sensors

MDPI

Article

Comparison of Representative Microservices Technologies in Terms of Performance for Use for Projects Based on Sensor Networks

Piotr Plecinski [1], Nataliia Bokla [2], Tamara Klymkovych [2], Mykhailo Melnyk [3] and Wojciech Zabierowski [1,*]

[1] Department of Microelectronic and Computer Science, Lodz University of Technology, ul. Wólczańska 221, 90–924 Łódź, Poland
[2] Department of Semiconductor and Optoelectronic Devices, Lodz University of Technology, ul. Wólczańska 211/215, 90-924 Łódź, Poland
[3] Department of Computer Aided Design Systems, Lviv Polytechnic National University, Mytropolyta Andreya St., 79013 Lviv, Ukraine
* Correspondence: wojciech.zabierowski@p.lodz.pl

Abstract: Reading and analyzing data from sensors are crucial in many areas of life. IoT concepts and related issues are becoming more and more popular, but before we can process data and draw conclusions, we need to think about how to design an application. The most popular solutions today are microservices and monolithic architecture. In addition to this choice, there is also the question of the technology in which you will work. There are more and more of them on the market and in each of them it is practically possible to achieve similar results, but the difference lies in how quickly it will be possible and whether the approach invented will turn out to be the most optimal. Making the right decisions at the beginning of application development can determine its path to success or failure. The main goal of this article was to compare technologies used in applications based on microservice architecture. The preparation of a book lending system, whose server part was implemented in three different versions, each using a different type of technology, helped to achieve this goal. The compared solutions were: Spring Boot, Micronaut and Quarkus. The reason for this research was to investigate projects using sensor networks, ranging from telemedicine applications to extensive sensor networks collecting scientific data, or working in an environment with limited resources, e.g., with BLE or WIFI transmitters, where it is critical to supply energy to these transmitters. Therefore, the issue of efficiency and hence energy savings may be a key issue depending on the selected programming technology.

Keywords: Spring Boot; Micronaut; Quarkus; microservices; web application; sensors networks

Citation: Plecinski, P.; Bokla, N.; Klymkovych, T.; Melnyk, M.; Zabierowski, W. Comparison of Representative Microservices Technologies in Terms of Performance for Use for Projects Based on Sensor Networks. *Sensors* **2022**, *22*, 7759. https://doi.org/10.3390/s22207759

Academic Editor: Diogo Gomes

Received: 15 September 2022
Accepted: 11 October 2022
Published: 13 October 2022

1. Introduction

One of the most important tasks before starting work on a project is to create an appropriate application architecture. The discussed issue has an impact on the functioning of the enterprise, the pace of changes, and security, among others. The consequences of this choice can be felt in everyday work, and changing the approach would involve long and costly changes that would cause a lot of problems and would slow down the development of applications from a business point of view.

In our department, we run many projects based on sensors or sensor networks. These include sensor network projects collecting data from complex high energy physics projects and large rotor machine endurance testing projects. Research related to smart clothing and telemedicine is also carried out. So far, in many situations, applications created by our teams have been executed in various programming technologies. Sometimes it was due to the technology being imposed by the project partner and sometimes simply because the language was better known or preferred in a given team. In addition to

the general knowledge and experience of our programmers, which generally gave the knowledge of the best choice of programming technology, we found that this was a further field for optimization. This was especially so, since in some projects the processing time and communication speed could and did have a critical importance. For example, if data collected from sensors placed on the patient's body sent data to a device and the device had to make a decision based on this analysis, time was of the essence. The delay in this case could mean the patient's fall. In another project, the data collected using information from BLE beacons and WiFi transmitters, apart from the fact that they should be processed relatively quickly, additionally, due to the difficulties with powering sensors and beacons, it was crucial to minimize energy consumption, so it was necessary to reduce the transmission time. Yet in another project, due to problems with power supply in the form of a battery, it was necessary to reduce energy consumption in general, so it was necessary to optimize resource consumption, processing times, etc. so that the control process itself consumed as little of such valuable energy as possible. Therefore, although the problem of performance is known among other studies and other teams, the research they conducted did not prove useful to us. The tests that were planned and carried out by us have been tailored to our needs, including, among others, database operations. In the following, we refer to the state of art, pointing to possible gaps in the research conducted for our needs, without in any way detracting from the achievements of other scientists. We just needed something else.

2. Related Works

Considering current solutions, there are several aspects to consider, such as: efficiency, scalability, manageability, and reliability. Currently, the best-known solutions include monolithic architecture and microservices.

This topic is very popular nowadays and there are many new, fast-developing technologies that are constantly being researched and tested in order to find the best one to use.

Matthias Graf conducted a comparison of technologies used in microservices in terms of performance and ease of implementation. In performance tests, he looked at compile time and application startup, as well as peak performance. These are important factors, but there are many more aspects to consider when choosing a form of technology [1]. A more interesting and useful study was conducted by Roman Kudryashov, because, in addition to measuring the times of specific actions, he also paid attention to memory consumption, which is very important if we use cloud services [2]. Similar tests were recently conducted by a Polish team comparing the above-mentioned technologies with the Javalin framework, which is not a microservices solution and therefore was not included in our comparison. The tests traditionally checked the speed of specific actions and resource consumption [3]. The previously mentioned research focused on checking which technology was faster and consumed less resources, but did not address the issue of application stability under a heavy load or the speed of database operations. In production systems these aspects are crucial and determine customer satisfaction. Spring Boot is one of the most popular technologies and you can find many publications comparing it with others in terms of performance. An interesting study was conducted by Hardeep Kaur Dhalla. He tested Spring Boot's capabilities in terms of load and resource consumption and compared it with the results of another well-known MS.NET technology. This was a very useful study, but unfortunately it did not include the research elements that are important to the developer, such as compilation time or test-run time [4]. The microservices architecture used by the authors is becoming more and more popular because it gives tremendous freedom in terms of technology choice. Aristide Massaga and Georges Edouard Kouamou are the authors of one of the most recent studies in this field. They decided to evaluate the adaptation of a given technology to microservices architecture. They approached the matter mathematically and developed a formula to present the results as a percentage. Unfortunately, they also focused on older technologies, such as Spring Boot and Java EE 7 [5].

Already in [6] they rightly note that the issue of the increasing use of microservices is worth examining. The authors tried to review the differences between the architecture of microservices and monolithic applications. They compared the performance of the monolithic application to the version in Consul and Euroreka when it comes to microservices. However, the developed comparison was not in-depth because, in principle, they focused only on the load test and concurrency testing. From our point of view, this is not enough. The authors [7] of the publication on Benchmarking for Microservices focused on the problem in a similar way. The fact of trying to answer the method of assessing the performance of this technology fits in with the demand for these results. Again, however, they did not address the key performance issues from our point of view. Our work presented in the article is a continuation of the problem of the performance of microservices for use in applications analyzing data from sensors. The issue of efficiency was worked on in relation to the solutions of the Amazon platform [8]. The authors [9] showed the efficiency of the applied microservice solutions in telemedicine with a cloud environment. They test their solution in a very general, and one can say comprehensive, perspective, i.e., the number of inquiries and the number of users. Although this is a very interesting and important study that gives an overview of the performance of the solution, we go further; that is, we consider the efficiency for atomic operations, and thus our research results provide a greater possibility to adapt the implementation of the sensor application, but also the design of the sensor network structure or even the returned data and their format. From this point of view, our research is more in-depth. The importance of using the sensor network in software via the microservices platform is also evidenced by its use in cattle breeding [10]. The authors of the publication in Applied Science [11] focused on the possibility and application of microservices for data exchange and production in industrial processes. Of course, it is no secret that microservices are suitable for such applications, but in this case, the authors were not limited by resources, as is often the case when analyzing data from a sensor network, when this analysis must be done quickly, and with the lowest consumption of resources and energy. Therefore, their research emphasizes the importance of the problem that was highlighted in this article. Finally, comparative performance studies can be found between monolithic and microservice architecture. Although the title [12] announces that it is a comparative study, it nevertheless focuses on examining general problems and benefits, and challenges in the transition from a monolithic to a microservice model. Although this is a very important voice in the discussion and contains important experimental data, they are, again, not related to the problem that can be encountered with the high requirements of applications that analyze ad hoc data from sensor networks, such as in medical applications, when the speed of data processing depends on the patient's reaction or a decision about some action, while taking care to use the lowest possible consumption of energy and therefore also physical resources, including sensors in the network or communication beacons. The same is the case in [13], confirming the popularity of microservice solutions, but without any significant reflection on the efficiency of nuclear activities. In the applications discussed by these authors it may not matter, but they differ significantly from the environment we are considering. The authors of the next publication make an interesting comparison of the microservices technology of the most popular platform, but they still lack some of the tests that the authors have made in this article. Besides, the study conducted in [14] concerns only one of the platforms and there is a much narrower comparison than in this article because it concerns three technologies. In another study [15], the authors dealt with the tuning of microservice solutions, noticing the problem with matching performance to the needs. The authors analyzed the Bayesian optimization approach both in terms of solution speed and resource allocation efficiency. The results obtained by the authors do not exhaust the issues discussed in this article. The authors collected data from sensors [16] geographically distant from each other due to the essence of the implemented environment. Although we are dealing with a network of sensors, the performance issues that were at the basis of our research, in this case are not crucial and Panduman and others do not think about it at all in this respect. The authors

analyzed [17] and collected runtime logs from six microservices in the retail domain. Although the study itself looks promising, it is quite superficial. It is an interesting approach in analyzing the impact of the communication protocol [18] used for microservices on the implementation performance. It shows that this has an impact on performance and resource consumption. The authors investigated and evaluated the microservice delivery performance of the selected platform [19] by considering the details of all layers in the modeling process. To this end, they first built a microservice platform on the Amazon EC2 cloud and then used it to develop a comprehensive performance model to perform conditioning analysis and capacity planning for large-scale microservice platforms. In other words, the proposed performance model provided a systematic approach to measuring the flexibility of a microservice platform by analyzing the provisioning performance on both the microservices platform and the back-end macro service infrastructure. From this point of view, this study is very useful, but it does not fit the microscale solutions that are so crucial for our tested solutions. The authors present the use of microservices [20] and their architecture from the sensor system and data processing using the developed environment. This demonstrates the effectiveness of this architecture for sensor network platforms.

