JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

# Benchmarks and performance metrics for assessing the migration to microservice-based architectures

**Nichlas Bjørndal**[*], **Luiz Jonatã Pires de Araújo**[†], **Antonio Bucchiarone**[‡], **Nicola Dragoni**[*], **Manuel Mazzara**[†], **and Schahram Dustdar**[**]

[*]Technical University of Denmark
[†]Innopolis University, Russia
[‡]Fondazione Bruno Kessler, Italy
[**]Technische Universität Wien, Austria

**ABSTRACT** The migration from monolithic to microservice-based systems have become increasingly popular in the last decade. However, the advantages of this type of migration have not been extensively investigated in the literature, to the best of the authors' knowledge. This paper aims to present a methodology and performance indicators to support better assessment on whether the migration from a monolithic to microservice-based architecture is beneficial. A systematic review was conducted to identify the most relevant performance metrics in the literature, validated in a survey with professionals from the industry. Next, this set of metrics, including latency, throughput, scalability, CPU, memory usage, and network utilization - were used in two experiments to evaluate monolithic and microservice versions of the same system. The results reported here contribute to the body of knowledge on benchmarking different software architectures. In addition, this study illustrates how the identified metrics can more precisely assess both monolithic and microservice systems.

**KEYWORDS** Benchmarking; Software Architecture; Monolith; Microservices; Software Engineering.

## 1. Introduction

Microservices are an architectural paradigm derived from Service-Oriented Architectures (SOAs) (MacKenzie et al. 2006) to bring *in the small*, or within an application, concepts that worked well *in the large* (Balalaie et al. 2016; Dragoni et al. 2017; Pahl & Jamshidi 2016). Such an approach has demonstrated to be suitable for cross-organization business-to-business workflow. On the other hand, monolithic architectures follow a modularization abstraction relying on sharing the same machine (memory, databases, files), where the components are not independently executable. The limitations of monolithic systems include *scalability* and aspects related to change (Kratzke & Quint 2017).

In the microservice paradigm, a system is structured of small independent building blocks, each with dedicated persistence tools and communicating exclusively via message passing. In this type of architecture, the complexity is moved to the level of services coordination. 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 does not impact other services and their releases. In general, it is also expected that a single service can be developed and managed by a single team (Parnas 1972; Dragoni et al. 2017). The idea to have a team working on a single microservice is rather appealing: to build a system with

**JOT reference format:**
Nichlas Bjørndal, Luiz Jonatã Pires de Araújo, Antonio Bucchiarone, Nicola Dragoni,
Manuel Mazzara, and Schahram Dustdar. *Benchmarks and performance metrics for assessing the migration to microservice-based architectures*.
Journal of Object Technology. Vol. 20, No. 3, 2021. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2021.20.3.a3
**JOT reference format:**
Nichlas Bjørndal, Luiz Jonatã Pires de Araújo, Antonio Bucchiarone, Nicola Dragoni,
Manuel Mazzara, and Schahram Dustdar. *Benchmarks and performance metrics for assessing the migration to microservice-based architectures*.
Journal of Object Technology. Vol. 20, No. 3, 2021. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2021.20.3.a3

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 (Conway 1968), directly impact the produced design. Therefore, microservice-based architectures promote more communication on both the team and the entire organization, leading to improved software design regarding modularity.

The study of methodologies addressing the adoption of microservices is a promising research area since several companies are engaged in an extensive effort for refactoring their back-end systems to implement this new paradigm. This study extends previous research (Bucchiarone et al. 2018; Mazzara et al. 2018) characterizing the software modernization of old systems by migrating from a monolithic to a microservice architecture. This paper addresses an interesting aspect that is often ignored in the literature: the lack of methodology and performance indicators to assess the monolithic-to-microservice redesign. This study presents a methodology that integrates the literature into a single framework and set of performance measurements for comparing both monolithic and microservice-based implementations during the migration. The proposed methodology is then validated against two open-source systems.

The remaining of this paper is organized as follows. Section 2 presents a systematic literature review (SLR) on benchmarks, metrics for evaluating software architectures, and examples in the literature about the migration from monolithic to microservice systems. Section 3 presents a survey conducted among professionals in the industry to validate the selected metrics used in this study. Section 4 presents a system especially developed for this study. Such a system has two versions: a monolithic and a microservices-based architecture. Then, two experiments were conducted to investigate how the performance metrics previously selected differ when running a small-scale system running on a local machine or a large-scale system running in the cloud. The results are shown in Section 5. Section 6 discusses the validity of the methodology and metrics to evaluate the migration from monolithic to microservice architecture for the investigated case studies. Finally, Section 7 summarises the benefits of the proposed methodology and future research directions.

## 2. Literature review

This section surveys the literature on methods and metrics for evaluating software architectures. First, it presents the adopted systematic for revising the literature (Section 2.1), then the existing benchmarks (Section 2.2), metrics (Section 2.3) and examples of migration reported in the literature (Section 2.4).

### 2.1. A systematic literature review

This study employs the PICO guidelines presented in (Petersen et al. 2008, 2015) to query academic databases for addressing the existing benchmarks and metrics in software engineering. In the PICO methodology, a population, intervention, comparison and outcomes are determined as follows:

- **Population**: Benchmarks used in the field of software engineering.
- **Intervention**: Methodologies and metrics for benchmarking in software engineering.
- **Comparison**: Their different benchmarking methodologies and metrics.
- **Outcomes**: A study presenting the different benchmarking methodologies and metrics and insights into which techniques are more suitable for monoliths and microservices.

Following this approach, the main keywords in the aspects above were selected and used for querying the following academic databases: Scopus[1], IEEE Xplore Digital Library[2], and ACM Digital Library[3]. Moreover, the following filter was applied to restrict the body of literature on user space applications such as web servers.

- Articles with actual measurement and not predictions.
- Articles published between 2010 and 2019.
- Peer-reviewed articles.
- Article in English.
- Articles not focusing on CPU architectures.
- Articles not focusing on OS Kernels.
- Articles not focusing on algorithm benchmarks.

The articles that result from the querying and filtering are shown in Table 1. Two metrics for measuring performance appear in the studies presented in Table 1: CPU utilisation and throughput. Therefore, this study will use these metrics as a core element in the proposed benchmark design. It is noteworthy that a similar concept is often referred to using different phraseology such as 'requests per second' and 'throughput'. In this example, both studies would be categorised as using the same metric.

The following sections present the main benchmarks and metrics in software engineering extracted from the papers in Table 1.

### 2.2. Benchmark in software engineering

In software engineering, benchmarking can be divided into the following categories: measurement, i.e. practical; analytical, i.e. theoretical; a junction of both analytical and measurement; and simulation. The latter category was ignored in this study since it consists of predictions rather than a characterisation of the actual system.

Some types of benchmarks are more recurrently used than others. The 'measurement' type, for example, is most popular and appear in 83.8% of the studies. This is not surprising since collecting data from a real-world system is a suitable way to understand the subject under consideration better. The second most common category is 'analytical', appearing in 12.5% of the studies. The study of this type of benchmark can be beneficial as it can enable creating a benchmark-suite by investigating the source code directly. On the other hand, some articles investigated benchmarking systems at design-time, i.e., performance metrics that can be calculated before the source-code is written. Finally, the type 'analytical and measurement'

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

| Metric type | Articles |
|---|---|
| Performance | (de Souza Pinto et al. 2018) (van Eyk et al. 2018) (Shukla et al. 2017) (Aragon et al. 2019) (Bermbach et al. 2017) (Bondi 2016) (Martinez-Millana et al. 2015) (Tekli et al. 2011) (Vasar et al. 2012) (Franks et al. 2011) (Gesvindr & Buhnova 2019) (Ibrahim et al. 2018) (Pandey et al. 2017) (Ferreira et al. 2016) (Ueda et al. 2016) (Hadjilambrou et al. 2015) (Amaral et al. 2015) (Brummett et al. 2015) (Vedam & Vemulapati 2012) (Sriraman & Wenisch 2018) (Düllmann & van Hoorn 2017) |
| Availability | (van Eyk et al. 2018) (Bermbach et al. 2017) (Bondi 2016) (Mohsin et al. 2017) (Düllmann & van Hoorn 2017) |
| Security | (Elsayed & Zulkernine 2019) (Antunes & Vieira 2012) (Curtis et al. 2011) |
| Scalability | (Bermbach et al. 2017) (Ueda et al. 2016) (Heyman et al. 2014) (Vedam & Vemulapati 2012) |
| Consistency | (Bermbach et al. 2017) (Vedam & Vemulapati 2012) |
| Quality | (Boukharata et al. 2019) (Cardarelli et al. 2019) (Ouni et al. 2018) (Aniche et al. 2016) (Dragomir & Lichter 2014) (Curtis et al. 2011) |
| Cost | (van Eyk et al. 2018) |

**Table 1** Selected articles in the systematic literature review.

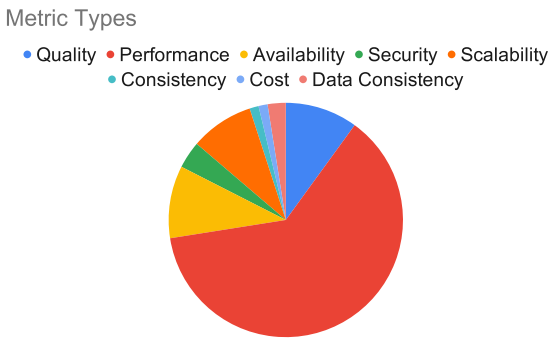


**Figure 1** Relative frequency of metric types' appearances among the studies.

appears in only 3.8% of the articles, even though such type of benchmarking combines analytical and performance metrics for a running system.

