applied sciences

MDPI

Article

From Monolithic Systems to Microservices: A Comparative Study of Performance

Freddy Tapia [1,2,*], Miguel Ángel Mora [2], Walter Fuertes [1], Hernán Aules [1,3,*], Edwin Flores [1] and Theofilos Toulkeridis [1]

1 Department of Computer Sciences, Universidad de las Fuerzas Armadas ESPE, Av. General Rumiñahui S/N, P.O. Box 17-15-231B, Sangolquí 171103, Ecuador; wmfuertes@espe.edu.ec (W.F.); egflores5@espe.edu.ec (E.F.); ttoulkeridis@espe.edu.ec (T.T.)

2 Department of Informatics Engineering, Universidad Autónoma de Madrid, Ciudad Universitaria de Cantoblanco, 28049 Madrid, Spain; miguel.mora@uam.es

3 Department of Mathematics, Universidad Central de Ecuador, Av. Universitaria, Quito 170129, Ecuador

* Correspondence: fmtapia@espe.edu.ec or freddy.tapia@estudiante.uam.es (F.T.); hmaules@uce.edu.ec (H.A.); Tel.: +593-998926648 (F.T.)

Received: 16 July 2020; Accepted: 18 August 2020; Published: 21 August 2020

Abstract: Currently, organizations face the need to create scalable applications in an agile way that impacts new forms of production and business organization. The traditional monolithic architecture no longer meets the needs of scalability and rapid development. The efficiency and optimization of human and technological resources prevail; this is why companies must adopt new technologies and business strategies. However, the implementation of microservices still encounters several challenges, such as the consumption of time and computational resources, scalability, orchestration, organization problems, and several further technical complications. Although there are procedures that facilitate the migration from a monolithic architecture to micro-services, none of them accurately quantifies performance differences. The current study aims primarily to analyze some related work that evaluated both architectures. Furthermore, we assess the performance and relationship between different variables of an application that runs in a monolithic structure compared to one of the micro-services. With this, the state-of-the-art review was initially conducted, which confirms the interest of the industry. Subsequently, two different scenarios were evaluated: the first one comprises a web application based on a monolithic architecture that operates on a virtual server with KVM, and the second one demonstrates the same web application based on a microservice architecture, but it runs in containers. Both situations were exposed to stress tests of similar characteristics and with the same hardware resources. For their validation, we applied the non-parametric regression mathematical model to explain the dependency relationship between the performance variables. The results provided a quantitative technical interpretation with precision and reliability, which can be applied to similar issues.

Keywords: architecture; containers; cloud; mathematical model; metrics; microservices; monolithic; performance

1. Introduction

Organizations require robust solutions based on technology. Therefore, software developers have designed and implemented different types of architectures overtime that help software products become resource-efficient and composed of functional requirements. Some architectures deploy their modules, either in one layer or distributed in different layers or tiers. Monolithic architecture, traditionally used in software production jointly with virtual machines, has been a successful

and effective formula for small and large-scale projects since the appearance of software systems. It is known that monolithic applications' performance is affected when the amount of data to be processed increases or exceeds a certain level of capacity. There have emerged diverse solutions through time, e.g., migration to new technologies, management of independent services, and more powerful servers. In the long term, a weak solution choice (i.e., defects in the software, configuration inconsistencies on modularity, cohesion, and coupling, and lack of security levels) could generate higher resource expenses. In the last two decades, innovation has given rise to new architectures that propose optimal solutions. In this sense, the microservices architecture is gaining space and will be part of the decision-making process in technical, financial, and marketing areas [1]. Microservices are replacing monolithic architecture systems, narrowly focused on the distributed system while having an isolated service [2].

One of the concerns when applying microservices is related to the efforts needed to implement and scale every microservice to the Cloud. Additionally, companies that deploy microservices may use different automation tools, such as DevOps, Docker, Chef, Puppet, and automatic scaling. The implementation of such tools saves time and resources. Unfortunately, it requires further development, migration, and integration. Therefore, to gain agility, independent growth, and scalability, the infrastructure costs are the main concern for companies that adopt the mentioned patterns [3]. Another issue is the orchestration of microservices in production [4]. Moreover, technical challenges in which microservices provide a solution lead to organizational challenges. Even if the organization may solve most or all of the appearing technical issues, the organization's structure and skills also need to be compatible with the new architecture [5]. Although various solutions already exist, the process of migrating from monolithic to microservices architectures still lacks accuracy measurement. Moreover, a brief literature review uncovered some differences in the performance of these architectures [6].

Based on the given issues, this study focuses primarily on a compilation of results from established research about the evaluation of both architectures. Secondly, it focuses on determining the performance and the relationship between different variables such as CPU, memory consumption, network performance, disk operations, and restrictions such as development time, integration, and migration effort of an application that proceeds from a monolithic structure towards one of the microservices. Indeed, this is complementary to [7] and an expansion of our previous work published in [8], where the data from those measurements will let us present a complete critical analysis, and new knowledge related to the evolution from Monolithic to Microservices architectures, including their performance, and development complexity. We also analyze a complete state-of-the-art to assume a critical position regarding what remains to be performed in the given context.

Furthermore, we evaluated two different scenarios, of which one comprises an application based on a monolithic architecture that works on a virtual server with KVM. The other scenario deploys the same application, based on a microservice architecture that runs in containers. Both cases were exposed to stress tests of similar characteristics and with the same hardware resources. For its validation, we applied the non-parametric regression mathematical model to explain the dependency relationship between the known performance variables. The results and the corresponding comparisons allowed a decision that directly focused on efficient resources management and software operation efficiency. Among the main contributions of the current study, we can consider: (1) The construction of a critical analysis of research production about the comparison of performance between monolithic versus microservices architecture. (2) A critical review of variables and constraints that impact the performance of architectures migration and a critical analysis of the evolution from monolithic to microservices architectures in terms of their applications, efficiency, and development complexity. (3) An evaluation of response times and resource consumption of microservices architecture; this would justify adopting such architecture based on its modularity as a new business technology or in applications of new technological trends. (4) A quantitative comparison of monolithic versus microservices architectures through the mathematical non-parametric

regression model that explains the dependency relationship between the selected variables, this will be required to decide when to use such architecture.

The remainder of this article is organized as follows. Section 2 discusses the state-of-the-art, while Section 3 presents the theoretical framework. Section 4 describes the experimental design, including an explanation of the mathematical model applied to validate the preliminary results. Section 5 details the evaluation process of both architectures as well as their findings and includes a discussion. Finally, Section 6 presents the conclusions and future work lines.

2. State of the Art

This section presents state-of-the-art defined as the contemporary research of the existing research in the field of interest. The current study has focused on reviewing the current situation of knowledge and research products to take the amount supply of it while creating new training and research scenarios that cover such prevailing problematic domain. Hence, several of them, given their familiarity, have been detailed described further below:

The study proposed by Stubbs et al. [2] explores container technology and the challenge of service discovery in microservices architectures. The authors introduce Serfnode, a decentralized open-source solution to the service detection problem, based on the Serf project. Serfnode is a non-intrusive Docker image that constructs one or more arbitrary Docker containers. With this, they described the building of a file systems synchronization solution between Docker containers using Git. Given the conclusions of this study contributed to our research as it allowed us to identify Serfnode that unites Docker containers with another group of existing clusters without affecting the original container's integrity. Likewise, Serfnode allowed monitoring and supervision mechanisms, which complemented the containers perfectly since they enable complete isolation and independence with the software running in each of the shared environments. While containers can simplify the deployment and distribution of individual components, they do little to address the concern of inter-service communication over a complex network. In conclusion, this study explores other options that allow us to achieve the most out of Microservices and containers.

Villamizar et al. [3] presented a cost comparison of a Web application developed and implemented using the identical scalable scenarios with three different approaches concerning the performance tests, the cost comparison, and the response time. The first evaluation included the testing and comparing the performance and infrastructure costs of the three architectures, defining three different business scenarios. Finally, the third evaluation, comprised to identify the performance of how each architecture affects the response time to requests during peak periods, as they determined the Average Response Time (ART) during performance tests. The test results indicated that the microservices have been able to help to reduce the costs of the infrastructure when compared to standard monolithic architectures. This given study contributed to ours as it encouraged us to consider different comparison scenarios and to establish evaluation metrics or indicators.

Al-Debagy and Martinek [6] presented a comparison between microservices and monolithic architectures in terms of performance. Two test scenarios were established to compare such performance. Another test scenario was created to examine the effect of different technologies on the performance of the microservices application. The authors concluded that microservices and monolithic applications might be able to have a similar performance within an average load in use. In the case of a lower load, with an amount of fewer than 100 users, the monolithic application may work slightly better than the microservices app. Therefore, the monolithic application is recommended for small applications as used only by a minor amount of users. In the second test scenario, the results were different in terms of performance, as the monolithic application indicated an improved performance on average. Therefore, the app can handle requests at a higher speed, so that the monolithic application may be used when the developer especially points to the application to process requests faster. A further test scenario included a comparison between two microservices applications with different service discovery technologies such as Eureka and Consul. This test

results indicated that the Consul application used as service discovery technology might be preferred compared to the microservices with Eureka service discovery technology. Although this study is similar to ours, the unique difference to our research is that we use containers, while they lack it.

Guaman et al. [9] developed a performance evaluation in the process of migrating a monolithic application to microservices. Hereby, the models such as NGINX and IBM were analyzed, which allowed the migration of Monolithic to Microservices. The services were implemented using RESTful web services to finally deploy microservices using technologies such as Spring Boot, Eureka, and Zuul. To obtain the performance-related metrics and analyze the advantages and disadvantages of each migration phase, they used as a tool the Apache JMeter. Each version of the application's performance was evaluated and compared with the performance of the metrics. This allowed them to determine which microservices architecture exhibits a better execution in terms of a performance quality attribute. The response time and error rate of unprocessed requests were faster compared to other applications involved in migration. This study is similar to ours. However, the significant variance with the current study is that the authors lacked to use containers. However, they agreed that before the use of the microservices approach, a model is required for the successful migration and, thus, evaluates performance and scalability. In the migration process, the study application was modified at the code and design level, including patterns such as Singleton, Facade, Choke, Single Service per Host, Service Discovery, and API Gateway. It has used to evaluate performance as an attributed quality in each phase of the migration process. This system was configured to generate results regarding the use of resources such as CPU, memory, network, and access to databases.

Akbulut and Perros [10] obtained performance results related to query response time, efficient hardware usage, hosting costs, and packet loss rate for three microservices. Complex architectures were associated with long-term development cycles and additional license expenses for third-party applications. Also, the employment of more qualified developers and test personnel in the team has been another factor that increased the total cost. However, such architectures increase productivity and lower prices as they are energy efficient in the long term. In general, there is no single microservices pattern that is better or more efficient than the others. Instead, each design pattern works best in different settings. They concluded that Microservices are still undeveloped, and therefore, best practices of their use are critical to their successful adaptation and incorporation in the future of Service-oriented architecture (SOA). This work contributed to ours due to their case studies' experience to obtain the results using design patterns. They used Node JS and Python to implement microservices, MongoDB for the data platform, and Docker Compose for the container environment. This study emphasizes that an entire ecosystem needs to be orchestrated with Kubernetes, Mesosphere, or Docker Swarm, together with monitoring at the System infrastructure level.

Singh and Peddoju [11] explained a comparison between deployments of microservice versus the implementation of a monolithic design. These authors deployed the proposed microservices on the Docker containers and tested them using a social networking application as a case study. Due to the comparison of the performance, they installed and used Jmeter8 to submit continuous requests to both designs. For the monolithic design, they used a web API to transmit the request to the application. In contrast, for a microservice design, they used HAProxy to send requests to the target service. The results indicated that the application developed using the microservice approach and deployed using the proposed design reduces the time and the effort for deployment and continuous integration of the app. Their results also confirmed that microservice-based implementation outperforms the monolithic model due to its low response time and high throughput. Our experimental results indicate that Containers are adequate launchers for microservice-based applications compared to virtual machines (VMs).

There are several proposed studies concerning the discussion of the advantages and disadvantages of migration from a monolithic architecture to one of the microservices. Ponce et al. [12], analyzed a rapid review of twenty migration techniques adapted from published studies about the migration from a monolithic architecture to microservices. Taibi et al. [13] investigated the technical debts

of migration from an inherited monolithic system to microservices. They were concluding that the movement to microservices allowed to reduce the technical debt in the long run. Lastly, Mazlami et al. [14] presented a formal microservice extraction model that allowed an algorithmic recommendation of candidates in a refactoring and migration scenario. The model is implemented in a web-based prototype.

In the same context, the study of Kalske et al. [4] has been about the evolution challenges when moving from monolith to microservice architecture. Hereby, the objective was to identify the reasons why companies decided to perform such a transition and identified the challenges such companies may face during this transition. The conclusion was that when a software company grows big enough and starts facing problems regarding the size of the codebase, which is the moment when microservices may be an excellent form to handle the complexity and size. Even though the transition provides challenges, they may be easier to solve than the obstacles that monolithic architecture presents to the companies. Bures et al. [15] proposed the identification of business applications transactional contexts for the design of microservices. The emphasis has been to drive the aggregation of domain entities by the transactional settings where they were executed. Fritzsch et al. [16] discussed the notion of architectural refactoring and subsequently compared ten existing refactoring approaches, which were recently proposed in the academic literature. The candidates for this review were obtained from only three academic search engines. Furthermore, the selected refactoring techniques have been investigated only theoretically.

Within the same circumstances, Kecskemeti et al. [17] developed a methodology to divide monolithic services into several microservices. The created microservices through the proposed method were derived from a common ancestor (previously available as a monolithic service). Kamimura et al. [18] presented a case study on a small application and a further case study on a practical industrial use with preliminary evaluation by developers. The method supported the extraction of candidates for microservices by analyzing the source code applicable to large inherited codes. Bucchiarone et al. [19] presented an experience report of a case study from the banking domain to demonstrate how scalability looks favorable when re-implementing a monolithic architecture in microservices. Djogic et al. [20] presented the architectural redesign of the SOA integration platform (service-oriented architecture) following the principles of microservice design. The microservices approach is the decomposition of software components and software development organizations' ability to consolidate themselves in small independent teams that may select different programming languages and technologies for operation and work independently. Acevedo et al. [21] implemented a development methodology of which the purpose has been to propose procedures that modify a monolithic system to an architecture based on microservices. The manuscript describes each of its phases briefly and explains its execution in an open-source monolithic application until the transformation to microservices is completed. The results yielded that microservices reduces software maintenance times, facilitates application scalability, and simplifies maintenance, while task automation reduces implementation times, allowing quality assurance.