Yilmaz and Buzluca [21] discussed the problem of maintaining microservices. In their work, they proved how important this aspect was in the maintenance of systems based on microservices and used fuzzy logic to do so. The topic of choosing the right microservice technology based on various frameworks, taking into account the response time, is discussed by Liu and his team [22]. This proved that this performance aspect of microservices should be taken into account in systems where processing time matters. Subsequent works deal in general with the design of microservice systems in order to ensure their best performance [23–26]. Another important aspect that appears in the literature and influences the design of sensor systems is security [27], in particular in edge computing, and the authors analyzed solutions based on AI.

It must also be said about the use of microservices in the production of similar ones to those we perform in our research. However, as you can see in these articles, the authors, although they show very interesting solutions, do not deal with performance in the way that we present. They treat microservices as a tool, without bending over details that could, or even certainly, affect the performance of the whole. First of all, we are talking about interesting works in the field of multisensory patient networks [28], very interesting work with the use of microservices for the measurement system "Nanoplasmonics" [29] or finally on the architecture of a system implementing multi-tenant vehicle monitoring [30].

The importance of the use of microservice solutions is demonstrated by, among others, Al-Debagy I Martinek and Mazlami with a team [31,32] proposing to migrate from a monolithic application to a microservices application. The methods and challenges in such a migration are also dealt with by Kyryk's [33] team, focusing on the processes and methods of such conversion. The aspects of software engineering in such a conversion are dealt with by De Lauretis [34], taking these aspects into account, disregarding performance issues. A similar approach to the subject can also be found in other works and other groups [35–37]. This only confirms that the topic of migration from monolithic applications to microservices is on time, and therefore how many challenges, including performance challenges, are to be solved and described.

The importance of the performance aspect is understood by, among others, Mostof and his team [15], but also Wan and others [38], who focused on load balancing in microservice applications and Khazaei with his team who provided an analysis of provisioning in microservices [14]. The use of technology in IoT solutions in edge computing is a very interesting work by Nisansal [39]. They dealt with an important problem of efficiency, but the results of their research, although very interesting, do not correspond to our needs in specific design choices. However, it is a very important voice that such research and the perception of these problems is noticeable. The use of microservices in IoT is also a presentation of the design methodology, taking into account the challenges of IoT [40].

A very interesting work [41] related to the general comparison of monolithic applications and those based on microservices, but the authors focused on the aspects of performance and scalability. Although it is a very interesting work, from our point of view, it does not deal with detailed issues related to the problems that we encountered when creating applications for our projects with sensor networks. Herbke and his team [42] are heading towards optimizing solutions based on microservices by correctly identifying the need to analyze and optimize microservice solutions. The work by Li [43], which introduces the concept of quality in these solutions of microservice architecture, resounds in this vein.

Important selections are the works cited in succession [44–48]. In the first one we can find a comparison of monolithic and microservice architectures, also in terms of certain performance aspects, but to a certain degree of generality due to the selection of only one of the microservice platforms. The next one is managing resources in edge computing with the use of microservices. This is an interesting voice corresponding to the work presented in this article, but dealing with other aspects of this management, also not referring, as in our article, to various microservice platforms.

The next three articles (listed above in the citation) are an attempt to estimate the performance and microservice solutions in various approaches. It contains very useful research and results providing a lot of help in optimizing processes and implementation decisions. However, they differ significantly in their approach to ours. It does not mean that theirs is worse or ours is better. Our approach and research are simply a response to a specific problem in our teams' research. Our results are a valuable answer for us and our projects, but also for other teams dealing with these problems. It is not an exaggeration theorem, because we obtained a positive reception of the results of our research through the exchange of experiences with other teams. The results of our research have often been met with positive curiosity and a positive response to the results and conclusions obtained in relation to various microservice platforms.

Nowadays, more and more institutions are using the cloud and that is why the authors thought it was worth checking and comparing currently available solutions. Micronaut and Quarkus are fairly young technologies and it is sometimes problematic to find research to help you decide which technology to choose for your project. Our study was conducted on three identical applications developed with three different technologies and was intended to extend the range of tests conducted so far and to identify the best application for the technologies.

3. Architecture

Many well-known companies have moved from monolithic architecture to microservices. One of the more famous examples is Netflix. The streaming service is connected to thousands of microservices and data store instances on AWS (Amazon Web Services) to serve millions of users around the world. The system is under heavy load due to tens of millions of real-time mutations, replicated globally, per second [49] and it still appears to be reliable.

3.1. Monolith

The term monolith refers to systems that are designed as single deployment units. Most often they have only one database, which is shared by all services. In theory, the biggest advantages of such architecture include its simplicity and accessibility. This approach does not require thinking about the division of services and functionalities, because everything takes place within one application. Unfortunately, in the case of an error occurring when even one service is called, the whole application usually stops working and finding the cause itself is often problematic. In such a central system, one technology is used and for some functionalities it is simply not advisable and in another one the given demand could be executed much faster and more efficiently. It is this lack of flexible approach to change and low scalability that are among the biggest arguments against this solution. The monolith is difficult to scale due to irregular consumption of different services deployed

on a single application. When there is demand for highly consumed services, additional infrastructure is needed for the entire application, regardless of the increase in consumption of services. For this reason, sometimes server resources are wasted on unused services [50].

3.2. Microservices

The term microservices describes an approach to software development that decomposes business domain models into smaller and consistent services. An application consists of multiple modules and each module is responsible for a single functionality. Microservices are separate and fully autonomous components that communicate with each other to form a coherent whole. Usually, a separate team is responsible for the implementation of one such service. Such an approach enables independence in system development and does not impose the technology when implementing a given service [49].

Advantages:

- Scalability;
- Freedom of technology choice;
- Code quality;
- Stability;
- Shortening the production cycle.

Disadvantages:

- No transactivity;
- Challenging design of architecture;
- Unprofitable in small projects.

3.2.1. Compared Technologies

In our study, we took the most popular, in our opinion, microservice technologies. We briefly outline them in the following sections.

3.2.2. Spring Boot

When discussing the first technology, Spring Boot, it is important to mention Spring itself. It is a popular, open-source, JVM-based framework for creating standalone, production-grade applications [51]. Its creators tried to solve the problems appearing in Java Enterprise Edition (JEE). The initial versions were clumsy and the application development itself was not an easy or pleasant task. The solutions used in JEE were complex and not easy to configure. The goal of the developers was to create an accessible product for everyone and that is why they decided to provide default configurations at the start, which made the work much easier. One of the most important advantages of Spring is their own IoC container, which is responsible for creating, managing and configuring bean objects. It manages the entire life cycle from the *new* operator to the *finalize* method. The IoC abbreviation refers to the Inversion of Control pattern, which in practice means the transfer of responsibility for object creation to the Spring container. Referring to the topic, Spring Boot is a tool that makes developing web application and microservices with the Spring Framework faster and easier [52].

3.2.3. Micronaut

It is a JVM-based technology that was created in 2018 and its creators are the people responsible for creating Grails. Micronaut is a modern framework for building modular and easily testable microservice applications. The supported programming languages are Java, Kotlin and Groovy [53].

The biggest differences between the Spring Boot and Micronaut include the method of bean creation. In Spring, during startup, classes with appropriate annotations are scanned, from which beans are then created. The described process is not the fastest one, because it is based on reflection. In Micronaut, a different approach was chosen. An

annotation processor was used, which takes care of these issues at the compilation level. The consequence of such action is a shorter application startup time.

Micronaut was designed in such a way that it harmonizes with cloud solutions. It has built-in integration components for GCP (Google Cloud Platform) and AWS as well as for Kubernetes. Thanks to fast startup time and dependency injection during compilation, Micronaut is very well suited for serverless environments. Spring Boot suffers a bit more in this area because it uses more memory and takes more time than Micronaut, and the application customization itself is more difficult and requires more attention.

3.2.4. Quarkus

RedHat is responsible for creating this technology and its first version was released in 2018. Quarkus was developed to work with the GraalVM universal virtual machine, but it is also possible to work with the JVM. The supported programming languages are Java and Kotlin. Quarkus does not use reflection, but injects dependencies at compile time. Thanks to that, the application starts up comparably as fast as Micronaut, and on native images startup times drop to the millisecond level. The presented technology was based on the Container First concept. It was created to have the best possible results in such categories as: memory consumption, runtime and docker image size. Such features have allowed it to become an effective platform for serverless, cloud and Kubernetes environments [54].

3.3. Popularity

The popularity of a given technology is worth noting, because when choosing technological solutions, the support of the community and the developers' plans for the future should also be taken into account. The greater the popularity, the greater the chance that the authors of the project will develop it further. Another noteworthy factor is the fact that in case of encountering problems while programming, it is much easier to find help on the internet for technologies with a large number of supporters.

Github

When comparing the popularity of the analyzed technologies, it is worth looking at their code repositories on Github. The service provides the possibility to mark a given repository with a star, which means that a given user considers it noteworthy and is satisfied with the content provided.

Spring Boot has about 57,000 such ratings, Micronaut 5000, and Quarkus 8400 (Figure 1). These results confirm that the product from Spring is the most popular, which cannot be a surprise since it has definitely been on the market longer. As for Micronaut and Quarkus, they are newer offerings; Quarkus came a little later, and you can see that it is already more popular than Micronaut by more than 50%.

