Received November 30, 2021, accepted February 16, 2022, date of publication February 18, 2022, date of current version February 28, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3152803

Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation

GRZEGORZ BLINOWSKI[1], (Member, IEEE), ANNA OJDOWSKA[2], AND ADAM PRZYBYŁEK[2]
[1]Institute of Computer Science, Warsaw University of Technology, 00-665 Warsaw, Poland
[2]Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, 80-233 Gdańsk, Poland

Corresponding author: Grzegorz Blinowski (grzegorz.blinowski@pw.edu.pl)

ABSTRACT **Context.** Since its proclamation in 2012, microservices-based architecture has gained widespread popularity due to its advantages, such as improved availability, fault tolerance, and horizontal scalability, as well as greater software development agility. **Motivation.** Yet, refactoring a monolith to microservices by smaller businesses and expecting that the migration will bring benefits similar to those reported by top global companies, such as Netflix, Amazon, eBay, and Uber, might be an illusion. Indeed, for systems that do not have thousands of concurrent users and can be scaled vertically, the benefits of such migration have not been sufficiently investigated, while the existing evidence is inconsistent. **Objective.** The purpose of this paper is to compare the performance and scalability of monolithic and microservice architectures on a reference web application. **Method.** The application was implemented in four different versions, covering not only two different architectural styles (monolith vs. microservices) but also two different implementation technologies (Java vs. C# .NET). Next, we conducted a series of controlled experiments in three different deployment environments (local, Azure Spring Cloud, and Azure App Service). **Findings.** The key lessons learned are as follows: (1) on a single machine, a monolith performs better than its microservice-based counterpart; (2) The Java platform makes better use of powerful machines in case of computation-intensive services when compared to .NET; the technology platform effect is reversed when non-computationally intensive services are run on machines with low computational capacity; (3) vertical scaling is more cost-effective than horizontal scaling in the Azure cloud; (4) scaling out beyond a certain number of instances degrades the application performance; (5) implementation technology (either Java or C# .NET) does not have a noticeable impact on the scalability performance.

INDEX TERMS Software architecture, microservices, monolith, software measurement, benchmarking, performance, scalability, cloud computing, Azure.

I. INTRODUCTION

The evolution of software architectures has been driven by the need to achieve a better separation of concerns. The term separation of concerns refers to the ability to decompose and organise systems into logically cohesive and loosely-coupled modules that hide their implementation from each other and present services through well-defined interfaces [1], [2].

Nowadays, two software engineering paradigms dominate modern enterprise application development: monolithic and microservice-based architecture [3]. The first is a traditional approach in which an application is built with a single code base that includes multiple services. These services are not independently executable [4]. They communicate with end-users and external systems via different interfaces, including HTTP(S)/HTML, Web services, and REST API [5].

A microservice architecture decomposes a business domain into small, consistently bounded contexts implemented by autonomous, self-contained, loosely coupled, and independently deployable services [6]–[8]. One of the pioneers of microservices was Netflix which began moving away from its monolithic architecture in 2009 when the term *microservice* did not even exist. The term was coined by a group of software architects in 2011 and officially announced a year later at the 33rd Degree Conference in Kraków [6]. However, it did not start gaining in popularity until 2014, when Lewis and Fowler published their blog on the topic [6], while Netflix shared their expertise from a successful transition [9] paving the way for other companies. Since that

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Ali Babar.

time, microservices have received significant attention from both academia and industry [4], [10]–[18], while the spread of container technologies, such as Kubernetes and Docker [19]–[22] has helped this new architectural style to gain even more momentum, especially in cloud-based environments [23]–[25]. Indeed, microservices have been successfully adopted by global companies, such as Amazon, eBay, Zalando, Spotify, Uber, Airbnb, LinkedIn, Twitter, Groupon, and Coca-Cola.

A. MOTIVATION AND PROBLEM STATEMENT

Inspired by success stories of tech giants, many small companies or startups are considering joining the trend and are adopting microservices as a game changer. They expect that it will help them improve scalability, availability, maintainability, and fault tolerance of deployed applications [26], [27] (which have been reported as difficult to achieve in IT systems [28], [29]). Yet, microservice-based applications come with their own challenges, including:

- identifying optimal microservice boundaries [30], [31];
- orchestration of complex services (the complexity of microservices applications is pushed from the components to the integration level [15], [23]);
- maintaining data consistency and transaction management across microservices [15], [19], [32], [33];
- the difficulty in understanding the system holistically [7], [27];
- increased consumption of computing resources [7], [11], [17].

For small scale systems, these challenges may outweigh the benefits [27].

On the contrary, the aforementioned global companies moved to microservices in response to the growth pressures they faced rather than jumping on the latest trend [32]. The volume of activity their systems performed outgrew the capacity of original technology choices, while the sheer size of the systems enormously slowed down the development carried out by multiple teams [30]. Therefore, microservices have been a compelling solution for them despite the added complexity of building and running a fine-grained distributed application [32].

On top of that, there are thousands of successful businesses around the world that are built on monolithic applications. So when does a company need microservices? Unfortunately, there is limited knowledge on this topic due to a lack of empirical evidence. Having witnessed this gap, we set forth to compare the performance and scalability of monolithic and microservice architectures in the context of a system that does not have thousands of concurrent users and can be scaled vertically. The following research questions were posed to guide the study:

- **(RQ1)** What is the performance difference between a monolithic application versus a microservice application?
- **(RQ2)** Which of the two architectures and scaling approaches should be chosen to best benefit an application from scaling?
- **(RQ3)** In what circumstances do the implementation technologies (Java vs. C# .NET) have any performance advantages or disadvantages?

B. OUTLINE OF THE PAPER

The rest of the paper is organised as follows. The following section describes monolithic and microservices architectural styles, and compares and contrasts their advantages and disadvantages. Section 3 discusses related work. The research method and the experimental design are explained in Section 4. This is followed by Section 5, which contains the experimental results. In Section 6, we discuss our findings. In Section 7, we elaborate on threats to validity that are relevant to our study and how we addressed them. Finally, Section 8 concludes the study along with suggestions for future research.

II. BACKGROUND

In the subsequent sections, we present more details on both the similarities and differences and the pros and cons of the two architectural styles, as well as two primary approaches to application scaling.

A. MONOLITHIC ARCHITECTURE

Enterprise applications are often internally built according to the classic three-tier model and hence consist of: (1) user interface code (typically HTML pages and JavaScript running in a browser on the user's machine); (2) server-side business logic which handles HTTP requests, executes domain logic, retrieves and updates data from the database, and selects and populates HTML views to be sent to the browser; and (3) database backend [34]. The server-side application is a monolith - a single logical executable [6].

From the operating system's point of view, a monolithic application runs as a single process in the application server's environment. When a new version of an application is deployed, it replaces the previous version of the application in a single step (for example, to deploy an application under the JEE Application Server, a single EAR/WAR file containing the application executable must be copied to a designated folder).

The most significant advantage of the monolithic architecture is its simplicity – in comparison to distributed applications of various genres, monolithic ones are much easier to test, deploy, debug and monitor. All data is retained in one database with no need for its synchronization; all internal communication is done via intra-process mechanisms. Hence it is fast and does not suffer from problems typical to inter-process communication (IPC). The monolithic approach is a natural and first-choice approach to building an application – all logic for handling requests runs in a single process. The basic features of the development team's preferred language can be used to structure the application into classes, functions, and namespaces.

However, as the application's size and complexity grow, problems start to arise – modifying the application's source becomes harder as more and more complex code starts to behave in unexpected ways. Changes in one module may lead to unexpected behavior in other modules and a cascade of errors. The very size of the monolith results in longer start-up time, which in turn slows down the development and becomes an obstacle to continuous deployment. Over time it is increasingly harder for the development team to keep changes that related to a particular module to only affect this very module, and in effect, to retain a modular structure of the application. Also, as the application grows, the number of developers increases, which often leads to unequal workforce utilization and, in effect, losses in productivity [34].

B. MICROSERVICE ARCHITECTURE

One of the first attempts to describe the microservice architectural style was by Lewis and Fowler [6]. In their famous blog post, they defined this new architectural term as "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API". Each microservice contains its own user handling functions, business logic, and backend functions. Microservice may also include its own database service (but it is also possible to share a single backend among multiple microservices).

The main principles of this architecture are:

- Single responsibility per service – according to the SOLID principles, a single unit should only have one responsibility and at no point should two units share one responsibility or one unit have more than one responsibility.
- Microservices are autonomous – they are self-contained and independently deployable services fully responsible for executing a given business [33], [35]–[37]. Because of their autonomy, they contain all dependencies such as: libraries, the execution environments – web servers and containers or virtual machines. Thereby, microservices increase the possibility of monetization of system parts, as access to relevant microservice APIs can be charged [38].
- Services are first-class citizens – they expose service endpoints as APIs and abstract all their implementation details. The internal structure: implementation logic, architecture, and technologies (including programming language, database, etc.) are completely hidden behind the API.

It is worth mentioning that the microservice communication paradigm differs significantly from approaches based on Service Oriented Architecture (SOA) [39] such as Enterprise Service Bus (ESB), which include sophisticated and "heavy-weight" facilities of message routing, filtering, and transformation. The microservice approach favors: "smart endpoints and dumb pipes" [3]. There is no standard for communication or transport mechanisms for microservices. Microservices communicate with each other using well-standardized lightweight internet protocols, such as HTTP and REST [5], or messaging protocols, such as JMS or AMQP.

The most attractive feature of the microservice architecture is the decomposition of complex applications into smaller components which are easier to develop, manage and maintain than a single monolith application [40]. As long as the public API does not change, internal modifications of one service are more straightforward, easier, and less costly than in the case of a similar change in a traditional model. Microservices are autonomous and communicate via open protocols, hence they can be developed fairly independently and even with different technologies [5], [41]–[43].

Microservice-based applications scale well horizontally, not only in the technical sense, but also concerning the organization's structuring of developer teams, which can be kept smaller and more agile [5], [22], [44], [45]. Efforts to seamlessly integrate such options into adaptive business process management have already found their way into business practice [46]. Furthermore, splitting large applications into individual microservices provides the next degree of independence to agile teams [7], [25], which supports scaling agile methods. Each team may work on different microservices and develops user stories that affect only their microservices [44]. As long as a team does not change the contracts among services, a decision can be made within the service team rather than among several teams working on a large monolithic application [7]. Accordingly, the adoption of microservice architecture implies reducing the need for inter-team coordination, which is a serious challenge in large-scale software development [47]–[52].

Another benefit of microservice applications is that its loosely coupled architecture makes them more fault-tolerant [37] – the failure of one component does not necessarily result in the unavailability of the whole system, as functioning services can still fulfill user requests. It is also possible to identify critical business functionality and deploy corresponding microservices in a more redundant environment.

Apart from numerous advantages, the microservice architecture has its caveats and disadvantages mainly related to its distributed nature. Deployment, scaling, and monitoring of a multi-service system is a more complex task than in the case of a monolithic application. For this reason, various automatization procedures in the continuous integration / continuous delivery (CI/CD) pipeline [53], monitoring, and demand-based autoscaling are used in the development of such applications [6], [34]. To fully take advantage of short development-to-operations, life-cycle testing must also be automated, which is a more challenging task in distributed environments [42], [54]. Another challenge lies in the design of data management facilities – the principles of microservice architecture state that maximum service isolation is preferred. Hence, multiple independent database systems are introduced into the distributed application, increasing complexity and reducing manageability [34].

In this work, we focus on application performance – IPC required between microservice components introduces substantial overhead when compared to intra-process communication (function calls and method invocations) used in monolithic applications. IPC is implemented as an operating system's kernel service. In most cases, it requires data copying between user and kernel space, and hence it reduces (in many cases significantly) application performance. Applications that do not service significant user traffic will show degraded throughput and response times when migrated from monolithic to a distributed architecture. Only as the user request increases, proper scaling of a microservice-based application can outweigh communication overhead – studying of this phenomenon is, in fact, the major topic of our work.