### 2.3. Benchmark metrics in software engineering

According to Bermbach et al. (2017), benchmark metrics can be categorised into the following types: availability, consistency, cost, performance, quality, scalability and security. The relative frequency of these metrics' appearance in the literature is shown in Figure 1.

Not surprisingly, **performance** is the metric mostly used in benchmarking studies in software engineering, appearing in approximately 62.5% of the investigated papers. One factor for its popularity is the preference by scholars for clearly measurable properties. Moreover, several related metrics can also be classified as measurement types, such as, for example, CPU performance.

The next three types of metrics in order of popularity among the selected studies are **availability** (10.0%), **quality** (10.0%) and **scalability** (8.8%). Availability refers to the capability of a system to run over a long period. Quality is a subjective metric that aims to assess the source-code, performance, and skill level of the development team. Scalability refers to how well a system scales as more computational resources are necessary to accommodate increasing demand. This requires both hardware, software architecture, and code to be designed to support many users. Like quality, precise quantification of this metric is subjective and complex. Among the metrics that appear less frequently there is **security** (3.8%), **data consistency** (2.5%). While some security-related metrics can be calculated based on the source code's characteristics, they still rely on subjective evaluation.

It is noteworthy to examine some of the metrics that have been categorised in the performance taxonomy since it is the most frequently used in software engineering benchmarking. The number of occurrences of these metrics is depicted in Figure 2. The remaining of this section presents the metrics and their meaning in the context of software engineering.

**Latency.** It is the time delay between the request and the response, which is essential in client-server communication. This metric is mostly affected by network speed, latency, and the server's amount of time to process the request. Moreover, it plays a significant role in the end-user experience, explaining its popularity among the examined articles, as shown in Figure 2.

**CPU utilization.** It indicates how efficient the software system is regarding the use of computational resources. It can also be used to assess the efficiency of different implementations for similar functionalities.

**Throughput.** It is a metric that informs the number of actions and requests that a system can process given a fixed workload and hardware resources. For example, a system that can process 100 requests per second performs is said to perform worse than a system that can process 1,000 requests per second, given the same computational resources.

**Network.** It indicates the number of networking resources used by a system. This metric is relevant in scenarios in which microservices communicate over HTTP/HTTPS, leading to high usage of the network and affecting the system's latency.

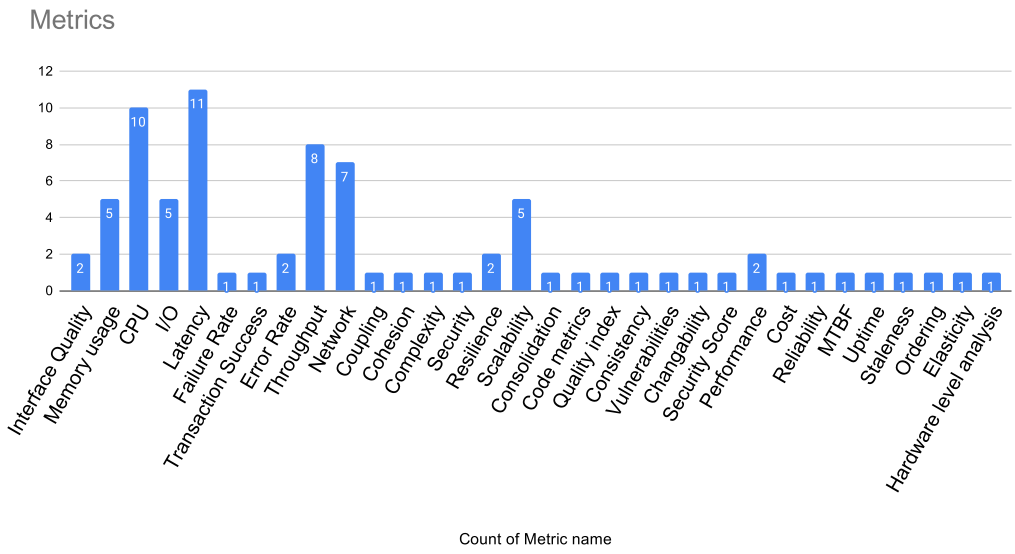


**Figure 2** Number of performance metrics' occurrences in the selected studies.

**Scalability.** It is the capability of a system to handle an increasing number of users while maintaining similar performance. The scalability can be measured by the number of requests that can be processed under a predetermined latency threshold. For example, the number of requests that a system can process before the latency rises is more than 200 ms. The term *elasticity* is often used to refer to how fast a cloud system can increase/decrease computing resources in the cloud to meet the demand.

**Memory.** It indicates the amount of RAM used by the system. It has gained increasing importance as virtualization (e.g. virtual machines and containers) becomes more prominent.

**Input/output (I/O).** This metric quantifies the use of input/output devices in the host machine. For example, how often the targeted system performs read or write operations using hard disks or network cards.

**Fault tolerance.** This umbrella term encompasses several metrics, including the following: failure or error rate, transaction success, resilience or reliability, mean time between failures (MTBF), and uptime. These metrics are usually measured by the number of occurrences of errors, frequency, and the impact on the target system.

**Code quality.** There have been some alternatives in the literature for assessing quality in the source-code. These include interface quality, coupling, cohesion, complexity, changeability, and quality index.

**Security.** This metric is often calculated by security experts or pen testers[4]. In the examined articles, two security metrics were found: security score and amount of vulnerabilities. The first metric is calculated according to acceptable architectural and coding practices. The latter metric is calculated according to metadata files and application byte code. The number of entry points, call-backs, dependencies, and information flow during the application's life cycle are considered when quantifying the number of vulnerabilities.

**Consistency.** This metric is often referred to as staleness or ordering. For example, in a distributed database, there are replicas of a data partition to improve performance and availability. Staleness quantifies how outdated a data partition is compared to the respective replicas. Ordering describes the order of occurring system events is consistent. For example, the order of the client-side events is the same as the order in which they are processed by the server-side.

**Hardware level.** This is a CPU and hardware-level metric. For example, how many CPU instructions are necessary to process a request or the number of data cache misses.

**Cost.** This metric is often one of the primary factors when stakeholders are making business decisions.

### 2.4. Monolithic to microservice migration

The selected literature contains some studies benchmarking the migration from monolithic to microservice architectures (Ueda et al. 2016; Villamizar et al. 2015; Zhou et al. 2018; Ren et al. 2018; Gan et al. 2019). For example, Ueda et al. proposed a workload characterization for microservices, in which a monolithic and microservice version the same system - which consists of a Java application using Node.js[5]. The authors analyzed the runtime and hardware-level performance of both systems according to throughput, scalability, path-length (i.e., amount of CPU instructions, the average amount of CPU clock cycles to

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

[5] https://github.com/acmeair/acmeair

complete an instruction, data cache miss cycles, code execution time per transaction.

Villamizar et al. (2015) evaluate monolithic and microservice architectures for web applications in the cloud. The authors presented a case study with a company to build a monolithic and a microservice reference system to support two services. The first service implemented a CPU bulky service with a relatively long processing time. The second service read from a relational database with a relatively short response time. Comparing these two services analyzed the throughput, the average and maximum response time for a set amount of requests per minute, and the cloud hosting cost defined as cost per million requests. Like the first study aforementioned, only a limited number of metrics was used, and the most commonly used metrics in the literature were assessed.

## 3. Relevant metrics for assessing monolithic-to-microservice migration

The systematic literature review presented in Section 2.1 enables identifying the most relevant metrics for assessing monolithic-to-microservice migration. The most popular metrics and the number of occurrences are shown in Table 2.

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

**Table 2** Frequency of the most popular metrics in the SLR.

The findings in Table 2 confirm the popularity of measurement and performance mentioned in Section 2.3. The results indicate that most of the metrics belong to both categories, indicating that generally, performance-related metrics are favoured by the research community. The authors have made the data and statistics on benchmarks and metrics publicly available to the software engineering community[6] [7].

### 3.1. Validating the relevant benchmarking metrics

This study surveyed to validate the choice of relevant metrics for assessing monolithic-to-microservice migrations. This survey aims to gain insights from real-world stakeholders such as customers and developers regarding these metrics' usefulness. Moreover, the survey enables the identification of which metrics are more relevant to each stakeholder role. Hence, an initial question asked in a free-text form which metrics the stakeholders perceived as the most important. Then, they were asked to rank the most popular metrics identified in this study on a scale from 1 to 5. The questions are shown as follows:

1. What is your role in the company?
2. Which metrics and qualities of a software system are most important to you?
3. Some of the most popular metrics are listed here. Please rank them according to how important you find them regarding software systems: latency, CPU utilization, throughput, network, scalability, memory usage, I/O, hosting cost, security, and 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 (Lavrakas 2008). The participants were selected from the author's current employment place and one of its clients. The survey can give insight through empirical software engineering into metrics that the software industry prefer.