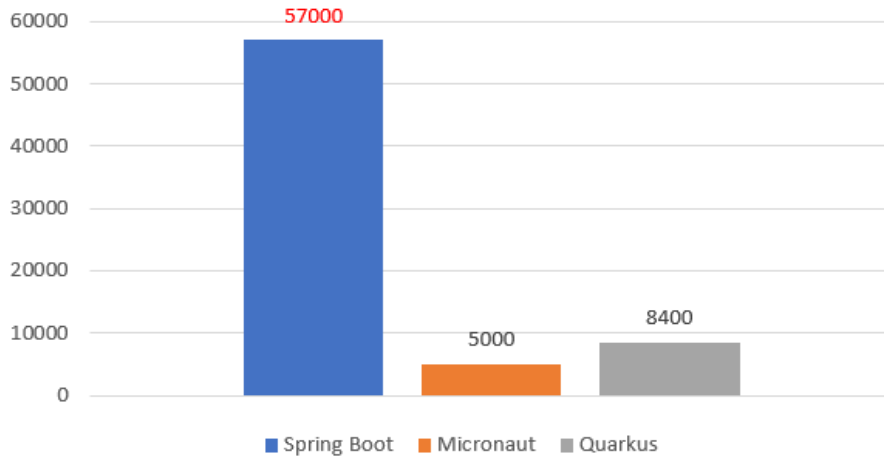


Figure 1. Popularity on Github.

An interesting comparison was presented by the JAXenter portal [55], which in 2020 conducted a survey on popular technologies used for application development. Participants could indicate their attitude towards each of them on a scale from not interesting at all through to neutral and to very interesting.

Also in this comparison Spring Boot definitely wins, but again the result of Quarkus is worth noting. Despite the fact that it was created only in 2018, it already managed to take the third place. For 22% of respondents, it was very interesting, and for 25%, it was at least interesting. Micronaut, on the other hand, ranked in the middle (Figure 2).

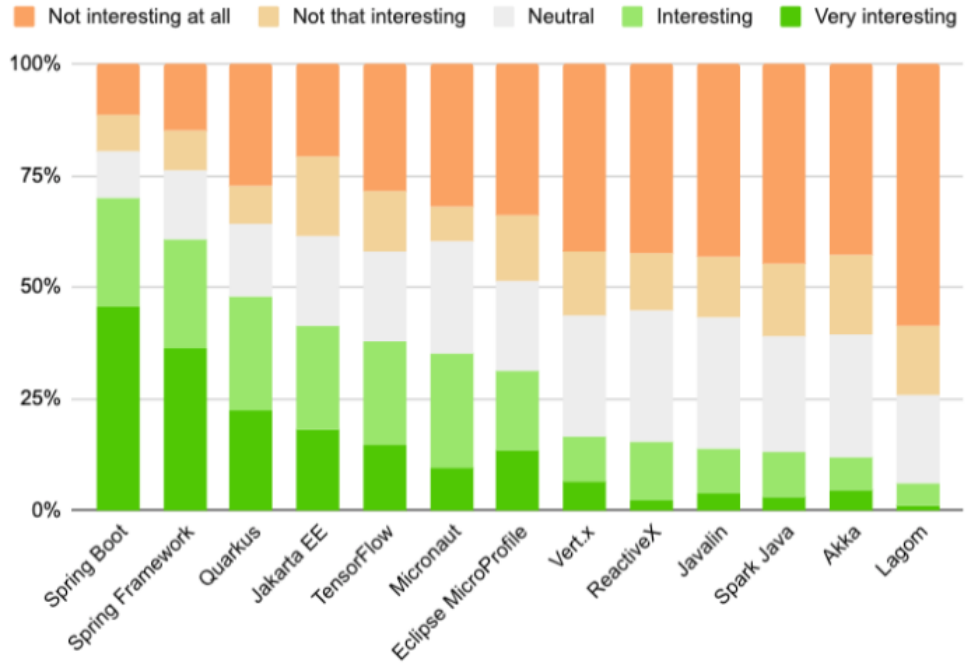


Figure 2. Results of a survey conducted by JAXenter [55].

4. Materials and Methods

Spring Boot (v. 2.3.10), Micronaut (v. 2.1.1) and Quarkus (v. 1.13.4) were tested for performance, stability and resource requirements. To test performance, three applications were developed that were as similar as possible. The projects were generated from the developers' websites with only those dependencies that were necessary. The application itself was very simple. It consisted of a controller, a service and a class-containing bean configuration. The prepared service tests were written using RestAssuredMvc library, which allowed us to test API calls. This application will be called "First type app" in this article. One of the most important aspects of running the application in production is its stability, and to establish this stability, the three applications were implemented as a book rental service, which were based on a microservices architecture. That will be the "Second type app".

The comparison was made under repeatable and identical conditions. In order to ensure real results, the tests were repeated serially, and the number of series was selected based on experience. So, for some tests, 5 was enough, and for others, as much as 1000 units. To increase the readability of the experiment, we first presented which tests were taken into account, and only later the results themselves, summarized in the form of graphs and with appropriate comments.

In our research work, equipment with the given specification was used. The presented tests took place on a laptop Lenovo Legion 5 15ARH05, whose parameters are as follows:

- CPU: AMD Ryzen™ 5 4600H 3.0–4.0 GHz;
- RAM: 16 GB;
- GPU: NVIDIA® GeForce GTX™ 1650 + AMD Radeon™ Graphics;

- Drive: 512 GB SSD;
- OS: Windows 10 Home Edition.

Due to the prior analysis of the performance needs of our projects, only some aspects affecting performance were tested. These are the aspects that constitute a bottleneck or a potential field for optimization of the final application.

Due to the convenience and speed of development and deployment, we also took the compilation time into account.

The results obtained were the arithmetic average of five attempts for "First type app" The *mvn clean compile* command was used to compile.

Due to the specificity of the results, also for the remaining tests, we did not subject the obtained results to a special statistical evaluation, but only used the arithmetic mean. In our opinion, this is completely sufficient to obtain knowledge about the goal set at the beginning of the research. Entering complex and elaborate statistics would be an artificial activity.

Testing plays a very important role in our projects. In many situations, we cannot afford to run tests after the final application has been deployed.

Therefore, an important aspect that we took into account due to frequent changes in the application environment was the testing time.

The tests were written using RestAssuredMvc and checked the returned value, which was a simple String test. The presented result is the average of five trials for the "First type app".

Due to possible restarts of the implemented system, sometimes the time of starting the application is of key importance. The result of the test is an average time needed to start the "First type app", based on five attempts.

In all or almost all our projects, the applications work with the database. The time, efficiency of basic database operations is of great importance for the overall data processing and obtaining the appropriate system decisions. The result of the test is an average time needed to start the "First type app", based on five attempts.

Therefore, the performance of save and read operations from the database was analyzed. Due to the high speed of the read operation, it was necessary to increase the number of cycles in order to obtain measurable results that could be compared.

For the "Save" operation, the result is the average of three attempts to add 1000 identical records to the database, and empty the *BOOKS* table.

On the other hand for the "Read" operation, the result of the test is an average time needed to read 10,000 identical records from the *BOOKS* table, based on three attempts.

Application stability is a very important aspect. Therefore, it is important to choose the technology with the greatest certainty of stability. For the same reason, C++ based systems do not fly into space.

Stability tests were conducted for the "Second type app", using the popular tool Gatling. It allowed us to introduce an actor model oriented towards sending requests instead of creating threads and thus generating more workloads. One of its biggest advantages is the ability to create or record test scenarios. The test scenario used to check the application overload was recorded while using the developed library and included routine user actions, such as displaying data from the database.

Each technology was tested under identical initial conditions, where the number of actors was 50 and the database had, respectively, 500 users, 1000 reservations and 2000 books.

For each application, tests were performed to find the threshold of first errors.

A very important factor in our projects is the high volume of received data. Therefore, in the study, we took into account requests per second. For this kind of testing Apache Benchmark was used, which tested the exposed API with huge amounts of requests. The console command *ab -k -c 20 -n 10000 http://localhost:8081/test* was sufficient to run the test. The parameter *n* specifies the total number of http requests that were made. The *c* parameter specifies the number of clients that were created to send requests in parallel. *http://localhost:8081/test* is the address that was tested with parameters c = 20 and *n* = 10,000

respectively. The results obtained are shown in the graph below and are the average result from five attempts for the "Second type app".

Of course, all the previously mentioned problems, whether related to energy management or response time, are influenced by the use of memory and processor—key resources. The Apache Benchmark tool presented in Section 4 was also useful for checking resource requirements. For each technology 20 such commands *ab -k -c 20 -n 10000 http://localhost:8081/test* were executed for each technology, and at this point the CPU and memory consumption was observed. VisualVM provides a way to see what is happening with applications running in a Java virtual machine. The initial results were not taken into account because the first queries are more for warming up the application.

5. Results

5.1. Compile Time

The first test focused on compile time (Figure 3). It was performed using *mvn clean compile* command. As you can see, the best time was achieved by Micronaut (2.2194 s), but its advantage over Spring Boot was minimal. Quarkus was slower than them by about 0.2 s.

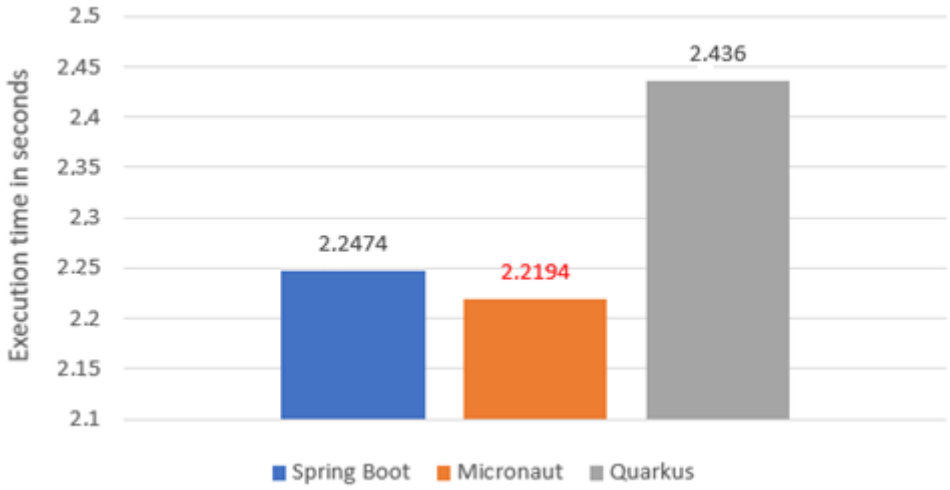


Figure 3. Average application compile time.