C. VERTICAL AND HORIZONTAL SCALING

Scalability is the property of a system to handle a growing amount of work by adding resources to the system [55]. The manner in which additional resources are added defines which of two scaling approaches is taken [8], [14], [56], [57] - *vertical scaling* or *horizontal scaling*. The former, also known as *scaling up*, refers to adding more resources (CPU, memory, and storage) to an existing machine. It is the more straightforward approach, but it is limited by the most powerful hardware available on the market [56]. As for the Azure App Service, the most powerful VM instance available has only eight cores and 32 GB of RAM. Additionally, beyond a specific configuration of hardware resources, costs increase tremendously. It is worth mentioning that in cloud platforms, vertical scaling allows for adding or removing virtual resources to a running virtual machine (VM) [58], thus it does not lead to downtime.

In contrast, *horizontal scaling*, also known as *scaling out*, refers to adding more machines and distributing the workload. It is more complex because it has an influence on the application architecture, but can offer scales that far exceed those that are possible with vertical scaling [56]. Horizontal scaling is more common with microservice applications [36], even though a monolith may be also scaled out by running many instances behind a load-balancer [6]. Nevertheless, scaling out a monolithic application may not be so effective because it commonly offers a lot of services - some of them more popular than others. In order to increase the availability of a monolithic application, the entire application needs to be replicated. This leads to over scaling in non-popular services, which consume server resources even when they are idle, and in effect results in sub-optimal resource utilization [59]. On the other hand, in order to increase the availability of a microservice application, only highly demanded microservices that consume a large amount of server resources will get more instances [18].

III. RELATED WORK

A. BENCHMARKING APPLICATIONS FOR COMPARING MONOLITHIC VERSUS MICROSERVICE ARCHITECTURE

To compare the performance and scalability of the two investigated architectures, we need both monolithic and microservices versions of the same application. When exhaustively searching the web, Aderaldo *et al.* [23] found only two such systems: *Acme Air* and *MusicStore*. Three years later, Francesco *et al.* [15] conducted a systematic mapping study to characterize state of the art on architecting with microservices. After screening 103 primary studies, they identified only one benchmarking application, namely *Acme Air*. Consequently, they called for developing open-source benchmarking applications that can be used for comparing monolithic versus microservice architecture. In addition, we identified one more benchmarking application, namely *JHipster*. All three aforementioned applications are shortly discussed below.

MusicStore was originally developed by Microsoft to demonstrate ASP.NET components. Later on, it was broken up into multiple independent services by the Steeltoe team[1] to illustrate how to use their open source library aimed at developing cloud native .NET microservice applications. Unfortunately, the original monolithic implementation has not been updated since 2018, while its GitHub repository has been archived. Seeing that the monolith uses obsolete technologies, whereas the microservice-based implementation uses the third party library, *MusicStore* is not a suitable benchmark application.

Acme Air[2] simulates the website for a fictitious airline company. It is available not only in two architectural styles but also in two different languages (i.e., Java EE and Node.js). Besides, it has already been frequently used and discussed in the microservices research literature [59], [60]. Unfortunately, *Acme Air* has also not been updated since August 2015, while Java has evolved significantly in that time. Accordingly, this application also cannot be used as a benchmark.

JHipster [41] is a development platform utilized to generate web applications. It is implemented with Java Spring Boot and Angular JS frameworks. Its source code is publicly available on GitHub in both monolithic and microservice versions.[3] Nonetheless, when the Java version of our benchmark application was developed [61], we were not aware of *JHipster*, as the paper that popularizes it was published later.

B. PERFORMANCE EVALUATION OF MONOLITHIC VERSUS MICROSERVICE ARCHITECTURE

The topic of performance and cost comparison of microservice and monolithic applications has already been tackled in the literature. In [41] Al-Debagy and Martinek compared the performance of an application built both in monolithic

[1]https://github.com/SteeltoeOSS/Samples/tree/main/MusicStore
[2]https://github.com/acmeair/acmeair
[3]https://github.com/eugenp/tutorials/tree/master/jhipster

TABLE 1. Independent and dependent variables by experiment.

	Independent variables					Dependent variables	
Deployment environment	**Architecture** {monolith, microservices}	**Service** {City, Route}	**Technology** {Java, .NET}	**VM type**	**#instances**	**Performance**	**Cost**
Local	✓	✓	✓			✓	
Azure Spring Cloud	✓	✓	[a]	✓[b]	✓[c]	✓	✓
Azure App Service	✓	✓	✓	✓[d]	✓[e]	✓	✓

[a] since Azure Spring Cloud did not support .NET, we tested only Java implementations; hence *technology* was not an independent variable in this experiment

[b] {1 vCPU 1GB RAM, 2 vCPU 2GB RAM, 3 vCPU 3GB RAM, 4 vCPU 6GB RAM} [c] {1, 2, 3, 6}

[d] {B1, S1, S2, S3, P3v2} [e] {1, 3, 6, 10, 15, 20}

and microservice style. The application was developed with Spring Boot and AngularJS with Apache JMeter used as a testbed. Tests were conducted in a local environment. Response time and throughput were used as performance metrics. In concurrency testing, the monolithic version of the application showed better performance by 6% in throughput with respect to the microservice-based variant, while in the load testing scenario, there was no significant difference between the two approaches. It is worth noting that in work quoted above, the application ran in a local (i.e., non-cloud) environment, whereby neither vertical nor horizontal scaling effects were evaluated.

In [62] Garces *et al.* have conducted a cost comparison of running web applications in three architecture variants: using both monolithic and client-operated microservice under AWS EC2 cloud as well as a provider-operated microservice under AWS Lambda cloud environment. The authors used Java technology with Play and Jax-RX frameworks and Node.js. The monolithic architecture was used as a baseline, to which subsequent microservice tests referred. The measure that was used was the maximum number of requests per minute supported by a given architecture. Test results have shown that client-operated microservices indeed reduce infrastructure costs by 13% in comparison to standard monolithic architectures and in the case of services specifically designed for optimal scaling in the provider-operated cloud environment, infrastructure costs were reduced by 77%.

In [59] Ueda *et al.* analyzed the behavior of an application implemented as a microservice and monolithic variant for two popular language runtimes – Node.js and Java Enterprise Edition (EE) using Acme Air benchmark suite and Apache JMeter for performance data collection. Additionally, tests were conducted both for native process and Docker container deployments. Throughput and cycles per instruction (CPI) were used as performance metrics; the test environment was equivalent to a private cloud deployment. The authors observed a significant overhead in the microservice architecture – on the same hardware configuration the performance of the microservice model was 79% lower with respect to the monolithic model. The microservice model spent a more significant amount of time in runtime libraries to process one client request than the monolithic model, namely – 4.22x on a Node.js application server and by 2.69x on a Java EE application server.

As this short review shows, the performance test results conducted with different assumptions, and in environments which are difficult to relate to each other give conflicting results. Some show significant performance improvement for microservice, while others for monolithic architectures. In this work, we present a more comprehensive benchmark environment focusing on the cloud deployment only, and spanning multiple variants of differently scaled deployments – more information follows in the next section.

IV. METHOD

A. RESEARCH DESIGN

To answer the research questions, we carried out three controlled experiments. Each experiment was conducted in a different deployment environment (see Table 1) and investigated the effects of several factors (i.e. independent variables), which varied among the deployment environments, on *performance* and *infrastructure cost*. *Performance* was calculated as the number of requests processed per second. In turn, *infrastructure cost* was determined by the number and type of virtual machine instances that were used to deploy an application.

When it comes to our local environment, we considered the effects of *architecture* (monolith vs. microservices), *service* (City vs. Route), and *technology* (Java vs. C# .NET). Accordingly, we used a 2 × 2 × 2 factorial design. In a factorial design, each level of one factor is combined with each level of the others to produce all possible combinations. Each combination, then, becomes a condition in the experiment [63]. Note, that the infrastructure cost was the same for all experimental runs in the local environment as the same hardware configuration was used, thus we did not calculate it.

When it comes to both cloud experiments, they involved two more factors:

- *VM type* - as for Azure Spring Cloud, it specifies the number of vCPU the amount of memory for a single VM instance, while in the case of Azure App Service, it refers to a generic description of a VM;
- *#instances* - denotes the number of instances of VM type.

Since all combinations of levels for all factors would require a large number of experimental runs and a significant amount of resources, a full factorial design was not practical in that case. Besides, not all possible combinations of levels across all factors were available or interesting. For example, as for B1 machines of Azure App Service, the maximum number of instances was three, while Azure Spring Cloud supported only Java Spring Boot applications. Thereby, we used a fractional factorial design consisting of an adequately selected subset of the experimental runs of a full factorial design. All scenarios, as well as independent and dependent variables, are discussed in detail in the following subsections.

B. EXPERIMENTAL OBJECT

Since none of the existing benchmarking applications were suitable to compare the performance of the investigated architectures fairly, we developed a new one. Our benchmarking application is implemented in four functionally identical versions, covering not only two different architectural styles (monolith and microservices) but also two leading technologies intended for the development of server-side software:

- Java – implemented in Java 8[4] with Spring Boot framework 2.3.0,
- .NET – implemented in C# version 8 with ASP.NET Core framework 3.1.

The rationale behind the selection above is the following: Spring Boot is the most dominant Java framework today [65], and is definitely the best documented and most common in cloud deployments [8]; ASP.NET Core is the new version of the popular and mature ASP.NET programming environment, its codebase is licensed as open-source and available on GitHub. Similar to Spring, it is well documented and has a large developer community backing it. It is also the first and natural choice for deployments of microservice applications in the Azure cloud.

All versions of our application expose two REST endpoints that return serialized JSON objects as shown on Figure 1. The REST endpoints correspond to two services:

- *City* service – simulates a simple single object query, the input contains city name string, while response contains city data (id, name, state and population)
- *Route* service – simulates computationally intensive query, the output contains a path (an ordered series of points) being the shortest route between 10,000 randomly chosen points, each time the route is computed by the heuristic algorithm of the traveling salesman class.

The basic scheme of monolithic and microservice versions is shown in Figure 2 – in the case of the monolith, the REST API directs client queries directly to the application's business logic. In the case of the microservice version, an API

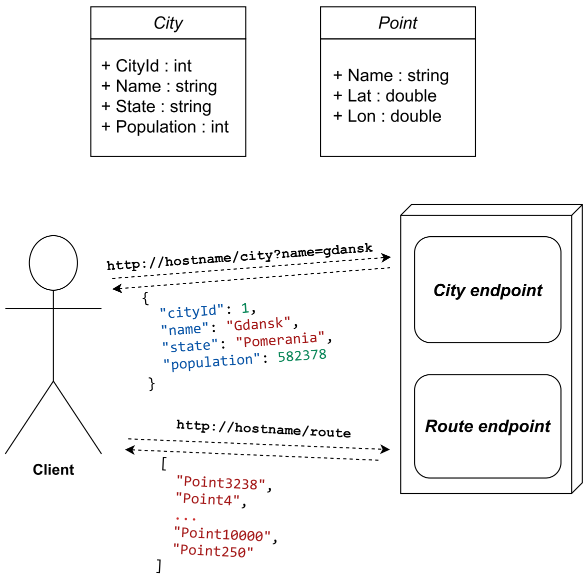


FIGURE 1. City and route services - response data and basic communication scheme.

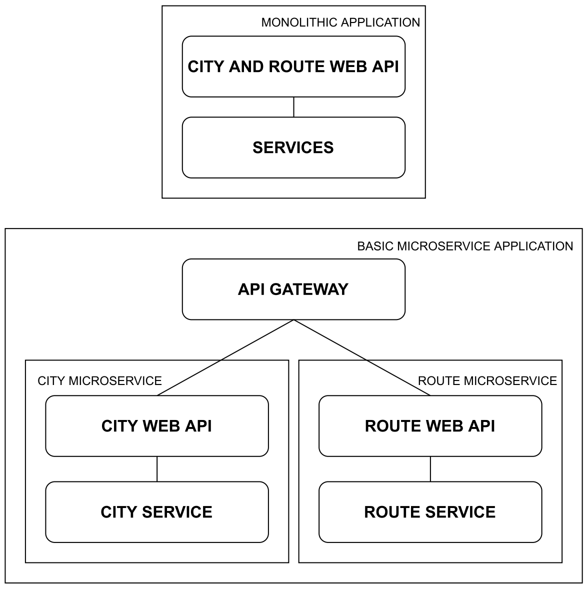


FIGURE 2. Monolithic and microservice logical architectures.

gateway is used to direct requests to the responsible service, which is a typical solution for this type of architecture.