Regarding the transition from monolithic to microservices, Sarkar et al. [22], considered a complex industrial application that runs on Windows and needs to be migrated to a microservices-based container architecture. The authors analyzed the architectural features of the app and examined its availability for migration. Escobar et al. [23] presented an approach to modernize legacy applications in microservices. The authors proposed a model-centered process to analyze and visualize the current structure and dependencies between business capacity and data capacity. The results indicated that the EJB data diagram improved the application's understanding and helped architects analyze the relationship between EJBs. Debroy and Miller [24] discussed architecture for the next generation of applications in Varidesk. The results reported details of how they addressed the challenges that arose due to the architectural change in microservices. They also presented a novel idea of how the infrastructure itself that supports continuous integration and implementation channels for the various applications may also have its own. Chen and Col. [25] proposed a top-down analysis

approach and developed a decomposition algorithm based on data flow. The comparison and evaluation proved that its identification mechanism based on the flow of data could offer candidates more rational, objective, understandable, and consistent microservices through a more rigorous and practical implementation procedure.

Finally, Sarita and Sunil in [26] presented an approach on how to transform an existing monolithic application into a microservices architecture. They also outlined a proposed strategy for building and shipping microservices using Docker. The proposed study presented an approach to transforming an existing monolith system into microservices architecture using Dockers in the cloud as a strategy. The focus of their article differs almost entirely from the current study. In this case, a proposal is formulated for a general procedure that may be used to transform a monolithic application to Microservices. The conclusions lack any contribution, meaning that there is nothing quantifiable to determine whether there were real advantages or disadvantages. The procedure that they exposed is literary. However, an interesting point was the inclusion of agile methodologies as a suggestion for its development. The architecture they proposed is identical to that presented on all official sites. However, our proposal is different as it demonstrates the interaction of a practical and real case (collaborative systems); this is due to the implementation of a management system that allows adaptation and obtaining a substantial benefit from Microservices.

Comparing such efforts with those obtained by the current study, we developed a comparative analysis of the performance of hardware resources and applications between monolithic and microservices architectures. We considered development and production environments. Besides, we evaluated a web application based on a monolithic architecture that runs on a virtual server with KVM. The second scenario documents the identical web application, based on the microservices architecture and running on Containers. The results indicate high productivity, cost reduction, efficiency in the use of hardware resources, and applications that use microservices in containers. Another important fact is that we determine the results during the execution of services in both architectures and contrast their accuracy using a mathematical model. To the best of our knowledge, this has not been addressed in previous studies.

3. Theoretical Framework

In this section, we briefly introduce some features of the monolithic and microservices architectures. Figure 1 is a holistic visual representation of the elements involved in our entire research process. This figure includes the characteristics of the architectures, performance metrics, assessments and comparisons, use of the mathematical model, and the relationships between them. Subsequently, we reviewed variables and constraints that impact the migration from a monolithic system to microservices. At last, we present a critical review of monolithic architectures' evolution to microservices in terms of their applications, efficiency, and development complexity.

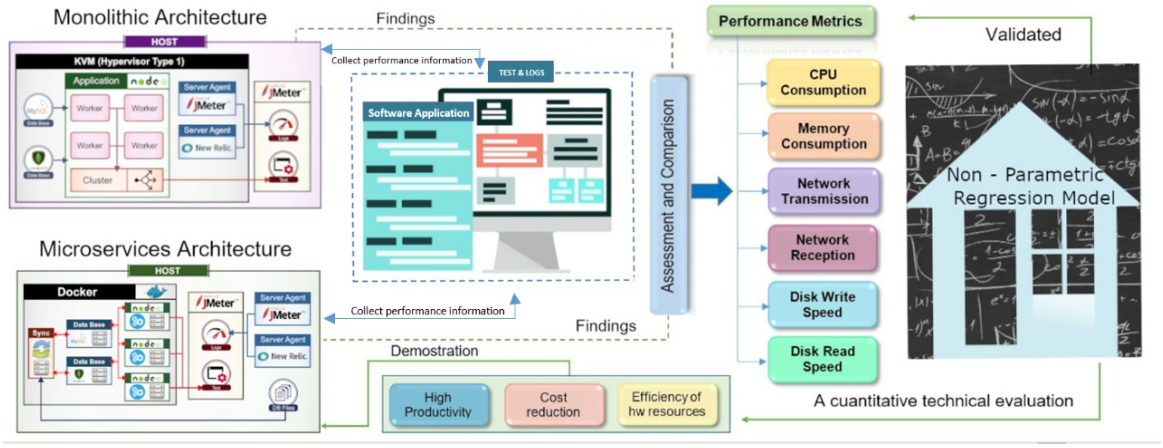

Figure 1. This is a visual representation of the elements involved in our whole research processes.

3.1. Monolithic Architecture

In general, the monolithic architecture uses a single development technology, limiting the availability of a suitable tool for each task that the system needs to execute. According to Nielsen [27], in a single logical executable, any change completed in a section of this kind of system involves the construction and deployment of a new version of the entire system. Most involved aspects or layers of presentation, processing, and storage are a single component of software required to run on a server (See Figure 2).

In concordance with the State of the Art section of our study, the monolithic architecture offers some advantages and disadvantages that give rise to confidence or technological drawbacks. It is necessary to know the risks that can lead to short or long-term problems [28]. Among the most significant advantages, there are stable systems, the full manage system, used by large companies in the world such as IBM, Sun Microsystems, and BMC. The professional services of these companies have a high level of knowledge about their products. Among some disadvantages: they are rigid systems and are difficult to adapt to new needs. The increase in its processing capacity goes through changing the current server for a larger one. Its technology is proprietary, which creates a significant dependency on the client towards the supplier company. Its acquisition, renewal, and support costs are high. We propose some examples of restriction of these kind systems in Section 3.5.

Every computer system is required to evolve and to follow trends such as solid-state disks surpassed punched cards. The level of complexity of modern systems requires improvements in both software production and performance. Thus, this means that it has found inevitable defects in the monolithic architecture [29]. Which, over time, will contrariwise to itself, giving rise to modern architectures such as microservices. There are cases in which applications are more efficient using this type of construction, e.g., firmware, key Logger, and viruses.

3.2. Microservices Architecture

As stated by Martin Fowler [30], "A microservice architectural style is an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery". This modern architecture enables creating large, complex, and scalable applications, comprised of small, independent, and highly uncoupling processes, communicating each one using APIs [31].

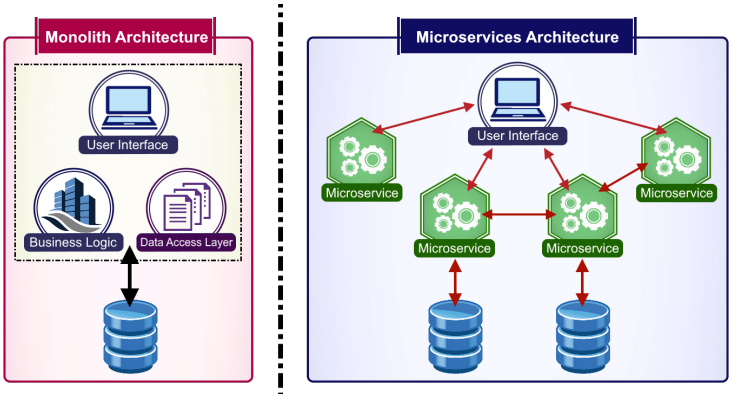

Figure 2. It is a visual representation of a monolithic (**left**) versus Microservices architecture (**right**).

Microservices work as an application composed of small and individual services running in their processes and communicating with light mechanisms. The independence of each one allows it to be tolerant of failures and increases its availability. It is a new culture of automation, with decentralized processes that concede independent deployments, where the whole is modeled around the business domain [5,32,33]. The structure of an application based on microservices differs significantly from

a monolithic architecture application (See Figure 2). Also, it can encapsulate complex and complete client business scenarios. It is unnecessary to update the entire system, except the component related to a microservice or specific container.

3.3. Software Tools for Research and Performance Measurements

In this study, we performed the monolithic application on a Kernel-based Virtual Machine (KVM) solution that implements native virtualization with Linux distributions. The other case is the application with Microservices Architecture orchestrated with Docker. Its purpose is to create lightweight and portable containers that can be executed independently and autonomous of the Operating System. Nevertheless, we mainly used open-source software tools to implement and configure virtual machines as an experimentation platform and for the implementation of the application (See Figure 3). Furthermore, they were used for stress tests, data collection, and performance measures. The only exception was NewRelic, where a trial version was applied. They will be described briefly further below, in the way as they contributed and supported within this study:

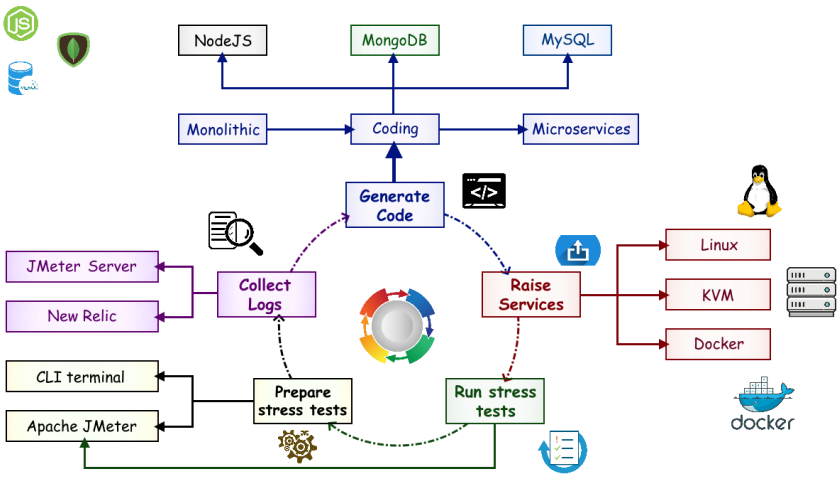

Figure 3. Software tools for research and performance measurements.

KVM The Kernel-based Virtual Machine is a virtualization technology for Linux systems that turn it into a hypervisor [34]. The hypervisor provides support for multiple operating systems and enables the virtualization of hardware [35]. Therefore, it has the advantage of being able to use the functionalities, solutions, and advances of the base operating system, without the need for additional engineering processes. A KVM needs specific system-level components for its operation, such as memory manager, process scheduler, input/output stack, device drivers, and security management. Each virtual machine runs as a Linux process and adheres to virtual hardware such as network cards, CPU, memory, disks, and graphics handler. Its allied application is QUEMU, which is also known as an operating system emulator [36]. A KVM supports hot migration, i.e., it allows physical servers or even complete data centers to perform maintenance tasks without interrupting the guest operating system [34]. For ease of use of the KVM, we used a graphical interface virtual machine manager application in the current study.

QEMU is an open-source, a generic virtualizer, emulator machine that allows the creation of the KVM with the Ubuntu Server. By default, this application is used through commands to generate virtual machines. Therefore it is necessary to use an additional application called Virtual Machine Manager (VMM). VMM is a desktop user interface for managing virtual machines through libvirt.

Docker Client and Server is an open platform for developers and system administrators to build, transport, and run distributed applications, whether on laptops, virtual machine data centers, or the cloud. In this research, Docker is focused on managing the containers used to run the various services.

Docker Compose is used to define and run multi-container Docker applications. In our study, it is used to define the configurations of the Microservices environment.

Sublime is a sophisticated text editor for code, markup, and prose. This was used to edit the application code and configurations to be performed to adapt to the environments.

Git is a free version control system distributed as open-source and is used to keep the application (code, files, resources) updated in all environments, thus maintaining the integrity and ensuring that both processes, as well as results, are adequate.

Java8 (JRE) is used only by the Java runtime environment to raise the Agent Server and run JMeter.

NodeJS allows running JavaScript code in the back end, outside the browser, using Google's V8 virtual machine. In addition, Node.js has several useful modules. Therefore it is not recommended to write everything from scratch. Therefore, it was used for two purposes being an execution environment and a library.

NPM, the JavaScript package manager, was used to handle the used library dependencies to deploy the application.

MySQL is an open-source relational database management system. It is used in this application in order to store the publications generated in the stress tests.

MongoDB is a NoSQL database system, which is a document-oriented and open source for querying and indexing. In this project, we used it to store the information generated for the Threads and Users.

During the conduction of the tests, we used JMeter [37] and NewRelic [38] applications. Both support data collection in real time of the performance of the resources that need to be analyzed. Hereby, the tests focus on CPU and Memory consumption, hard disk drive reading and writing speed, network, and application performance. Following is a brief description of the mentioned tools:

Apache JMeter is a Java-based open-source software tool designed to load the test's functional behavior and measure performance. We used this tool to configure stress tests and collect information on the application's performance and response in real time. With JMeter, the scripts were generated for stress tests, allowing them to specify the request conditions in each environment. Also, Server Agent version 2.2 was included as a component of JMeter to collect performance information. The current project was used as a service that supports the collection and interpretation of data generated during stress tests.

Server Agent is a component of JMeter in order to collect performance information. Hereby, we used it as a service that helps to collect and interpret data generated during stress tests.

NewRelic provides a performance analysis service that allows viewing and analyzing amounts of data and obtaining useful information in real time. In the current study, it is used such as JMeter. However, in this case, it predominantly focuses on collecting information about the performance of the environment where the application runs during the stress tests and later analyzing them in real-time graphs.