The results of this test are not surprising, as one would have expected Spring Boot to be no worse than its competitors in this category. This is due to the fact that both Micronaut and Quarkus deal with bean issues at the compilation level, while Spring Boot only deals with them at runtime.

5.2. Test Time

The second aspect compared was test execution time (Figure 4). In this case, the Micronaut proved to be by far the fastest (7.852 s). Its result was by more than 2 s better than Quarkus, which was slightly ahead of Spring Boot. This may be due to the less complicated configuration of the class loader.

5.3. Startup of the Application

Next, the timing of one of the most important actions for a developer, which is launching the application, was examined (Figure 5).

As mentioned earlier, Spring Boot scans annotated classes at startup from which it later creates beans, while the other technologies tested inject dependencies at compile time. So one would have guessed that in this comparison Spring Boot would turn out to be the slowest and that is exactly what happened, with Micronaut again being the fastest. This test would probably end up with a different result if the applications were run on native images, where Quarkus could present its full potential. According to some sources, its results then drop to the millisecond level.

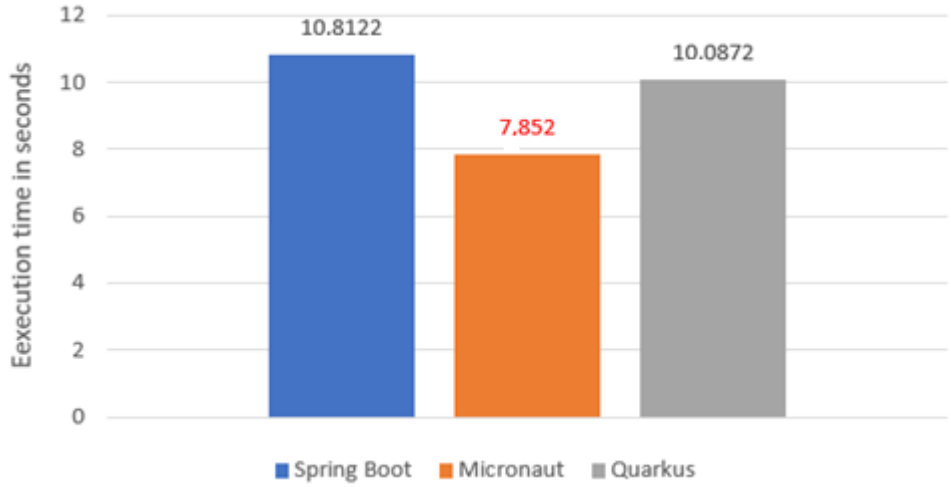


Figure 4. Average application test time.

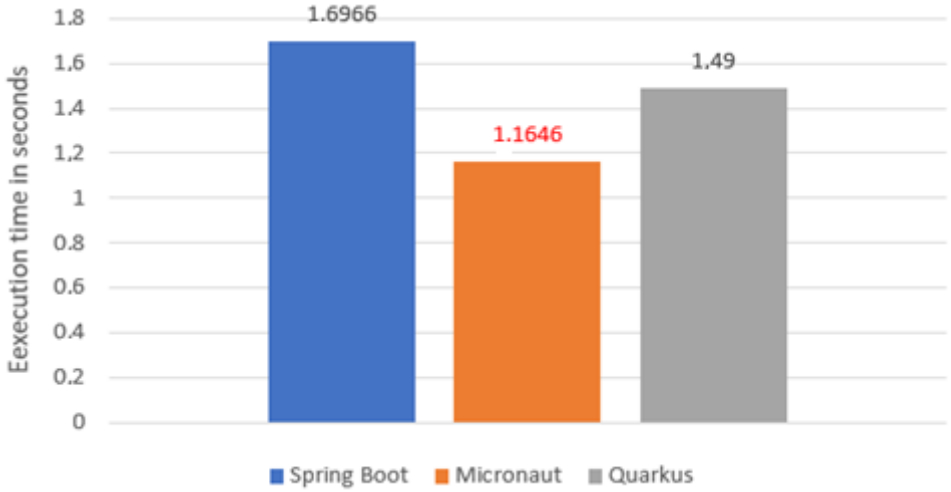


Figure 5. Average startup of the application time.

5.4. Database Operations

5.4.1. Save

Considering process of saving 1000 books (Figure 6), Quarkus proved to be the most efficient, with saving 50% better than Spring Boot and about twice as fast as Micronaut.

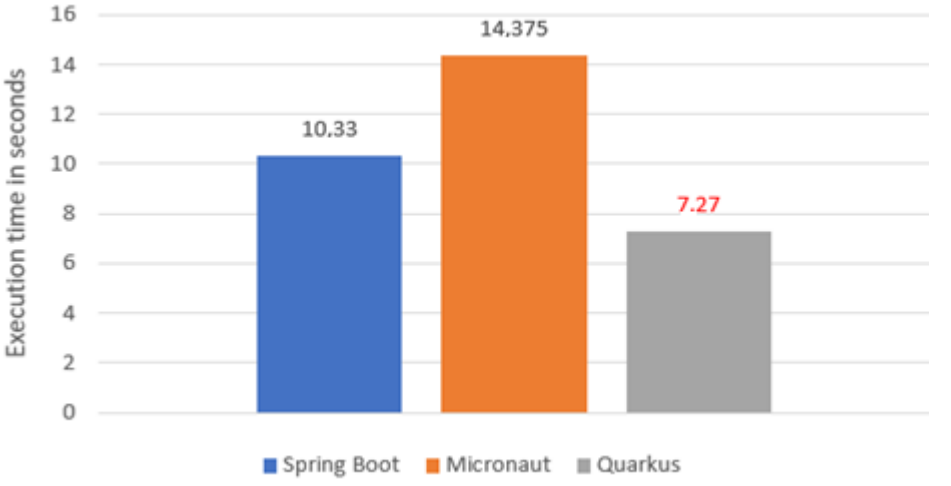


Figure 6. Average time of saving data to database.

5.4.2. Read

In the case of reading data, Quarkus was also the fastest, but in this case it had a decisive advantage over Spring Boot and was much slower than Micronaut (Table 1). When analyzing the results, it is worth noting that Quarkus uses PanacheRepository to manage data on the database, which is their own overlay on Hibernate. The developers' goal was to create the simplest possible mechanism for communicating with the database. The results show that one of the biggest advantages of Quarkus is its speed, which makes it seem like an ideal candidate for native solutions.

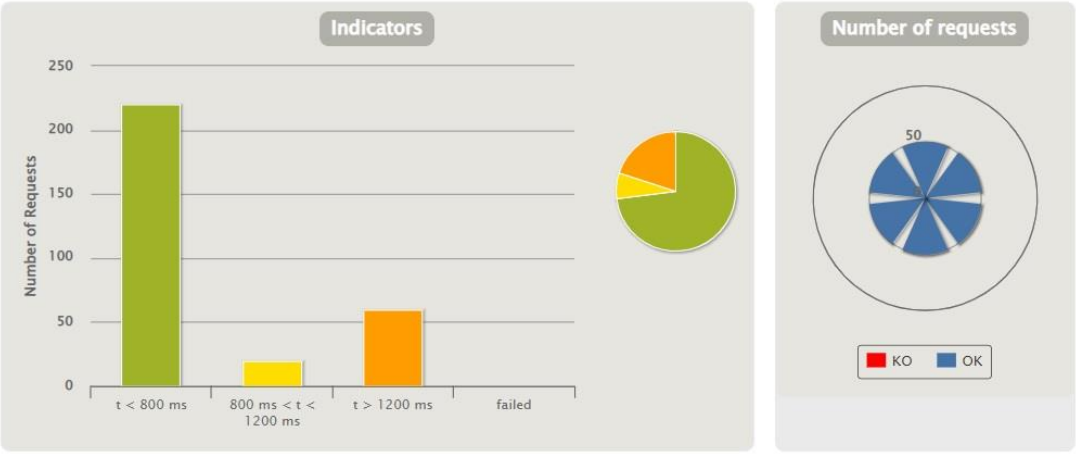
Table 1. Average time of reading and writing data to/from database.

	Spring Boot [s]	Micronaut [s]	Quarkus [s]
Write (1000 cycles)	10.330	14.375	7.270
Read (10,000 cycles)	0.152333	2.665000	0.024817

5.5. Stability

5.5.1. Tests for Identical Data

The following figure presents the results from the test run for Spring Boot (Figure 7). The presentation itself is detailed and easy to read for the user. As you can see, Spring Boot handled this test without any major problems. Analyzing the figure, you can see the division into three parts. In the upper left corner, there is a bar chart that represents how many queries were executed in under 800 ms, how many in the 800–1200 ms range, and how many in over 1200 ms. The fourth bar reports errors, but none appeared here. In the upper right corner is a slightly modified pie chart where you can see how many queries of a given type were correct. In the case of Spring Boot, they all executed and there were no KOs. Below the graphs are more detailed statistics, visualized by the previously mentioned charts. Below are also the results for Micronaut (Figure 8) and Quarkus (Figure 9).



