool[53], which is a tool allowing to tests with thousands of concurrent users, will be used to test the 2 systems at scale with different VM configurations. It is determined that it was only really feasible to measure the three most important metrics with this tool: Throughput, Latency and Scalability all which the tool automatically captures. Due to the way the performance

[52]A microservice that requires a lot of communication with other services - https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

[53]https://docs.microsoft.com/en-us/azure/azure-monitor/app/performance-testing

tool work, times few things are slightly changed compared to experiment 1: Throughput is defined in this experiment as all the successful requests for a given interval.

D. Benchmark Design

The performance testing tool also only allowed HTTP GET requests, so the workload used in this experiment was an light version of the simple workload from experiment 1.

The 2 versions of the LibraryService are each configured with two different hardware configurations. The monolithic version is first configured with 1 CPU core, 3.50 GB memory, and 12 server instances. Then configured with 4 CPU cores, 14 GB memory, and 30 server instances meaning there is a four times increase in resources. The second configuration is the highest configuration possible for the Azure subscription used for this experiment.

The microservice version is first configured with two nodes of 2 CPU cores, 7 GB memory, 24 BookService pods, and 3 pods of each of the other services. Then it is configured with five nodes of the same configuration and 144 BookService pods yielding a 2.5 times increase in computational power. The five nodes with 10 CPU cores (in total) and roughly 30 pods per node was the maximum amount allowed on the Azure subscription used for this experiment.

The lower hardware configurations (low config) which consist of P1V2 or two DSv2 VMs, are tested with 2,500, 5,000, and 7,500 concurrent users for 2 minutes generating GET requests to the BookService API end point, and the high hardware configurations (high config) which consist of P3V2 or five DSv2 nodes VMs, are tested with 10,000, 15,000 and 20,000 concurrent users for 1 minute (to conserve the cloud credits) reaching the same API end point.

E. Latency results

The average latency for the low config can be seen on Figure 14. Here it can be observed how the monolith version performs better initially but as the amount of requests increases the difference in latency between the two systems decreases.

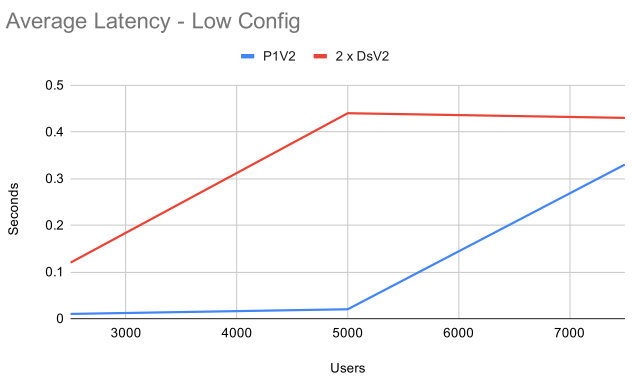

Fig. 14: Latency for low config in experiment 2

The average latency for the high config can be seen on Figure 15. Here it can be observed how the two different systems performs somewhat close to each other, but the monolith version performs somewhat better at 20,000 concurrent users.

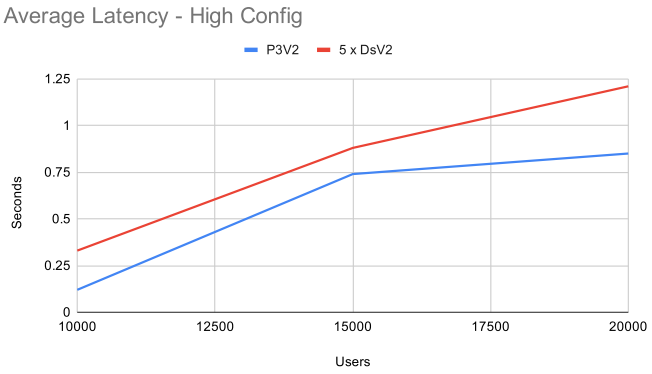

Fig. 15: Latency for high config in experiment 2

F. Throughput results

The throughput for the low config can be seen on Figure 16 which shows that, like with the latency, once again the monolithic system performs better than the microservice system.