The survey was given to four selected stakeholders: A systems architect, a product owner and two developers. These stakeholders are in the same project developing an application that serves the national prison system with approximately 4,000 registered users. While the survey asks about metrics in general, it is assumed that survey participants carry some bias toward the project they are currently working on. The survey data can be seen at GitHub[8].

The answers for the second question - metrics and qualities perceived as the most important - the there were four answers: latency, throughput, code quality and other metrics such as user's experience needs. These answers reinforce the importance of latency and throughput and stress code quality when maintainability and development speed are essential requirements.

For the third question, in which the participants were asked to rank the metrics, it can be seen that latency, throughput and scalability were ranking highly. The average rating given to each metric in the survey can be seen in Table 3. Interestingly, security and code quality are also highly ranked metrics.

## 4. A case study

A benchmark experiment was conducted using the most relevant metrics identified in Section 3 to compare monolith and microservice-based software architectures. The experiment was performed on reference systems designed and implemented for this study using the targeted software architectures. The fact that the authors had full access and control over all the software artefacts addressed the limitations of analysing third-party applications in previous migration experiences (Mazzara et al. 2018).

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

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

[8] 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 3** Average ranking scores of the metrics in the survey.

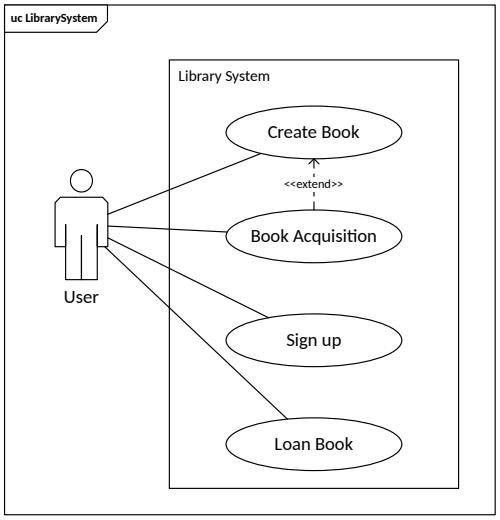


**Figure 3** Use case diagram of the *LibraryService* system.

The first application developed for this experiment was a library management system so-called *LibraryService* in both monolithic and microservice-based versions. The differences between the monolithic and microservice-based versions have been kept to a minimum to enable an analysis of the performance differences at an architectural level. Such a methodology does not entirely prevent criticism regarding the validity of the comparison between the architectures mentioned above. However, it is noteworthy that a similar strategy has been consistently adopted in several studies in the literature (Flygare & Holmqvist 2017; Fan & Ma 2017; Taibi et al. 2017). The system can be used by regular library users interested in registering and also borrowing books. The following entities have been identified: user, book, loan, author, physical book and order. These entities will also drive the database design and service endpoints. The system's functionalities are described by the use case shown in Figure 3.

This experiment focuses performance measurement metrics since latency and throughput are among the most popular metrics in the literature (see Table 2) and also in the survey (see Table 3). Other relevant metrics used in the experiment include scalability, CPU, memory and network utilisation, security and code quality.

### 4.1. Monolithic system version

The monolithic system follows standard enterprise systems in which the entire application runs in one processor coupled to one technology stack and connected to a single database. This version of the *LibrarySystem* was built using ASP.NET Core[9], and Web API using .NET Core 3[10]. This system is connected to a SQL Server[11] database. The interaction between the database and the web API uses the Entity Framework[12], which is an object-relational mapping (ORM) tool, which provides the create, read, update and delete (CRUD) operations.

### 4.2. Microservice system version

The second version for the proposed *LibraryService* decomposes the domain model into several microservices via domain-driven design. It consists of the following microservices: BookService, UserService, LoanService, and OrderService.

The microservices were built in ASP.NET Core Web APIs and Entity Framework. Each microservice has its own dedicated SQL Server database to enable scaling-up or scaling-down their respective data stores if necessary. The microservices run in Docker[13] containers, yielding a Docker image per microservice which are stored in registries such as Docker Hub[14] and Azure Container Registry (ACR)[15]. This setup allows easy distribution of the microservices by running in similar environments across different machines and hosts. Kubernetes has been used to orchestrate the microservices and allow easy horizontal scaling, load balancing, health checks, among other requirements.

Each service has its deployment workload[16], which is responsible for metadata, pulling the Docker image from the registry, internal and external communication and replication, ensuring multiple pods[17] to run. Then, each deployment is exposed to the external network through a Kubernetes service[18] that routes external traffic to Kubernetes pods, which can handle each request. Kubernetes also supports load balancing (Iqbal et al. 1986), which attempts to spread the load among the microservice replicas. Each deployment is also configured to perform a periodic liveliness check. If a replica fails, a liveness check Kubernetes will shut down the pod and create a new one instance of the microservice.

[9] https://docs.microsoft.com/en-us/aspnet/
[10] https://docs.microsoft.com/en-us/dotnet/core/about
[11] https://www.microsoft.com/en-us/sql-server/sql-server-2019
[12] https://github.com/aspnet/EntityFrameworkCore
[13] https://www.docker.com/
[14] hub.docker.com
[15] https://azure.microsoft.com/en-us/services/container-registry/
[16] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[17] https://kubernetes.io/docs/concepts/workloads/pods/pod/
[18] https://kubernetes.io/docs/concepts/services-networking/service

Locally, the load balancing is achieved by using Ingress[19], which is an alternative way for handling load balancing by routing the external requests to inside the Kubernetes cluster. An Azure Service Bus Message Queue[20] is used for asynchronous communication between *Book Service* and *Order Service*, and to illustrate the effects of using message queues over HTTP(S) communication between services. The standard structure of a *LibraryService* microservice is an ASP.NET Core Web API image containing all the dependencies required for its execution. Each service connects to its own SQL Server database.

The microservice system version of the Library system is built into an image through a Docker file[21] and Docker Compose[22]. Then, Kubernetes runs several instances of each microservices to handle redundancy, scalability, among other features.

### 4.3. System Deployment

The *LibraryService* is deployed as a local version running on a desktop computer and a cloud version running in Azure. This section describes the technical details of both deployments.

***4.3.1. Local Version*** A server is a desktop machine with 8GB DDR3 1600 MHZ RAM, an Intel i5-4590 @3.3 GHz quad-core CPU and 250GB SSD. It runs both the web API and the database server. The client machine is a laptop with 8GB DDR 3200MHZ RAM, an Intel i5-7300HQ @2.50 GHz quad-core CPU, and 250GB SSD. The server and the client connected via WLAN. The local version is deployed on a desktop computer running Docker Desktop[23] which also has a built-in version of Kubernetes. All the images are hosted on Docker Hub registries that are pulled down and executed by local Kubernetes. Kubernetes is configured through a series of deployment and service files: one deployment and service file for each service.

An ingress solution by NGINX[24] was used to achieve local load balancing locally. For this to work locally, the Kubernetes service is configured as NodePorts on port 80, which lets the ingress controller route properly. The source code of the monolith and microservice system version deployed locally can be found at GitHub[25][26].

***4.3.2. Cloud Version*** A professional-grade hosting solution is necessary to support a large user base such as a data-centre or a cloud service. This study uses Microsoft Azure[27] the cloud provider for easy integration and leverage from the authors' expertise with the platform. Amazon's cloud service[28] AWS or Google's cloud service Google Cloud[29] pose as viable alternatives.

***4.3.3. Monolith Cloud Deployment*** The monolith cloud version was built with Azure App Service[30] to host the web-server. App Services generally allows running between one and 30 instances of the web server, depending on the virtual machine hosting it. The app service instances connect to an Azure SQL Database[31], which is configured as serverless rather than running in a dedicated virtual machine to reduce the cloud credits that are spent. The database is configured to run with six virtual CPU cores (vcores[32]), 32 GB of storage and a maximum of 18 GB of memory. This version was hosted on two different hosting configurations[33]. The first configuration is a P1V2 service plan with one CPU core, 3.50 GB memory and 12 server instances. The second configuration is a P3V2 service plan with four CPU core, 14 GB memory and 30 instances (maximum amount of instances). Figure 4 presents the monolithic app running on Azure's App Service in multiple instances. The source code of the monolith system deployed in the cloud has been made available[34] publicly by the authors.