STATISTICS — Expand all groups | Collapse all groups

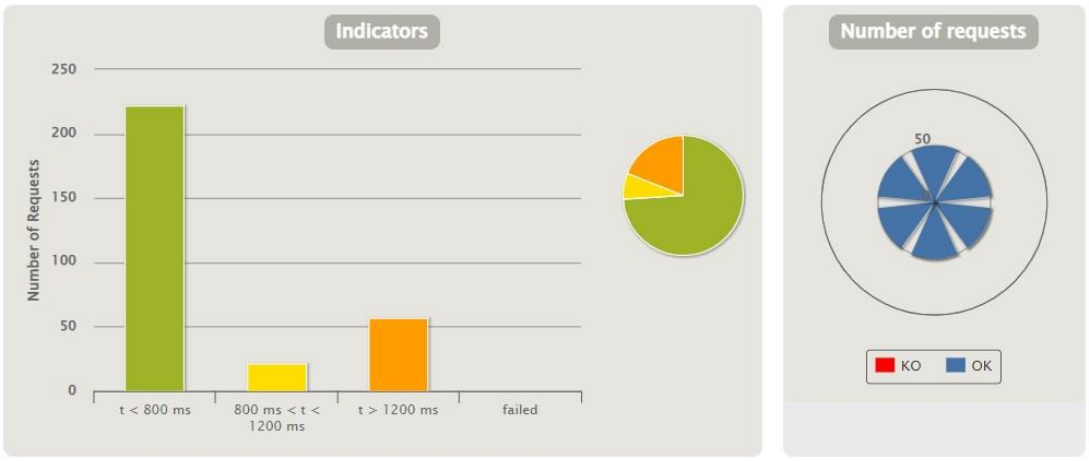
Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	300	300	0	0%	10.714	1	52	957	2758	3388	3613	566	890
request_0	50	50	0	0%	1.786	21	69	249	280	281	281	148	103
GET_USERS	50	50	0	0%	1.786	393	1040	1343	1636	1651	1658	1032	423
request_2	50	50	0	0%	1.786	1	2	3	4	5	5	2	1
GET_BOOKS	50	50	0	0%	1.786	945	2144	2767	3509	3613	3613	2149	875
request_4	50	50	0	0%	1.786	1	1	2	2	4	4	1	1
GET_RESERVATIONS	50	50	0	0%	1.786	25	31	38	202	221	222	62	63

Figure 7. Results for test data: 500 users, 1000 reservations, 2000 books, 50 actors—Spring Boot.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	250	208	42	17%	5.208	2	1395	11127	14470	24185	27052	5385	6437
GET_USERS	50	20	30	60%	1.042	11109	11573	12723	23564	27048	27052	13838	4713
request_1	50	50	0	0%	1.042	130	267	340	725	1019	1054	323	210
GET_BOOKS	50	40	10	20%	1.042	876	10529	11417	17483	22802	24153	9614	4782
request_3	50	50	0	0%	1.042	2	7	2094	7661	10054	11233	1449	2630
GET_RESERVATIONS	50	48	2	4%	1.042	19	46	1327	10294	11038	11041	1702	3332

Figure 8. Results for test data: 500 users, 1000 reservations, 2000 books, 50 actors—Micronaut.



STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	300	300	0	0%	11.111	0	32	862	3612	5048	5074	681	1204
request_0	50	50	0	0%	1.852	22	65	240	262	264	264	142	98
GET_USERS	50	50	0	0%	1.852	356	866	1134	1245	1258	1267	858	278
request_2	50	50	0	0%	1.852	1	2	2	4	6	7	2	1
GET_BOOKS	50	50	0	0%	1.852	1472	3029	3886	5049	5074	5074	3052	1149
request_4	50	50	0	0%	1.852	0	1	1	3	5	5	1	1
GET_RESERVATIONS	50	50	0	0%	1.852	27	29	30	32	35	35	30	2

Figure 9. Results for test data: 500 users, 1000 reservations, 2000 books, 50 actors—Quarkus.

The figures above (Figures 7–9) show the results of tests conducted using Gatling for the three technologies for an identical test scenario. The database had 500 users, 1000 reservations and 2000 books. The number of actors in the script was set to 50. The interesting requests sent from the application were those related to users, reservations and books. The abbreviations were taken from the view representing the data by Gatling. As can be seen, Spring Boot and Quarkus handled this task with 100% efficiency. Micronaut's results, on the other hand, are already much worse in comparison. In this case, 42 queries failed. Analyzing the logs received in the Gateway microservice console, we were able to determine that the errors were caused by exceeding the default time limit.

5.5.2. Achieved Limits

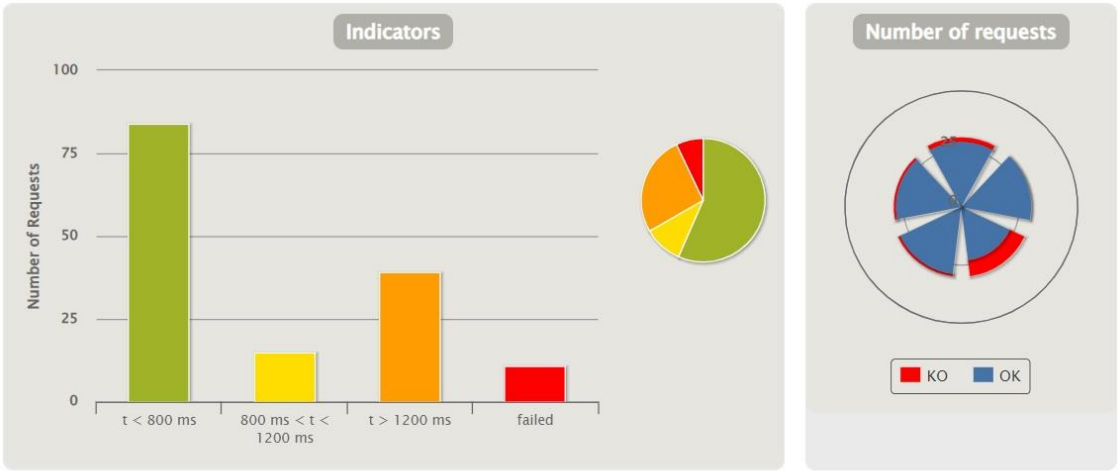
After the first tests with identical test data, further tests were performed to find the threshold of occurrence of the first errors. During the tests the resources stored on the database and the number of actors were increased accordingly. The achieved limits are presented in the following graphs—Figure 10 for Sping Boot, Figure 11 for Micronaut and Figure 12 for Quarkus.

In the load test (Table 2), the clear winner was Spring Boot. The first problems started to appear when the number of actors was equal to 200 and the number of records stored on the database was as follows: 2000 users, 4000 reservations and 8000 books. The cause of the errors was the default timeout set to 60,000 ms.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1200	1174	26	2%	11.215	0	156	10489	41401	60009	60022	8160	14883
request_0	200	200	0	0%	1.869	81	180	266	297	342	347	192	71
GET_USERS	200	200	0	0%	1.869	1206	13981	22190	29405	30618	31132	14511	9101
request_2	200	200	0	0%	1.869	1	2	2	4	16	16	2	2
GET_BOOKS	200	174	26	13%	1.869	4421	35784	42638	60011	60015	60022	34100	16501
request_4	200	200	0	0%	1.869	0	1	2	3	4	9	1	1
GET_RESERVATIONS	200	200	0	0%	1.869	73	149	159	185	338	1122	151	93

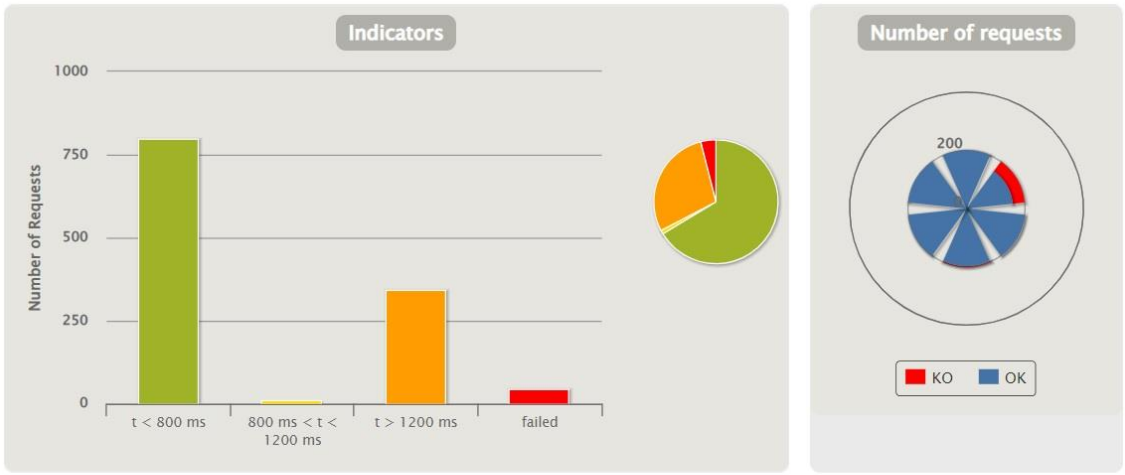
Figure 10. Results for test data: 2000 users, 4000 reservations, 8000 books, 200 actors—Spring Boot.



STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	149	138	11	7%	1.886	2	204	1590	60006	60016	60017	7994	19041
GET_USERS	30	28	2	7%	0.38	584	1267	1567	33940	60017	60017	5097	14684
request_1	30	30	0	0%	0.38	125	158	183	203	208	209	164	24
GET_BOOKS	30	23	7	23%	0.38	855	2003	57114	60010	60014	60015	19149	26344
request_3	30	29	1	3%	0.38	2	4	6	54479	58448	60003	9076	20163
GET_RESERVATIONS	29	28	1	3%	0.367	19	22	77	55312	58861	60015	6429	17227

Figure 11. Results for test data: 500 users, 1000 reservations, 2000 books, 30 actors—Micronaut.



STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1200	1156	44	4%	17.91	0	115	4565	26241	39361	44209	4163	8619
request_0	200	200	0	0%	2.985	41	110	158	266	281	310	136	59
GET_USERS	200	159	41	21%	2.985	760	5105	10268	14259	14329	14362	6511	4373
request_2	200	200	0	0%	2.985	0	1	2	62	220	802	16	71
GET_BOOKS	200	197	3	2%	2.985	2998	12491	28168	40182	42527	44209	18165	12577
request_4	200	200	0	0%	2.985	0	1	1	9	36	52	3	7
GET_RESERVATIONS	200	200	0	0%	2.985	42	109	182	508	597	802	150	142

Figure 12. Results for test data: 1000 users, 2000 reservations, 4000 books, 200 actors—Quarkus.

Table 2. Load limits for tested technologies.