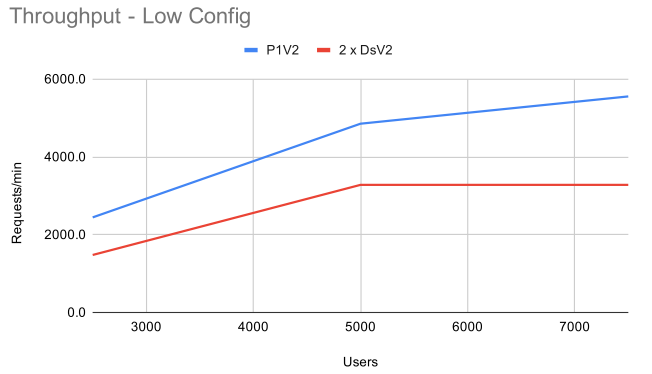

Fig. 16: Throughput for low config in experiment 2

The throughput for the high config can be seen on Figure 17 where the monolith again initially performs better, but as the amount of concurrent users increases the two systems throughput are close to converge.

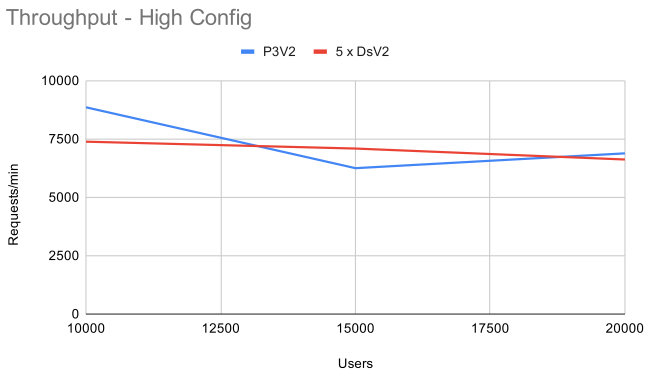

Fig. 17: Throughput for high config in experiment 2

G. Scalability results

The cloud systems scalability were determined by the ration between amount of concurrent users while maintaining an average latency of no more than 200 ms. Polynomial regression was used to determine the amount of users a system and its specific hardware configuration can handle. Looking at Figure 18 and Figure 19 the polynomial trend lines can be seen alongside their R^2 values.

