electronics

MDPI

Article

A Container Orchestration Development That Optimizes the Etherpad Collaborative Editing Tool through a Novel Management System

Freddy Tapia [1,2,*], Miguel Ángel Mora [2,*], Walter Fuertes [1,*], Jorge Edison Lascano [1] and Theofilos Toulkeridis [1]

[1] Deparment of Computer Sciences, Universidad de las Fuerzas Armadas ESPE, Av. General Rumiñahui S/N, P.O. Box 17-15-231B, Sangolquí 171103, Ecuador; jelascano@espe.edu.ec (J.E.L.); ttoulkeridis@espe.edu.ec (T.T.)

[2] Department of Informatics Engineering, Universidad Autónoma de Madrid, Ciudad Universitaria de Cantoblanco, 28049 Madrid, Spain

* Correspondence: fmtapia@espe.edu.ec (F.T.); miguel.mora@uam.es (M.Á.M.); wmfuertes@espe.edu.ec (W.F.); Tel.: +593-998926648 (F.T.)

Received: 21 February 2020; Accepted: 1 May 2020; Published: 17 May 2020

Abstract: The use of collaborative tools has notably increased recently. It is common to see distinct users that need to work simultaneously on shared documents. In most cases, large companies provide tools whose implementations have been a very complicated and expensive task. Likewise, their platform deployment requirements should be robust hardware infrastructures. It becomes even more critical when their main target is to reach scalability and highavailability. Therefore, this study aims to design and implement a microservices-based collaborative architecture using assembled containers in the cloud, enabling them to deploy Etherpad instances to guarantee high availability. To ensure such a task, we developed and optimized a central management system that creates Etherpad instances and continuously interacts with other Etherpad tools running on Docker containers. This design goes from the monolithic Etherpad instantiation and handling towards a service architecture, where every Etherpad is offered as a microservice. Furthermore, the management system follows (implements) the Observer, Factory Method, Proxy, and Service Layerpopular design patterns. This allows users to gain more privacy through access to validations and shared resources. Our results indicate both the correct operation in the automation of containers' creation for new users who register in the system and quantifiable improvement in performance.

Keywords: collaborative tools; containers; cloud computing; Etherpad; management systems; microservices; monolithic

1. Introduction

Microservices have increasingly become popular in recent years. Some companies, such as Netflix, Amazon, and The Guardian, have successfully been early adopters of microservices in large-scale software systems environments [1]. The microservices architecture (MSA) is an approach in software development that consists of building a distributed application as a set of small services, each one running in its process and communicating with light mechanisms; they are often HTTP API resources [2]. Each service is responsible for the implementation of full business functionality and is deployed independently. Furthermore, they can be programmed in different languages and use diverse data storage technologies [3].

However, a microservices-based distributed architecture inevitably inherits the problems and complexities of distributed systems. For example, they are much more complicated to design; they

include a high risk of failure with performance and scalability problems, especially as the system grows in size [4].

Additionally, the dilemma of the design of a microservice architecture versus a monolithic architecture persists [5]. In the first case, the offered software package is not presented as an individual product, as each function responds autonomously to others. This leads to an inefficientlymaintainable solution and may even react with different levels of effectiveness depending on the generated activity [6]. A monolithic architecture, on the other hand, is an architecture where software is structured in such a way that every functional aspect is coupled and packaged inside the very same program. Either way, the information that these systems need to run correctly is steadily hosted on a single server. Due to its nature, there is no separation of modules of any kind. Therefore, the different layers of a program depend entirely on each other [7]. Testing is generated based on a monolithic architecture, which causes performance problems that prevent being scaled vertically or horizontally. The problem has been corrected by introducing microservices-based containers, obtaining a vertical or horizontal growth on demand.

Based on the given scenario, this study aims to design and implement a microservices-based collaborative architecture, through the use of assembled containers in the cloud, enabling deployment of Etherpad instances. To do this, we develop and optimize a central management system that creates Etherpad instances and continuously interacts with Etherpad tool containers. Specifically, our management system implements four popular design patterns that together solve the aforementioned problems: Observer pattern, Factory Method pattern, Proxy pattern, and Service Layer pattern. On the other side, Monitortools are only based on the Observer design pattern. A monitortool does not make any decision, and it only informs other applications about monitored objects' states. It is worth noting that the design patterns we used are oriented towards the object-oriented paradigm; hence, we adapted the patterns for our purposes. Our system mainly implements services.

Furthermore, this study demonstrates that Etherpad, an open-source collaborative real-time web text editor, developed under a monolithic architecture [8], is able to perform in a cloud-based microservices architecture. We expect that scaling the number of Etherpad nodes to a simultaneously large number of editions/editors allows the application to operate regularly and without delay [9]. It is also expected to reduce the consumption of hardware resources in comparison to a monolithic architecture. We, as well, intend to reveal how Etherpad can perform as a container that guarantees its availability and allows for easy scalability in terms of adaptability for a scalable distributed system.