	Spring Boot [s]	**Micronaut [s]**	**Quarkus [s]**
Database load	2000 users	500 users	1000 users
	4000 reservations	1000 reservations	2000 reservations
	8000 items	2000 items	4000 items
Actors	200	30	200
OK requests	574	30	200
KO requests	26	11	34

Successful amount for queries is OK, and for invalid KO. The abbreviations have been taken from the Gatling data view.

With Quarkus, the first failed queries appeared with 200 active actors and twice as little data stored in the database as with Spring. Application logs allowed us to discover the cause of the problems and it was a broken connection to the database.

Micronaut at the initial test already noted problems with exceeding the time threshold on the gateway, so here we had to look for limits by reducing the parameters. For the initial data stored on the base and with 50 actors, significant problems were noted, but when reducing the latter value to 30, the situation improved significantly, and it can be considered that the maximum values for Micronaut oscillate within these limits.

To sum up, Gatling turned out to be a great and simple tool for application overload testing. Created test scenarios allowed us to mimic real user traffic as it occurs in systems already running on the client's production environment. The winner of the test was Spring Boot. It maintained stability and responded to queries for the longest time, and the problems encountered were caused by the timeout set at 60,000 ms. It was followed by Quarkus, which also started to report the first irregularities while handling 200 actors, but twice as little data. The most surprising results concern Micronaut, because it achieved much worse results than the rest. The reasons for such results can be found in the default configuration of each technology. The main reason for the errors was exceeding the time limit or breaking the connection with the database. Potentially after adjusting the configuration to the application architecture, the results could look different.

5.6. Request per Second

In this test (Figure 13), Micronaut proved to be the best, achieving a result about 20% better than Quarkus and almost twice as good as Spring Boot.

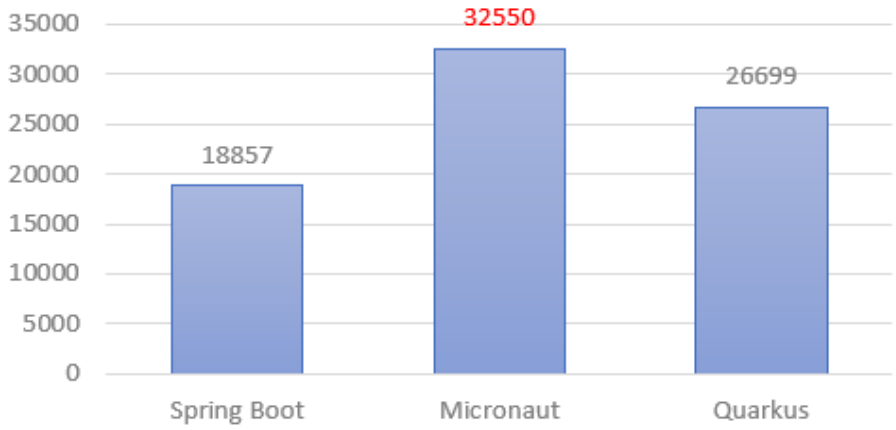


Figure 13. Average amount of requests per second.

5.7. CPU and Memory Usage

The graphs (Figures 14–16) show that Micronaut consumed the least resources. Quarkus was slightly worse, but its results in comparison with those achieved by Spring Boot were

still much better. These results confirm the trend that newer technologies are geared towards running in serverless environments. They are designed to keep startup time as short as possible and memory consumption as low as possible, since charges are incurred for the direct execution time of given functions. One of the reasons Spring consumes so much memory is the already mentioned reflection mechanism, which is not ideal when it comes to optimization.

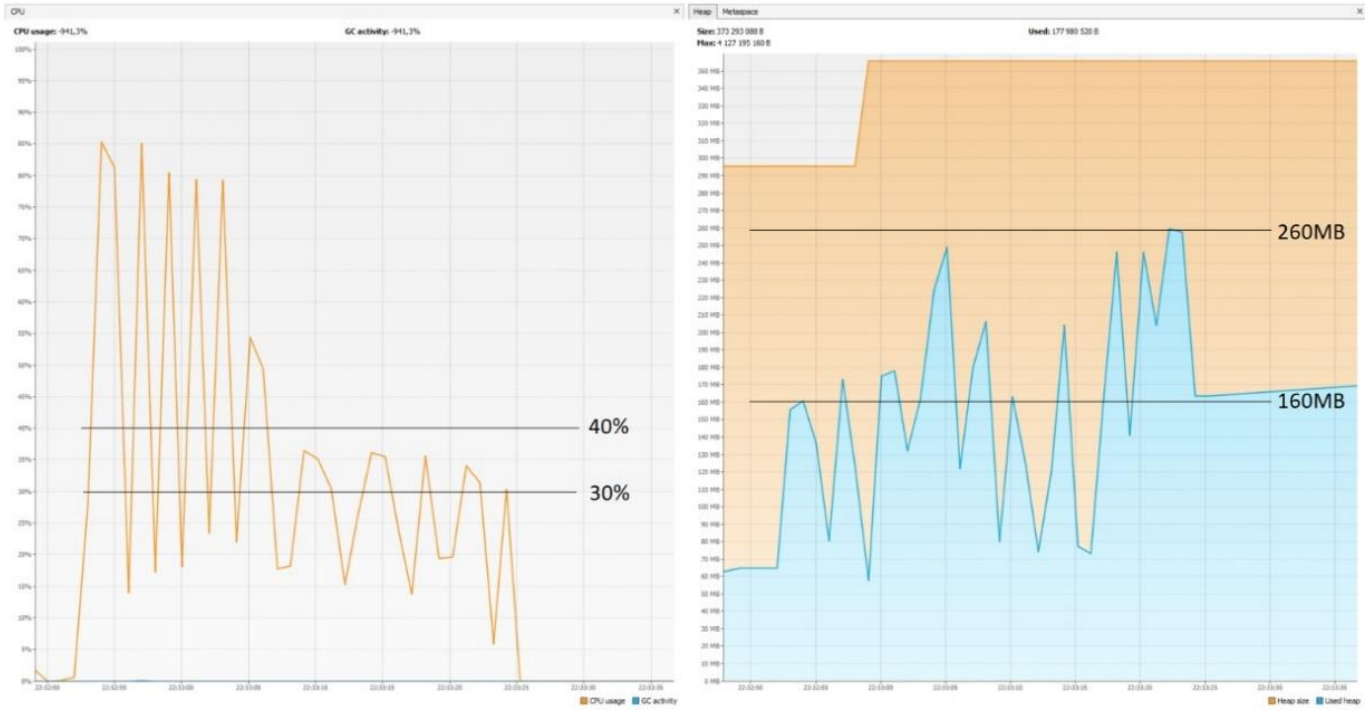


Figure 14. CPU usage: 30–40%; memory usage: 160–260 MB—Spring Boot.

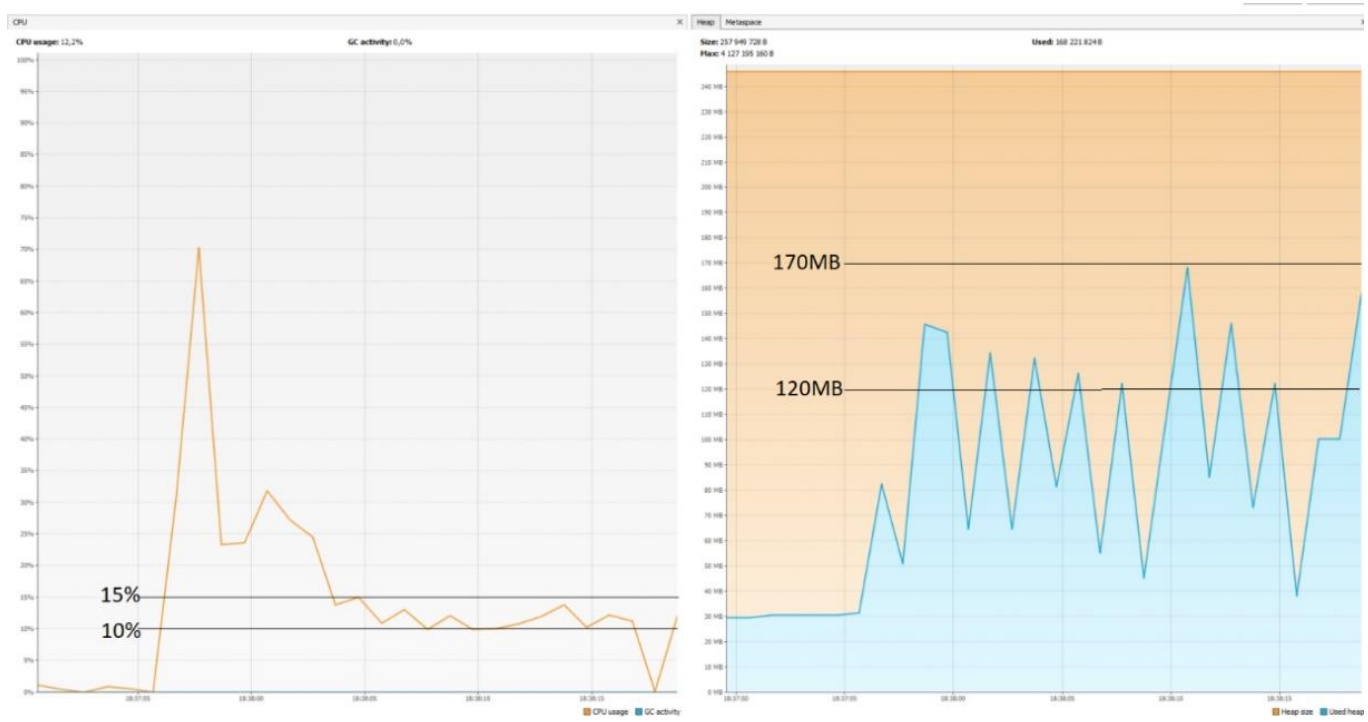


Figure 15. CPU usage: 10–15%; memory usage: 120–170 MB—Micronaut.