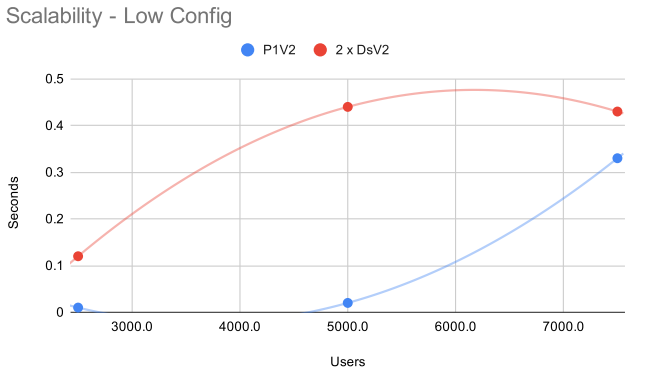

Fig. 18: Scalability for low configuration in experiment 2

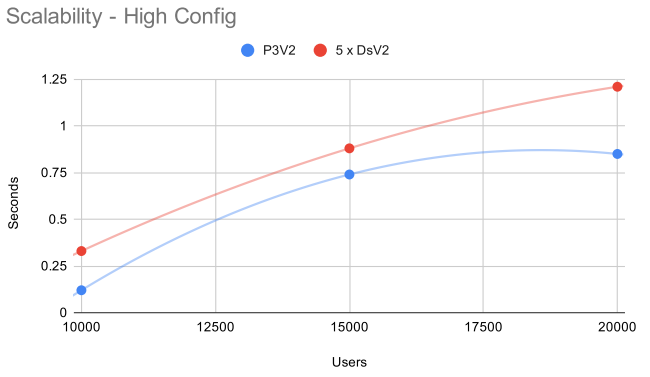

Fig. 19: Scalability for high configuration in experiment 2

The amount of concurrent users a system can handle with a maximum average latency of 200 ms with a given hardware configuration can then be calculated. The results can be seen in Table IX. Note that the results from the Result 1 column are used, as it is desired to get the amount of concurrent users before the polynomial reaches its peak, and thus the system's break off point), rather than after.

Hardware Configuration	Result 1	Result 2
P1V2	6713	621
2 x DSv2	2939	9410
P3V2	10470	26747
5 x DSv2	9046	40954

TABLE IX: Amount of concurrent users each configuration can handle at 200 ms average latency

The scalability can now be calculated as the ratio between increase in concurrent users and increase in hardware power:

$$scalability = \frac{IncreaseInConcurrentUsers}{AmountOfSystemResources} \quad (2)$$

The results from using Equation 2 can be seen in Table X. The scalability ratio of the microservice system is over six times bigger than the monolith system, meaning that the microservice version scales vastly better with an increase in hardware resources.

System Type	Users Increase	Resource Increase	Scalability
Monolith	3757	4x	0.14
Microservice	6107	2.5x	0.83

TABLE X: Scalability ratio for the monolith and microservice systems

H. Threat to validity

One threat to validity is the amount of pods/app service instances used. It was difficult getting exact CPU and Memory usage to optimise the exact amount of pods/instances, so the amount was chosen after a bit of experimentation. One way to test this would be to use App Service and Kubernetes auto horizontal scaling features based on CPU and memory usage. However, with the system properties only being sampled once a minute in Azure it would take a while for the auto-scaler to reach the ideal amount. Another threat to validity is the allocation of the cloud resources. The databases are serverless[54], meaning that they take up no resources when they have not been used in a while, however upon being used again there is an initial startup period where the resources must re-allocated.

VIII. Discussion

The results presented the previous sections show that the monolith system performs better in all metrics except for scalability. This outcome can possibly be explained by a number of reasons: the reference systems are not enterprise sized systems, Kubernetes/Docker overhead, and the hardware configurations used for experiments.

The reference systems are relatively small compared to large enterprise systems consisting of millions of lines of code and

[54]https://martinfowler.com/articles/serverless.html

heavy use of libraries/other dependencies. This means that spinning up additional instances of a monolithic web server often comes with a bigger resource costs than the microservice equivalent. However, with both reference systems being fairly small, there is not a big difference in the resource cost of scaling up.

This leads to the second point being that Docker and Kubernetes comes with an extra resource overhead, meaning that potentially the overhead of spinning up an extra microservice instance might be more costly than spinning up an extra monolithic server instance.

The increased overhead becomes visible in experiment 1 where the resource usage is higher to provide a similar, or even lower, throughput for the microservice compared to the monolith. The higher resource usage most likely lead the SQL server to bottleneck [47] as it did not have access to enough system resources to properly perform. The under-performing database is also a likely reason for the microservices in experiment 1 performing worse compared to having a separate machine as a dedicated database server like in experiment 2.

The resource usage of experiment 1 was as expected considering that Kubernetes is made for a cluster rather than a single computer as mentioned in Section **??**. However the overhead for managing a cluster comes with better scalability as the microservice scaled three times better in experiment 1 and six times in experiment 2 compared to the monolith system. This indicates that despite the increased resource cost, the microservice architecture is worth using if the system needs to support a large amount of concurrent users.

Experiment 2 was a simpler version of experiment 1, since it used a light version of the simple workload. This setup gives a less nuanced picture of the overall performance of the system, but it allowed to test the scalability better with more and faster hardware while handling a much higher load.