The application was tested in a local (i.e., non-cloud) environment to established baselines and in the Azure cloud. In the case of the Java version of the application, we have tested two deployment variants: Azure Spring Cloud and Azure App Service. In the case of the monolithic version, there is no difference between the local and cloud version. In the case of microservice variants, additional components were added to enable horizontal scaling, namely – the application was extended to use Sprint Cloud framework, which

[4]The annual survey on the state of the JVM ecosystem conducted in 2020 [64], shows that 64% of developers still use Java 8 in production environments.

includes: Zuul load balancer, Spring Cloud Config, and Eureka[5] – a registry providing service discovery.

In the case of Java, the difference between Azure Spring Cloud and Azure App Service deployment is that the latter offers a "built-in" application gateway component which is a part of the framework. Still, in the case of Spring cloud, the gateway must be deployed manually as a service provided by the user. The architecture of the microservice version of our test application under Azure Sprint Cloud is shown in Figure 3.

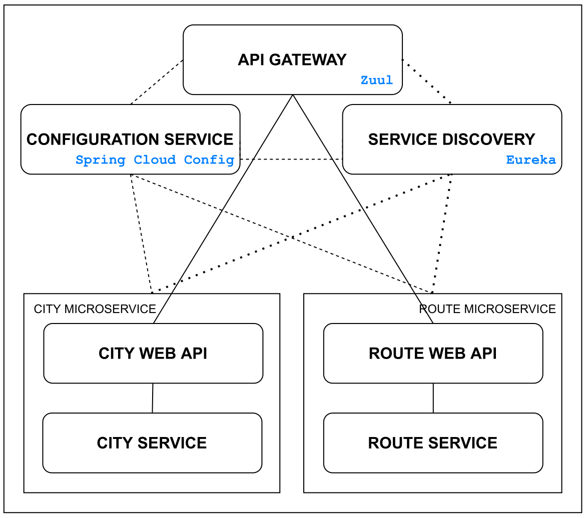


FIGURE 3. Microservices application deployed to Azure spring cloud.

C. EXPERIMENTAL SETUP

1) LOCAL ENVIRONMENT

Local tests were conducted on a PC machine running under Microsoft Windows version 10 Enterprise with the following hardware parameters: Intel(R) Core(TM) i7-9850H CPU 2.60GHz, six physical, 12 logical cores, and 32 GB RAM (Dell Precision 7540). During the local tests, only one instance of each service without scaling was used.

2) AZURE SPRING CLOUD ENVIRONMENT

Test environment configurations for Azure Spring Cloud are shown in Table 2. Unfortunately, Azure Spring Cloud does not offer fine-grained resource scaling. In the Standard Tier, if the size of consumed resources does not exceed 16 cores and 32 GB RAM, Azure Spring Cloud charges a fixed fee per hour. It means that according to the pricing list available to the US East Region in September 2020, the infrastructure costs of running our application in each deployment scenario were fixed at 1.65 USD per hour (39.6 USD per 24h). Thereby, to compare the cost effectiveness of each deployment scenario, we calculated "24H Effective Cost" by assuming fine-grained resource prices as follows: 0.08 USD per core per hour and 0.01 USD per 1GB RAM per hour. This

TABLE 2. Deployment scenarios for Azure spring cloud.

Architecture	# cores	RAM [GB]	# instances	24H Effective Cost [USD]
monolithic	1	1	1	2.16
monolithic	3	3	1	6.48
monolithic	4	6	1	9.12
microservice	1	1	1	4.32
microservice	1	1	3	8.64
microservice	2	2	2	10.8
microservice	3	3	1	8.64
microservice	1	1	6	15.12

calculation takes into account provider's overage memory price (0.00825 USD per GB-hour) and overage vCPU price (0.0783 USD per vCPU-hour). Note, that "24H Effective Cost" refers to a total cost, including the cost of API Gateway service configured with 1 CPU core and 1 GB RAM.

3) AZURE APP SERVICE

In the case of Azure App Service, the cost is based on the size and number of instances according to the *pricing tiers* shown in Table 3. (Please note that values from the "Identifier" column will be further used in this work to designate hardware resource configurations).

TABLE 3. Azure app service tiers.

Name	Identifier	# cores	RAM [GB]	Price [USD/h]	Max # instances
B1	B1:1.75	1	1.75	0.08	3
S1	S1:1.75	1	1.75	0.10	10
S2	S2:3.5	2	3.5	0.20	10
S3	S4:7	4	7	0.40	10
P3v2	P4:14	4	14	0.80	30

In the case of the total cost of microservice variants, we must also account for resources needed by the application gateway, which is 0.025 USD per work-hour (the application gateway needs only one core and 1 GB RAM).

Different deployment variants with respect to architecture, technology and resources which we tested are listed in Table 4. "24H Cost", is the cost of a 24 hour continuous run of the application with all required components.

D. PROCEDURE

We have used Apache JMeter[6] as a testing tool. JMeter is a popular open-source software tool that can simulate a load on a server, group of servers, network, or object. We have selected it because of its flexibility – it allows us to precisely configure the load configuration [66], i.e.:

- the total number of requests,
- the number of threads that will be used to execute the test, which simulates the number of users accessing the application simultaneously,
- the method of result verification.

[5]https://github.com/Netflix/eureka

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

TABLE 4. Deployment scenarios for Azure app service.

Architecture	Technology	Pricing tier	# instances	24H Cost
monolithic	.NET	B1	1	1.8
monolithic	.NET	S2	1	4.8
monolithic	.NET	S3	1	9.6
monolithic	.NET	P3v2	1	19.2
monolithic	Java	B1	1	1.8
monolithic	Java	S2	1	4.8
monolithic	Java	S3	1	9.6
monolithic	Java	P3v2	1	19.2
microservice	.NET	B1	1	2.4
microservice	.NET	B1	3	6
microservice	.NET	S1	6	15
microservice	.NET	S1	10	24.6
microservice	.NET	S2	1	5.4
microservice	.NET	S2	3	15
microservice	.NET	S2	6	29.4
microservice	.NET	S2	10	48.6
microservice	.NET	S3	1	10.2
microservice	.NET	S3	3	29.4
microservice	.NET	S3	6	58.2
microservice	.NET	S3	10	96.6
microservice	.NET	P3v2	1	19.8
microservice	.NET	P3v2	3	58.2
microservice	.NET	P3v2	6	115.8
microservice	.NET	P3v2	10	192.6
microservice	.NET	P3v2	15	288.6
microservice	.NET	P3v2	20	384.6
microservice	Java	B1	1	2.4
microservice	Java	B1	3	6
microservice	Java	S1	6	15
microservice	Java	S1	10	24.6
microservice	Java	S2	1	5.4
microservice	Java	S2	3	15
microservice	Java	S2	6	29.4
microservice	Java	S2	10	48.6
microservice	Java	S3	1	10.2
microservice	Java	S3	3	29.4
microservice	Java	S3	6	58.2
microservice	Java	S3	10	96.6
microservice	Java	P3v2	1	19.8
microservice	Java	P3v2	3	58.2
microservice	Java	P3v2	6	115.8
microservice	Java	P3v2	10	192.6
microservice	Java	P3v2	15	288.6
microservice	Java	P3v2	20	384.6

As the *performance* measure, we have used throughput, which is calculated by JMeter as the number of requests processed by the server, divided by the total time in seconds to process the requests. The time is measured from the start of the first request to the end of the last request. This includes any intervals between requests, as it is supposed to represent the load on the server.

Throughput is the most commonly used measure in similar works – see for example [42], [59]. Other measures include response time and cycles per instruction, CPI) but throughput is best suited when we also want to compare the overall costs of running the application in a given technology and configuration.

To gather performance data, we used the following test procedure: the *City* service was invoked 1000 times, the *Route* service was invoked 100 times, the number of threads was set to 10 for both scenarios. Each of the test runs was repeated 20 times. Since the cloud environments do not necessarily guarantee stable performance due to many factors related to network stability, virtual server availability, etc. – to compensate for various unexpected variations, we repeated each test series five times, computed the median of the results, and finally, we chose the test with the median being the median of obtained medians of all test runs. The validity of such an approach is discussed in "Software Microbenchmarking in the Cloud" work by Laaber *et al.* [67].

Note that when testing the performance of the microservice-based application, only one service was running. This is a common approach when it comes to horizontal scaling of microservices – each popular microservice gets its own virtual machines according to the associated load [18], [68]. Nevertheless, this approach slightly handicapped the monolithic application, where both services were running at one time even though either was idle.

V. RESULTS

In this chapter, we present and analyze the results of application performance and scalability tests.

A. LOCAL ENVIRONMENT

Tests executed in a local (non-cloud) environment let us establish a performance baseline. In Figure 4 we present throughput for monolithic and microservice applications written in .NET and Java, both for *City* and *Route* services.

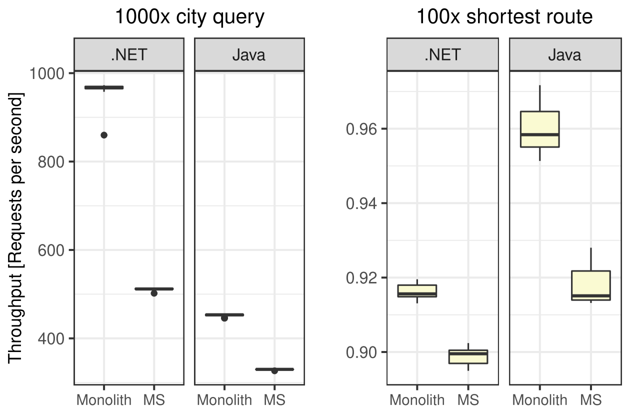


FIGURE 4. Throughput in the local environment, city service (left), route service (right). Hereinafter, MS denotes microservices.

In the microservice application, requests are passed between API Gateway and the backend service, which imposes a communication overhead that deteriorates the performance. As for the computationally intensive *Route* service, this overhead is minor compared to the service execution time. Therefore, throughput is only slightly higher for the monolithic application. In contrast, in the case of non-computationally intensive *City* service, the CPU time required to execute the service may be the same order of magnitude as the CPU time, to pass the data to and from the backend service. Accordingly, the monolith system handled on average over 2 times more requests in the .NET version, and 1.37 times more requests in the Java version.

Comparing .NET and Java, we can conclude, that in the case of the non-computationally intensive *City* service, .NET is more efficient in communication request handling than Java. This can be attributed to a high level of performance optimization in .NET Core libraries, especially regarding network request handling[7] and JSON data serialization.[8,9] On the other hand, with the *Route* service, Java implementations show better throughput than .NET both for monolithic and microservice applications – respectively by 5% and 1.5%. Some insight into CPU utilization gives an answer regarding better Java throughput in this case – CPU usage in .NET variant was noticeably lower; hence we can conclude that Java, in the case of computationally intensive applications, is more aggressive in allocating this resource to the application (RAM utilization was similar for both cases).

B. AZURE SPRING CLOUD ENVIRONMENT

As Spring is a Java-specific framework, only the Java version of the application was tested in the Azure Spring Cloud environment. We tested each service under four different CPU/RAM resource allocation variants. The highest throughput in City service 5 was achieved by microservice configured on two instances of 2 CPU & 2 GB RAM machines; next was the configuration of six instances of 1 CPU & 1 GB RAM, whereas the worst performer was one microservice instance configured with 1 CPU & 1 GB RAM. We can notice that in the case of this lightweight application, we can achieve throughput improvement when a higher number (six in this case) of instances is used – throughput of MSx3 was comparable to the throughput of the monolithic variant. We can also state that adding CPU and RAM to the monolithic variant improves performance only to a specific limit – compare 4 CPU & 6 GB RAM and 4 CPU & 6 GB RAM. Finally, the best throughput is achieved in a configuration scaled vertically and horizontally – MSx2 2 CPU & 2 GB RAM.