3.4. Containers

Containers resemble virtual machines concerning the services they provide, except that they lack a penalty (overhead) produced by the execution of a separate kernel and the virtualization of the hardware components [39]. Containers enable each workload to hold exclusive access to resources such as processor, memory, service account, and libraries, which is essential for the development process [40]. They also run as groups of isolated processes within an operating system, allowing them to start fast and maintain. Each container is a package of libraries and dependencies necessary for its operation, resulting in being independent and allowing the end of isolated processes. One of the most critical risks in containers is the poor visibility of the processes which run inside it to the limit of

host resources [41]. Additionally, there are cybersecurity issues as containers lack to include a secure network communication system, which results in leaving them vulnerable to the computer (network) attacks [42]. For safety, group and permissions are handled with namespaces, which means that users do not have the same treatment inside and outside the container.

Docker is widely gaining ground in terms of container handling; this is how it has developed from being an ambiguous and unknown term Linux Containers (LXC) to become a container manager. Another similar solution is known as Kubernetes. Thus, this is a container orchestration platform that allows us to deploy, scale, and manage containers on a large scale [43]. This open-source technology is enhancing to be the most used tool by DevOps for large-scale container projects. It has even been suggested as a solution for more interoperable cloud applications packaging [44,45]. Initially, Docker used LxC as the default execution environment. However, continuous changes of distributions represented a problem when trying to standardize a generic version.

3.5. A Review of Variables and Constraints That Impact Performance in the Migration from a Monolithic System to Microservices

When a company is created, depending on different circumstances, its applications generally start being Monolithic; this is reasonable, as these systems initially work well within limited settings, requiring less of their equipment. However, as companies grow and evolve, so they also need their application architecture. As systems become large and complex, companies turn into Microservices as a long-term infrastructure solution.

Whitin this context, as an alternative to justify such migration, it is required to determine both architectures' performance. Therefore, according to [39,46], performance can be measured quantitatively, based on some variables such as CPU and memory consumption, network performance, and disk operations speed. **CPU consumption** refers to the percentage measured at which the central processing unit was used to process the system's instructions. **Memory consumption** is the percentage of memory that has been used in the execution of a process. **Network performance** is the measurement of data transfer, both in transmission and reception of data. **Disk performance** is used to yield the efficiency of sequential read and write operations to the disk.

However, to measure qualitatively, in the current study, some examples were considered to be Wix.com, Bes tbuy, and Cloud Elements [47,48]. **Wix.com** [49] is an Israelian software company providing cloud-based web development services). **Best Buy** [50] is an American multinational consumer electronics retailer). **Cloud Elements** [51] is a cloud API integration platform that enables developers to publish, integrate, aggregate, and manage all of their APIs through a unified platform. These companies had to migrate from a monolithic architecture to microservices.

Among its motivations for migration, Wix.com adopted microservices to reverse massive technical problems that had created instability. In 2010, the company started dividing parts down into smaller services to manage scalability better. Similarly, the Best Buy architecture became a bottleneck for deployments. Its downtime was far too long to keep business online. At Cloud Elements, the transition also arose out of necessity, as it was a company experiencing exponential growth. A microservices design would help iterate continuously to meet new demands [47,48].

Wix.com had to invent new patterns of integration, analysis, and fostering a new internal culture among the constraints or obstacles encountered along with the development. In Cloud Elements, its modern design generated debate and conflict. For Best Buy, one of the significant challenges was creating trust. Cultural resistance to the way software is developed and implemented is challenging changes the way people do their jobs. For these companies, almost no support tools existed when they set out on this path. The migration process was not easy. However, in a view towards the past, it has been fortunately a wise decision. However, migrating to a microservices design does not guarantee benefits for all companies. A monolithic codebase may be more efficient and with less overhead for smaller-scale applications. Although implemented, microservices can provide a backbone upon which innovation can thrive. However, the benefits of microservices must outweigh the time spent migrating

and maintaining new issues. Staying with a monolithic architecture is probably a much easier task than managing a distributed system, which allows us to deduce that a progressive migration should be generated [47,48].

3.6. A Critical Review of the Evolution from Monolithic to Microservices Architectures in Terms of Their Applications, Efficiency and Development Complexity

There exist some of the companies such as Walmart, Spotify, Netflix, Amazon, and eBay, among others [52–54], and many other large-scale websites and applications that have evolved from a monolithic architecture to microservices. The following describes some of the success stories and how their design scheme might be.

Walmart needs to handle some six million page views per minute, with currently approximately some four billion people connected and more than 25 million applications available.

Spotify, which built a microservice architecture with autonomous equipment, provides services to more than 75 million active users per month, with an average session duration of 23 min.

Netflix receives an average of one billion calls to its different services. It is capable of adapting to more than 800 types of devices through its video streaming API. To offer a more stable service, for each given request, it performs five requests to different servers to avoid the loss of the continuity of the transmission [55].

Amazon, which migrated three years ago to the microservice architecture, is one of the first of the large companies that implemented it in production. There is no approximation of the number of applications they can receive daily [56].

The company, eBay, is one of the corporations with the most optimal vision for the future, being a pioneer in the adoption of technologies such as Docker. Its main application includes several autonomous services, and each one is executing the logic of each functional area that is offered to customers.

For a critical analysis, the design of a microservice oriented application, according to Microsoft [57], is here described as more detailed. A crucial app that may meet the requirements of the analyzed success stories should be able to control requests by executing business logic, accessing databases, and returning HTML, JSON, or XML responses [58]. The application needs to support a variety of clients, including desktop browsers running single-page applications (SPA), traditional web applications, mobile web applications, and native mobile applications. It is also possible that the application exposes an API for third-party consumption. Furthermore, it should be able to integrate the external microservices or applications asynchronously. Thus, this approach allows the resistance of the microservices in case of partial errors. The application should also consist of a variety of types of components, such as:

- Presentation components are responsible for the control of the user interface and the consumption of remote services;
- Domain or business logic, which refers to the domain logic of the application;
- The access logic to databases is the data access components responsible for obtaining access to the databases (SQL or NoSQL); and.
- Application integration logic, which includes a messaging channel, based mainly on message agents [59]. The application will require scalability, allowing its vertical subsystems to scale horizontally autonomously as some subsystems will require greater scalability than others [60].

Considering, for example, a mobile shopping application with the following microservices: (a) A Shopping cart service that maintains the number of items. (b) A Shipping service that supports for handling shipments and cargo. (c) An Inventory service that allows an organization to react to customer demand rapidly. (d) A recommendation service, which provides recommendations to users. (e) An Order service, which maintains an order history. (f) A review service designed to identify potential service delivery improvements. (g) A Catalogue service that keeps product catalogs [61].

Moreover, also considering, e-commerce Web applications [62], which need to be developed independently with the freedom in the use of technologies. This composition requires programming language on the server-side and the framework on the client-side. Likewise, the authentication service should be used, which subsequently would allow security in access to microservices.

Finally, the orchestration of the containers is required, where the microservices would be deployed. This solution may work significantly better than a monolithic model concerning performance, availability, and response time indicators. Finally, considering rendering applications with mobile platforms such as ANDROID and IOS. On the server-side, there are non-functional components that support the microservices (i.e., Config, Eureka, Gateway, and Composition) and the functional elements. The last ones are authentication microservices, assists, notes, co-participation, surveys, notification, and reports developed with spring boot. Each one will have private access to its database. How could access security be added to services [63,64]?

4. Experimental Design

This section describes the research process developed to perform the comparative performance analysis among monolithic versus microservices architectures. It starts explaining the application architecture design used for the evaluation. The scenarios developed to fulfill the proofs of concepts, the performance testing environment, and the procedure followed to collect the corresponding data, the experiments, and preliminary results. Finally, it explains the mathematical model used and its employability. Figure 4 illustrates a workflow diagram of the research process conducted, methods, and evaluation.

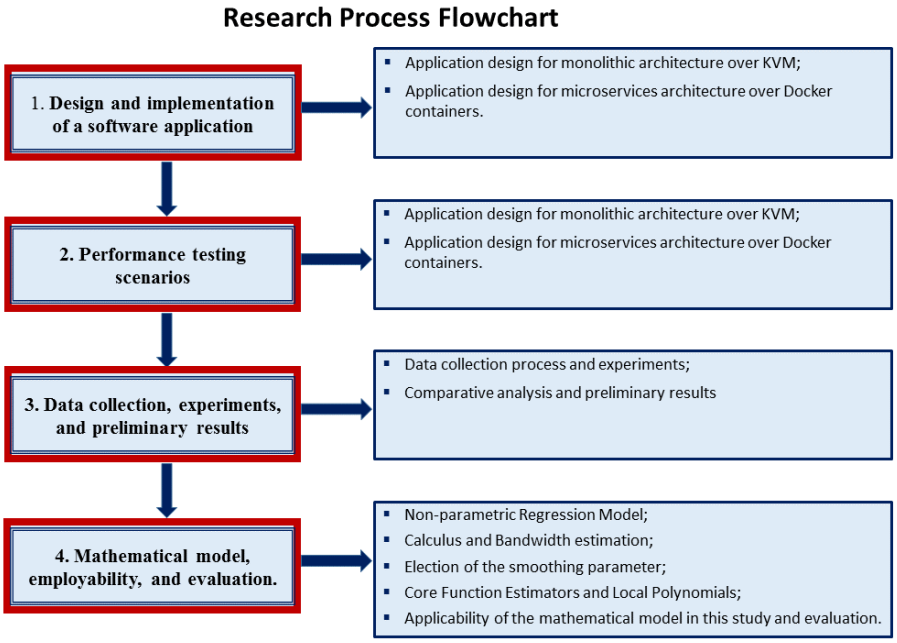

Figure 4. WorkFlow diagram of the process research conducted, methods, and evaluation.

4.1. Design and Implementation of a Software Application

This subsection presents an app's evolution based on a monolithic architecture and the result, which is the identical app using microservices. Besides, the configuration of the virtual environment is presented where the services were built, and the tests were run.

The application represents the basic design for handling forums, chats, comments, or notifications. A forum is an online site created to allow discussion, where users can have conversations in the form of posted messages [65]. Forums are a place for users where they discuss issues of various kinds. Forum users may interact by posting questions and subsequent answers and leave comments related to

a given topic. A thread is a grouping of posts configured as a conversation between users. A discussion forum has a hierarchical or tree-like structure. A forum contains multiple sub-forums, where each of them may have multiple topics. Within the forum topic, every new discussion started is called a thread and maybe answered by as many people as desired. In this way, three main elements are identified, being the user, thread, and post. The user is who generates the publication (posts) for each thread. The thread determines a conversation by storing as reference the identifier of both the user and the related publication. The number of both users and publications is unlimited for each conversation or thread. The post is the information generated by the user in a specific thread.

For both implementations, an elementary back-end-application was obtained, being capable of supporting stress loads that simulate situations that may happen in reality. A back-end application is a type of programming focused on functionality and services issues that run mostly on the server-side, which has no significant concern with the interface's aesthetics during a user's interaction [66]. In this case, we used Node-JS as a scripting Language.

4.1.1. Application Design for Monolithic Architecture over Kvm

The entire node.js application runs in a KVM. The following elements intervene in the general information of the architecture (presentation, application, and data layers depicted in Figure 5).

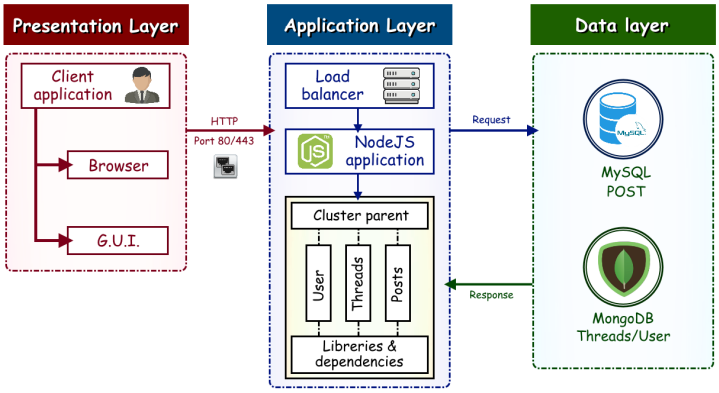

Figure 5. The monolithic application software architecture implemented for stress testing.

Client: The client requests the application through port 80 to the load balancer.

Load Balancer: serves as a single point of contact for customers. Clients send requests to the load balancer, and it distributes the requests to the client's overall available ports.

Target Groups: Each target group is used to route requests to one or more registered targets. Instances are registered in the application's target group.

Node.js: The central node.js cluster is responsible for distributing traffic to workers within the monolithic application.

Concerning implementation, by definition, the monolithic App appears with a simple directory structure and with a reduced number of files as there is no code segmentation. Figure 6) illustrates the hierarchical directory tree structure. The main files and the content of each one of the monolithic application are described below:

src/dumbdata.json: contains sample data in JSON format, and is used when generating content in the three services, being users, threads, and posts.

src/mongoApp.js: contains the source code to connect to the database in MongoDB, generating random content and stores it.

src/MySQL-database.sql: contains in code the SQL statements in order to create the Database and Table in MySQL, which are used in this experiment.

src/mysqlApp.js: contains the source code to connect to the database in MySQL, generating random content and stores it.

Index.js: contains the source code in order to initialize and start the services. package.json: describes the library dependencies required to run this application. In addition, this file indicates through a script the file that needs to be executed in order to initialize the services.

server.js: contains the source code that implements the endpoints for each service in order to generate the data and realize query calls.

To run this application, it is necessary to have installed a browser, Git, NodeJS, MongoDB, and MySQL. The steps to run the application are the following. Steps 1 to 5 are the same as those to be followed to lift services when KVM is available.

1. Install and configure NodeJS, MongoDB, and MySQL;
2. Clone the project from the repository on GitHub;
3. Use the CLI and go to the root folder of the Monolithic Application (consider /monolith as the root folder);
4. Manage dependencies, while it is needed to execute the command: npm install;
5. Raise the services;
6. Generate users, threads, or publications;
7. Select users, threads, or publications.