Based on the results from these experiments it would seem that a smaller to medium sized monolith generally uses less system resources while performing better than its microservice counterpart. However the microservice architecture redeems itself with its great scalability, allowing it to easier scale up. It also showed how a microservice migration can go wrong, as the complex microservice workload performed badly across the board. In the case of the complex microservice it means that it is important to design the microservices with well bounded contexts to avoid communication between services unless necessary.

The chosen metrics seem to be good choices for evaluation with particular Latency, Throughput and Scalability being important to evaluate how well a system performs. CPU, Memory and Network are also good performance indicators, but these serve just as much as a way to explain a system's Latency, Throughput and Scalability.

IX. Future Work

While the results from the two experiments mostly are as expected, it would be interesting to measure the metrics for systems which are closer to the kind of systems used by companies. To make the system closer to a real-life system, the reference system would benefit from a series of improvements. The first improvement is the sheer size of the system, adding bigger and more components with more dependencies would make it closer an enterprise system. This bigger system would also benefit from having a bigger and more complex database with real, or close to real, data. This would impact query time and thus latency, but it would make it closer to a real-life system.

To better simulate the possibilities of a polyglot microservice architecture it would also be beneficial to implement one or more services in another programming language than C# such as Java or Python. If the local experiment is desired to be repeated it would be beneficial to have a more professional setup with server-grade hardware, wired connections and a separate machine for the database server. Ideally for clustering there should be multiple machines for Kubernetes nodes, or a server machine powerful enough to run multiple virtual machine nodes.

A final important improvement to make the system more real-life-like is adding security; currently the system neither uses encryption or authentication. Ideally all communication would go through TSL[55]. Additionally the system should require authentication e.g. through a JSON web token[56]. To further test the scalability it would be interesting to add more and/or more powerful nodes to AKS in order to see how the microservice system scales at 10, 50 or even 100 nodes.

Another scalability measure to be measured would be to use auto-scale for App Service instances and for Kubernetes rather than the current manual tuning. However this would also require longer test runs to let the auto-scale scale the system to a proper size.

X. Conclusion

In this article different experiments have been realized to evaluate whether or not the migration from a monolith architecture to a microservice architecture is beneficial. To perform this evaluation, a methodology was constructed based on a *Structured Literature Review*, performing a small-scale survey amongst IT professionals, and looking at existing approaches. This led to a methodology consisting of benchmarking a monolith and microservice system and comparing them based on the following metrics: Latency, Throughput, Scalability, CPU, Memory and Network.

It was decided to build two new reference systems for the benchmarking. This led to the creation of the Monolithic- and Microservice version of the *LibraryService*.

Two benchmarking experiments were performed: one using a local desktop PC and one using the cloud. In the first experiment the monolith came out ahead, while the microservice struggled, particularly with the complex workload. The microservice system did however in the best case have a three times higher scalability ratio compared to its monolith counterpart.

In the second experiment, using the power of the cloud, the monolith still had better performance, but the microservice

[55]https://tools.ietf.org/html/rfc8446

[56]https://jwt.io/

system once again had a favorable scalability ratio. In this experiment the microservice system had a six times higher scalability score compared to the monolith system.

In conclusion, the monolithic architecture appear to perform better for small to medium sized systems as used in this article. However, the much higher scalability ratio of the microservice system indicates, that the hypothesis of the microservice architecture performs better than monolith architecture, is correct for systems which must support a large amount of concurrent users.

References

[1] Carlos M Aderaldo et al. "Benchmark requirements for microservices architecture research". In: *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*. IEEE Press. 2017, pp. 8–13.

[2] M. Amaral et al. "Performance Evaluation of Microservices Architectures Using Containers". In: *2015 IEEE 14th International Symposium on Network Computing and Applications*. Sept. 2015, pp. 27–34.

[3] M. Aniche et al. "SATT: Tailoring Code Metric Thresholds for Different Software Architectures". In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Oct. 2016, pp. 41–50.