Application daily cost vs. performance is shown in Figure 6, where the pricing was calculated according to data presented in Table 2. Here, and in subsequent similar cases, we always use the obtained throughput's median for a cost comparison. On this and subsequent figures, Pareto dominated configurations marked in red relate to cases where the same or better performance can be achieved by a cheaper – Pareto efficient – configuration. Comparing the costs, we can conclude that in the Spring Cloud environment, horizontal scaling of low-performance machines (1 CPU & 1 GB RAM) always leads to Pareto dominated configurations.

[7]https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-core

[8]https://devblogs.microsoft.com/dotnet/try-the-new-system-text-json-apis

[9]https://www.techempower.com/blog/2019/07/09/framework-benchmarks-round-18

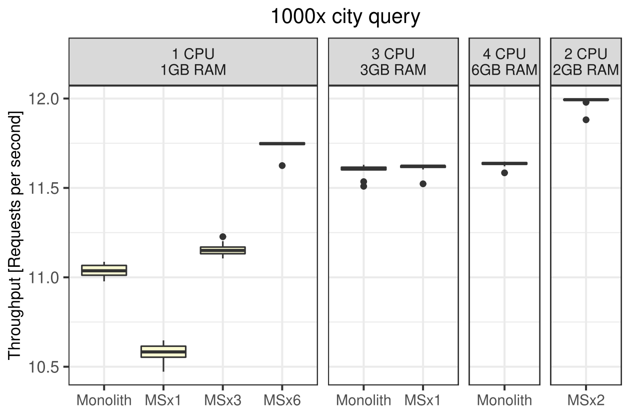


FIGURE 5. Throughput in the Azure spring cloud environment—city application.

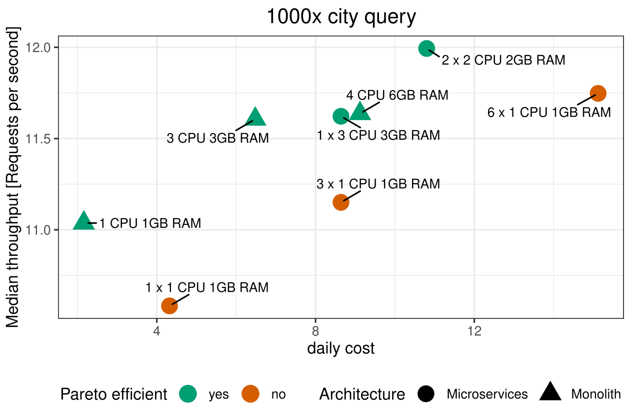


FIGURE 6. Throughput and cost in the Azure spring cloud environment—city application.

The *Route* test gives different results – see Figure 7 – the best performing configuration is that of six instances of 1 CPU & 1 GB RAM machines, good throughput results were also obtained by two instances of 2 CPU & 2 GB RAM and the monolithic version. Comparing these results to the previous ones, we can conclude that in the case of computationally intensive applications, positive results of horizontal scaling are more apparent – throughput scales almost linearly with the number of instances. Also, there is no substantial performance difference between monolithic and microservice applications with the same CPU resources.

Route service daily cost to performance ratio is shown in Figure 8. In this case, three microservice configurations were Pareto dominated, whereas their resource-corresponding monolith versions were Pareto efficient. It is worth noting that MSx6 1 CPU & 1 GB RAM turned out to be the most powerful configuration, but MSx2 2 CPU & 2 GB RAM and monolithic version with 4 CPU & 6 GB RAM delivered only slightly lower performance with substantially lower cost.

C. AZURE APP SERVICE ENVIRONMENT

Figure 9 illustrates throughput results for *City* service under Azure App Service environment. As in the previous

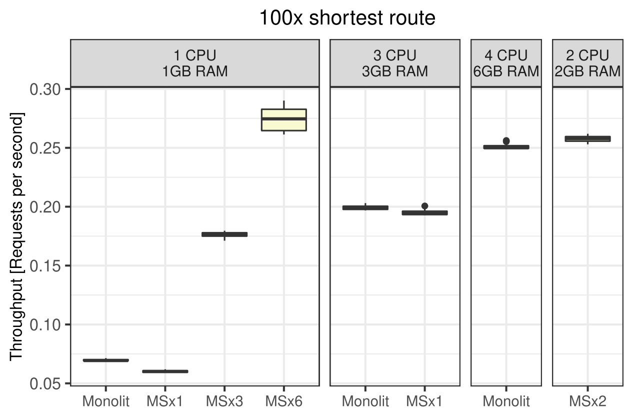


FIGURE 7. Throughput in the Azure spring cloud environment–route service.

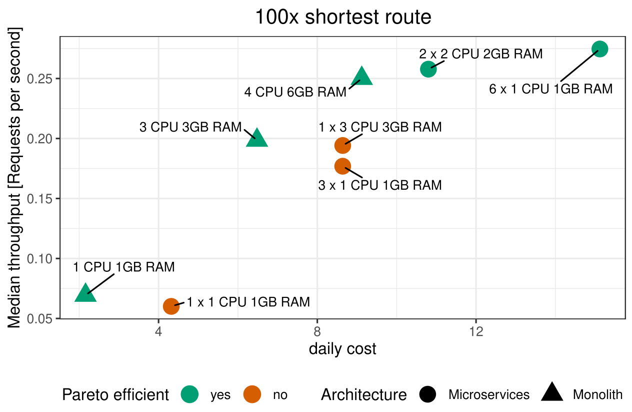


FIGURE 8. Throughput and cost in the Azure spring cloud environment–route service.

section – monolith and MSx1, MSx2, …, MSx20 relate to the number of microservice instances ranging from 1 to 20. The results are grouped by virtual machine configuration (i.e., number of CPUs and RAM size) shown as a label – see table 4. This form of presentation lets us analyze the effects of horizontal scaling.

The same information is presented in Figure 10 to analyze the effects of vertical scaling, as a function of ascending virtual machine computing power and grouped by the number of microservice instances. The full dataset and box plots figures of measured throughput under Azure App Service are available at our GitHub repository.[10]

The top-performing configuration in the *City* test is 6 × S4:7 written in .NET. A very similar throughput was obtained by 3 × S4:7 also under .NET. The best configuration for Java application was 6 × S4:7 – ranked fifth. Worst performing were Java applications running on machines B1:1.75 and S1:1.75. In most configurations, .NET applications performed better than their Java counterparts. This is consistent with the results of local tests – here, also Java applications performed worst in low power configurations, and similar or better than .NET in environments that were as well vertically scaled as P4:14. This result is consistent with the earlier observation that Java applications have notable resource overhead and require more computing power.

A surprising result is that in the highest power configuration – P4:14, both Java and .NET applications have shown performance loss with respect to S4:7 even though P4:14 comes from the Premium v2 Service Plan, which provides faster processors and SSD storage compared to VMs offered in the Standard Service Plan. Our results suggest that CPU types used in P4:14 configuration are less suited to servicing a large number of short requests and heavy network traffic.

To illustrate the effect of scaling, we have prepared two additional plots – Figure 11 shows how throughput's median changes in % relative to previous configuration – when the number of instances increases – from 1 to 3, from 3 to 6, …, and finally from 15 to 20; respectively. Figure 12 shows how throughput's median changes (also %-wise) when a single virtual machine configuration is upgraded – from: B/S1:1.75 to S2:3.5, from S2:3.5 to S4:7, and from S4:7 to P4:14. From Figure 11 we can conclude that horizontal scaling of simple& short request applications is most beneficial when the number of instances is moderately increased, here: from 1 to 3 and from 3 to 6. A further increase of the number of instances is not notably beneficial, and in half of the cases, it resulted in performance decrease (.NET S2:3.5, Java S2:3.5, .NET S4:7, Java S4:7, Java P4:14, Java B/S1:1.75 with ten instances) – this is caused by communication overhead due to load balancing and the need of request passing. The first obvious conclusion from Figure 12 relating to vertical scaling is that the changes are more significant when compared to horizontal scaling – compare the increase of throughput by 24% in the case of Java Monolith and MSx1 architecture up-scaled to S2:3.5 configuration with relation to maximum increase by 9% in case of the same architectures horizontally scaled from 1 to 3 instances. Also, we do not observe a notable performance increase with further configuration up-scaling. However, we should be careful with the interpretation of these results – a significant performance increase, especially for the Java architecture, is caused by the effect of "right-scaling" – the basic configuration B/S1:1.75 does not fully meet requirements of the application, especially under the heavier Java run-time environment. Finally, we should also mention the negative impact of moving to the P4:14 configuration caused by the previously discussed change of the CPU architecture in the Premium v2 Service Plan.

Application daily cost vs. performance is shown in Figure 13. As in previous cases, the pricing was calculated according to data presented in Table 2 and the obtained throughput median was used. Comparing the costs of running the *City* service in different configurations, we can conclude that Pareto efficient configuration variants for .NET were: monolithic B1:1.75, S2:3.5, S4:7 and microservice-based:

[10]https://github.com/przybylek/Monolith-vs-Microservices

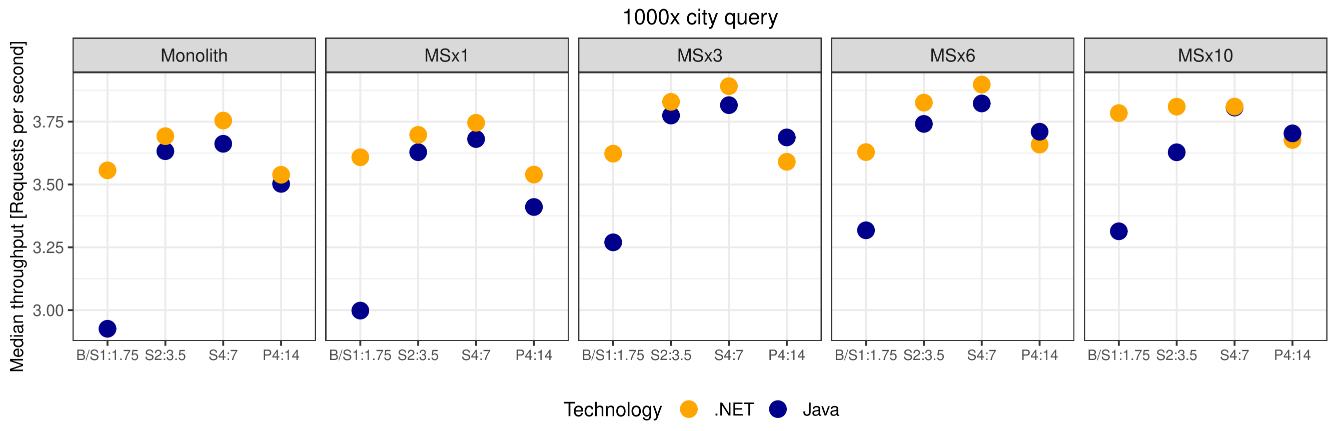


FIGURE 9. Throughput vs. horizontal scaling in the Azure app service environment—city service.