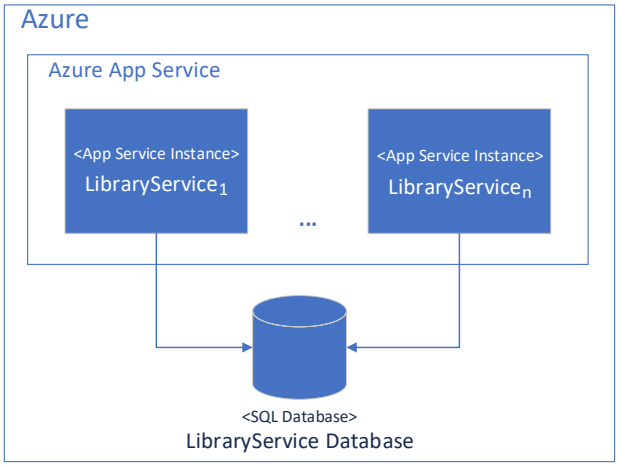


**Figure 4** Monolithic version deployed on the cloud.

***4.3.4. Microservice cloud deployment*** The microservice version is built with Azure Kubernetes Service[35] (AKS) to manage Kubernetes and associated containers. The container images are hosted on Azure registry ACR, from where AKS pull the

[19] https://kubernetes.io/docs/concepts/services-networking/ingress/
[20] https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions
[21] https://docs.docker.com/engine/reference/builder/
[22] https://docs.docker.com/compose/
[23] https://www.docker.com/products/docker-desktop
[24] https://kubernetes.github.io/ingress-nginx/
[25] https://github.com/NichlasBjorndal/LibraryService-Monolith
[26] https://github.com/NichlasBjorndal/LibraryService-Microservice-DotNet
[27] https://azure.microsoft.com/en-us/
[28] https://aws.amazon.com/
[29] https://cloud.google.com/
[30] https://azure.microsoft.com/en-us/services/app-service/
[31] https://azure.microsoft.com/en-us/services/sql-database/
[32] https://docs.microsoft.com/en-us/azure/sql-database/sql-database-service-tiers-vcore
[33] https://azure.microsoft.com/en-us/pricing/details/app-service/windows/
[34] https://github.com/NichlasBjorndal/LibraryService-Monolith-Cloud
[35] https://azure.microsoft.com/en-us/services/kubernetes-service/

containers images. The databases are configured with similar resources to those shown in Section 4.3.3. The microservice version was hosted on two different configurations[36]. The first configuration consists of two DSv2 virtual machines nodes which each has two CPU cores, 3.50 GB memory. The second configuration consists of five DSv2 virtual machine nodes. Each service has its external IP, mapped to a subdomain[37] where the load balancer distributes incoming requests amongst the nodes and their pods. Figure 5 summarises the microservice version deployed in the cloud and presents the similarity to the locally deployed microservice version[38]. Unlike the previous setup, there are multiple nodes, while Kubernetes services use Azure's load balancer to route traffic.

## 5. Results

The metrics selected in Section 3 are used to compare the performance and features of the implemented versions of the *LibraryService*. Two benchmarking experiments were performed: one running locally, another in the cloud resembling a real-world system. The data collected during the experiments have been publicly available[39]. The metrics collected during the experiments are measured as follows:

- Throughput: number of successful requests per second over a period. In this study, a successful request is defined as the HTTP response code 2xx[40] with latency lower than or equal to 200 ms. The higher throughput, the better the system.
- Latency: the difference in milliseconds between the moment when the client sends the HTTP request and receives the HTTP response. The lower the latency, the better the system.
- Scalability: the ratio between the percentage of additional throughput and the percentage of additional resources (Bermbach et al. 2017). The higher the scalability, the better the system.
- CPU, memory and network: These hardware-related metrics are calculated according to their use in the host system.

### 5.1. Experiment 1: Local deployment

This experiment consists of testing both versions of the *LibraryService* deployed on regular consumer-grade hardware. The benchmark uses two synthetic workloads - a simple and a complex. 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 generated and executed by Apache JMeter[41] (JMeter) on the client. JMeter uses a constant throughput timer to calculate the number of desired requests over a minute, HTTP responses, latency and other metrics used to determine the throughput and scalability. The server uses Window built-in monitoring app and performance monitor to record the CPU utilization, memory usage and network traffic.

The so-called 'simple workload' consists of two actions: collecting data about a specific book and creating a new user. Getting the information about a book generates an HTTP GET request that results in a read operation database. Creating a new user generates an HTTP POST request that results in a write operation in the database. The 'complex workload' consists of collecting information about a specific loan and create an order and a physical book. Retrieving a loan generates an HTTP GET request that requires a join of several tables in the monolithic version and multiple HTTP GET requests for the microservice version. Creating an order generates an HTTP POST request that has write operations on multiple tables. A microservice requires communication between the Order service and Book service using a message queue to create the order and the physical book.

The workloads were implemented on 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 tested against many requests per minute. The samples are collected in batches of 2500 requests per minute until the average latency rises above 200 ms, indicating that the system has achieved maximum throughput (also referred to as break-off point). While the pre-determined batch size allows finding patterns, it is not considerably time-consuming. In each test iteration, latency, memory usage, CPU utilization, and network usage are measured.

The monolith system is executed in release mode through the *dotnet run*[42] command in PowerShell[43]. The microservice version is deployed on a locally running instance of Kubernetes through Docker Desktop. Each microservice is replicated to three running instances, using a ReplicaSet[44] configured to run four CPU cores and 3GB RAM. During a workload, the microservices were configured with nine pods, and unused services were configured with three pods. This experiment collected 15,000 entries through the Entity Framework, a number considerably higher while it prevents stack overflow exceptions during database migration.

#### 5.1.1. Latency results

Figure 6 presents the results regarding latency metric. Both simple workloads and complex monolithic workload perform similarly. However, the simple microservice workload crosses the 200ms threshold much earlier than the monolithic counterpart. The monolith workload has a higher latency than the simple workload, but they 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.

[36] https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-general
[37] https://tools.ietf.org/html/rfc1034#section-3.1
[38] https://github.com/NichlasBjorndal/LibraryService-Microservice-DotNet-Cloud
[39] https://github.com/NichlasBjorndal/LibraryService-Appendices/tree/master/Benchmarking%20Data
[40] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
[41] https://jmeter.apache.org/
[42] https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-run?tabs=netcore30
[43] https://docs.microsoft.com/en-us/powershell/
[44] https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

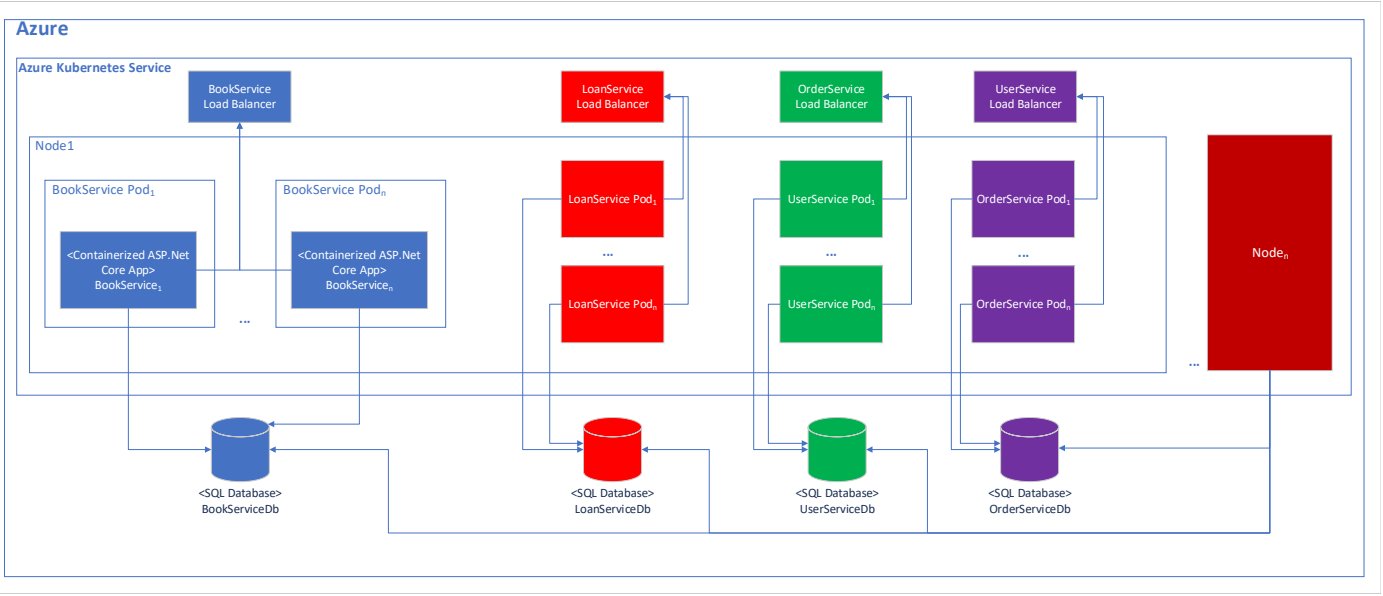


**Figure 5** Summary of the microservice system deployed in the cloud.

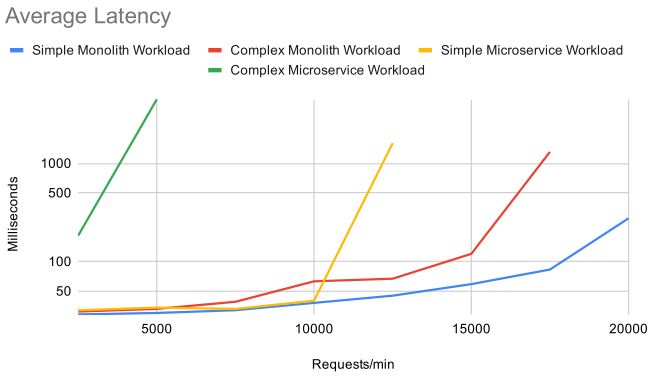


**Figure 6** Average latency per workload in experiment 1.


**Figure 7** Throughput per workload in experiment 1.

It can be observed that it is not always favourable migrating to a microservice architecture if it is not done correctly. The general lower latency of the monolithic system shows that the monolith performs better for this metric.