[4] N. Antunes and M. Vieira. "Detecting Vulnerabilities in Service Oriented Architectures". In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*. Nov. 2012, pp. 134–139.

[5] Harold Aragon et al. "Workload Characterization of a Software-as-a-Service Web Application Implemented with a Microservices Architecture". In: *Companion Proceedings of The 2019 World Wide Web Conference*. WWW '19. San Francisco, USA: ACM, 2019, pp. 746–750.

[6] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52.

[7] D. Bermbach, E. Wittern, and S. Tai. *Cloud service benchmarking: Measuring quality of cloud services from a client perspective*. cited By 16. 2017, pp. 1–167.

[8] André B. Bondi. "Incorporating Software Performance Engineering Methods and Practices into the Software Development Life Cycle". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE '16. Delft, The Netherlands: ACM, 2016, pp. 327–330.

[9] S. Boukharata et al. "Improving web service interfaces modularity using multi-objective optimization". In: *Automated Software Engineering* 26.2 (2019). cited By 0, pp. 275–312.

[10] T. Brummett et al. "Performance Metrics of Local Cloud Computing Architectures". In: *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*. Nov. 2015, pp. 25–30.

[11] Antonio Bucchiarone et al. "From monolithic to microservices: An experience report from the banking domain". In: *Ieee Software* 35.3 (2018), pp. 50–55.

[12] Mario Cardarelli et al. "An Extensible Data-driven Approach for Evaluating the Quality of Microservice Architectures". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. Limassol, Cyprus: ACM, 2019, pp. 1225–1234.

[13] Melvin E Conway. "How do committees invent". In: *Datamation* 14.4 (1968), pp. 28–31.

[14] B. Curtis, J. Sappidi, and J. Subramanyam. "An evaluation of the internal quality of business applications: does size matter?" In: *2011 33rd International Conference on Software Engineering (ICSE)*. May 2011, pp. 711–715.

[15] A. Dragomir and H. Lichter. "Towards an Architecture Quality Index for the Behavior of Software Systems". In: *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 2. Dec. 2014, pp. 75–82.

[16] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and Ulterior Software Engineering*. Ed. by Bertrand Meyer and Manuel Mazzara. Springer, 2017.

[17] Thomas F. Düllmann and André van Hoorn. "Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17 Companion. L'Aquila, Italy: ACM, 2017, pp. 171–172.

[18] M. Elsayed and M. Zulkernine. "Offering security diagnosis as a service for cloud SaaS applications". In: *Journal of Information Security and Applications* 44 (2019). cited By 1, pp. 32–48.

[19] Erwin van Eyk et al. "A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: ACM, 2018, pp. 21–24.

[20] C. H. G. Ferreira et al. "PEESOS-Cloud: A Workload-Aware Architecture for Performance Evaluation in Service-Oriented Systems". In: *2016 IEEE World Congress on Services (SERVICES)*. June 2016, pp. 118–125.

[21] Greg Franks, Danny Lau, and Curtis Hrischuk. "Performance Measurements and Modeling of a Java-based Session Initiation Protocol (SIP) Application Server". In: *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*. QoSA-ISARCS '11. Boulder, Colorado, USA: ACM, 2011, pp. 63–72.

[22] D. Gesvindr and B. Buhnova. "PaaSArch: Quality Evaluation Tool for PaaS Cloud Applications Using Generated Prototypes". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Mar. 2019, pp. 170–173.

[23] Z. Hadjilambrou, M. Kleanthous, and Y. Sazeides. "Characterization and analysis of a web search bench-

mark". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 328–337.

[24] T. Heyman, D. Preuveneers, and W. Joosen. "Scalability Analysis of the OpenAM Access Control System with the Universal Scalability Law". In: *2014 International Conference on Future Internet of Things and Cloud*. Aug. 2014, pp. 505–512.

[25] A. A. Z. A. Ibrahim et al. "PRESEnCE: Performance Metrics Models for Cloud SaaS Web Services". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. July 2018, pp. 936–940.