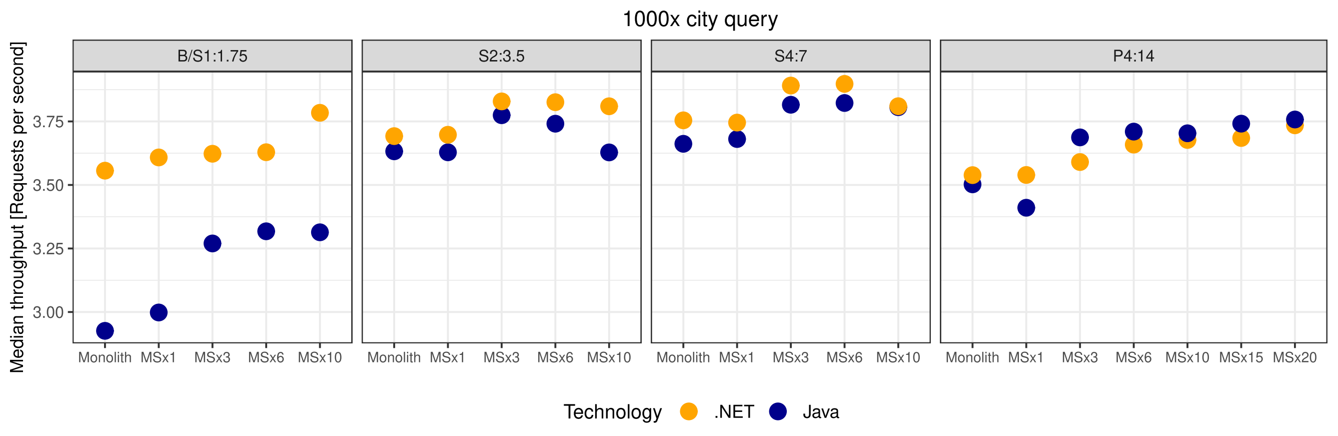


FIGURE 10. Throughput vs. vertical scaling in the Azure app service environment—city service.

B1:1.75, 1 x S2:3.5, 3 x S2:3.5, 3 x S4:7 and 6 x S4:7. In the case of Java, an almost identical set of configurations with an addition of 1 x S4:7 was Pareto efficient.

Now we will focus on tests of the *Route* service – refer to Figure 14 which presents throughput median as a function of the number of instances grouped by instance configuration (horizontal scaling) and to Figure 15 which presents the same data but grouped by the number of instances (vertical scaling). Java is a better performer here – attaining better results in all but one case. The effects of both horizontal and vertical scaling are clearly visible – the more powerful the configuration and the larger the number of microservice instances, the better throughput. The best performance was registered for 10 x P4:14 Java configuration. This result is consistent with those obtained in the local configuration of the *Route* where Java was also a better choice for this computationally intensive service. We can also observe an interesting trend: for low-power configuration, throughput is similar for .NET and Java variants, but as the power of the virtual machine is increased, Java variant shows large performance gain over .NET – compare results for S4:7 and P4:14 on Figure 14. Java also exhibits better vertical scaling – see results for P4:14 on Figure 15.

Similar to the *City* service in the case of *Route* service, we have also visualized the effect of scaling on two additional plots – Figure 16 shows how throughput's median changes in % relative to the previous value when the number of instances increases; Figure 17 shows how throughput's median changes (also %-wise) when single virtual machine configuration is upgraded. Values on the x-axis are identical, as with the *City* service. The largest throughput increase was measured when the number of instances was increased from 1 to 3 and from 3 to 6 – this is consistent with the previous results. The scale of the increase is much more prominent with the *Route* service. In five cases (both .NET and Java), performance increased by over 100% relative to the previous configuration. The most significant observed scaling gain was 148% (compared to maximum 9% gain in the *City* service). Again, similarly to the previous service, we also observe no or minimal gain when the number of

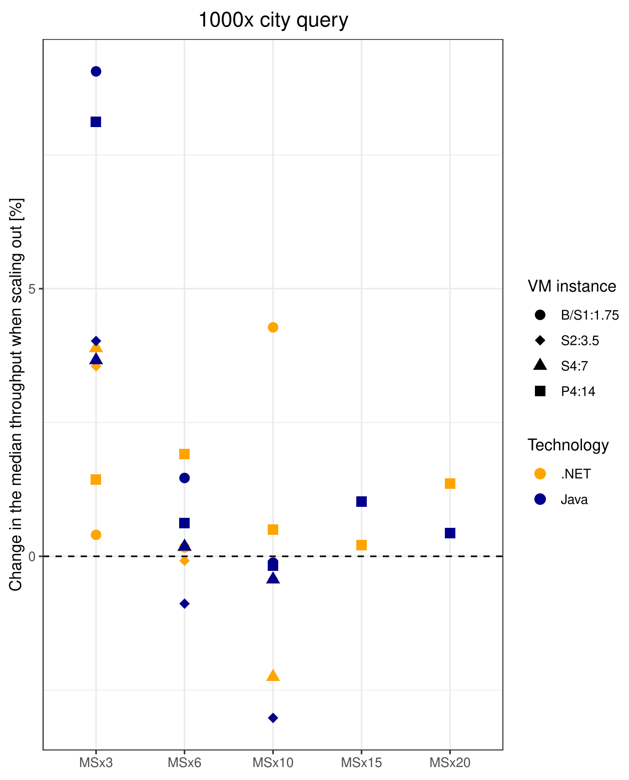


FIGURE 11. Throughput's median change as an effect of horizontal scaling in the Azure app service environment—city service.

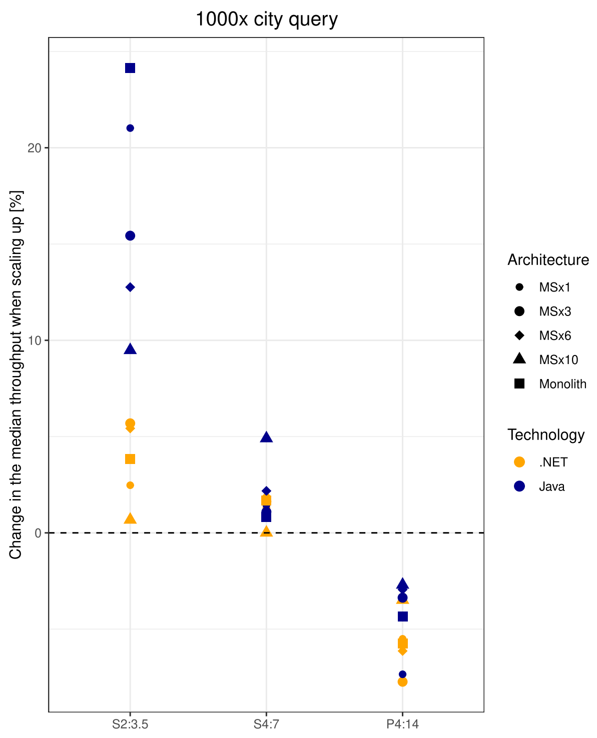


FIGURE 12. Throughput's median change as an effect of vertical scaling in the Azure app service environment—city service.

instances increases to 15, and throughput reduction with the increase to 20 instances. The effects of vertical scaling are also notably larger in the case of the *Route* service – we have observed 200% performance gain with the change from 1 × B1.1:75 to 1 × S2:3.5 configuration in the case of Java platform, and significant gains no smaller than 48% in case of other horizontal configurations. Note that in the case of *City* test, vertical up-scaling resulted in an actual performance decrease in eleven cases. Here, only in one case, a minimal performance reduction was measured. In general, vertical up-scaling almost always proved beneficial in this test, and its results were meaningful (at least 30% improvement) in all but four cases.

Finally, in this section, we will analyze the costs of various configurations of the *Route* service – see Figure 18. Comparing the costs for the *Route* service we can conclude that Pareto efficient configuration variants were: monolithic B1:1.75, S2:3.5, S4:7 and P4:14 as well as microservice-based: 1 × S2:3.5 i 3 × S2:3.5, 3 × P4:14, 6 × P4:14, 10 × P4:14. In the case of Java: an almost identical set of configurations with an addition of 1 × S4:7 - both for .NET and Java. Additionally, 15 × P4:14 may be selected for microservice .NET and 1 × P4:14 for microservice Java. Almost all high-power configurations (except for 20 × P4:14) are Pareto efficient – this is in contrast to the *City* test where high-power configurations were Pareto dominated.

VI. DISCUSSION

This section discusses the results of our study – we address research questions stated in the introduction regarding performance differences between monolithic and microservice applications, scaling approaches, and implementation technologies. Here, we generalize the naming of our benchmark services: *City* is referred to as a *simple* service and *Route* as a *complex* service.

A. (RQ1) WHAT IS THE PERFORMANCE DIFFERENCE BETWEEN A MONOLITHIC APPLICATION VERSUS A MICROSERVICE APPLICATION?

On a single machine, the monolith performs better than the microservices (see Figure 4) because of the additional overhead of request passing between microservice components. Note that this difference is not noticeable in our cloud benchmarks because the application gateway was deployed on a separate virtual machine, which relieved the primary machine that hosted microservices.

Thereby, as for both cloud experiments, rather than comparing the performance of both architectures running on the same VM type, we must compare the performance of the architectures hosted on configurations that have similar infrastructure costs.

When such comparisons are made, the monolith outperforms the microservices (see Figures 6, 8, 13, and 18), even though the latter was still privileged, because when testing the performance, only one service was running (for details see Section IV-D).

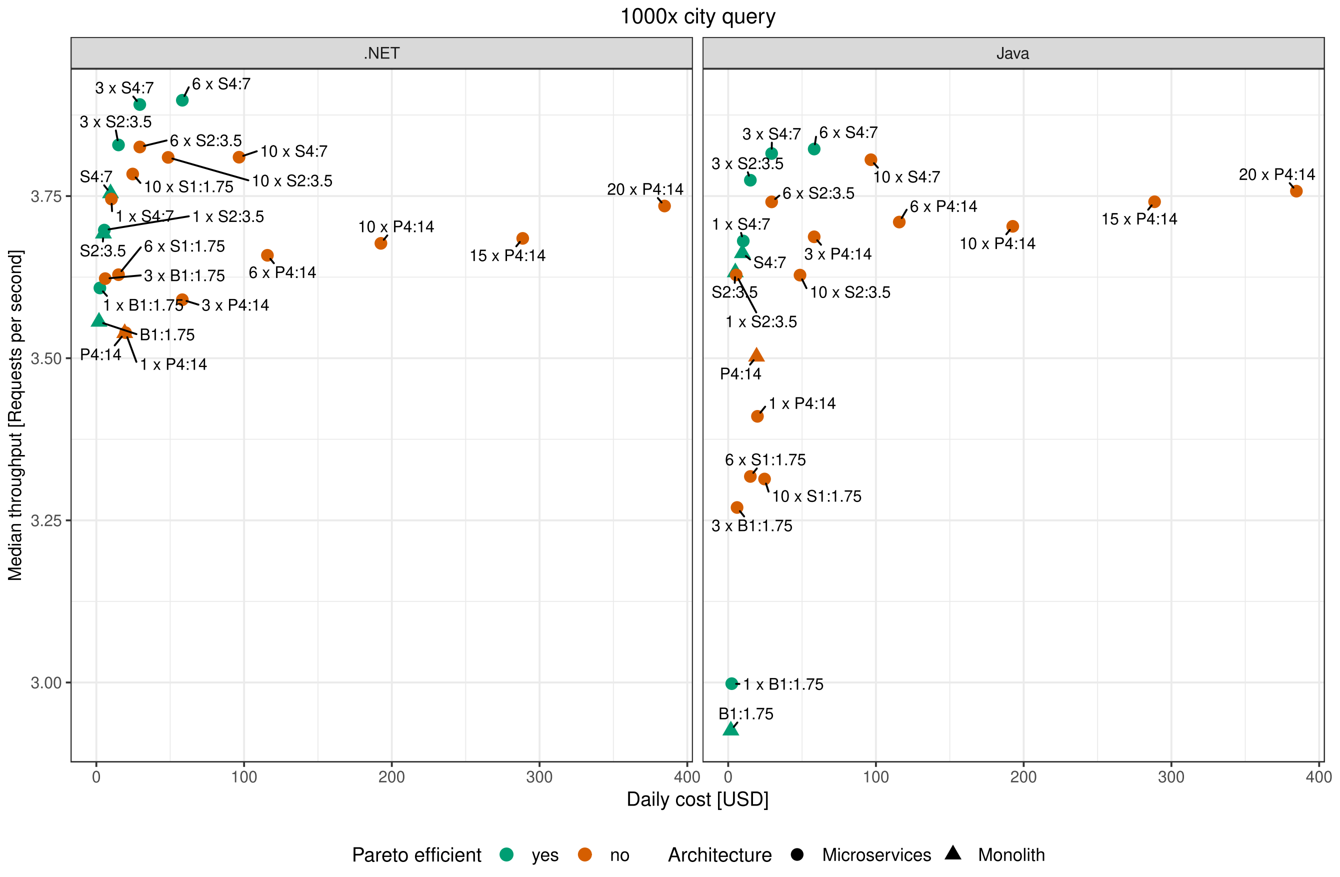


FIGURE 13. Throughput and cost in the Azure app service environment—city service.