***5.1.2. Throughput results*** Figure 7 presents the throughput for different workloads, which apparent to mirror the latency in Figure 6 regarding the break-off points. The selected workloads perform similarly until 10,000 requests per minute, except by the complex microservice. The monolith system handled more than 1.5 times successful requests for the simple workload than its microservice counterpart. On the other hand, the microservice system underperformed during the complex workload according to metrics trend. Higher throughput in both simple and complex workloads favour the monolith architecture for the setup used in this experiment.

***5.1.3. Scalability results*** In this experiment, the increase in performance is measured by the number of requests per minute a system can handle with an average latency of 200 ms. The two workloads were benchmarked a second time, only granting the two reference systems access to 2 CPU cores, and the scalability will be calculated as the ratio between the increase of requests per minute and the increase in hardware capacity. Figure 8 shows the two workloads being run on a 2 and 4 CPU core on the monolith system. The results show an extensive range of latency values. Moreover, the simple monolith workload with four cores performs the best, and the complex workload with two cores performs the worst, as one could expect. Interestingly, the simple monolith workload with two cores performs similar to the complex workload with four cores.

Figure 9 shows the performance of two workloads being executed on a 2 and 4 CPU core on the microservice system. It can be seen that the simple workload with four cores is the only setup that the microservice system can handle at higher than 2,500 requests per minute. The worst performance comes from

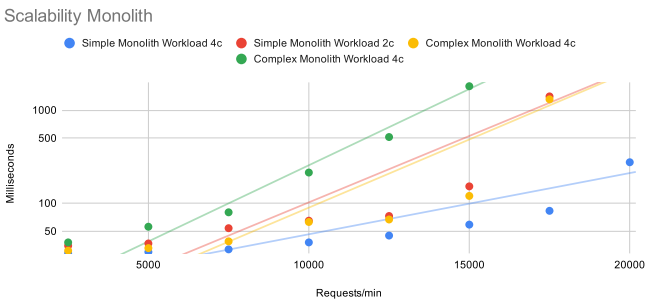


**Figure 8** Scalability for the monolith system in experiment 1.


**Figure 9** Changes in the response time per number of requests for the microservice system in the experiment 1.

the complex workload of two cores, where the peak latency is 9,011 ms at 5,000 requests per minute, which indicates that the microservice is not suited to handle a complex workload on a setup as the one used in this experiment.

Exponential regression has been applied to generate trendlines, which will be used to calculate the number 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. The number 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 are presented in Table 4 and Table 5.

The scalability is calculated as the ratio between the increase in requests per minute and increase in CPU cores. For simple workload, the microservice scales approximately three times better than the monolith system, as expected by the hypothesis mentioned in Section 1. The results shown in Table 6 also suggest that Kubernetes/Docker only having access to two desktop CPU cores is not adequate and lead to under-performance, reducing the number of requests per minute it can handle.

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

**Table 4** Number of requests per minute of the monolith system under 200 ms latency.

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

**Table 5** Number of requests per minute that the microservice system can support under 200 ms latency

| Workload | Request Increase | CPU Cores Increase | Scalability |
|---|---|---|---|
| Monolith Simple | 7,649 | 2x | 0.32 |
| Monolith Complex | 3,042 | 2x | 0.16 |
| Microservice Simple | 5,968 | 2x | 1.14 |
| Microservice Complex | 271 | 2x | 0.06 |

**Table 6** Scalability rate per workload.

***5.1.4. CPU results*** Figure 10 presents the CPU utilisation under different workloads. It can be seen that the monolithic version uses less CPU for handling the same amount of requests per minute than the microservice version. The CPU usage of the simple monolith, complex monolith and simple microservice increases until it reaches maximum throughput. The simple microservice workload decreases after 10,000 requests per minute when it reaches its maximum throughput, indicating that other factors but the CPU impose limitations on its performance. It is noteworthy that the use of 12 pods to improve CPU utilization caused the system to halt with 100% usage at a relatively low to medium amount of requests per minute, between 7,500 and 10,000 requests.

The results also showed that the simple microservice workload reduces CPU utilization when it reaches instead of increasing like the three other workloads. A similar trend is observed for network usage, as shown in Figure 12. These indicate that the microservice system, possibly the load balancer or Kubernetes/Docker network service, is overloaded and, as a result, stops being able to handle more network requests. It can also be noted that the complex workload reaches nearly 80% CPU usage, preventing the system from handling more than 5,000 requests per minute. Therefore, the CPU usage of the monolithic system is lower, which is another metric in favour of the monolith.

***5.1.5. Memory results*** Figure 11 presents the memory utilization under different workloads. Both monolithic and microservice systems use approximately the same memory, with

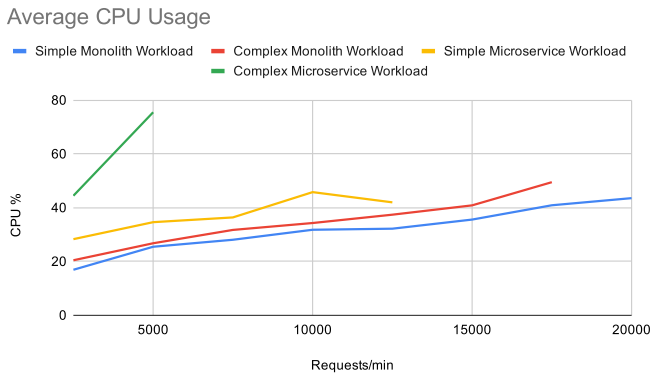


**Figure 10** CPU utilization per different workloads in experiment 1.

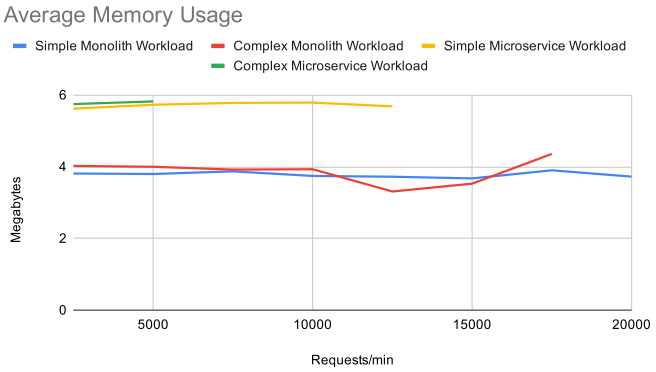


**Figure 11** Memory utilisation per workload in experiment 1.

the latter using approximately 2 GB more of memory. This results from the overhead caused by virtualization since Containerization/Kubernetes require additional resources. Interestingly, the memory consumption remains approximately constant throughout the workload, except for the complex monolith workload (see Section 5.1.7).

***5.1.6. Network results*** The network usage of the different workloads can be seen in Figure 12. The network usage increases to approximately 10,000 requests per minute, except for the complex microservice workload. The complex workload shows the network impact of a microservice that requires much communication with other services [45] and requires higher network usage. The decline 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.

***5.1.7. Threat to validity*** Some design decisions and infrastructure used in experiment 1 can offer some threats to the

[45] https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

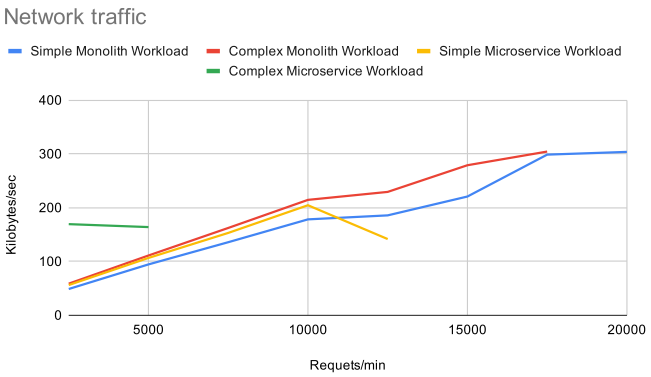


**Figure 12** Network utilisation per workload in experiment 1.

results' validity. First, the use of the monolithic library's domain model does not comply with a domain-drive design and might incur threats to the current approach. Such a threat will be addressed in future research. Second, the desktop machine did not run the two LibraryService systems in complete isolation. Other background processes executed in the background, affecting CPU and memory baseline in experiment 1. Computationally expensive services not related to the Library system, i.e., those requiring more than 5% CPU and 100 MB, were interrupted. However, many low demand processes could slightly affect the performance metrics, mainly CPU and memory usage. Finally, network communication was based on a local 2.4 GHz WiFi network, resulting in higher latency and a higher risk of packet loss than a wired connection. This setup can impact the average latency and throughput.