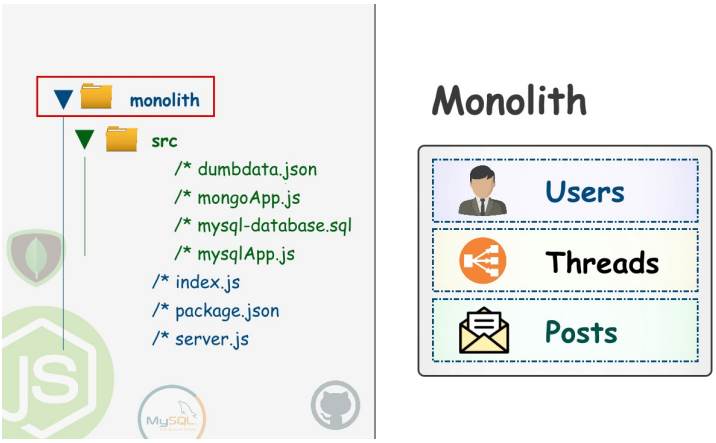

Figure 6. The hierarchical structure of directories of the Monolithic application implemented.

4.1.2. Application Design for Microservices Architecture over Docker Containers

For this implementation, the example published by Amazon Web Services (AWS) Github was taken as a foundation in its public repository. There are enough steps to migrate a monolithic application into microservices using Amazon Elastic Container Service, Docker, and Amazon EC2 [67]. To achieve such a procedure, we performed the necessary modifications and adaptations to fit the application for experimental purposes. When implementing only the App in its final version, without virtualization, a cyclical workflow based on five steps is followed: (1) Generate or modify code; (2) Raise services; (3) Prepare stress tests; (4) Run stress tests, and (5) Collect and Analyze Logs.

The application is structured with three layers: presentation, application, and data (see Figure 7). As established in step 3, segmenting the monolithic, and in step 4, implementing microservices of [67], the following elements intervene in the architecture's general information:

Client: The client makes requests for traffic through port 80.

Load Balancer: Directing external traffic to the correct service. Inspects the client's request and uses the routing rules to direct the request to an instance and port in the target group that matches the rule.

Target groups: Each service has a target group that tracks the instances and ports of each container running for the service.

Microservices: Each service is deployed in a container within an EC2 cluster. Each container manages a single feature

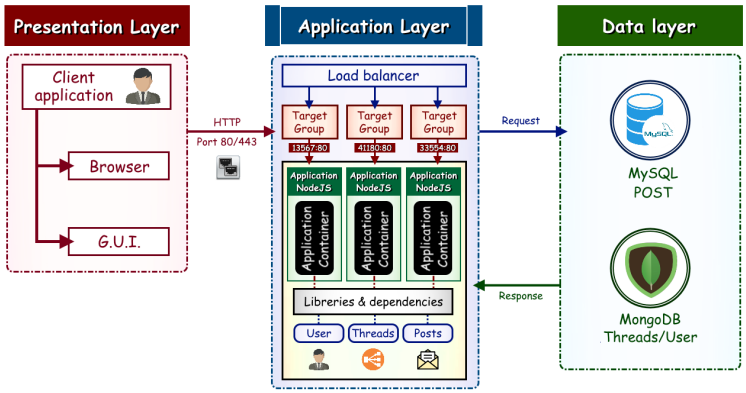

Figure 7. The microservices application software architecture implemented for stress testing. An adaptation from AWS in [67] .

In this version, the application has its files segmented in a directory structure that is more organized by functionalities. During this particular case, each indicated element (i.e., users, threads, and posts) of the application are considered a part that should be segmented and executed as an independent service. Within the given implementation, it should be noted that each feature of the Node.js application runs as a separate service in its container. Services can be scaled and updated independently of each other. The directory structure and file location changes are illustrated in Figure 8. The main files and details about the content of each one are described further below.

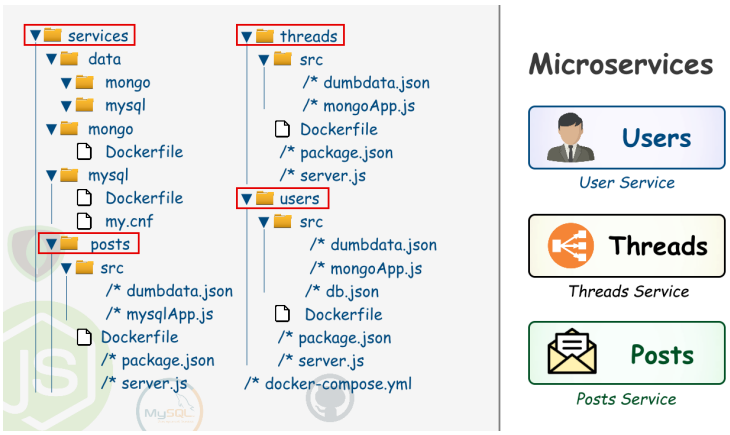

Figure 8. The hierarchical structure of directories of the Microservice application implemented.

services/[post, thread, user]/Dockerfile: For the case of the three services, being User, Thread, and Publication, the same Docker file configuration is used. Hereby, they use the same base image to instantiate their Containers. The file configures NodeJS and runs it as a server for each service.

services/[post, thread, user]/src/dumbdata.json: Contains sample data in JSON format, and is used when generating content in the three services users, threads, and posts.

services/docker-compose.yml: Contains the main configuration to build the three services individually, as well as the Database and volume services that will keep the settings and data synchronized.

services/data/mongoservices/data/mysql: Contains configurations and files of the corresponding databases. Docker synchronizes this directory with the existing one inside the container in order to keep the host's information and not in the container. Docker allows it to deploy volumes, and every time

it starts a Container, the state of the latest version of the Database will be reused. In this particular case, the configuration is specified in the docker-compose.yml file.

To execute this application, it is required to have installed a browser, Git, Docker, and Docker-Compose. The summarized steps to run the application are

1. Install Docker CE (Docker Community Edition, the free version of Docker) and Docker-Compose;
2. Clone the project from the repository on GitHub;
3. Use the CLI and enter the services/post folder. Here is the Dockerfile, which contains the script to generate our Docker image for its use in each service;
4. Generate the Docker image for the services;
5. Enter the root folder of the application with Microservices. This directory appears the docker-compose.yml file, which contains the main settings to start the application services;
6. Run the application and services;
7. Generate users, threads;
8. Select users, threads, or publications.

The following section describes the scenarios for the stress test.

4.2. Performance Testing Scenarios

The first version of the application is executed on the monolithic architecture. In this case, the entire structure of the project directory, and the layout of the files with the code, which covers all the functionality requirements, are grouped as a result of being a single component (see Figure 9). The second version is the application based on the microservice architecture; here, the directory structure and the code distribution in the files become different, giving rise to its provision for later coupling with the Docker containers (see Figure 10). This solution is adapted to virtual environments [68], where services and tests will be executed. As explained in [8], the monolithic application runs on a KVM, and the app with microservices runs in containers managed by Docker. In both cases, they have identical characteristics of hardware resources.

4.2.1. Scenario 1: Application on Monolithic Architecture over Kvm

Figure 9 illustrates the monolithic scenario, which is up to the present day a traditional version used in technological projects of different sizes and purposes. In such a scenario, KVM contains the necessary applications and libraries for the application to run, exposing two ports that allow calls from the host. The databases, both MySQL and MongoDB, start their services when the virtual platform is turned on. These databases respond to requests that are sent from workers. To take full advantage of the resources configured in KVM, we use a cluster managed by a NodeJS library. With this, a worker can be generated for each existing core. A worker is a process that runs on a logical core, providing the connection service to interact with the application. In this experiment, there are four logic cores. Therefore, there are four workers who not only process connection requests but also allow general stability to the application. Hence, availability is guaranteed if a worker fails due to an error in the processing of requests. Each worker has established a connection with the databases and the endpoints that support, specifying the service to which the request is sent from the client.

The cluster operates as a load balancer and assigns the request processing to the workers to finally send a response to the client. The port 3000, which is by default for NodeJS servers, is the same used to send requests from the host. In this case, using the JMeter Server Agent service, located inside the KVM, exposes port 4444. Then, it runs a daemon that receives instructions sent from JMeter (on the host) when running the stress tests in order to start the collection of data related to the performance of the resources running these tests. Furthermore, a NewRelic Server Agent, which, similar to the JMeter Server Agent, collects performance data with the difference that this data is general. Therefore, it collects data from the moment it starts as a service when the Operating System starts. The data collected is stored in log files on the host and synchronized in real time with its server

within the Cloud (NewRelic Server). It analyzes, interprets, and displays the results with interactive graphs, being very useful for quick analysis and understanding. This service allows us to observe the server's behavior in general and effectively see the performance of the resources when executing the stress tests. The purpose of using this service in coordination with JMeter was to corroborate the results and their analysis to demonstrate the feasibility of the present investigation. On the host side is JMeter, an application based on Java technology that allows editing of the scripts to create stress tests and collect the results when they run. Likewise, it allows the collection of logs with data from the processes executed and the resources' performance.

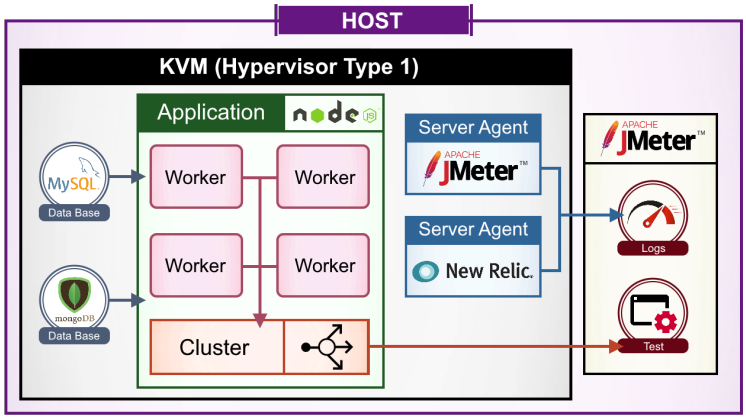

Figure 9. First Scenario: Monolithic application on a KVM. An adaptation from [8].

4.2.2. Scenario 2: Application with Microservices Architecture Using Containers

In this scenario, it is proposed to transform an application based on a functional monolithic architecture on a KVM into an application with a functional microservices architecture on Containers. Figure 10 represents the structure of the approach applied to the experiment. In this case, the Docker Engine performs Container orchestration. It also provides them with the necessary resources so that each container meets the required conditions and follows the guidelines that this technology requires.

Unlike the first scenario, the cluster and workers helping conduct most of the system resources are no longer used. In this scenario, Docker handles this issue by using the resources efficiently. As a result of this, ports are exposed for each new microservice. The databases also run inside containers, and each one exposes a port through which it can be accessed from the host. Besides, there is an extra container, namely Sync Volumes, to keep the data and files generated by the databases in the container synchronized with a directory on the host. This occurs to not lose the information generated once the container ends its execution.

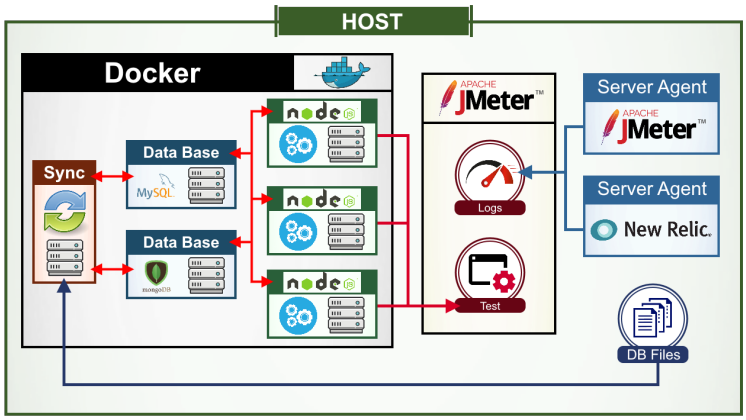

Figure 10. Second Scenario: Application with microservices on containers. An adaptation from [8].

For the interconnection and intercommunication between containers, it is defined in the docker-compose.yml file. As indicated in Figure 4, the containers are connected over a private network generated by Docker when the services start. In such a network, ports are exposed that lack access from an external Host, since Docker exposes ports for each container, so that they can be accessed from the host. Hence, there are three NodeJS applications, of which each one is independent and has its libraries and internal and external connection ports. The JMeter Server Agent service is located on the host since Docker is part of the primary system. It exposes its port 4444 to collect the resource performance information when stress tests run.

Nonetheless, it should be noted that this does not affect the results obtained in the conducted tests since JMeter focuses on the resources' behavior concerning the test script that is executed under its supervision. The NewRelic Server Agent service is also on the host side. In this case, the information sent to the NewRelic Server is affected since it contains global information, which includes processes outside the experiment that are executed and generate resource consumption. Hereby, a filter is used on the server-side of NewRelic to capture the resources' behavior at the exact moment when the stress tests run. This allows a better idea and supports a comparison to be performed concerning the results obtained within the first scenario.

In terms of resources, it is about maintaining the same characteristics in both scenarios so that the data generated in the tests are as accurate as possible.

4.3. Data Collection Process, Experiments, and Preliminary Results

4.3.1. Data Collection Process and Experiments

The stress tests were configured with the identical procedure for both scenarios. These are tests generated in JMeter using the graphical interface. Hereby, two different scenarios were configured. The first scenario generates GET requests via HTTP to the endpoints of each service to generate the databases and data. The number of data is variable and can be edited in the source code. The second scenario generates GET requests via HTTP to the endpoints of each service to select the information. In this case, it selects all the generated data that are stored in MySQL and MongoDB. Furthermore, these tests are intended solely to collect information on the performance of resources on the server-side. As a result, once a response is obtained from the server, this process will be terminated without considering the time it needs to process the information on the client-side or more commonly known as the browser. The reason for collecting behavior data on the server-side is that the Ubuntu Server is used in the first scenario, which by default, lacks a graphical interface. In the second scenario, we have Docker, which similarly lacks a graphical interface. For both scenarios, commands were executed only through a command-line interface (CLI), including the command that executes the stress tests. Nonetheless, we never used a GUI. Once the tests were accomplished, applications with a graphical interface were used to process, analyze, interpret, and generate reports.

For Case 1, the stress test was executed with 273 requests, 10,000 generated data, 30 repetitions, three threads, and three terminals, in order to collect enough information to perform a first interpretation and analysis. Table 1 lists the results obtained in logs on the requests executed from JMeter to the application. These results were obtained through the JMeter GUI use, which automatically processes the data stored in the log file and presents them to the user for a subsequent interpretation.