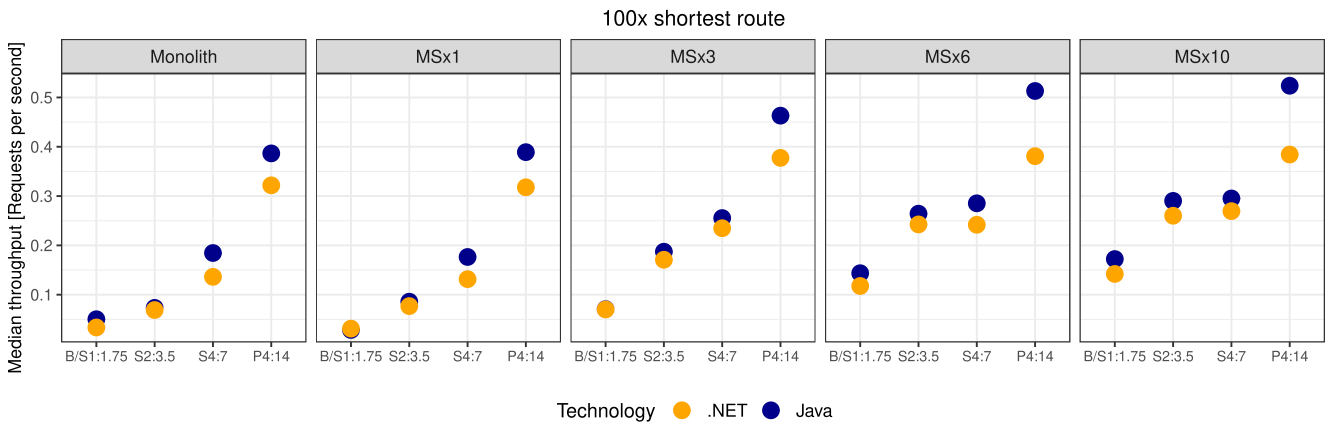


FIGURE 14. Throughput vs. horizontal scaling in the Azure app service environment—route service.

B. (RQ2) WHICH OF THE TWO ARCHITECTURES AND SCALING APPROACHES SHOULD BE CHOSEN TO BEST BENEFIT AN APPLICATION FROM SCALING?

The monolith scaled vertically was Pareto efficient in servicing both simple and complex requests on all hardware configurations except P4:14 regardless of the implementation technology. Also note, that all experimental runs executed on P4:14 turned out to be Pareto dominated.

Likewise, scaling up the microservice-based application performed well and was more cost-efficient than horizontal scaling. Nevertheless, vertical scaling was limited by the most powerful VM available; therefore the best performance

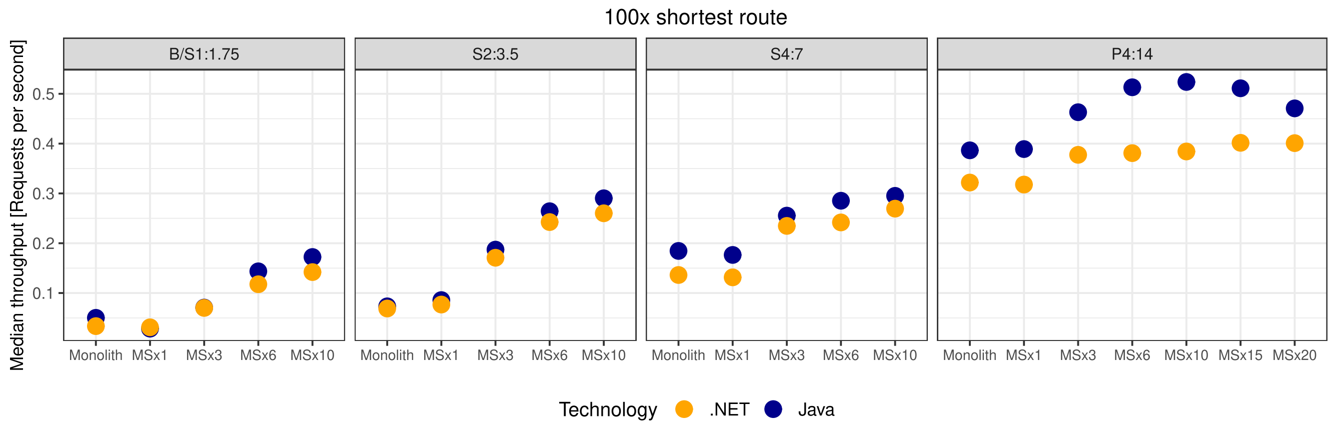


FIGURE 15. Throughput vs. vertical scaling in the Azure app service environment—route service.

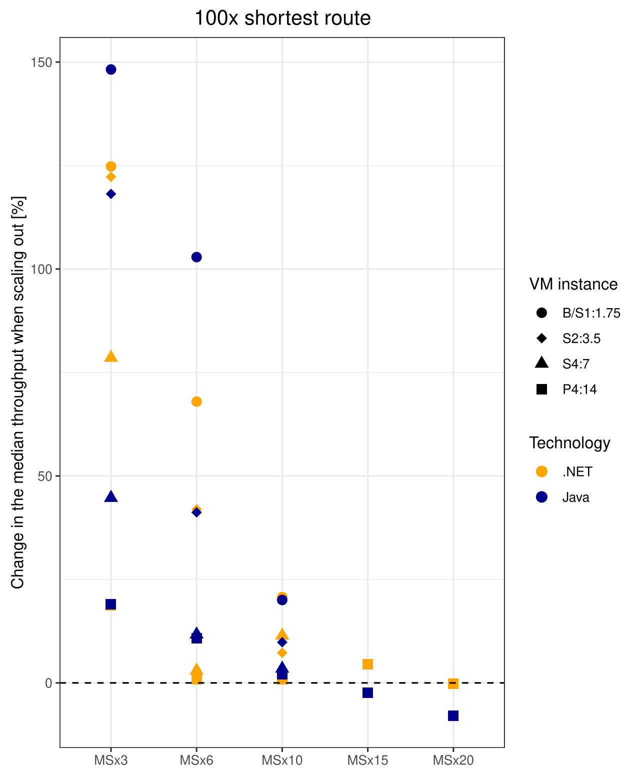


FIGURE 16. Throughput's median change as an effect of horizontal scaling in the Azure app service environment—route service.

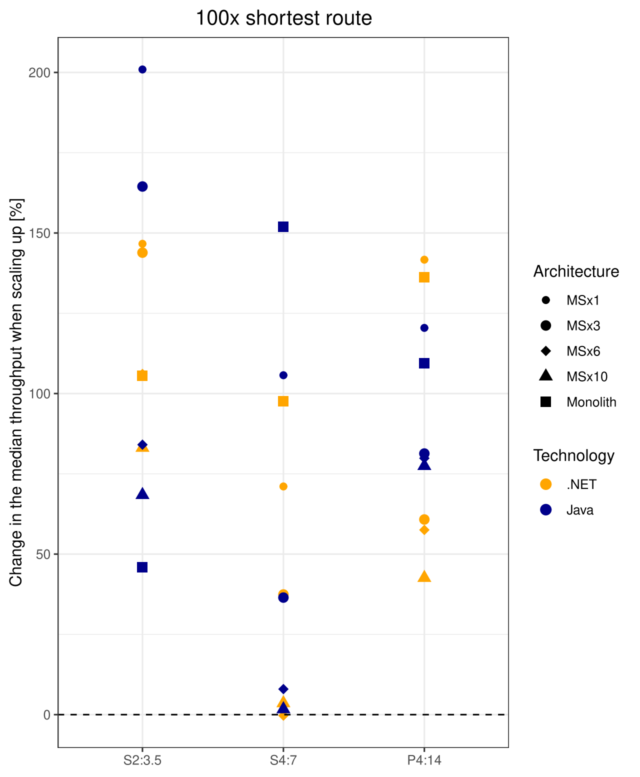


FIGURE 17. Throughput's median change as an effect of vertical scaling in the Azure app service environment—route service.

was achieved by the microservice architecture that was both vertically and horizontally scaled.

Our experiments have also shown that the best cost-to-performance ratio was achieved with three instances of the most powerful platform. Moreover, both horizontal and vertical scaling exhibit significant throughput gain in the case of complex services than in simple ones.

Furthermore, with microservice architecture, horizontal scaling effects differed significantly in the case of simple and complex services. Concerning the number of instances, in the case of the simple service, top performance was achieved with a smaller number of virtual machines than in the complex service. There is a visible cap on horizontal scaling for both service types, where the further increase of the number of instances does not improve and may even degrade performance. This effect of over-scaling manifests itself when CPU overhead resulting from load distribution exceeds the benefits of increasing the total processing power.

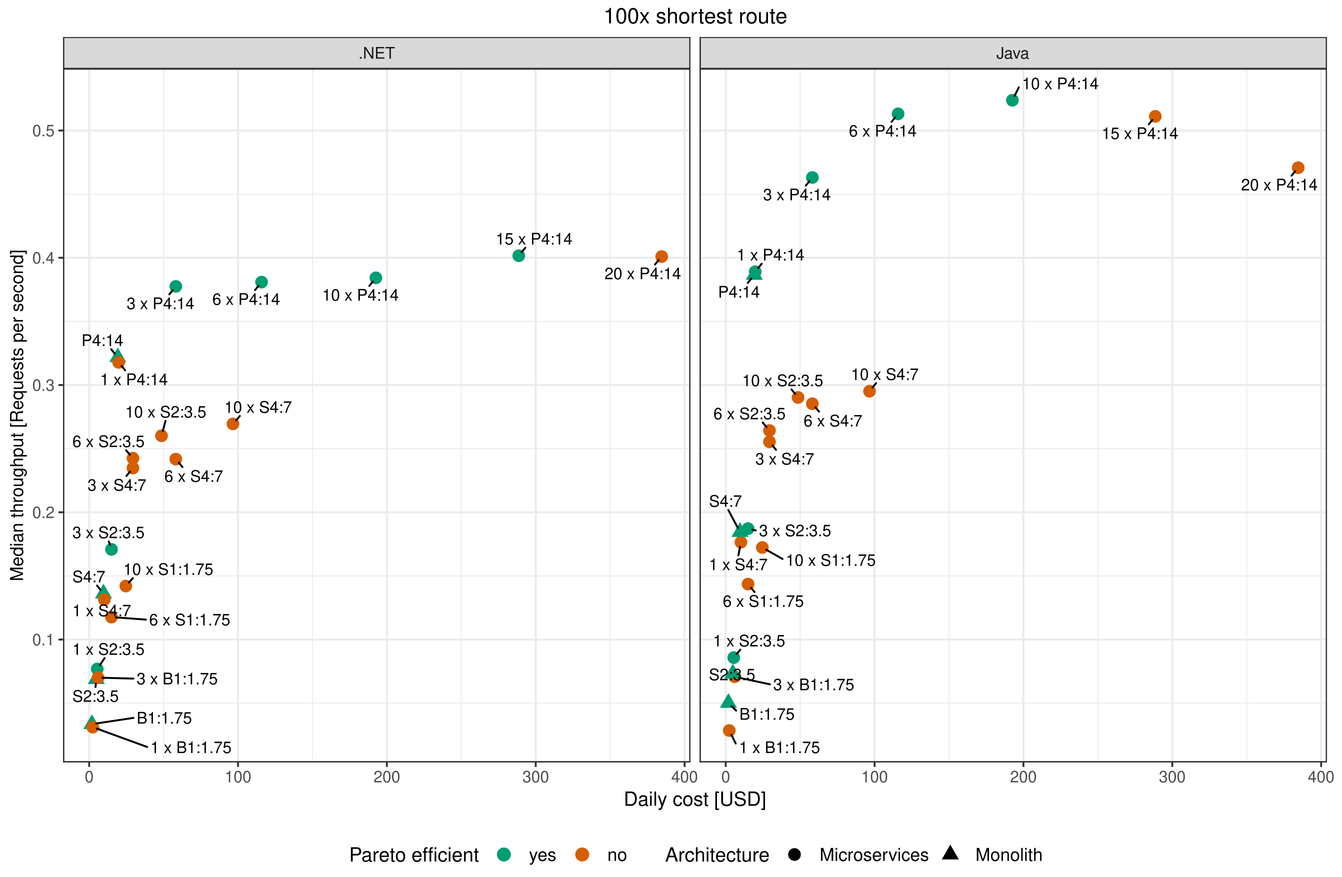


FIGURE 18. **Throughput and cost in the Azure app service environment—route service.**

Finally, the implementation technology (Java vs. C# .NET) did not have a noticeable impact on the scalability performance.