### 5.2. 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[46] will be used to test the two systems at scale with different VM configurations. It is determined that it was only feasible to measure the three most important metrics that are automatically captured: throughput, latency and scalability. Differently from experiment 1, throughput is defined as all the successful requests for a given interval.

***5.2.1. Benchmark Design*** The performance testing tool allowed only HTTP GET requests. As a result, the workload used in this experiment was a lighter version of the simple workload from experiment 1. Both versions of the LibraryService (monolithic and microservice) are configured with two different hardware configurations. The monolithic version is first configured with one CPU core, 3.50 GB memory, and 12 server instances. An alternative configuration had four CPU cores, 14 GB memory, and 30 server instances, i.e., a four times increase in resources. The latter configuration is the highest possible configuration for the Azure subscription.

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

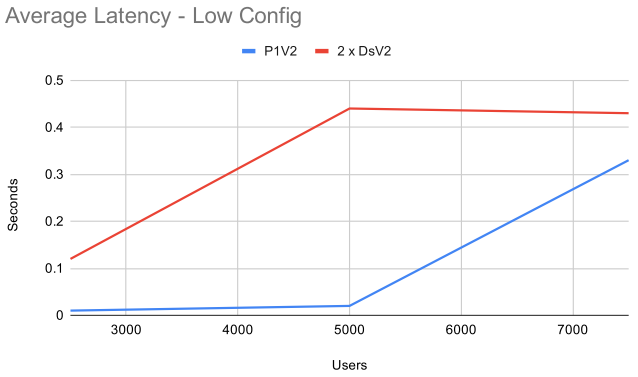


**Figure 13** Latency for low configuration in experiment 2.

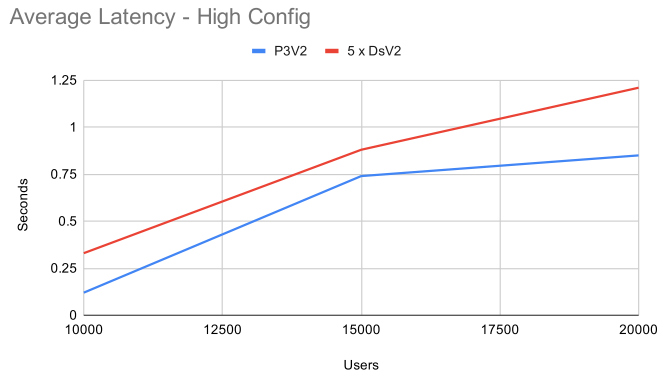


**Figure 14** Latency for high configuration in experiment 2.

The microservice system's first configuration has two nodes of 2 CPU cores, 7 GB memory, 24 BookService pods, and three pods for each of the other services. The second setup had five nodes of similar capacity and 144 BookService pods, representing a 2.5 times increase in computational power. The five nodes with 10 CPU cores in total and 30 pods per node were the maximum capacity allowed by the Azure subscription used for this experiment. The lower hardware configurations, which consist of P1V2 or two DSv2 VMs, were tested with 2,500, 5,000, and 7,500 concurrent users generating GET requests BookService API endpoint for 2 minutes. The high hardware configurations, which consist of P3V2 or five DSv2 nodes VMs, were tested with 10,000, 15,000 and 20,000 concurrent users for 1 minute (to conserve the cloud credits) reaching the same API endpoint.

### 5.3. Latency results

The average latency for the lower config is shown in Figure 13. The graph shows that the monolith version performs better in the beginning. However, the difference number between the two systems decreases while the amount of requests increases.

Figure 14 shows that both systems present similar average latency for the high configuration. However, the monolith version performs slightly better for 20,000 concurrent users.

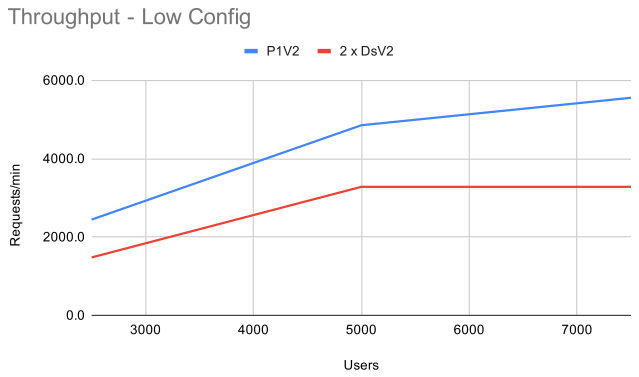


**Figure 15** Throughput for low configuration in experiment 2.

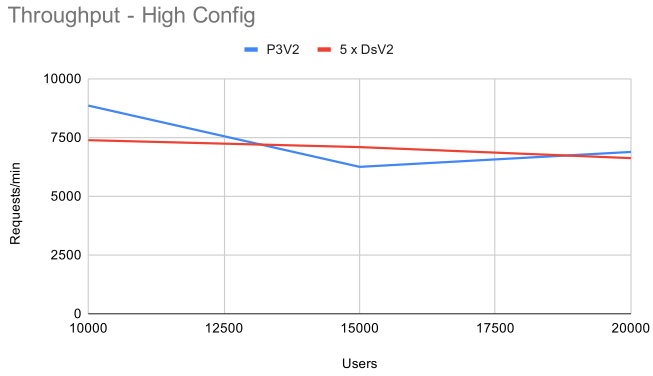


**Figure 16** Throughput for high configuration in experiment 2.

### 5.4. Throughput results

The throughput for the lower configuration can be seen in Figure 15, which shows that, like with the latency, the monolithic system performs better than the microservice system.

The throughput for the higher configuration can be seen in Figure 16 where the monolith again initially performs better. However, both systems have similar throughput as the number of concurrent users increases.

### 5.5. Scalability results

The ration between the number of concurrent users while maintaining an average latency of no more than 200 ms was determined by the cloud systems scalability. This study used polynomial regression to determine the number of users a system and hardware configuration can handle. Figure 17 and Figure 18 show the polynomial trend lines with respective $R^2$ values.

Table 7 presents the number of concurrent users a system can handle while maintaining a maximum average latency of 200 ms for the given hardware configuration.

The scalability is calculated by the ratio between the increase in concurrent users and the resultant increase in hardware power. The results before and after the increase are shown in Table 8. The microservice system's scalability ratio is over six times bigger than the monolith system, indicating that the microservice version scales considerably better with increased hardware

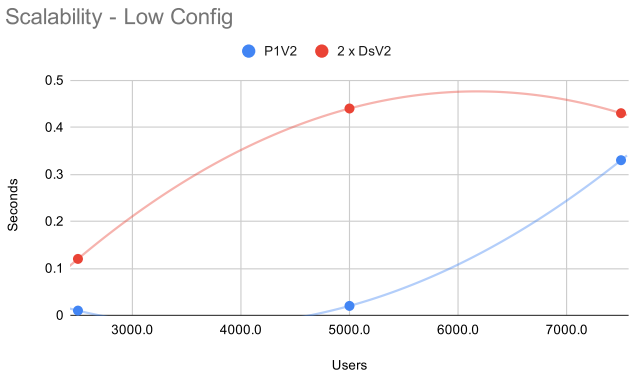


**Figure 17** Scalability for low configuration in experiment 2.

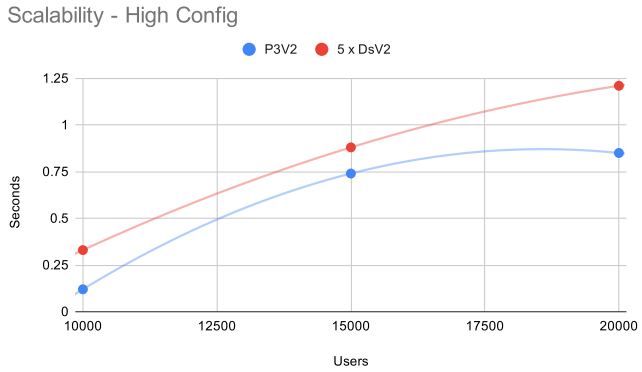


**Figure 18** Scalability for high configuration in experiment 2.

resources.

***5.5.1. Threat to validity*** The first threat to validity is the number of pods/app service instances used in experiment 2. It was challenging to collect the specific CPU and memory utilisation to optimise the exact amount of pods/instances, which required preliminary testing. App Service and Kubernetes auto horizontal scaling features based on CPU and memory usage were used. However, Azure requires a fair amount of time for the auto-scaler to reach the ideal performance, with the system properties only being sampled once a minute. Another threat to validity is the allocation of cloud resources. The databases are serverless, requiring no additional resources when they have not been used in a while. However, there is an initial startup period where the resources must re-allocated.

| Hardware configuration | Result 1 | Result 2 |
|---|---|---|
| P1V2 | 6,713 | 621 |
| 2 x DSv2 | 2,939 | 9,410 |
| P3V2 | 10,470 | 26,747 |
| 5 x DSv2 | 9,046 | 40,954 |

**Table 7** Number of concurrent users that each configuration can handle under 200 ms average latency.

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

**Table 8** Scalability ratio for the monolith and microservice systems.

## 6. Discussion

The results presented in this study demonstrate that the monolith system performed better for all the selected metrics, except scalability. Several factors can cause this outcome, such as, for example, the reference systems are not enterprise-sized, Kubernetes/Docker overhead, and the hardware configurations used for experiments. Abbott & Fisher (2015) describe the three main implementations for scalability: running multiple copies of an application load-balanced across servers; running identical copies of code across multiple servers with each server responsible for only a subset of the application; breaking the application into components and microservices. Moreover, upscaling microservices require handling several components and services, which should be done simultaneously or by identifying individual components to upscale (Dragoni et al. 2017).