For case 2, the amount of data to be generated is doubled, and the script execution configuration is modified with the stress tests. The stress test runs with 1053 requests, 20,000 data generated, 70 repetitions, five threads, and three terminals. The configurations were given in a maximum limit specified by the researcher in the Script. This is based on the characteristics of the hardware and software used. Modifying with higher values generates errors of different types, such as memory overload (JMeter or NodeJS), request processing error, unexpected server shutdown due to process overload, among others. Furthermore, Table 1 indicates the results obtained in logs on the requests executed from JMeter to the application. These results are obtained through the JMeter GUI use, which

automatically processes the data stored in the log file, subsequently presenting them to the user for the corresponding interpretation.

Table 1. Results of the stress test of both scenarios.

	Case 1		Case 2	
	Scenary 1	**Scenary 2**	**Scenary 1**	**Scenary 2**
Total Requests			1053	1053
OK	273	273	1051	1053
Error	0	0	2	0
Duration time	00:01:50	00:01:29	00:14:17	00:12:08
Requests/s (average)	2.5/s	3.1/s	1.2/s	1.4/s
Duration per request				
Min	7 ms	4 ms	22 ms	8 ms
Max	3677 ms	2793 ms	35,784 ms	10,832 ms
Average	1150 ms	936 ms	3934 ms	3411 ms
Median	695 ms	312 ms	2548 ms	715 ms
Standard deviation	1094.66 ms	1082.4 ms	38.05 ms	4220.21 ms
Total data				
Received	85,014.43 KB/s	105,193.29 KB/s	86,342.29 KB/s	102,433.12 KB/s
Sent	0 KB/s	0 KB/s	0 KB/s	0 KB/s

4.3.2. Comparative Analysis and Preliminary Results

To perform the comparative analysis, it is essential to consider that the results on which the analysis is based are those generated with the JMeter tool. Additionally, the analysis is generated with NewRelic to corroborate and validate the given results. The application's performance consists of various tasks, such as analyzing the results of the stress test, focusing on the number of requests generated, and the time it takes to process them. If they finished successfully, it releases a response from the server.

In Case 1, a total of 273 of the requests generated have been processed successfully in both scenarios. In Case 2, there were 1053 requests. The application with microservices has successfully terminated all requests, while in Scenario 1 with the Monolithic application, two requests ended with an error. The tests' execution time demonstrates that with Monolithic, it delayed some 00:01:50 while in microservices, it lasted as low as 00:01:29, with some 00:00:21 seconds of difference in favor. Similarly, the number of requests processed per second improves using the Microservices Application. Case 1 performed with 2.5 s in Monolithic and 3.1 s in microservices (difference of 0.6 s), while in Case 2, we reached 1.2 s in Monolithic and 1.4 s in microservices (0.2 s in favor).

For the analysis of the CPU performance, memory, disk operations, and network performance, a mathematical model was chosen considering that the given information recorded in the log files related to each of the scenarios could not be sufficiently clear. Besides, while having high amounts of data, it was decided to apply this mathematical formula to express the relationships between the variables and their measurements to study both architectures' performance.

The following matrices are examples of the data initially collected and refined, corresponding to the data sets of the microservices variables performance (Table 2) and monolithic performance Table 3. The number of elements listed in the tables is barely the first ten when demonstrating the examples of the collected data. From these data, the application of the chosen mathematical model will be explained in the next section. This will determine the relationships between subsets of data and the following variables identified in [8] and described further below.

- CPU (%)
- Disc reading speed (MB/s)
- Disc writing speed (MB/s)
- Memory (MB)

- Network reception (B/s)
- Network transmission (B/s)

Table 2. Example of data recorded in the performance measurement of microservices architecture [8].

Time	CPU%	Disk-Read	Disk-Write	Memory-MB	Network-Rx	Network-Tx
1	39.593	0	0	7621.167	0	0
2	37.810	31.50	0	7621.167	0	0
3	31.578	35.29	50600	7813.601	68100	68100
4	32.338	37.56	10900	7894.582	72793	73509
5	40.722	52.87	1179648	8006.667	64500	64500
6	63.805	47.74	2994176	8648.839	423000	224000
7	74.193	45.17	2781184	8997.964	360000	187000
8	73.880	42.61	3366912	9471.820	498000	247000
9	61.369	37.56	4349952	9304.417	430000	222000
10	67.910	47.20	2727936	9316.832	444000	229000

Table 3. Example of data recorded in the performance measurement of monolithic architecture [8].

Time	CPU%	Disk-Read	Disk-Write	Memory-MB	Network-Rx	Network-Tx
1	31.313	0	0	867.207	0	0
2	35.820	0	0	867.207	0	0
3	36.616	1084	22.500	1025.601	84600	84600
4	38.000	444	13.500	1047.164	6591820	6593210
5	33.583	0	77.824	1176.492	167	256
6	51.133	216	159.744	1387.386	62500	62500
7	77.611	40	614.400	1921.847	212000	212000
8	50.670	0	512.000	2386.394	29700	150000
9	43.271	192	528.384	2367.207	35800	188000
10	57.692	0	253.952	1922.867	129000	173000

4.4. Mathematical Model

This section describes how raw data, corresponding to the result of determining performance when comparing the Web application from a monolithic architecture to microservices, need to be validated through a mathematical model. Thus, this is necessary since we use a mathematical formulation to express variable relationships to study complex system behaviors in situations that are complex to observe in reality. It consists of four subsections, being the Non-parametric Regression Model, the Calculus and Bandwidth estimation on a non-parametric regression model, the Election of the smoothing parameter, and Bandwidth estimation, as well as the Applicability of the mathematical model in the current study.

Given the rapid progress of computational statistics in mathematical models, the technique used in this study was the non-parametric model for the respective regression analysis. This occurred as it achieves a better adaptation to the available data, highlighting its flexibility essentially to various situations and problems that arise in society. It is conducted by obtaining estimates closer to the underlying regression curve. Therefore, we used the virtues and capabilities offered by the R and RStudio language.

4.4.1. Non-Parametric Regression Model

According to [69], a non-parametric regression model is a form of regression analysis in which the predictor does not have a predetermined way, but it is constructed according to information derived

from the data. Specifically, the goal of regression analysis is to produce a reasonable explanation of the relationship between a dependent variable (response variable y) and a set of independent variables (explanatory variables) $x1, \ldots, Xn$.

We assume that n pairs of data are observed (x_i, y_i) then it is able to establish the non-parametric regression model (1):

$$y_i = m(x_i) + \varepsilon_i \tag{1}$$

where $\varepsilon_i \ldots \varepsilon_n$ is the random variable distributed identically with $E(\varepsilon_i) = 0$ and $E(\varepsilon_i) = \sigma^2$, while the values of the variable $x_i \ldots x_n$ are known, leading to a design that is considered fixed. Additionally, the variance has constant errors. Therefore, the model presents homoscedasticity (i.e., when the variance of the conditional error to the explanatory variables is constant throughout the observations).

Given (X, Y) are bivariate random variables (i.e., a bivariate distribution represents the joint probability distribution of a pair of random variable) with joint density $f(x, y)$, the regression function is defined as well:

$m(x) = E(Y/X = x)$, which is the expected value of Y when X takes the known value x. Thus $E(Y/X) = m(X)$, and defining $\varepsilon = Y - m(X)$ it has to, $Y = m(X) + \varepsilon, E(\varepsilon/X) = 0, V(\varepsilon/X) = \sigma^2$, Therefore, the non-parametric regression model is (2):

$$Y_i = m(X_i) + \varepsilon_i, i = 1, 2, \ldots, n \tag{2}$$

As we already established the model, it is now necessary to determine the estimate or to adjust it based on the n observations available in the exploration. Therefore, an estimator m $\hat{m}(x)$ of the regression function and an estimator σ^2 of the error variance need to be generated. These estimation procedures of $m(x)$ are known as smoothed methods. Among the various non-parametric smoothing techniques, local adjustment is considered, which only considers data close to the point where it needs to estimate the function [70]. Therefore, we focus on research with the core function, which is reasonable to assume that the density function of the variable being estimated is continuous. These core type estimators (or Kernel) were designed to solve models with such characteristics within the defined variables.

Given the variables $X_i, \ldots, X_n$ with density f, its density at point t will be estimated using the estimator (3)

$$\hat{f}(t) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{t - X_i}{h}\right) \tag{3}$$

where h is a succession of smoothing parameters, called bandwidths that needs to tend slowly to zero [70], $h \to 0; nh \to \infty$ ensuring that $\hat{f}$ tends towards the true density f of the variables X_i and K is a function that fulfills $\int K = 1$. Thus, several nuclei may be taken into account, among which the following are considered:

Gaussian-core model (4):

$$\frac{1}{2\pi} e^{\frac{-u^2}{2}} \tag{4}$$

Or by the Epanechnikov Core (5):

$$\frac{3}{4}\left(1 - u^2\right) I_{|u|<1} \tag{5}$$

The directories of each scenario demonstrated a significant difference. Although that does not necessarily identify the application and its functionalities, it allows us to detail the project's structural granularity in general.

4.4.2. Calculus and Bandwidth Estimation in a Non-Parametric Regression Model

As noted in previous sections, the current study is innovative, being an extended development of our previous published work [8]. With this, we described the application design, the runtime environment, and conducted stress tests. Additionally, we described the tools such as JMeter, NewRelic, and ServerAgent, which allowed the collection of log files that were subsequently processed and analyzed for their proper interpretation. This finalized with the assessment and the performed comparison. These obtained results will be validated in this section by applying the Non-Parametric Regression Method.

From data sets of the variables Microservices performance Table 2 and Monolithic performance Table 3, through applying the kernSmooth library (i.e., Smooth Core Functions), the bandwidth h is calculated, while the h is the so-called smoothing parameter, also known as bandwidth or window [70]. This parameter is of significant importance for the performance of the regression core estimator. Depending on the weight function's choice, there will be different subtypes of the general class of core estimators [69]. Then, in order to determine the appropriate value of the h window needs to follow the subsequent model (see Equation (6))

$$h = \delta_k \left(\frac{3}{8}\right) \pi^{\frac{1}{10}} S_n n^{-\frac{1}{5}} \tag{6}$$

where:

- n: is the width of the sample
- $S_n = \left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^{\frac{1}{2}}$ standard deviation
- δ_k depends on the coreK,and it is calculated as:

$$\delta_k = \left(\frac{\int K^2(t)dt}{\left(\int u^2 K(t)dt\right)^2}\right)^{\frac{1}{5}}$$

So that:
If K is the Gaussian core, then

$$\delta_k = \left(\frac{1}{4\pi}\right)^{\frac{1}{10}} \tag{7}$$

If K is the Epanechnikov core, then

$$\delta_k = (15)^{\frac{1}{5}} \tag{8}$$

For exploration, it is considered a sample of $X_1, \ldots, X_n$, i.e., for the CPU variable (%), with some 88 observations. Therefore, the standard deviation is: $S_n = 8.089328$. Subsequently, using the core function of Epanechnikov, the performance for h will be (9):

$$h = \delta_k \left(\frac{3}{8}\right) \pi^{\frac{1}{10}} S_n n^{-\frac{1}{5}} \tag{9}$$

$$h = (15)^{\frac{1}{5}} \left(\frac{3}{8}\right) \pi^{\frac{1}{10}} (8.089328)(88)^{-\frac{1}{5}} = 2.3877$$

And for the Gaussian model, it has

$$h = \left(\frac{1}{4\pi}\right)^{\frac{1}{10}} \left(\frac{3}{8}\right) \pi^{\frac{1}{10}} (8.089328)(88)^{-\frac{1}{5}} = 1.0786$$

Hereby, a grid is generated for t, between the interpolating values, with semi-spaced points. Thus, for each t_j, $K\left(\frac{t_j - X_i}{h}\right)$, K is calculated. Afterwards, the estimate of f is obtained in order to generate the graphic representation.

$$\hat{f}(t) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{t - X_i}{h}\right) \tag{10}$$

The resources may be considered to be a minimum for a test with these characteristics. The used values to generate the workload are moderate, and in a certain way, they limit the level of work of the resources.

4.4.3. Election of the Smoothing Parameter

The smoothing parameter estimator h has crucial importance in the appearance and properties of the regression function estimator. The small values of h allow the estimator to display more flexibility, allowing them to approach all the observed data. However, they cause high prediction errors (overestimation). Thus, high values of h offer a lower degree of adjustments to the data. Nonetheless, they predict better. In contrast, if h is too high, there is a lack of adjustment to the data (sub-estimation). Therefore, within the research, depending on the values of h, both samples are obtained for the estimation of the model (i.e., training samples) and others for the comparison of the samples to predict (i.e., the sample test) (see Table 4). The quality of smoothing parameter h is the mean square error of the test sample (11).

$$ECMP_{test}(h) = \frac{1}{n}\sum_{i=1}^{nt} \left(Y_{i,t} - \hat{m}\left(X_{i,t}\right)\right)^2 \tag{11}$$

where $(X_{i,t}, Y_{i,t})$;$i = 1 \ldots n_t$ is the test sample and $\hat{m}(X)$ It is the non-parametric estimator built by the training sample. The value of h which minimizes the mentioned error would be the chosen smoothing parameter. With these arguments, we proceeded to calculate the different performance values *(h)* of the variables (see Table 4) in order to perform the respective mathematical analyses.

4.4.4. Core Function Estimators and Local Polynomials

Among the non-parametric alternative to the regression models, it is assumed that $Y = m(X) + e$ where m is a function which is not confined within a parametric family. Subsequently, we need to estimate m from a sample $(X_i, Y_i), \ldots, (X_n, Y_n)$, *where* is the kernel estimators which are established by the weight of (X_i, Y_i), in the estimate m, as considered by (12):

$$W_i\left(t, X_i\right) = \frac{\frac{1}{h}K\left(\frac{t - X_i}{h}\right)}{\hat{f}(t)} \tag{12}$$

where $K(t)$ is a symmetric density function and $\hat{f}(t)$ is a kernel density estimator. Furthermore, $W_i(t, X_i)$, and for each i, a weighting function that gives greater importance to the values X_i of the auxiliary variable that are close to t.