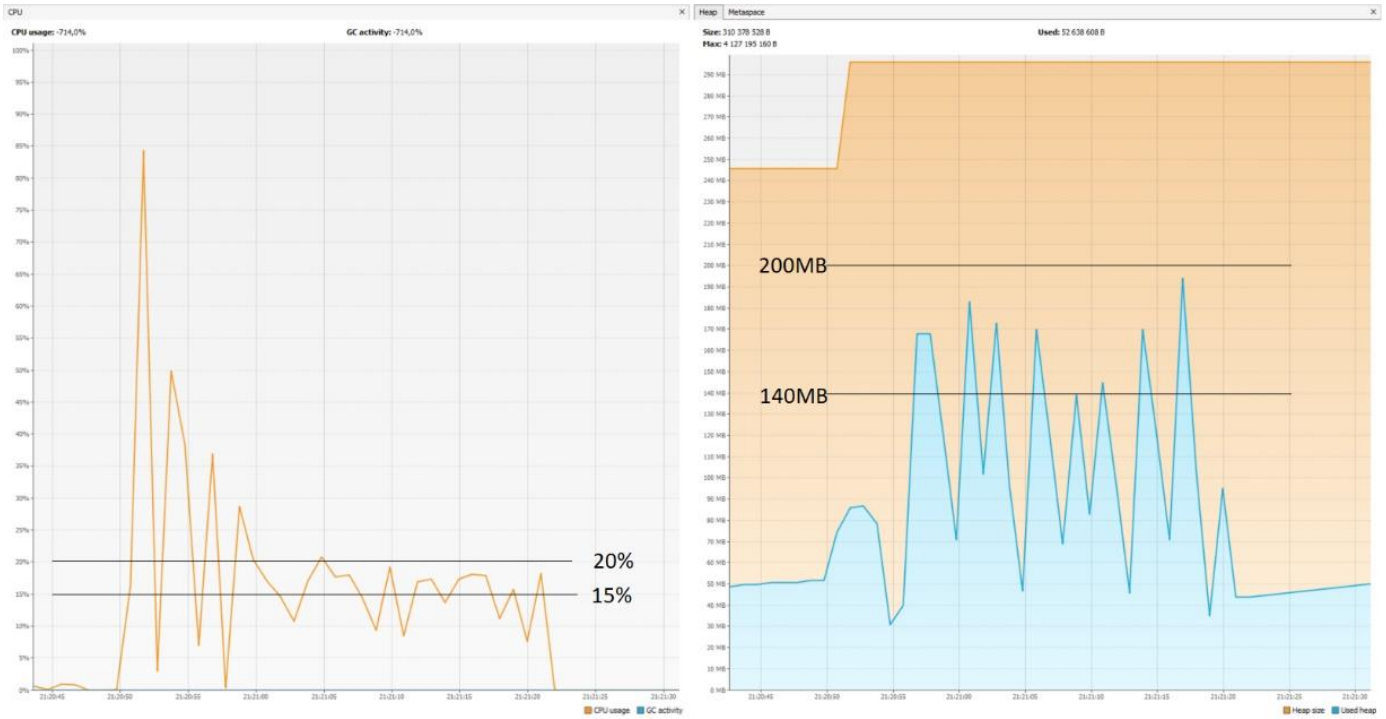


Figure 16. CPU usage: 15–20%; memory usage: 140–200 MB—Quarkus.

6. Discussion

At the beginning it is worth noting that in each of these technologies it was possible to implement the assumed functionalities in the test application and the way of implementation looked similar everywhere. The biggest differences could be seen in the used annotations and in the configuration issues. From the perspective of the authors, who had the most experience with Spring Boot, getting started with the other technologies was not too difficult because the developers provide project generators and support the developer with rich documentation.

What turned out to be problematic was finding solutions when errors occurred in the application. Spring Boot is a few years older than the other two items, and because of that it was much easier to find help on the Internet. It is more popular and has decidedly more community support at this point, but in a few years, this should change in favor of Micronaut and Quarkus, as they have a lot to offer.

The tests conducted have shown that Spring Boot's younger rivals perform better in several key elements, such as application startup time and resource consumption. This is due to the fact that dependencies are injected at the compilation level, which helps to speed up the process. However, when tested for application robustness to overloads, Spring Boot proved to be the more stable solution.

From the analysis of all these results, a very important point emerges. It is true that some results could be expected, but it is one thing to suppose and another to prove, and this is the role of the research process. Most of the results, however, were not at all obvious and both our own tests and the tools used have shown the real features of each technology.

7. Conclusions

When choosing the right technology for a project with microservices and lots of sensor data, you need to consider what it will entail. If you intend to run in the cloud, it is better to use Micronaut or Quarkus, as they are designed to run in the cloud, where costs are incurred based on time of use and resources consumed. For server-side solutions, the proven Spring Boot may be a more efficient choice.

Our research showed very important information related to the performance of each of the tested technologies, when used precisely for applications related to sensor networks. Of course, the conclusions drawn from the research are applicable to each application written in the technologies mentioned, but most importantly, the specificity of tests selected in terms of the challenges of our projects has proved that the choice of technology is not arbitrary. When developing applications on the edge of performance or with limited resources, the conclusions of our research can and are crucial.

Our work allowed for better selection of technologies in terms of various requirements in our projects. Each project has different requirements, ranging from efficiency, speed and energy consumption. The presented study enables such a selection of technologies to optimize certain aspects of our projects based on microservices. These are projects ranging from small networks based on sensors used in telemedicine to large networks of sensors used in industry. The authors plan further research in this area in order to check other microservice platforms and their suitability for our research. The conducted research is in line with the research carried out by our team to research the efficiency of various pro-programming technologies and programming languages and to use them to optimize the systems we design. The results of these studies will certainly be useful and are now useful for other teams, for which the authors already had information.

The results of our research are applicable in our work on the projects mentioned in the introduction to this article, and also for future implementations. We believe that the results and conclusions obtained as a result of this study will be useful also for others, due to the universality of this comparison, which certainly fills a certain gap in information on this type of problem.

Author Contributions: Conceptualization, W.Z.; Data curation, P.P. and M.M.; Formal analysis, T.K.; Funding acquisition, W.Z.; Investigation, P.P.; Methodology, N.B. and W.Z.; Project administration, W.Z.; Software, P.P.; Supervision, W.Z.; Validation, W.Z.; Writing—original draft, W.Z.; Writing—review & editing, N.B., T.K., M.M. and W.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Which Java Microservice Framework Should You Choose in 2020? Exploring Micronaut and Quarkus vs. Spring Boot—How Good Are They? Available online: https://betterprogramming.pub/which-java-microservice-framework-should-you-choose-in-2020-4e306a478e58 (accessed on 3 September 2021).
2. Review of Microservices Frameworks: A Look at Spring Boot Alternatives. Available online: https://dzone.com/articles/not-only-spring-boot-a-review-of-alternatives (accessed on 3 September 2021).
3. Błaszczyk, M.; Pucek, M.; Kopniak, P. Comparison of lightweight frameworks for Java by analyzing proprietary web applications. *J. Comput. Sci. Inst.* **2021**, *19*, 159–164. [CrossRef]
4. Dhalla, H.K. A Performance Comparison of RESTful Applications Implemented in Spring Boot Java and MS.NET Core. *J. Phys. Conf. Ser.* **2021**, *1933*, 2–7. [CrossRef]
5. Massaga, A.; Kouamou, G.E. Towards a Framework for Evaluating Technologies for Implementing Microservices Architectures. *J. Softw. Eng. Appl.* **2021**, *14*, 442–453. [CrossRef]
6. Al-Debagy, O.; Martinek, P. A Comparative Review of Microservices and Monolithic Architectures. In Proceedings of the 18th IEEE International Symposium on Computational Intelligence and Informatics, CINTI 2018, Budapest, Hungary, 21–22 November 2018.
7. Grambow, M.; Wittern, E. Benchmarking the Performance of Microservice Applications. *ACM SIGAPP Appl. Comput. Rev.* **2020**, *20*, 20–34. [CrossRef]
8. Kubiak, D.; Zabierowski, W. A Comparative Analysis of the Performance of Implementing a Java Application Based on the Microservices Architecture, for Various AWS EC2 Instances. In Proceedings of the 2021 IEEE XVIIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH), Polyana, Ukraine, 12 May 2021.