The reference systems are relatively small compared to large enterprise systems consisting of millions of code lines and heavy use of libraries/other dependencies. Hence, executing instances of a monolithic web server often comes with higher resource costs than the microservice equivalent. However, there is no significant difference in scaling up the resource cost. This leads to the second point being that Docker and Kubernetes come with an extra resource overhead. Hence, the overhead of executing an additional microservice instance might be more costly than adding a monolithic server instance.

Experiment 1 showed an increased overhead to provide a similar, or even lower, throughput for the microservice than the monolith version. One of the reasons for the higher resource utilisation is the SQL server bottleneck (Wescott 2013) as it did not have access to enough system resources to perform correctly. The under-performing database is also a likely reason for the microservices in experiment 1 performing worse than having a separate machine as a dedicated database server like in experiment 2.

The metrics of resource utilisation in experiment 1 were not surprising, considering that Kubernetes is appropriate for a cluster rather than a single computer. 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 suitable when it requires several concurrent users. Experiment 2 was a simpler version of experiment 1 since it used a lighter version of the simple workload. Although this setup gives a less nuanced picture of the system's overall perfor-

mance, it allowed testing the scalability better with more and faster hardware while handling a much higher load.

While analysing the results reported here, the reader should bear in mind the study's limitations. Sections 5.1.7 and 5.5.1 presented threats to validity, which include some design decisions and infrastructure, and the difficulty measuring some performance indicators for remote computer resources. Firstly, the results show that a small to medium-sized monolith system generally uses less computational resources while performing better than its microservice counterpart. However, the microservice architecture redeems itself with its scalability, allowing it to more straightforward scale-up. It also showed the risks involved in a microservice migration, such as the complex microservice workload underperformed across the board. In the case of the complex microservice, it is essential to design the microservices with well-bounded contexts to avoid communication between services unless necessary.

Second, the experiments conducted in this study demonstrated that the selected performance metrics are valuable to benchmark different software architectures. Some metrics are beneficial for evaluation, including latency, throughput and scalability. CPU, memory and network are additional performance indicators that explain a system's latency, throughput and scalability. An automatic auto-scale for App Service instances and Kubernetes instead of a manual tuning could leverage from the collection of these features, leading to optimized configurations.

## 7. Conclusion

This study demonstrated the different performances obtained during the migration from a monolith architecture to a microservice architecture. An extensive and structured literature review was conducted to identify the most relevant performance indicators to be considered in such a type of project. Also, a survey conducted among professionals in the real-world confirmed the relative importance of these features. Then, two experiments were conducted to compare the performance of both a monolithic and microservice-based version of a system developed for this study. The results stressed the relevance of the following metrics for assessing architecture migration: latency, throughput, scalability, CPU, memory and network utilization.

Two reference systems were developed for the benchmarking, a monolithic and a microservice-based version of the *LibraryService*. Then, two experiments were performed - the first using a local desktop machine and the second being cloud-based. In the first experiment, the monolith outperformed the microservice version, especially for complex workloads. However, the microservice system had a three times higher scalability ratio compared to its monolith counterpart, demonstrating a scenario in which the migration is strongly recommended. In the second experiment, the monolith had better performance leveraging from more robust IT infrastructure. However, the microservice system again had a better scalability ratio compared to the monolith system.

In summary, the monolithic architecture appears 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 microservice-based architectures outperform monolith-based architectures for systems that must support many concurrent users.

Future work can explore how the metrics assess real-world (i.e. complex and large) systems used in the industry. Another research direction is to investigate the effects of using a polyglot microservice architecture, i.e., which performance metrics are mostly affected by diverse programming languages. Lastly, a significant extension would be adding security as a requirement. Currently, the system neither uses encryption nor authentication. Ideally, all communication would go through TSL[47].

## References


Abbott, M. L., & Fisher, M. T. (2015). *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Addison-Wesley Professional.

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., & Steinder, M. (2015, Sep.). Performance evaluation of microservices architectures using containers. In *2015 ieee 14th international symposium on network computing and applications* (p. 27-34). doi: 10.1109/NCA.2015.49

Aniche, M., Treude, C., Zaidman, A., v. Deursen, A., & Gerosa, M. A. (2016, Oct). Satt: Tailoring code metric thresholds for different software architectures. In *2016 ieee 16th international working conference on source code analysis and manipulation (scam)* (p. 41-50). doi: 10.1109/SCAM.2016.19

Antunes, N., & Vieira, M. (2012, Nov). Detecting vulnerabilities in service oriented architectures. In *2012 ieee 23rd international symposium on software reliability engineering workshops* (p. 134-139). doi: 10.1109/ISSREW.2012.33

Aragon, H., Braganza, S., Boza, E., Parrales, J., & Abad, C. (2019). 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* (pp. 746–750). New York, NY, USA: ACM. doi: 10.1145/3308560.3316466

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, *33*(3), 42–52.

Bermbach, D., Wittern, E., & Tai, S. (2017). *Cloud service benchmarking: Measuring quality of cloud services from a client perspective*. doi: 10.1007/978-3-319-55483-9

Bondi, A. B. (2016). 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* (pp. 327–330).

Boukharata, S., Ouni, A., Kessentini, M., Bouktif, S., & Wang, H. (2019). Improving web service interfaces modularity using multi-objective optimization. *Automated Software Engineering*, *26*(2), 275–312.

Brummett, T., Sheinidashtegol, P., Sarkar, D., & Galloway, M. (2015, Nov). Performance metrics of local cloud computing architectures. In *2015 ieee 2nd international conference on*


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

*cyber security and cloud computing* (p. 25-30). doi: 10.1109/CSCloud.2015.61

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

Cardarelli, M., Iovino, L., Di Francesco, P., Di Salle, A., Malavolta, I., & Lago, P. (2019). An extensible data-driven approach for evaluating the quality of microservice architectures. In *Proceedings of the 34th acm/sigapp symposium on applied computing* (pp. 1225–1234). New York, NY, USA: ACM. doi: 10.1145/3297280.3297400

Conway, M. E. (1968). How do committees invent. *Datamation*, *14*(4), 28–31.

Curtis, B., Sappidi, J., & Subramanyam, J. (2011, May). An evaluation of the internal quality of business applications: does size matter? In *2011 33rd international conference on software engineering (icse)* (p. 711-715). doi: 10.1145/1985793.1985893

de Souza Pinto, R., Botazzo Delbem, A., & Monaco, F. (2018). Characterization of runtime resource usage from analysis of binary executable programs. *Applied Soft Computing Journal*, *71*, 1133-1152. doi: 10.1016/j.asoc.2017.12.040

Dragomir, A., & Lichter, H. (2014, Dec). Towards an architecture quality index for the behavior of software systems. In *2014 21st asia-pacific software engineering conference* (Vol. 2, p. 75-82). doi: 10.1109/APSEC.2014.97

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, 195–216.

Düllmann, T. F., & van Hoorn, A. (2017). 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* (pp. 171–172). New York, NY, USA: ACM. doi: 10.1145/3053600.3053627

Elsayed, M., & Zulkernine, M. (2019). Offering security diagnosis as a service for cloud saas applications. *Journal of Information Security and Applications*, *44*, 32-48. doi: 10.1016/j.jisa.2018.11.006

Fan, C.-Y., & Ma, S.-P. (2017). Migrating monolithic mobile application to microservice architecture: An experiment report. In *2017 ieee international conference on ai & mobile services (aims)* (pp. 109–112).

Ferreira, C. H. G., Nunes, L. H., Pereira, L. A., Nakamura, L. H. V., Estrella, J. C., & Reiff-Marganiec, S. (2016, June). Peesos-cloud: A workload-aware architecture for performance evaluation in service-oriented systems. In *2016 ieee world congress on services (services)* (p. 118-125). doi: 10.1109/SERVICES.2016.25

Flygare, R., & Holmqvist, A. (2017). *Performance characteristics between monolithic and microservice-based systems.*

Franks, G., Lau, D., & Hrischuk, C. (2011). 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* (pp. 63–72).

Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., ... others (2019). An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems* (pp. 3–18).