An alternative expression for $W_i(t, X_i)$, is $W_i(t, X_i) = \frac{K\left(\frac{t - X_i}{h}\right)}{\sum_{j=1}^{n} K\left(\frac{t - X_i}{h}\right)}$, is given after having calculated the weights W_i, where the exploration may be resolved by weighted least squares as follows (13):

$$\min_{a,b} \sum_{i=1}^{n} W_i\left(Y_i - (a + b\left(t - X_i\right))\right)^2 \tag{13}$$

where the parameters obtained depend on t, because of the weights W_i, they also need to depend on t. Therefore, the locally adjusted regression model around t will be: $I_t(X) = a(t) + b(t)(t - X)$ and

the estimation of the function at the point of $X = t$, is $\hat{m}(t) = l_i(t) = a(t)$, where the functions of the nucleus in the parametric estimation of the regression are identical as in the density.

When generalizing to the local adjustment of polynomial regressions of a higher degree, meaning if it is intended to estimate type models $\beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_q X^q$, where instead of value X_i it is used the value of $(t - X_i)$. The local polynomial estimator of grade q assigned weights W_i is obtained through the nucleus function of the polynomial regression that it can be solved hence (14).

$$\min_{\beta_0,\ldots,\beta_q} \sum_{i=1}^{n} W_i \left(Y_i - \left(\beta_0 + \beta_i(t - X_i) + \cdots + \beta_q(t - X_i)^q\right)\right)^2 \tag{14}$$

The parameters $\hat{\beta}_j = \hat{\beta}_j(t)$ depend on the point t, where the estimate is considered, and the locally adjusted polynomial around t, will be (15):

$$P_{q,t}(t - X) = \sum_{j=1}^{q} \hat{\beta}_j (t - X)^j \tag{15}$$

Being m(t) the estimated polynomial value at the point where $X = t$:$\hat{m}_q(t) = P_{q,t}(0) = \hat{\beta}_0(t)$ and for the particular case, it is adjusted for zero-grade polynomials, obtaining the Madaraya-Watson estimator, or known as the regression core estimator, using the model (16):

$$\hat{m}_k(t) = \frac{\sum_{i=1}^{n} K\left(\frac{t-X_i}{h}\right) Y_i}{\sum_{i=1}^{n} K\left(\frac{t-X_i}{h}\right)} = \sum_{i=1}^{n} W(t, X_i) Y_i \tag{16}$$

4.4.5. Applicability of the Mathematical Model in This Study

The following matrices are available, corresponding to the data sets of the variables:

- Microservices performance;
- Monolithic performance.

In each of them, we received subsets with the following variables, which were determined in [8], and transformed after the use of the Non-parametric Regression Model in the data listed in Table 4: CPU (%), Disc read speed (MB/s), Disc write speed writing (MB/s), Memory (MB), Network reception (B/s), and Network transmission (B/s).

For the case of CPU consumption (%), as instance, the fifth degree interpolated smoothed best fit polynomial based on the variables $(X_i, P(X_i))$. We considered the following mathematical model:

$$P(X) = beta_0 + beta_1 X + beta_2 X^2 + \cdots + beta_q X^5$$

$$P(X) = 65.9551 + 23.8631X - 32.1553X^2 + 33.1659X^3 - 24.2761X^4 + 19.8721X^5$$

Analogically, it is calculated for the other variables in the calculation of performance values. The identical ones which are represented in the different Tables 5–9, represented in Figures 11–16 in Section 5 (Results).

Table 4. Calculation of performance values

Calculation	CPU Consumption (%)	Disk Reading (MB/s)	Disk Writing (MB/s)	Memory Consumption (MB)	Network Reception (B/s)	Network Transmission (B/s)
Monolithic	2.3877	3.833604	12.40030	4.525155	3.285871	3.129852
Microservices	7.851149	3.833604	3.168329	2.887646	10.48274	4.505547

In the current research, the relevant data are used for n = 88, with the Elapsed time variables given in seconds, being the initial stabilization from 17 for up to 104 s. Furthermore, the CPU variable (%)

is considered; therefore, the relationship between CPU %—Elapsed time (sec) is initially regarded. Afterward, the core regression estimator is obtained, using the model (17)

$$\hat{m}_k(t) = \frac{\sum_{i=1}^{n} K\left(\frac{t-X_i}{h}\right) Y_i}{\sum_{i=1}^{n} K\left(\frac{t-X_i}{h}\right)} = \sum_{i=1}^{n} W(t, X_i) Y_i \tag{17}$$

where X_i is the Elapsed time value (sec.) and Y_i is the variable CPU (%). Subsequently a core function of Epanechnikov will be used, of which h window, will be (18):

$$h = \delta_k \left(\frac{3}{8}\right) \pi^{\frac{1}{10}} S_n n^{-\frac{1}{5}} \tag{18}$$

with: $\delta_k = (15)^{\frac{1}{5}}$ ¸ $S_n = 8.089328$, thus

$$h = (15)^{\frac{1}{5}} \left(\frac{3}{8}\right) \pi^{\frac{1}{10}} (8.089328) (88)^{-\frac{1}{5}} = 2.3877$$

Subsequently, for each age, (t) is calculate $K\left(\frac{t-X_i}{h}\right)$ and then $K\left(\frac{t-X_i}{h}\right) Y_i$ and finally, we get (19):

$$\hat{m}_k(t) = \frac{\sum_{i=1}^{n} K\left(\frac{t-X_i}{h}\right) Y_i}{\sum_{i=1}^{n} K\left(\frac{t-X_i}{h}\right)} \tag{19}$$

Successively, in order to operationalize the procedures of the mathematical calculations and for the respective graphs, the *R* language is used for all the variables involved in the present study (see Table 4).

5. Results

5.1. Cpu Consumption

The CPU (%) consumption in microservices is 5.4 units higher than in the monolithic scenario. There is a negligible difference in favor of the monolithic scenario, which generates lower consumption of this resource.

Table 5. CPU Consumption.

(h1)Calculation	CPU (%) Consumption
Monolithic	2.387700
Microservices	7.851149

The performance value using Monolithic = 2.387700, gives the estimator immense flexibility compared to the observed data concerning the microservices values = 7.851149 (see Table 5) that are higher. These data offer a lower degree of adjustment to the data by a difference of 5.4634. Individually, they consume more computing resources (See Figure 11) (20).

The results in Figure 11 are reasonable since, with the use of the application with microservices, a more significant number of requests are processed per second. Additionally, as each microservice has its resources and databases and the complexity of managing and integrating them, it also affects the CPU consumption. This means that although the application's performance obtained advantages, as a large network of services that depend on each other is built, the use of resources will increase when compared to the monolithic architecture.

$$\Delta h = h_2 - h_1 = 7.851149 - 2.387700 = 5.4634 \tag{20}$$

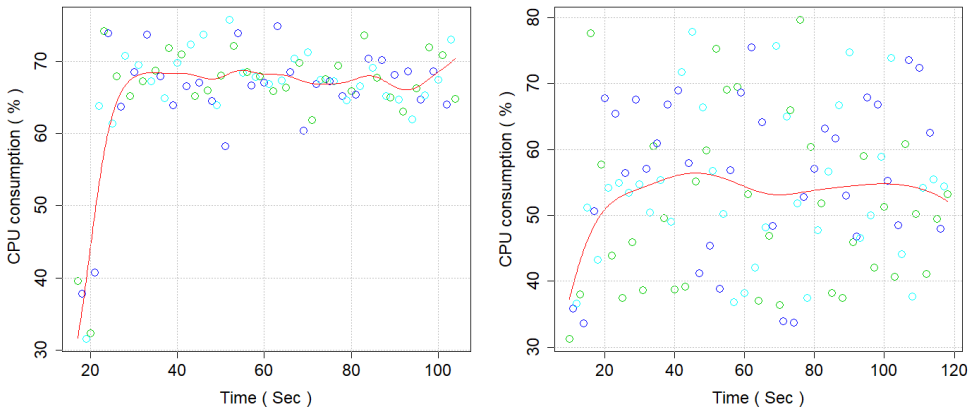

Figure 11. CPU Consumption (**left side**, Microservices, and **right side** Monolithic Architecture).

5.2. Memory Consumption

In contrast to CPU consumption, the average memory consumption is minor with 1.63 units (MB) in the scenario with microservices, compared to the monolithic proposal. This indicates that the data are better adjusted (see Figure 12). Then, we may conclude that the Monolith option consumes more memory as they are better adjusted to the research data. Therefore, the differences between memory consumption is (see Equation (21)) and Table 6) :

$$\Delta h = h_2 - h_1 = 2887646 - 4525155 = -1.63750 \tag{21}$$

Table 6. Memory Consumption (MB).

Memory Consumption (h)	Memory (MB)
Monolithic	4.525155
Microservices	2.887646

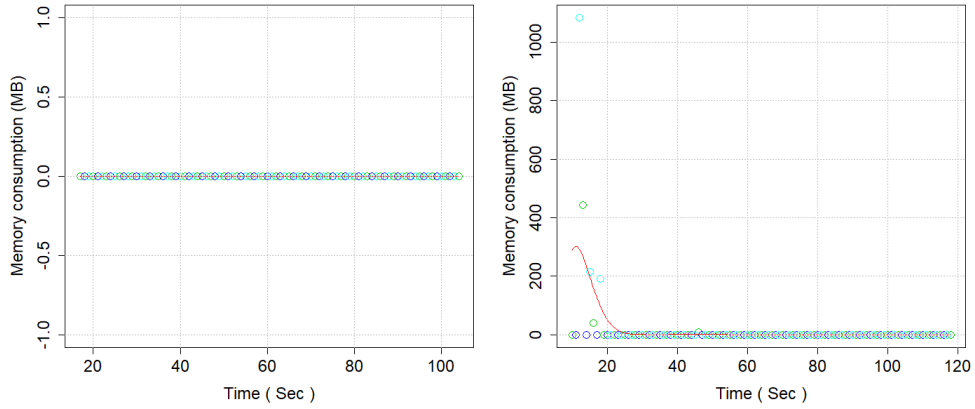

Figure 12. Memory Consumption (MB) (**left side**, Microservices, and **right side** Monolithic Architecture).

5.3. Network Transmission/Reception

In the case of the scenario using microservices, the determination of network transmission and reception (number of bytes per second) is more used than the Monolithic platform (7.19 units more in reception, and 1.37 units more in transmission process). This results since Docker generates a private network for the execution of their containers, which serves the same communication medium.

This allows that the data reception and or transmission capacity to be more efficiently used (see (22), (23), and Table 7). In other words, the data are more adjusted in the Microservices variables in both Network transmission and reception, which indicates that there is a higher speed in the delivery of packet reception than Monolithic option.

$$\Delta h = 10.48274 - 3.285871 = 7.1969 \tag{22}$$

$$\Delta h = 4.505547 - 3.129852 = 1.3757 \tag{23}$$

Table 7. Network Transmission/Reception (Bytes/s).

Calculation (h)	Network Reception	Network Transmission
Monolithic	3.285871	3.129852
Microservices	10.48274	4.505547

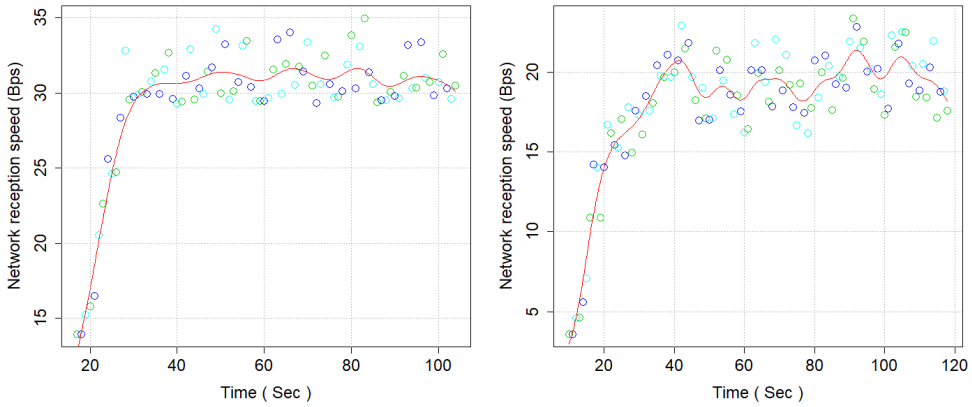

Figure 13. Network Received (Bytes/s) (**left**), Microservices, and Monolithic Architecture (**right**).

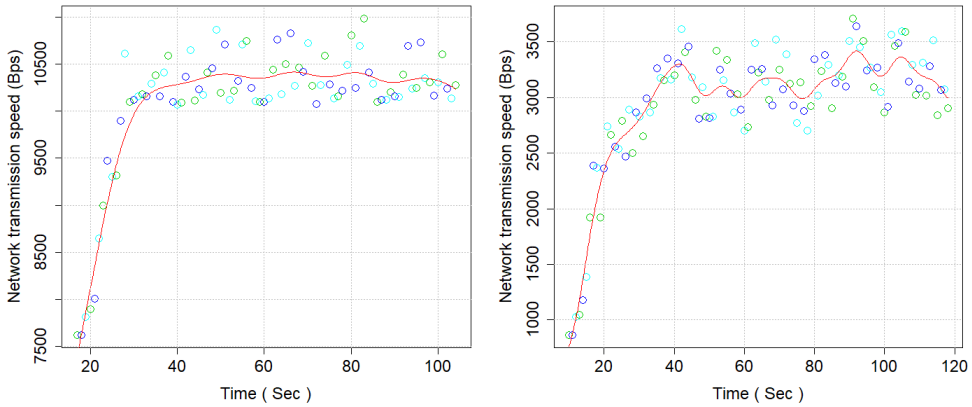

Figure 14. Network transmission (Bytes/s) (**left**), Microservices, and Monolithic Architecture (**rigth**).

5.4. Disk Writing and Reading Speed

In regards to the process of reading/write hard disk speed (i.e., throughput), within the case of disk reading speed, there is no difference in both architectures (h = 3.833604 (see Table 8) and Figure 15).

$$\Delta h = h_2 - h_1 = 12.40030 - 3.168329 = 9.2320 \tag{24}$$

Table 8. Disk reading speed.