[26] M Ashraf Iqbal, Joel H Saltz, and SH Bokhart. "Performance tradeoffs in static and dynamic load balancing strategies". In: (1986).

[27] Nane Kratzke and Peter-Christian Quint. "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study". In: *Journal of Systems and Software* 126 (2017), pp. 1–16.

[28] Paul J Lavrakas. *Encyclopedia of survey research methods*. Sage Publications, 2008.

[29] Matthew C. MacKenzie et al. "Reference model for service oriented architecture 1.0". In: *OASIS Standard* 12 (2006).

[30] A. Martinez-Millana et al. "Performance assessment of a closed-loop system for diabetes management". In: *Medical and Biological Engineering and Computing* 53.12 (2015). cited By 10, pp. 1295–1303.

[31] Manuel Mazzara et al. "Microservices: Migration of a mission critical system". In: *IEEE Transactions on Services Computing* (2018).

[32] A. Mohsin et al. "A comprehensive framework to quantify fault tolerance metrics of web centric mobile applications". In: *2017 International Conference on Communication Technologies (ComTech)*. Apr. 2017, pp. 65–71.

[33] Ali Ouni et al. "A Hybrid Approach for Improving the Design Quality of Web Service Interfaces". In: *ACM Trans. Internet Technol.* 19.1 (Dec. 2018). cited By 1, 4:1–4:24.

[34] Claus Pahl and Pooyan Jamshidi. "Microservices: A Systematic Mapping Study". In: *CLOSER 2016, April 23-25, 2016.* Vol. 1. 2016, pp. 137–146.

[35] A. Pandey et al. "An Automation Framework for Benchmarking and Optimizing Performance of Remote Desktops in the Cloud". In: *2017 International Conference on High Performance Computing Simulation (HPCS)*. July 2017, pp. 745–752.

[36] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058.

[37] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. "Guidelines for conducting systematic mapping studies in software engineering: An update". In: *Information and Software Technology* 64 (2015), pp. 1–18.

[38] Kai Petersen et al. "Systematic mapping studies in software engineering." In: *Ease*. Vol. 8. 2008, pp. 68–77.

[39] A. Shukla, S. Chaturvedi, and Y. Simmhan. "RIoTBench: An IoT benchmark for distributed stream processing systems". In: *Concurrency Computation* 29.21 (2017). cited By 16.

[40] R. de Souza Pinto, A.C. Botazzo Delbem, and F.J. Monaco. "Characterization of runtime resource usage from analysis of binary executable programs". In: *Applied Soft Computing Journal* 71 (2018). cited By 0, pp. 1133–1152.

[41] A. Sriraman and T. F. Wenisch. "µ Suite: A Benchmark Suite for Microservices". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. Sept. 2018, pp. 1–12.

[42] J.M. Tekli et al. "SOAP processing performance and enhancement". In: *IEEE Transactions on Services Computing* 5.3 (2012). cited By 26, pp. 387–403.

[43] T. Ueda, T. Nakaike, and M. Ohara. "Workload characterization for microservices". In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. Sept. 2016, pp. 1–10.

[44] M. Vasar, S.N. Srirama, and M. Dumas. "Framework for monitoring and testing web application scalability on the cloud". In: 2012, pp. 53–60.

[45] V. Vedam and J. Vemulapati. "Demystifying Cloud Benchmarking Paradigm - An in Depth View". In: *2012 IEEE 36th Annual Computer Software and Applications Conference*. July 2012, pp. 416–421.

[46] Mario Villamizar et al. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *2015 10th Computing Colombian Conference (10CCC)*. IEEE. 2015, pp. 583–590.

[47] Bob Wescott. *Every Computer Performance Book: How to Avoid and Solve Performance Problems on The Computers You Work With*. CreateSpace Independent Publishing Platform, 2013.

[48] Claes Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering". In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. Citeseer. 2014, p. 38.