C. (RQ3) IN WHAT CIRCUMSTANCES THE IMPLEMENTATION TECHNOLOGIES (Java VS. C# .NET) HAVE ANY PERFORMANCE ADVANTAGES OR DISADVANTAGES?

In the selection of the technology platform, characteristics of network load vs. CPU load should be taken into account – for intensive network services with a low CPU load .NET is a better choice than Java; on the other hand, Java consistently proved to utilize CPU better in computation-intensive services..NET also better utilizes cheap hardware with lower computational capacity as long as the computation is not intensive (see the first plot on Figure 10). On the other hand, The Java platform makes better use of powerful machines in the case of computation-intensive services (see the last plot on Figure 15).

D. OTHER FINDINGS

Java implementations deployed in the Azure Spring Cloud performed better (especially in the case of the simple service) with respect to their counterparts deployed on similar machines under the Azure App Service environment. Also, the cost of running the application under a given hardware configuration in Azure Spring Cloud is lower than in Azure App Service.

VII. THREATS TO VALIDITY

In this section, we report on the threats to the validity of our study. We distinguish between three types of threat to validity [2], [69]:

- construct validity – the extent to which the independent and dependent variables accurately measure the concepts they purport to measure;
- internal validity – the extent to which the observed effects are caused only by the experimental treatment conditions;
- external validity – the extent to which the findings of the study can be generalized outside the experimental setting.

A. CONSTRUCT VALIDITY

We narrowed the performance assessment to measuring the throughput. Although throughput is the predominant performance metric, other metrics, such as response time (latency) and CPU utilization, have also been used in prior studies

[4], [14], [17], [18], [42], [70]. Our motivation was that throughput had also been commonly used as the scalability measure [18], while in this study, we were also interested in comparing the scalability of both architectural styles. Nevertheless, we are aware that throughput does not capture all aspects of the performance, and future work should investigate other performance metrics.

B. INTERNAL VALIDITY

Readers need to keep in mind that experiments in a public cloud cannot be fully controlled [67]. Indeed, in a pilot study, we observed that the performance results varied considerably between different runs of the experiment. For instance, some test runs showed worse measured throughput for a higher-end virtual machine than for a low-end machine. The reason for this phenomenon is unknown. We may suspect both variability between VM instances or random network disturbances. On the other hand, variability within a single test series was very stable. To eliminate the influence of such random bias, we have employed test repetitions and median of medians selection method described in Section IV-D.

Another factor that might have influenced the experiment's result is the random selection of route points in the *Route* benchmark. We have verified in the local environment that repeating a series of identical queries leads to better throughput results. However, such a coincidence is very improbable, given that 2000 queries are sent a single test series.

C. EXTERNAL VALIDITY

The most important threat to the external validity of our study concerns the representativeness of the experimental object. As it consists of only two independent services, our application is far away from being a realistic benchmark. In a real application, dozens (if not hundreds) of microservices need to keep communicating by protocols such as HTTP. This communication may result in overhead due to inter-service communication when compared with method calls performed locally in the monolith [13], [71], [72]. However, our intention was to solely isolate the scalability and eliminate factors that could affect the dependent variables. Accordingly, we traded some external validity for more internal validity.

Our analysis of the impact of vertical and horizontal scaling on application performance is based to a large extent on test results conducted under the Azure App Service cloud. Because of this, our conclusions concerning scaling are applicable to applications deployed in a similar environment and on virtual machines of comparable characteristics. However, our cost analysis is strictly tied to the pricing models of the Azure platform. Accordingly, before generalizing our results to other public clouds, the reader should carefully compare their pricing models. On the other hand, our conclusions concerning the impact of chosen technology (Java vs. C# .NET) on application performances can be generalized to other cloud environments and application types.

VIII. CONCLUSION AND FUTURE WORK

Although microservices are gaining more and more momentum in the IT industry, empirical studies evaluating their performance and scalability are still rare, while hands-on experiences come only from global IT giants with millions of concurrent users. Accordingly, shifting towards microservices by small companies is often based on intuition rather than solid information. To support software architects in rational decision-making about migrating systems to microservices, or developing an entire application from scratch under this architectural style, we carried out a series of controlled experiments in three different deployment environments (local, Azure Spring Cloud, and Azure App Service). We extensively investigated the effects of architectural style (monolithic vs. microservice) and implementation technology (Java vs. C# .NET) on the application performance and scalability. Our key lessons learned are as follows:

- on a single machine, a monolith performs better than its microservice-based counterpart;
- Java platform makes better use of powerful machines in case of computation-intensive services when compared to .NET; the platform effect is reversed if non-computationally intensive services are run on hardware with low computational capacity;
- vertical scaling is more cost-effective than horizontal scaling in the Azure cloud;
- scaling out beyond a certain number of instances degrades the application performance;
- the implementation technology does not have a noticeable impact on the scalability performance.

In conclusion, a microservice architecture is not the best suited for every context. A monolithic architecture seems to be a better choice for simple, small-sized systems that do not have to support a large number of concurrent users. We hope that our findings will help companies avoid jumping into microservices simply because they are trendy, especially if better results can be obtained by scaling up their monoliths.

We also believe that both researchers and practitioners can benefit from our reference benchmarking application,[11] and use it as a starting point for further experimentation.

In future work, we intend to make our benchmarking application more complex so that it will closely resemble real-life systems used by companies. The first extension will be to develop more microservices, of which some will communicate with each other. The reality of the application will also benefit from having a database. Another interesting future direction is to provide new implementation alternatives for other programming languages recommended for developing microservices, e.g., Golang, Python, and Node.js. Moreover, we plan to deploy and benchmark our application on other cloud platforms, including Amazon Web Services, Google Cloud Platform, and Alibaba Cloud. It would be also interesting to adopt more performance assessment metrics, including response time (latency) and CPU utilization. Finally,

[11]https://github.com/annaojdowska/monolith-vs-microservices

to address performance fluctuation issues due to a shared environment, in a follow-up study, we consider running VMs in a private environment (e.g. Azure App Service Environment) dedicated exclusively to a single customer.

ACKNOWLEDGMENT

The authors would like to thank Szymon Okrój, who developed Java implementations of their benchmark application as a part of his master's thesis project [61].

REFERENCES

[1] A. Przybyłek, "Where the truth lies: AOP and its impact on software modularity," in *Fundamental Approaches to Software Engineering*, D. Giannakopoulou and F. Orejas, Eds. Berlin, Germany: Springer, 2011, pp. 447–461.

[2] A. Przybyłek, "An empirical study on the impact of AspectJ on software evolvability," *Empirical Softw. Eng.*, vol. 23, no. 4, pp. 2018–2050, 2018.

[3] M. Fowler. *Microservice Premium*. Accessed: May 30, 2020. [Online]. Available: https://Martinfowler.com/bliki/MicroservicePremium.html

[4] N. Bjørndal, A. Bucchiarone, M. Mazzara, N. Dragoni, S. Dustdar, F. B. Kessler, and T. Wien, "Migration from monolith to microservices: Benchmarking a case study," Tech. Rep., 2020. [Online]. Available: https://www.researchgate.net/profile/Manuel-Mazzara/publication/339749917_Migration_from_Monolith_to_Microservices_Benchmarking_a_Case_Study/links/5e6359034585153fb3c8515f/Migration-from-Monolith-to-Microservices-Benchmarking-a-Case-Study.pdf

[5] B. Terzić, V. Dimitrieski, S. Kordić, G. Milosavljević, and I. Luković, "Development and evaluation of microbuilder: A model-driven tool for the specification of rest microservice software architectures," *Enterprise Inf. Syst.*, vol. 12, nos. 8–9, pp. 1034–1057, 2018.

[6] J. Lewis and M. Fowler. (Mar. 2014). *Microservices: A Definition of This New Architectural Term*. [Online]. Available: https://www.Martinfowler.com/articles/microservices.html

[7] C. Posta, *Microservices for Java Developers: A Hands-on Introduction to Frameworks Containers*. Newton, MA, USA: O'Reilly Media, 2016.

[8] R. Rajesh, *Spring Microservices*. London, U.K.: Packt, 2016.

[9] A. Cockroft. (Aug. 2004). *Migrating to Microservices*. [Online]. Available: https://youtu.be/1wiMLkXz26M

[10] P. Z. Y. N. Doro ski, A. Brzeski, J. Cychnerski, and T. Dziubich, "Towards healthcare cloud computing," in *Proc. 36th Int. Conf. Inf. Syst. Archit. Technol.*, J. Świątek, L. Borzemski, A. Grzech, and Z. Wilimowska, Eds. Cham, Switzerland: Springer, 2016, pp. 87–97.

[11] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 146, pp. 215–232, Dec. 2018.

[12] H. Vural, M. Koyuncu, and S. Misra, "A case study on measuring the size of microservices," in *Computer Science Application*, O. Gervasi, B. Murgante, S. Misra, E. Stankova, C. M. Torre, A. M. A. Rocha, D. Taniar, B. O. Apduhan, E. Tarantino, and Y. Ryu, Eds. Cham, Switzerland: Springer, 2018, pp. 454–463.

[13] L. Carvalho, A. Garcia, W. K. G. Assunç ao, R. de Mello, and M. J. de Lima, "Analysis of the criteria adopted in industry to extract microservices," in *Proc. Joint 7th Int. Workshop Conducting Empirical Stud. Ind.*, 2019, pp. 22–29.

[14] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Dec. 2019, pp. 80–90.

[15] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *J. Syst. Softw.*, vol. 150, pp. 77–97, Apr. 2019.

[16] J. Jaworski, W. Karwowski, and M. Rusek, "Microservice-based cloud application ported to unikernels: Performance comparison of different technologies," in *Proc. 40th Anniversary Int. Conf. Inf. Syst. Archit. Technol.*, L. Borzemski, J. Świątek, and Z. Wilimowska, Eds. Cham, Switzerland: Springer, 2019, pp. 255–264.

[17] M. Jayasinghe, J. Chathurangani, G. Kuruppu, P. Tennage, and S. Perera, "An analysis of throughput and latency behaviours under microservice decomposition," in *Web Engineering*, M. Bielikova, T. Mikkonen, and C. Pautasso, Eds. Cham, Switzerland: Springer, 2020, pp. 53–69.

[18] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to microservices: An assessment framework," *Inf. Softw. Technol.*, vol. 137, Dec. 2021, Art. no. 106600.

[19] M. Viggiato, R. Terra, H. Rocha, M. Tulio Valente, and E. Figueiredo, "Microservices in practice: A survey study," 2018, *arXiv:1808.04836*.

[20] M. Jagiełło, M. Rusek, and W. Karwowski, "Performance and resilience to failures of an cloud-based application: Monolithic and microservices-based architectures compared," in *Computer Information Systems and Industrial Management*, K. Saeed, R. Chaki, and V. Janev, Eds. Cham, Switzerland: Springer, 2019, pp. 445–456.

[21] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: Intentions, strategies, and challenges," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Dec. 2019, pp. 481–490.

[22] A. Poniszewska-Marańda and E. Czechowska, "Kubernetes cluster for automating software production environment," *Sensors*, vol. 21, no. 5, p. 1901, 2021.

[23] C. M. Aderaldo, N. C. Mendonca, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *Proc. IEEE/ACM 1st Int. Workshop Establishing Community-Wide Infrastruct. Archit.-Based Softw. Eng. (ECASE)*, May 2017, pp. 8–13.