Calculation (h)	Disk Reading (MB/s)
Monolothic	3.833604
Microservices	3.833604

Table 9. Disk writing speed.

Calculation (h)	Disk Reading (MB/s)
Monolothic	12.400365
Microservices	3.168329

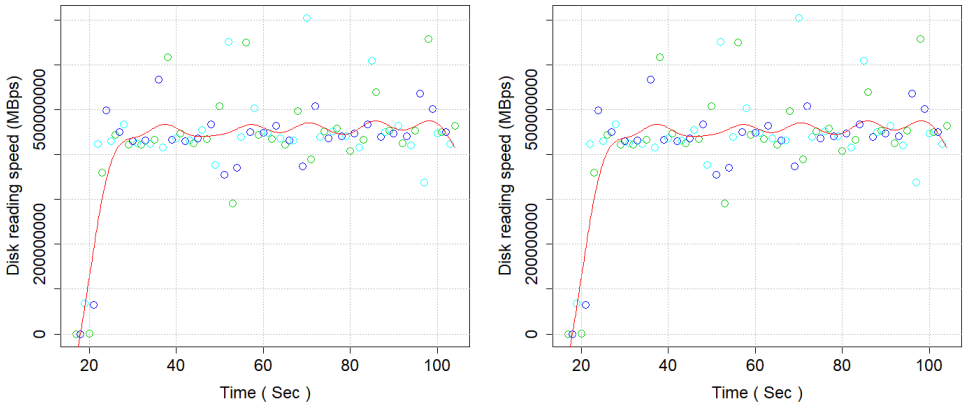

Figure 15. Disk reading speed (MB/s) (**left side**, Microservices, and **right side** Monolithic Architecture).

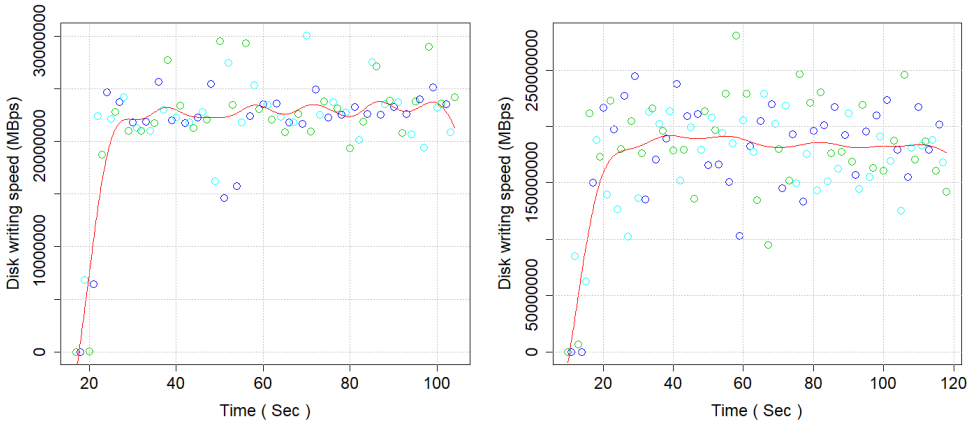

Figure 16. Disk writing speed (MB/s) (**left side**, Microservices, and **right side** Monolithic Architecture).

However, the disk writing speed in microservices is lowest (9.23 units less), instead of the one obtained by the Monolithic (see Figure 16). Consequently, the rate of disk writing is higher in Monolithic. Thus, the writing speed on disk h_2 = 12.40030 of Monolithic is faster than the one f the Microservices platform (see Table 9, Figure 16, and (24)). This indicates a finer dispersion of the Monolithic data concerning the Microservices platform, which is better adjusted to the data. During the interpretation of these results, such observation outstands, considering that applications need to handle amounts of data that constantly travel between them in microservices. This is able to lead to a series of issues when integrating all this information, such as synchronization, scaling, or data processing. Thus, there is a disadvantage compared to the best writing speed obtained in a monolithic architecture.

5.5. Discussion

5.5.1. Experience, Knowledge and Learning

After a critical and thorough review of various bibliographic sources, it is evident that multiple positions determine the benefits and limitations of each of the analyzed architectures. For example, when talking about microservices architecture, there is no defined model. Each company needs to determine which model is best suited to the business model they are implementing. Instead, the Monolithic architecture, which by its nature concedes secure handling, allows the implementation of a base model for almost unlimited use. However, a high amount of companies lack to conceive of a base model since their founding within the business, as a future level of growth of an enterprise remains initially unknown. This occurred to all companies which opted for this new architecture, where its growth might have been exponential, while they required adaptation and rapid change. Subsequently, such companies search for adaptive technologies that allow growth simultaneously with the companies development.

When we compare the monolithic with the microservice architecture, we observed that either have their merits and demerits, as characterized by the entire system's volume and complexity.

Among the advantages of microservices are scalability, modular functionality with independent modules, and containers' use, allowing the application's deployment and development quickly. They will enable the use of different technologies and languages. They can be deployed as needed; consequently, they work satisfactorily within agile methodologies. They also allow rapid and continuous improvement of each functionality. Maintenance is more straightforward and cheaper compared with the microservices. Therefore it will enable improvements to one module at a time, leaving the rest to work regularly. Additionally, the developer can benefit from the functionalities that third parties have already developed.

Among the disadvantages of applying microservices, we may mention that global tests are more complicated as the components are distributed. It is necessary to control the number of microservices that are managed. The more microservices appear in a solution, the more difficult it will be to manage and integrate them entirely. Microservices require developers with a high level of experience. This may result in being expensive to implement due to licensing costs for third-party applications. The benefits may seem to outweigh the cons. However, there are specific problems inherent in microservices-based solutions such as cost, efficiency, and response times analyzed in this study. The most important fact to consider is about which solution may be best suited to the specific project's needs and which solutions will help achieve the planned business objectives.

Another issue has been determined to be the attitude, culture, or professionalism of the technical personnel, often avoiding an easy adaptation and migration to other technological platforms since there is an inevitable rejection of change. Nonetheless, usually, companies are aware of the modularity and independence that microservices promote. However, it is also true that more advanced knowledge and experience are required in adapting to this new architecture, which is why a gradual and controlled migration is suggested.

It is generally considered that the microservice is an evolutionary approach compared to the monolithic. Moreover, while the application of microservices has been developed, a set of new complexities, data redundancy, transmission, reception of data, and more interoperability and data classification challenges emerged. Microservices communicate through the network, which may cause communication failures or, in cases worse, to some slow communications. The testing in a microservice architecture becomes more complicated due to its distributed nature. Hereby, not only are some unit tests of services necessary, but there are also integration needed. Furthermore, there is more activity of CPU consumption and data transmission/reception in the microservices system. However, in the application performance, such a phenomenon is reversed, as it consumes more CPU, but uses less memory, leading to better use of connectivity.

The use of containers allows jobs to be conducted independently; this is because there is no need to expand the memory on raising another operating system since everything performs at a level of software and libraries. This attracts the business sector since the hardware resources are then optimized, and frequently their use is hereby more efficient.

Based on the results obtained during the tests, we observed that the reading process evolves continuously and simultaneously, regardless of the storage medium. Instead, the writing process always varies, as it repeatedly depends on the storage medium. Therefore, the transfer process also changes.

5.5.2. Mathematical Modeling and Multidisciplinary Integration

Concerning the application of the mathematical model, the non-parametric models, which are between one of the many applications in statistics, they allow solving the existing relationships between the variables of interest by using flexible functionalities that approximate unknown relationships between the variables. This entails the need to estimate the density function, where these functions tend to fit the data smoothly, allowing a minimum error to be generated. However, the sample size is also analyzed according to the quantity of the involved random variables. For these aspects, the application of the non-parametric regression model in the current research is considered, establishing the estimates of $\hat{m}(x)$ of the regression function and an estimator δ^2 of the error variance from the n observations. These procedures of estimation $m(x)$ are known as smoothed methods h (bandwidth of the function) of local adjustment, considering only data close to the point where it needs to estimate the function. Therefore, the research focuses on the core function. These core type estimators were designed to solve models with the previously mentioned characteristics.

Nonetheless, the smoothing parameter estimator h is fundamental in the appearance and properties of the regression function estimator, where small values of h generate greater flexibility for the estimator—hence adjusting better to the observed data. At the same time, they may cause high prediction errors, also known as overestimation. However, the high values of h offer a lower adjustment to the data, but they predict more adequately. If h is too high, the data being a sub estimation, are not adjusted. Then, depending on the values of h, and the samples for the evaluation of both architectures, the respective analysis between the different variables may be performed.

At the end of the application of the model, the results allow us to infer that the Monolithic architecture uses 5.47 units less CPU than microservices. In disk reading, they present the same results. In disk writing, however, the Monolithic uses 9.24 units more than microservices. Monolithic uses 1.64 units more than microservices in memory consumption, while the microservices receive 7.2 more units of packages per second compared to the monolithic. Finally, microservices transmit 1.38 packet units per second more compared with a monolithic architecture. The results indicate that the microservices architecture leverages the CPU and packet transmission/reception resources more efficiently compared to the monolithic architecture, which consumes more memory resources and disk writing speed; this would yield a decrease in system performance. Therefore, microservices architecture is an excellent alternative to computer innovation.

Finally, the application of the non-parametric mathematical procedure, which considers the original data measured in the two analyzed architectures, given the characteristics of its variables, generated greater precision minimizing the error and demonstrating the benefits of the current research.

6. Conclusions and Future Work

In the current study, computational metrics such as CPU, disk reading speed, disk writing speed, memory, network reception, and transmission are evaluated for comparing a web application in a monolithic architecture versus the same application using microservices. The mathematical model based on the Non-Parametric Regression Method was applied to validate these studies' findings. Also, we could accurately validate the results and provide a quantitative technical interpretation. The architecture of microservices with containers proved to be more efficient. This gives rise to

discussion topics and is part of the planning that leads to DevOps' use when time is a priority factor. Our findings indicate that using microservices with containers generally applied in demanding computer systems, at massive scale, and within high traffic allows jobs to be conducted independently hence faster than serialized processed. Containers were the methods used to encapsulate microservices, which simplified configuration. Since it is necessary to deploy the applications to be uploaded to the cloud, it is preferable to use containers and upload them instead of using virtual machines. Its purpose is feasible if the goal is to process a large number of requests in the shortest time possible, regardless of resource limitations and where scalability is a priority. We could also determine that both platforms (monolithic and micro-services) could be used for different purposes and different technological conditions. However, our study demonstrates more efficiency concerning hardware resources, cost reduction, and high productivity when using microservices.

As future work, we plan to optimize a collaborative native monolithic tool by migrating it to microservices. With this, we intend to continue showing the strengths and weaknesses of each of these architectures. Also, we plan the extension of the design and implementation of a users and services management system that optimizes the creation and deployment of containers, using MongoDB as a back end deployed in a Docker container, developed in Node.js, managed in cluster mode for high availability, and works through HTTPS. To solve the overload problem, we will design and implement a load balancer, whose primary purpose will be to respond to requests that come from the management system. The balancer will reply to the management system, with the address URL of the container with the least amount of users connected to it. It will also communicate with each container, every time a user connects to one of them. A high transactional rate must characterize this load balancer. Also, it must be under the principle of microservices and operate independently. Besides, it must collect each node's status and the number of users per node. Finally, given the increase in cyber-attacks on microservices applications, we will design software solutions to increase these systems' information security.

Author Contributions: F.T. is the initiator of the project. *Conceptualization:* F.T., W.F. and E.F. identified the theoretical constructs and the different elements. *Validation:* M.Á.M. and H.A. conducted the verification and validation that the system complies with the requirements and specifications. *Formal analysis:* F.T., E.F. and M.Á.M. developed such analysis. *Resources and funding:* M.Á.M. and W.F. obtained financial, and economic resources. *Writing—initial draft preparation:* F.T. and W.F. edited the first deliverable draft of the manuscript. *Writing—review and editing:* T.T. re-edited the paper and performed proofreading. *Supervision:* M.Á.M. and W.F. supervised the technical and scientific quality assurance of the study. All authors have read and agreed to the proposed version of the manuscript.

Funding: The funding of this research is provided from the Madrid Regional Government co-funded through the e-Madrid-CM Project under Grant S2018/TCS-4307, co-funded by the European Structural Funds (FSE and FEDER), and also from the Mobility Regulation of the Universidad de las Fuerzas Armadas ESPE, from Sangolqui, Ecuador.

Acknowledgments: The authors would like to thank the academic and technical support from the Specialized Laboratories of the Universidad de las Fuerzas Armadas ESPE, from Sangolquí, Ecuador.

Conflicts of Interest: We ensure that our manuscript has not been submitted simultaneously for publication anywhere else, containing original data, and the content of this paper has not been even presented previously in any symposium or congress. Finally, we do not have any material (figures, images, or tables) included in the manuscript that may require to obtain permission to reproduce copyrighted material from other sources. There is no conflict of interest with any party; there is no problem with the ethical standards of any kind.

References

1. Teixeira, S.; Martins, J.; Branco, F.; Gonçalves, R.; Au-Yong-Oliveira, M.; Moreira, F. A theoretical analysis of digital marketing adoption by startups. In *International Conference on Software Process Improvement*; Springer: Cham, Switzerland, 2017; pp. 94–105.
2. Stubbs, J.; Moreira, W.; Dooley, R. Distributed systems of microservices using docker and serfnode. In Proceedings of the IEEE 2015 7th International Workshop on Science Gateways, Budapest, Hungary, 3–5 June 2017; pp. 34–39.