Gesvindr, D., & Buhnova, B. (2019, March). Paasarch: Quality evaluation tool for paas cloud applications using generated prototypes. In *2019 ieee international conference on software architecture companion (icsa-c)* (p. 170-173). doi: 10.1109/ICSA-C.2019.00038

Hadjilambrou, Z., Kleanthous, M., & Sazeides, Y. (2015, March). Characterization and analysis of a web search benchmark. In *2015 ieee international symposium on performance analysis of systems and software (ispass)* (p. 328-337). doi: 10.1109/ISPASS.2015.7095818

Heyman, T., Preuveneers, D., & Joosen, W. (2014, Aug). Scalability analysis of the openam access control system with the universal scalability law. In *2014 international conference on future internet of things and cloud* (p. 505-512). doi: 10.1109/FiCloud.2014.89

Ibrahim, A. A. Z. A., Wasim, M. U., Varrette, S., & Bouvry, P. (2018, July). Presence: Performance metrics models for cloud saas web services. In *2018 ieee 11th international conference on cloud computing (cloud)* (p. 936-940). doi: 10.1109/CLOUD.2018.00140

Iqbal, M. A., Saltz, J. H., & Bokhart, S. (1986). Performance tradeoffs in static and dynamic load balancing strategies.

Kratzke, N., & Quint, P.-C. (2017). Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. *Journal of Systems and Software*, *126*, 1–16.

Lavrakas, P. J. (2008). *Encyclopedia of survey research methods*. Sage Publications.

MacKenzie, M. C., Laskey, K., McCabe, F., Brown, P. F., Metz, R., & Hamilton, B. A. (2006). Reference model for service oriented architecture 1.0. *OASIS Standard*, *12*.

Martinez-Millana, A., Fico, G., Fernández-Llatas, C., & Traver, V. (2015). Performance assessment of a closed-loop system for diabetes management. *Medical & biological engineering & computing*, *53*(12), 1295–1303.

Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S. T., & Dustdar, S. (2018). Microservices: Migration of a mission critical system. *IEEE Transactions on Services Computing*.

Mohsin, A., Naqvi, S. I. R., Khan, A. U., Naeem, T., & AsadUllah, M. A. (2017, April). A comprehensive framework to quantify fault tolerance metrics of web centric mobile applications. In *2017 international conference on communication technologies (comtech)* (p. 65-71). doi: 10.1109/COMTECH.2017.8065752

Ouni, A., Wang, H., Kessentini, M., Bouktif, S., & Inoue, K. (2018, December). A hybrid approach for improving the design quality of web service interfaces. *ACM Trans. Internet Technol.*, *19*(1), 4:1–4:24. doi: 10.1145/3226593

Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic

mapping study. In *Closer (1)* (pp. 137–146).
Pandey, A., Vu, L., Puthiyaveettil, V., Sivaraman, H., Kurkure, U., & Bappanadu, A. (2017, July). An automation framework for benchmarking and optimizing performance of remote desktops in the cloud. In *2017 international conference on high performance computing simulation (hpcs)* (p. 745-752). doi: 10.1109/HPCS.2017.113
Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. In *Pioneers and their contributions to software engineering* (pp. 479–498). Springer.
Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008). Systematic mapping studies in software engineering. In *Ease* (Vol. 8, pp. 68–77).
Petersen, K., Vakkalanka, S., & Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, *64*, 1–18.
Ren, Z., Wang, W., Wu, G., Gao, C., Chen, W., Wei, J., & Huang, T. (2018). Migrating web applications from monolithic structure to microservices architecture. In *Proceedings of the tenth asia-pacific symposium on internetware* (pp. 1–10).
Shukla, A., Chaturvedi, S., & Simmhan, Y. (2017). Riotbench: An iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, *29*(21), e4257.
Sriraman, A., & Wenisch, T. F. (2018, Sep.). µ suite: A benchmark suite for microservices. In *2018 ieee international symposium on workload characterization (iiswc)* (p. 1-12). doi: 10.1109/IISWC.2018.8573515
Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, *4*(5), 22–32.
Tekli, J. M., Damiani, E., Chbeir, R., & Gianini, G. (2011). Soap processing performance and enhancement. *IEEE Transactions on Services Computing*, *5*(3), 387–403.
Ueda, T., Nakaike, T., & Ohara, M. (2016, Sep.). Workload characterization for microservices. In *2016 ieee international symposium on workload characterization (iiswc)* (p. 1-10). doi: 10.1109/IISWC.2016.7581269
van Eyk, E., Iosup, A., Abad, C. L., Grohmann, J., & Eismann, S. (2018). 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* (pp. 21–24). New York, NY, USA: ACM. doi: 10.1145/3185768.3186308
Vasar, M., Srirama, S., & Dumas, M. (2012). Framework for monitoring and testing web application scalability on the cloud. In (p. 53-60). doi: 10.1145/2361999.2362008
Vedam, V., & Vemulapati, J. (2012, July). Demystifying cloud benchmarking paradigm - an in depth view. In *2012 ieee 36th annual computer software and applications conference* (p. 416-421). doi: 10.1109/COMPSAC.2012.61
Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th computing colombian conference (10ccc)* (pp. 583–590).
Wescott, B. (2013). *Every computer performance book: How to avoid and solve performance problems on the computers you work with*. CreateSpace Independent Publishing Platform.
Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., & Zhao, W. (2018). Poster: Benchmarking microservice systems for software engineering research. In *2018 ieee/acm 40th international conference on software engineering: Companion (icse-companion)* (pp. 323–324).

## About the authors

**Nichlas Bjørndal** received his B.Eng in Information Technology at Aarhus Universty, Aarhus, Denmark in 2017 and a M.Sc.Eng in Computer Science and Engineering from the Technical of Denmark (DTU), Copenhagen, Denmark in 2020. The author has also studied abroad at Temple University, Philadelphia, PA, USA and University of Massachusetts Amherst, Amherst, MA USA. He currently works as an software developer at NNIT A/S, working on a similar migration as the one described in this article. You can contact him at s173086@student.dtu.dk.

**Luiz Jonatã Pires de Araújo** has a PhD in Computer Science at the University of Nottingham (UK), where he integrated the Computational Optimisation and learning (COL) lab. Currently, Dr Araujo is an Assistant Professor at Innopolis University, Russian Federation, in the *Machine Learning and Knowledge Representation* lab. Areas of interest include optimisation, evolutionary algorithms, meta-learning and algorithm selection, cutting and packing, three-dimensional irregular packing in 3D Printing applications. You can contact him at l.araujo@innopolis.university.

**Antonio Bucchiarone** is a senior researcher in the Motivational Digital Systems (MoDiS) research unit of FBK in Trento, Italy. His main research interests are: Self-Adaptive Systems, Domain Specific Languages for Socio-Technical System, and AI planning techniques for Automatic and Runtime Service Composition. He has been actively involved in various research projects in the context of Self-Adaptive Systems, Smart Mobility and Constructions and Service-Oriented Computing. He is an Associate Editor of IEEE Software, IEEE Transactions on Intelligent Transportation Systems, and IEEE Technology and Society Magazine. You can contact him at bucchiarone@fbk.eu.

**Manuel Mazzara** is a professor of Computer Science at Innopolis University (Russia) with a research background in software engineering, service-oriented architectures and programming, concurrency theory, formal methods and software verification. Manuel received a PhD in computing science from the University of Bologna and cooperated with European and US industry, plus governmental and inter governmental organizations such as the United Nations, always at the edge between science and software production. The work conducted by Manuel and his team in recent years focuses on the development of theories, methods, tools and programs covering the two major aspects of software engineering: the process side, describing how we

develop software, and the product side, describing the results of this process. You can contact him at m.mazzara@innopolis.ru.

**Nicola Dragoni** is Professor in Secure Pervasive Computing at DTU Compute, Technical University of Denmark and, part-time Professor in Computer Engineering at Centre for Applied Autonomous Sensor Systems, Örebro University, Sweden. He is also affiliated with the Copenhagen Center for Health Technology (CACHET) and the Nordic IoT Hub. He got a M.Sc. Degree (cum laude) and a Ph.D. in Computer Science at University of Bologna, Italy. His main research interests lie in the areas of pervasive computing and security, with focus on Internet-of-Things, Fog computing and mobile systems. He has co-authored 100+ peer-reviewed papers in international journals and conference proceedings, he has edited 3 journal special issues and 1 book. He is active in a number of national and international projects. You can contact him at ndra@dtu.dk.

**Schahram Dustdar** is Full Professor of Computer Science heading the Research Division of Distributed Systems at the TU Wien, Austria. He holds several honorary positions: University of California (USC) Los Angeles; Monash University in Melbourne, Shanghai University, Macquarie University in Sydney, and University of Groningen (RuG), The Netherlands (2004-2010). From Dec 2016 until Jan 2017 he was a Visiting Professor at the University of Sevilla, Spain and from January until June 2017 he was a Visiting Professor at UC Berkeley, USA. He is founding co-Editor-in-Chief of the new ACM Transactions on Internet of Things (ACM TIoT) as well as Editor-in-Chief of Computing (Springer). He is an Associate Editor of IEEE Transactions on Services Computing, IEEE Transactions on Cloud Computing, ACM Transactions on the Web, and ACM Transactions on Internet Technology, as well as on the editorial board of IEEE Internet Computing and IEEE Computer. Dustdar is recipient of the ACM Distinguished Scientist award (2009), the IBM Faculty Award (2012), an elected member of the Academia Europaea: The Academy of Europe, where he is chairman of the Informatics Section, as well as an IEEE Fellow (2016). You can contact him at dustdar@dsg.tuwien.ac.at.