[24] A. Poth, H. Urban, and A. Riel, *Make Product Service Requirements Shippable—From Cloud Service Vision to a Continuous Value Stream Which Satisfies Current Future User Needs*. Cham, Switzerland: Springer, 2021.

[25] Y. Wang, H. Kadiyala, and J. Rubin, "Promises and challenges of microservices: An exploratory study," *Empirical Softw. Eng.*, vol. 26, no. 4, pp. 1–44, Jul. 2021.

[26] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 22–32, Sep. 2017.

[27] J. Ghofrani and D. Lübke, "Challenges of microservices architecture: A survey on the state of the practice," in *Proc. ZEUS*, 2018, pp. 1–8.

[28] S. Butt, S. Abbas, and M. Ahsan, "Software development life cycle & software quality measuring types," *Asian J. Math. Comput. Res.*, vol. 11, no. 2, pp. 112–122, 2016.

[29] A. Jarzębowicz and P. Marciniak, "A survey on identifying and addressing business analysis problems," *Found. Comput. Decis. Sci.*, vol. 42, pp. 315–337, Dec. 2017.

[30] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, and Z. Shan, "A dataflow-driven approach to identifying microservices from monolithic applications," *J. Syst. Softw.*, vol. 157, Nov. 2019, Art. no. 110380.

[31] H. Stranner, S. Strobl, M. Bernhart, and T. Grechenig, "Microservice decompositon: A case study of a large industrial software migration in the automotive industry," in *Proc. 15th Int. Conf. Eval. Novel Approaches Softw. Eng.*, 2020, pp. 498–505.

[32] M. Bruce and P. A. Pereira, *Microservices in Action*. New York, NY, USA: Simon and Schuster, 2018.

[33] A. Banijamali, P. Kuvaja, M. Oivo, and P. Jamshidi, "Kuksa: Self-adaptive microservices in automotive systems," in *Product-Focused Software Process Improvemen*, M. Morisio, M. Torchiano, and A. Jedlitschka, Eds. Cham, Switzerland: Springer, 2020, pp. 367–384.

[34] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *Current Trends in Web Engineering* (Lecture Notes in Computer Science), vol. 10544, I. Garrigós and M. Wimmer, Eds. Cham, Switzerland: Springer, 2018, doi: 10.1007/978-3-319-74433-9_3.

[35] W. Karwowski, M. Rusek, G. Dwornicki, and A. Orłowski, "Swarm based system for management of containerized microservices in a cloud consisting of heterogeneous servers," in *Proc. 38th Int. Conf. Inf. Syst. Archit. Technol.*, L. Borzemski, J. Świątek, and Z. Wilimowska, Eds. Cham, Switzerland: Springer, 2018, pp. 262–271.

[36] B. Terzic and V. Dimitrieski, "A model-driven approach to microservice software architecture establishment," in *Proc. Ann. Comput. Sci. Inf. Syst.*, Sep. 2018, p. 73.

[37] M. Štefanko, O. Chaloupka, and B. Rossi, "The saga pattern in a reactive microservices environment," in *Proc. 14th Int. Conf. Softw. Technol.*, 2019, pp. 483–490.

[38] C. Rajasekharaiah, *Case Study: Energence*. Berkeley, CA, USA: Apress, 2021, pp. 1–12.

[39] A. Poniszewska-Marańda, P. Vesely, O. Urikova, and I. Ivanochko, "Building microservices architecture for smart banking," in *Advances in Intelligent Networking and Collaborative Systems*, L. Barolli, H. Nishino, and H. Miwa, Eds. Cham, Switzerland: Springer, 2020, pp. 534–543.

[40] J. Ghofrani and A. Bozorgmehr, "Migration to microservices: Barriers and solutions," in *Applied Informatics*, H. Florez, M. Leon, J. M. Diaz-Nafria, and S. Belli, Eds. Cham, Switzerland: Springer, 2019, pp. 269–281.

[41] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *Proc. IEEE 18th Int. Symp. Comput. Intell. Informat. (CINTI)*, Nov. 2018, pp. 149–154.

[42] A. de Camargo, I. Salvadori, R. D. S. Mello, and F. Siqueira, "An architecture to automate performance tests on microservices," in *Proc. 18th Int. Conf. Inf. Integr. Web-Based Appl. Services*, Nov. 2016, pp. 422–429.

[43] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does migrating a monolithic system to microservices decrease the technical debt?" *J. Syst. Softw.*, vol. 169, Nov. 2020, Art. no. 110710.

[44] V. Lenarduzzi and O. Sievi-Korte, "On the negative impact of team independence in microservices software development," in *Proc. 19th Int. Conf. Agile Softw. Develop., Companion*, New York, NY, USA, May 2018, pp. 1–4.

[45] F. Ramin, C. Matthies, and R. Teusner, "More than code: Contributions in scrum software engineering teams," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. Workshops*, New York, NY, USA, Jun. 2020, pp. 137–140.

[46] B. Marcinkowski and B. Gawin, "A study on the adaptive approach to technology-driven enhancement of multi-scenario business processes," *Inf. Technol. People*, vol. 32, no. 1, pp. 118–146, Feb. 2019.

[47] M. Kalenda, P. Hyna, and B. Rossi, "Scaling agile in large organizations: Practices, challenges, and success factors," *J. Softw., Evol. Process*, vol. 30, no. 10, p. e1954, Oct. 2018.

[48] P. Diebold, A. Schmitt, and S. Theobald, "Scaling agile: How to select the most appropriate framework," in *Proc. 19th Int. Conf. Agile Softw. Develop., Companion*, New York, NY, USA, May 2018, pp. 1–4.

[49] S. Theobald, A. Schmitt, and P. Diebold, "Comparing scaling agile frameworks based on underlying practices," in *Agile Processes in Software Engineering and Extreme Programming*, R. Hoda, Ed. Cham, Switzerland: Springer, 2019, pp. 88–96.

[50] A. Poth, M. Kottke, and A. Riel, "Scaling agile—A large enterprise view on delivering and ensuring sustainable transitions," in *Agile Processes in Software Engineering and Extreme Programming*, A. Przybyłek and M. E. Morales-Trujillo, Eds. Cham, Switzerland: Springer, 2020, pp. 1–18.

[51] A. Khalid, S. A. Butt, T. Jamal, and S. Gochhait, "Agile scrum issues at large-scale distributed projects: Scrum project development at large," *Int. J. Softw. Innov. (IJSI)*, vol. 8, no. 2, pp. 85–94, 2020.

[52] M. Kowalczyk, B. Marcinkowski, and A. Przybyłek, "Scaled agile framework: Dealing with software process-related challenges of a financial group with the action research approach," *J. Softw., Evol. Process*, 2022.

[53] M. Kösling and A. Poth, "Agile development offers the chance to establish automated quality procedures," in *Systems, Software and Services Process Improvement*, J. Stolfa, S. Stolfa, R. V. O'Connor, and R. Messnarz, Eds. Cham, Switzerland: Springer, 2017, pp. 495–503.

[54] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2016, pp. 57–66.

[55] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *Proc. 2nd Int. Workshop Softw. Perform.*, New York, NY, USA, 2000, pp. 195–203.

[56] B. Wilder, *Cloud Architecture Patterns: Using Microsoft Azure*. Newton, MA, USA: O'Reilly Media, 2012.

[57] L. Lu, X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, and E. Smirni, "Application-driven dynamic vertical scaling of virtual machines in resource pools," in *Proc. IEEE Netw. Oper. Manage. Symp. (NOMS)*, May 2014, pp. 1–9.

[58] S. Spinner, N. Herbst, S. Kounev, X. Zhu, L. Lu, M. Uysal, and R. Griffith, "Proactive memory scaling of virtualized applications," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 277–284.

[59] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10.

[60] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards recovering the software architecture of microservice-based systems," in *Proc. IEEE Int. Conf. Softw. Archit. Workshops (ICSAW)*, Apr. 2017, pp. 46–53.

[61] S. Okrój, "A comparative analysis of the performance of monolithic and microservice architecture," M.S. thesis, Gdansk Univ. Technol., Gdańsk, Poland, 2018.

[62] M. Villamizar, O. Garces, and L. Ochoa, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *Proc. 16th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGrid)*, 2016, pp. 179–182.

[63] J. Shaughnessy, E. Zechmeister, and J. Zechmeister, *Research Methods Psychology*. Surrey, BC, Canada: Kwantlen Polytechnic Univ., 2019.

[64] B. Vermeer. (2020). *JVM Ecosystem Report 2020*. Accessed: Oct. 22, 2021. [Online]. Available: https://snyk.io/wp-content/uploads/jvm_2020.pdf

[65] B. Vermeer. (2021) *JVM Ecosystem Report 2021*. Accessed: Oct. 22, 2021. [Online]. Available: https://res.cloudinary.com/snyk/image/upload/v1623860216/reports/jvm-ecosystem-report-2021.pdf

[66] A. Derezinska and K. Kwaánik, "Performance-based refactoring of web application: A case of public transport," in *Proc. 15th Int. Conf. Eval. Novel Approaches to Softw. Eng.*, 2020, pp. 611–618.

[67] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. How bad is it really?" *Empirical Softw. Eng.*, vol. 24, no. 4, pp. 2469–2508, Aug. 2019.

[68] M. R. López and J. Spillner, "Towards quantifiable boundaries for elastic horizontal scaling of microservices," in *Proc. 10th Int. Conf. Utility Cloud Comput.*, New York, NY, USA, 2017, pp. 35–40.

[69] Y. Y. Ng and A. Przybyłek, "Instructor presence in video lectures: Preliminary findings from an online experiment," *IEEE Access*, vol. 9, pp. 36485–36499, 2021.

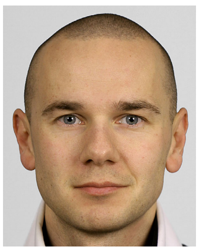
[70] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, "Performance comparison between container-based and VM-based services," in *Proc. 20th Conf. Innov. Clouds, Internet Netw. (ICIN)*, 2017, pp. 185–190.

[71] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *Int. J. Open Inf. Technol.*, vol. 2, pp. 24–27, Oct. 2014.

[72] H. Knoche, "Sustaining runtime performance while incrementally modernizing transactional monolithic software towards microservices," in *Proc. 7th ACM/SPEC Int. Conf. Perform. Eng.*, New York, NY, USA, Mar. 2016, pp. 121–124.

GRZEGORZ BLINOWSKI (Member, IEEE) was born in Warsaw, Poland. He received the M.Sc. and Ph.D. degrees in computer science from the Faculty of Electronics and Information Technology, Institute of Computer Science, Warsaw University of Technology, Poland, in 1993 and 2001, respectively. He has held a Certified Information Systems Security Professional (CISSP) Certificate, since 2014. Since 2001, he has been an Assistant Professor at the Parallel and Distributed Computing Research Group, Institute of Computer Science. He is the author of two books. His research and scientific interests include: distributed memory systems, software engineering, internet technology, and networks and systems security, especially in context of WSN and the IoT and recently VLC systems. He has received the Warsaw University of Technology Rector's Award for Academic Achievements twice.

ANNA OJDOWSKA received the M.Sc. degree in computer science from the Gdańsk University of Technology, in 2020. She has been working as a Software Engineer with IHS Markit, since June 2018. She specializes in .NET technology. Her main professional interests include software architecture, functional programming and its practical applications, and impact on software engineering.



ADAM PRZYBYŁEK received the master's degree in management information systems and the Ph.D. degree in software engineering, in 2002 and 2011, respectively. Between 2002 and 2011, he was a Network Consultant and an Instructor at the Cisco Networking Academy. He has been working as an Assistant Professor with the Gdańsk University of Technology, Poland, since October 2012. His main research interests include empirical software engineering with a focus on software modularity, post object-oriented paradigms, and agile methods. He is the Founder of the International Conference on Lean and Agile Software Development (https://lasd.pl). He has also served on the program committees of ENASE and ACM SAC, since 2015, and MADEISD@ADBIS since its origin, in 2019.