3. Villamizar, M.; Garces, O.; Ochoa, L.; Castro, H.; Salamanca, L.; Verano, M.; Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In Proceedings of the 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, Colombia, 16–19 May 2016; pp. 179–182.
4. Kalske, M.; Mäkitalo, N.; Mikkonen, T. Challenges when moving from monolith to microservice architecture. In *International Conference on Web Engineering*; Springer: Cham, Switzerland, 2017; pp. 32–47.
5. Newman, S. *Building Microservices: Designing Fine-grained Systems*; O'Reilly Media, Inc.: Newton, MA, USA, 2015.
6. Al-Debagy, O.; Martinek, P. A Comparative Review of Microservices and Monolithic Architectures. In Proceedings of the 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 21–22 November 2018; pp. 149–154.
7. Tapia, F.; Mora, M.Á.; Fuertes, W.; Lascano, J.E.; Toulkeridis, T. A Container Orchestration Development that Optimizes the Etherpad Collaborative Editing Tool through a Novel Management System. *Electronics* **2020**, *9*, 828.
8. Saransig, A.; Tapia, F. Performance analysis of monolithic and microservice architectures–containers technology. In *International Conference on Software Process Improvement*; Springer: Cham, Switzerland, 2018; pp. 270–279.
9. Guaman, D.; Yaguachi, L.; Samanta, C.C.; Danilo, J.H.; Soto, F. Performance evaluation in the migration process from a monolithic application to microservices. In Proceedings of the IEEE 2018 13th Iberian Conference on Information Systems and Technologies (CISTI), Caceres, Spain, 13–16 June 2018; pp. 1–8.
10. Akbulut, A.; Perros, H.G. Performance Analysis of Microservices Design Patterns. *IEEE Internet Comput.* **2019**, *23*, 19–27.
11. Singh, V.; Peddoju, S.K. Container-based microservice architecture for cloud applications. In Proceedings of the IEEE 2017 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, 5–6 May 2017; pp. 847–852.
12. Ponce, F.; Márquez, G.; Astudillo, H. Migrating from monolithic architecture to microservices: A Rapid Review. In Proceedings of the 2019 IEEE 38th International Conference of the Chilean Computer Science Society (SCCC), Concepcion, Chile, 4–9 November 2019; pp. 1–7.
13. Taibi, D.; Lenarduzzi, V.; Pahl, C. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Comput.* **2017**, *4*, 22–32.
14. Mazlami, G.; Cito, J.; Leitner, P. Extraction of microservices from monolithic software architectures. In Proceedings of the 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 25–30 June 2017; pp. 524–531.
15. Bures, T.; Duchien, L.; Inverardi, P. (Eds.) *Software Architecture—Proceedings of the 13th European Conference, ECSA 2019, Paris, France, 9–13 September 2019*; Springer Nature: Cham, Switzerland 2019; Volume 11681.
16. Fritzsch, J.; Bogner, J.; Zimmermann, A.; Wagner, S. From monolith to microservices: A classification of refactoring approaches. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*; Springer: Cham, Switzerland, 2018; pp. 128–141.
17. Kecskemeti, G.; Marosi, A.C.; Kertesz, A. The ENTICE approach to decompose monolithic services into microservices. In Proceedings of the IEEE 2016 International Conference on High Performance Computing & Simulation (HPCS), Innsbruck, Austria, 18–22 July 2016; pp. 591–596.
18. Kamimura, M.; Yano, K.; Hatano, T.; Matsuo, A. Extracting Candidates of Microservices from Monolithic Application Code. In Proceedings of the IEEE 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, 4–7 December 2018; pp. 571–580.
19. Bucchiarone, A.; Dragoni, N.; Dustdar, S.; Larsen, S.T.; Mazzara, M. From monolithic to microservices: An experience report from the banking domain. *IEEE Softw.* **2018**, *35*, 50–55.
20. Djogic, E.; Ribic, S.; Donko, D. Monolithic to Microservices redesign of event driven integration platform. In Proceedings of the IEEE 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 21–25 May 2018; pp. 1411–1414.
21. Acevedo, C.A.J.; y Jorge, J.P.G.; Patiño, I.R. Methodology to transform a monolithic software into a microservice architecture. In Proceedings of the 2017 IEEE 6th International Conference on Software Process Improvement (CIMPS), Zacatecas, Mexico, 18–20 October 2017; pp. 1–6.
22. Sarkar, S.; Vashi, G.; Abdulla, P.P. Towards Transforming an Industrial Automation System from Monolithic to Microservices. In Proceedings of the 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), Turin, Italy, 4–7 September 2018; pp. 1256–1259.

23. Escobar, D.; Cárdenas, D.; Amarillo, R.; Castro, E.; Garcés, K.; Parra, C.; Casallas, R. Towards the understanding and evolution of monolithic applications as microservices. In Proceedings of the 2016 XLII Latin American Computing Conference (CLEI), Valparaiso, Chile, 10–14 October 2016; pp. 1–11.
24. Debroy, V.; Miller, S. Overcoming Challenges with Continuous Integration and Deployment Pipelines When Moving From Monolithic Apps to Microservices: An experience report from a small company. *IEEE Softw.* **2019**, *37*, 21–29.
25. Chen, R.; Li, S.; Li, Z. From monolith to microservices: A dataflow-driven approach. In Proceedings of the 2017 IEEE 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, China, 4–8 December 2017; pp. 466–475.
26. Sarita; Sebastian, S. Transform Monolith into Microservices using Docker. In Proceedings of the International Conference on Computing, Communication, Control and Automation (ICCUBEA), Pune, India, 17–18 August 2017; pp. 1–5.
27. Nielsen, D. Investigate Availability and Maintainability within a Microservice Architecture. Ph.D. Thesis, Aarhus University, Aarhus, Denmark, 2015.
28. Khazaei, H.; Barna, C.; Beigi-Mohammadi, N.; Litoiu, M. Efficiency analysis of provisioning microservices. In Proceedings of the 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg, 12–15 December 2016; pp. 261–268.
29. Sun, L.; Li, Y.; Memon, R.A. An open IoT framework based on microservices architecture. *China Commun.* **2017**, *14*, 154–162.
30. Fowler, M.; Lewis, J. Microservices a Definition of This New Architectural Term. 2014; Volume 22. Available online: Http://martinfowler.Com/articles/microservices.html (accessed on 1 June 2020).
31. Kratzke, N. About Microservices. *Contain. Their Underestim. Impact* **2015**, *961*, 165–169.
32. Henry, A.; Ridene, Y. Assessing Your Microservice Migration. In *Microservices*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 73–107.
33. Fussell, M. Why a Microservices Approach to Building Applications? 2020. Available online: https://docs.microsoft.com/es-es/azure/service-fabric/service-fabric-overview-microservices (accessed on 1 June 2020).
34. Burvall, B. Improvement of Container Placement Using Multi-Objective Ant Colony Optimization. KTH School of Electrical Engineering and Computer Science (EECS). 2019. Available online: https://www.diva-portal.org/smash/get/diva2:1305653/FULLTEXT01.pdf (accessed on 1 June 2020).
35. Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An updated performance comparison of virtual machines and linux containers. In Proceedings of the 2015 IEEE international symposium on performance analysis of systems and software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015; pp. 171–172.
36. Bartholomew, D. Qemu: A multihost, multitarget emulator. *Linux J.* **2006**, *145*, 3.
37. Erinle, B. *Performance Testing with JMeter*; Packt Publishing Ltd.: Birmingham, UK, 2015; ISBN 978-1-78216-584-2.
38. Rajput, D. *Hands-On Microservices–Monitoring and Testing: A performance Engineer's Guide to the Continuous Testing and Monitoring of Microservices*; Packt Publishing Ltd.: Birmingham, UK, 2018.
39. Desai, P.R. A survey of performance comparison between virtual machines and containers. *Int. J. Comput. Sci. Eng.* ***2016***, 4, *55–59.*
40. Scott, J. A Practical Guide to Microservices and Containers: Mastering the Cloud, Data and Digital Transformation. 2017. Available online: Https://pdfs.semanticscholar.org/9757/5b5231c40d9fa8e340528fa47a85c4bd2de2.pdf?_ga=2.180544677.324332155.1597956977-214923530.1581080317 (accessed on 1 June 2020).
41. Vaughan-Nichols, S.J. New approach to virtualization is a lightweight. *Computer* **2006**, *39*, 12–14.
42. Rea-Guaman, A.M.; San Feliu, T.; Calvo-Manzano, J.A.; Sanchez-Garcia, I.D. Systematic review: Cybersecurity risk taxonomy. In Proceedings of the International Conference on Software Process Improvement, Zacatecas, Mexico, 18–20 October 2017; Springer: Cham, Switzerland, 2017; pp. 137–146.
43. Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Comput.* **2014**, *1*, 81–84.
44. Pahl, C. Containerization and the paas cloud.*IEEE Cloud Comput.* **2015**, *2*, 24–31.
45. Gerlach, W.; Tang, W.; Keegan, K.; Harrison, T.; Wilke, A.; Bischof, J.; DSouza, M.; Devoid, S.; Murphy-Olson, D.; Desai, N.; et al. Skyport-container-based execution environment management for multi-cloud scientific workflows. In Proceedings of the 2014 IEEE 5th International Workshop on Data-Intensive Computing in the Clouds, New Orleans, LA, USA, 21 November 2014; pp. 25–32.

46. Joy, A.M. Performance comparison between Linux containers and virtual machines. In Proceedings of the 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, India, 19–20 March 2015; pp. 342–346, doi:10.1109/ICACEA.2015.7164727.
47. Matusiak, P. Monoliths to Microservices: What to Know about Modernising Enterprise Applications. Information Age. 2019. Available online: Https://www.information-age.com/modernising-enterprise-applications-123486138/ (accessed on 1 June 2020).
48. Doerrfeld, B. From Monolith to Microservices: Horror Stories and Best Practices. Teachbeacon. 2019. Available online: Https://techbeacon.com/app-dev-testing/monolith-microservices-horror-stories-best-practices (accessed on 1 June 2020).
49. Wix. Available online: Https://www.wix.com/ (accessed on 7 May 2020).
50. Best Buy. Available online: Https://www.bestbuy.com/ (accessed on 7 May 2020).
51. Cloud Elements. Available online: Https://cloud-elements.com/ (accessed on 7 May 2020).
52. Villamizar, M.; Garcés, O.; Castro, H.; Verano, M.; Salamanca, L.; Casallas, R.; Gil, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In Proceedings of the 2015 IEEE 10th Computing Colombian Conference (10CCC), Bogota, Colombia, 21–25 September 2015; pp. 583–590.
53. Levcovitz, A.; Terra, R.; Valente, M.T. Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv* **2016**, arXiv:1605.03175.
54. Schäffer, E.; Leibinger, H.; Stamm, A.; Brossog, M.; Franke, J. Configuration based process and knowledge management by structuring the software landscape of global operating industrial enterprises with Microservices. *Procedia Manuf.* **2018**, *24*, 86–93.
55. Leung, A.; Spyker, A.; Bozarth, T. Titus: Introducing containers to the Netflix cloud. *Commun. ACM* **2018**, *61*, 38–45.
56. Uotila, T. Designing scalable microservices: Case: AWS with Python. 2019. Available online: Hhttps://www.theseus.fi/bitstream/handle/10024/169685/Uotila_Timsa.pdf?sequence=2&isAllowed=y (accessed on 1 June 2020).
57. Chawla, H.; Kathuria, H. *Building Microservices Applications on Microsoft Azure*; Apress: New York, NY, USA, 2019; doi:10.1007/978-1-4842-4828-7, ISBN 978-1-4842-4828-7/978-1-4842-4827-0.
58. Pathak, N.; Bhandari, A. Consuming Microsoft Cognitive APIs. In *IoT, AI, and Blockchain for. NET*; Apress: Berkeley, CA, USA, 2018; pp. 125–146.
59. Chawla, H.; Kathuria, H. Evolution of Microservices Architecture. In *Building Microservices Applications on Microsoft Azure*; Apress: Berkeley, CA, USA, 2019; pp. 1–20.
60. Al-Dhuraibi, Y.; Paraiso, F.; Djarallah, N.; Merle, P. Autonomic vertical elasticity of docker containers with elasticdocker. In Proceedings of the 2017 IEEE 10th international conference on cloud computing (CLOUD), Honolulu, CA, USA, 25–30 June 2017; pp. 472–479.
61. Bicocchi, N.; Boese, S.; Cabri, G. User-Aware Comfort in Retail Environments. In *Modeling and Using Context—Proceedings of the 11th International and Interdisciplinary Conference, CONTEXT 2019, Trento, Italy, 20–22 November 2019*; Springer: Cham, Switzerland, 2019; pp. 14–25.
62. Sharma, S.; Rajesh, R.; Gonzalez, D. *Microservices: Building Scalable Software*; Packt Publishing Ltd.: Birmingham, UK, 2017.
63. Johansson, P. Efficient Communication with Microservices. Master's Thesis, Umea University, Umea, Sweden, 2017.
64. Yadav, R.R.; Sousa, E.T.G.; Callou, G.R. A.Performance comparison between virtual machines and docker containers. *IEEE Lat. Am. Trans.* **2018**, *16*, 2282–2288.
65. Verberne, S.; Krahmer, E.; Wubben, S.; Van Den Bosch, A. Query-based summarization of discussion threads. *Nat. Lang. Eng.* **2020**, *26*, 3–29.
66. Sugijarto, D.P.; Mukhtar, M.; Safie, N.; Sulaiman, R. Developing Context Awareness Mobile Application for Blood Donation. JOIV. *Int. J. Inform. Vis.* **2018**, *2*, 118–126.
67. Amazon Web Service (AWS). Break a Monolith Application into Microservices with Amazon Elastic Container Service, Docker y Amazon EC2. Available online: Https://aws.amazon.com/getting-started/ (accessed on 8 August 2020).
68. W. Fuertes, J. E. López de Vergara, F. Meneses and F. Galán, A generic model for the management of virtual network environments. In Proceedings of the IEEE Network Operations and Management Symposium (NOMS 2010), Osaka, Japan, 19–23 April 2010; pp. 813–816, doi:10.1109/NOMS.2010.5488367.

69. Zhang, X.; King, M.; Shang, L. A sampling algorithm for bandwidth estimation in a nonparametric regression model with a flexible error density. *Comput. Stat. Data Anal.* **2014**, *78*, 218–234; ISSN 0167-9473.
70. Lee, C.T. Smoothing parameter selection for smoothing splines: A simulation study. *Comput. Stat. Data Anal.* **2003**, *42*, 139–148.