9. Calderón-Gómez, H.; Mendoza-Pittí, L.; Vargas-Lombardo, M.; Gómez-Pulido, J.M.; Sención, D.R.G.; Polo-Luque, M. Evaluating Service-Oriented and Microservice Architecture Patterns to Deploy eHealth Applications in Cloud Computing Environment. *Appl. Sci.* **2021**, *11*, 4350. [CrossRef]
10. Pratama, Y.P.; Basuki, D.K.; Sukaridhoto, S.; Yusuf, A.A.; Yulianus, H.; Faruq, F.; Putra, F.B. Designing of a Smart Collar for Dairy Cow Behavior Monitoring with Application Monitoring in Microservices and Internet of Things-Based Systems 2019 International Electronics Symposium (IES). In Proceedings of the 2019 International Electronics Symposium (IES), Surabaya, Indonesia, 27–28 September 2019.
11. Li, P.; Jiang, P.; Liu, J. Mini-MES: A Microservices-Based Apps System for Data Interconnecting and Production Controlling in Decentralized Manufacturing. *Appl. Sci.* **2019**, *9*, 3675. [CrossRef]
12. Tapia, F.; Mora, M.Á.; Fuertes, W.; Aules, H.; Flores, E.; Toulkeridis, T. From Monolithic Systems to Microservices: A Comparative Study of Performance. *Appl. Sci.* **2020**, *10*, 5797. [CrossRef]
13. Herrera-Quintero, L.F.; Vega-Alfonso, J.C.; Banse, K.B.A.; Zambrano, E.C. Smart ITS Sensor for the Transport ation Planning Based on IoT Approaches Using Serverless and Microservices Architecture. *IEEE Intell. Transp. Syst. Mag.* **2018**, *10*, 17–27. [CrossRef]
14. Khazaei, H.; Barna, C.; Beigi-Mohammadi, N.; Litoiu, M. Efficiency Analysis of Provisioning Microservices. In Proceedings of the 2016 IEEE 8th International Conference on Cloud Computing Technology and Science, Luxembourg, 12–15 December 2016.
15. Mostof, V.M.; Krishnamurthy, D.; Arlitt, M. Fast and Efficient Performance Tuning of Microservices. In Proceedings of the 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), Chicago, IL, USA, 5–10 September 2021.
16. Panduman, Y.Y.F.; Albaab, M.R.U.; Besari, A.R.A.; Sukaridhoto, h.S.; Tjahjono, A. Implementation of Microservice Architectures on SEMAR Extension For Air Quality Monitoring. In Proceedings of the 2018 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC), Bali, Indonesia, 29–30 October 2018.
17. Samardzic, M.; Sajina, R.; Tankovic, N. Microservice Performance Degradation Correlation. In Proceedings of the MIPRO 2020, Opatija, Croatia, 28 September–2 October 2020.
18. Kumar, P.K.; Agarwal, R.; Shivaprasad, R.; Sitaram, D.; Kalambur, S. Performance Characterization of Communication Protocols in Microservice Applications. In Proceedings of the 2021 International Conference on Smart Applications, Communications and Networking (SmartNets), Glasgow, UK, 22–24 September 2021; IEEE: Glasgow, UK, 2021.
19. Khazaei, H.; Mahmoudi, N.; Barna, C.; Litoiu, M. Performance Modeling of Microservice Platforms. *IEEE Trans. Cloud Comput.* **2020**, 1. [CrossRef]
20. Miyagoshi, K.; Teranishi, Y.; Kawakami, T.; Yoshihisa, T.; Shimojo, S. Proposal of a Logical Sensor Architecture using WoT-based Edge Microservices. In Proceedings of the 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid, Spain, 13–17 July 2020.
21. Yilmaz, R.; Buzluca, F. A Fuzzy Quality Model to Measure the Maintainability of Microservice Architectures. In Proceedings of the 2021 2nd International Informatics and Software Engineering Conference (IISEC), Ankara, Turkey, 16–17 December 2021.
22. Liu, X.; Jiang, S.; Zhao, X.; Jin, Y. A Shortest-Response-Time Assured Microservices Selection Framework. In Proceedings of the 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), Guangzhou, China, 12–15 December 2017.
23. Haselbock, S.; Weinreich, R.; Buchgeher, G. An Expert Interview Study on Areas of Microservice Design. In Proceedings of the 2018 IEEE 11th International Conference on Service-Oriented Computing and Applications, Paris, France, 20–22 November 2018.
24. Bilgin, B.; Ünlü, H.; Demirörs, O. Analysis and Design of Microservices: Results from Turkey. In Proceedings of the 2020 Turkish National Software Engineering Symposium (UYMS), Istanbul, Turkey, 9 November 2020.
25. Munaf, R.M.; Ahmed, J.; Khakwani, F.; Rana, T. Microservices Architecture: Challenges and Proposed Conceptual Design. In Proceedings of the 2019 International Conference on Communication Technologies (ComTech 2019), Rawalpindi, Pakistan, 20–21 March 2019.
26. Sill, A. The Design and Architecture of Microservices. *IEEE Cloud Comput.* **2016**, *3*, 76–80. [CrossRef]
27. Al-Doghman, F.; Moustafa, N.; Khalil, I.; Tari, Z.; Zomay, A.Y. AI-enabled Secure Microservices in Edge Computing: Opportunities and Challenges. *IEEE Trans. Serv. Comput.* **2022**. [CrossRef]
28. Kazlauskas, M.; Navakauskas, D. Case Study of a Multisensor Patient Network and Microservices Managed by Fog Computing. In Proceedings of the 2021 IEEE 9th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), Riga, Latvi, 14 January 2022.
29. Bolesta, I.; Kushnir, O.; Bavdys, M.; Khvyshchun, I.; Demchuk, A. Computational-Measurement System "Nanoplasmonics". Part 2: Structure of Microservices. In Proceedings of the 2019 XIth International Scientific and Practical Conference on Electronics and Information Technologies (ELIT), Lviv, Ukraine, 16–18 September 2019.
30. Chen, C.; Cai, J.; Ren, N.; Cheng, X. Design and Implementation of Multi-tenant Vehicle Monitoring Architecture Based on Microservices and Spark Streaming. In Proceedings of the 2020 International Conference on Communications, Information System and Computer Engineering (CISCE), Kuala Lumpur, Malaysia, 3–5 July 2020.
31. Al-Debagy, O.; Martinek, P. Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach. In Proceedings of the IEEE 15th International Conference of System of Systems Engineering, SoSE 2020, Budapest, Hungary, 2–4 June 2020.
32. Mazlami, G.; Cito, J.; Leitner, P. Extraction of Microservices from Monolithic Software Architectures. In Proceedings of the 2017 IEEE 24th International Conference on Web Services, Honolulu, HI, USA, 25–30 June 2017.

33. Kyryk, M.; Tymchenko, O.; Pleskanka, N.; Pleskanka, M. Methods and process of service migration from monolithic architecture to microservices. In Proceedings of the 2022 IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET), Lviv-Slavske, Ukraine, 22–26 February 2022.
34. De Lauretis, L. From Monolithic Architecture to Microservices Architecture. In Proceedings of the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Berlin, Germany, 27–30 October 2019.
35. Chen, R.; Li, S.; Li, Z. From Monolith to Microservices: A Dataflow-Driven Approach. In Proceedings of the 2017 24th Asia-Pacific Software Engineering Conference, Nanjing, China, 4–8 December 2017.
36. Ponce, F.; Márquez, G.; Astudillo, H. Migrating from monolithic architecture to microservices: A Rapid Review. In Proceedings of the 2019 38th International Conference of the Chilean Computer Science Society (SCCC), Concepcion, Chile, 4–9 November 2019.
37. Filippone, G.; Autili, M.; Rossi, F.; Tivoli, M. Migration of Monoliths through the Synthesis of Microservices using Combinatorial Optimization. In Proceedings of the 2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Wuhan, China, 25–28 October 2021.
38. Wan, F.; Wu, X.; Zhang, Q. Chain-Oriented Load Balancing in Microservice System. In Proceedings of the 2020 World Conference on Computing and Communication Technologies, Warsaw, Poland, 13–15 May 2020.
39. Nisansala, S.; Chandrasiri, G.L.; Prasadika, S.; Jayasinghe, U. Microservice Based Edge Computing Architecture for Internet of Things. In Proceedings of the 2022 2nd International Conference on Advanced Research in Computing (ICARC), Belihuloya, Sri Lanka, 23–24 February 2022.
40. Cabrera, E.; Cardenas, P.; Cedillo, P.; Pesantez-Cabrera, P. Towards a Methodology for creating Internet of Things (IoT) Application based on Microservices. In Proceedings of the 2020 IEEE International Conference on Services Computing (SCC), Beijing, China, 7–11 November 2020.
41. Blinowski, G.; Ojdowska, A.; Przybyłek, A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access* **2022**, *10*, 20357–20374. [CrossRef]
42. Dinh-Tuan, H.; Katsarou, K.; Herbke, P. Optimizing microservices with hyperparameter optimization. In Proceedings of the 2021 17th International Conference on Mobility, Sensing and Networking (MSN), Exeter, UK, 13–15 December 2021.
43. Li, S. Understanding Quality Attributes in Microservice Architecture. In Proceedings of the 2017 24th Asia-Pacific Software Engineering Conference Workshops, Nanjing, China, 4–8 December 2017.
44. Gos, K.; Zabierowski, W. The Comparison of Microservice and Monolithic Architecture. In Proceedings of the 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH), Lviv, Ukraine, 22–26 April 2020.
45. Liu, C.-C.; Huang, C.-T.; Tseng, C.-W.; Yang, Y.-T.; Chou, L.-D. Service Resource Management in Edge Computing Based on Microservices. In Proceedings of the 2019 IEEE International Conference on Smart Internet of Things (SmartIoT), Tianjin, China, 9–11 August 2019.
46. Zhou, X.; Peng, X.; Xie, T.; Sun, J.; Xu, C.; Ji, C.; Zhao, W. Benchmarking Microservice Systems for Software Engineering Research. In Proceedings of the 2018 ACM/IEEE 40th International Conference on Software Engineering: Companion Proceedings, Gothenburg, Sweden, 30 August 2018.
47. Akbulut, A.; Perros, H.G. Performance Analysis of Microservice Design Patterns. *IEEE Internet Comput.* **2019**, *23*, 19–27. [CrossRef]
48. Ueda, T.; Nakaike, T.; Ohara, M. Workload Characterization for Microservices. In Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC), Providence, RI, USA, 25–27 September 2016.
49. Papapanagiotou, I.; Chella, V. NDBench: Benchmarking Microservices at Scale. *ResearchGate* **2018**, 1.
50. Villamizar, M.; Garcés, O.; Ochoa, L.; Castro, H.; Salamanca, L.; Verano, M.; Casallas, R.; Gil, S.; Valencia, C.; Zambrano, A.; et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *ResearchGate* **2017**, *11*, 233–247. [CrossRef]
51. Posta, C. Microservices for Java Developers. In *Microservices for Java Developers*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2016; pp. 6–7.
52. What is Java Spring Boot? Available online: https://www.ibm.com/cloud/learn/java-spring-boot (accessed on 3 September 2021).
53. Micronaut Tutorial: How to Build Microservices with This JVM-Based Framework. Available online: https://www.infoq.com/articles/micronaut-tutorial-microservices-jvm (accessed on 3 September 2021).
54. Implementing Microservicilities with Quarkus and MicroProfile. Available online: https://www.infoq.com/articles/microservicilities-quarkus (accessed on 3 September 2021).
55. Available online: https://jaxenter.com/java-trends-top-10-frameworks-2020-168867.html (accessed on 3 September 2021).