In concordance with Lascano [10], adaptability for a scalable distribution, i.e., scalability, means that a system is scalable when it provides more or fewer resources and users than it was originally designed for, yet, the systems perform properly. When we design a client-server or a distributed application, we have to use communication protocols. Since RESTful apps work at the application level, we follow application level communication protocols (ACPs) that offer scalability by providing location transparency and/or replication transparency [11]. This scalability is able to refer to resources such as data, operations, or objects that are able to be distributed across multiple hosts. Another way to support scalability is to break up complex resources into smaller components (services or microservices) and distribute them across multiple servers [10].

To demonstrate the proposed scalability, it is possible to have N layers, but only one layer (database) is currently shown. This does not imply that such an architecture is able to grow according to the requirements. Therefore, we deploy Etherpad instances so that improvement in performance, scalability, and availability can be demonstrated. In this scenario, the application allows distinct authors to edit a text document simultaneously and monitor all participants in real-time.

The experiment results show both the correct execution of the editor in the automation of containers' creation for new users registered in the system, as well as quantifiable improvement in performance, compared to a traditional (monolithic) architecture under similar conditions. Therefore, usability is not affected by allowing various services to work on demand and the containers to be responsible, unlike a monolithic system, which is affected by the usability of the application.

Among the main contributions of this study, we have the following: First is the design and implementation of a free microservices-oriented software architecture that uses cloud-centered containers to deploy Etherpad instances through GET http requests providing in this way high availability. Second is the implementation of a central user management system, responsible for installing Etherpad instances according to the users' creation and their login into the platform. As a consequence, it performs as a service using containers and giving better use to hardware resources. Third, we are providing a robust and well-designed software architecture based on proven design patterns, and this architecture integrates Docker as a deployment platform for Etherpad microservices orchestrated by a management system that provides high availability through PM2.

The rest of this article is organized as follows. Section 2 describes the related work. Section 3 presents the theoretical framework for this study. Section 4 explains the design and implementation of the management system for the microservices architecture. Section 5 provides the results. Section 6 discusses the findings. Finally, Section 7 closes with conclusions and future work lines.

2. Related Work

In 1990, Knister and Prakas dealt with distributed collaborative tools for text editing [12]. They developed a toolkit called DistEdit. This tool provided a set of primitives for collaborative text editors, without dealing with distributed problems such as communication protocols and fault tolerance. They tested this approach by modifying two editors: MicroEmacs and GNU Emacs. The resulting editor allowed users to reach changes simultaneously while other users were able to observe those changes. In a similar context, Gianoutsos and Grundy in 1996 [13] developed W4, an extended WWW browser, that provided a variety of computer-supported cooperative work (CSCW) extensions for collaborative document editing. These extensions included synchronous and asynchronous communication mechanisms (i.e., text chats, notes, and collaboration messages). It also involved the WYSIWYG editing software (i.e., What You See Is What You Get) and collaboration editors such as whiteboards and text editors. Existing WWW pages were annotated with URL links, notes, text chats, brainstorming sessions, and whiteboards. These annotations and the shared documents for whiteboards and text editors were stored in a GroupKit conference and, therefore, were independent of HTML documents. In addition, users were enabled to join and leave a W4 meeting at any time.

Preeth et al. [14] evaluated the performance of containers used in Docker versus the performance of virtualized host systems. To accomplish such a task, the authors ran tests on memory usage performance, CPU consumption, CPU number, disk usage, boot time, and others. In most aspects, Docker performance was more efficient, since it decreased the use of resources and obtained improved results in availability, usability, response times, adaptation between several programming languages, and ease of use. Our research compares CPU, RAM, and network consumption produced by a Docker container versus a bare computer machine, while Bonnie ++ used it for stress tests. CPU and RAM consumption were also determined. Stress tests were not performed automatically because actual users accessed one shared document and edited it at the same time. Later, performance levels and accessibility were yielded with cAdvisor, which demonstrated the benefits of using containers.

Among research works that aim to allow multiple users to edit text documents in real-time from multiple devices collaboratively, the study proposed by Gadea et al. [15] presented the architecture and implementation of a text editor so that various users collaboratively edited rich-text documents in real-time from multiple devices that used microservices. The architecture used Docker containers to allow the development and testing of individual services as separate containers, which allowed a smooth implementation in the available network of computers and other computing devices. The system demonstrated how microservices allowed multiple users to co-edit a document in which images containing faces were added and recognized as part of the document content, supporting the creative process of the document. Compared with our proposal, this research also used Etherpad. Nevertheless, the differences are very noticeable. Although the authors talked about architecture implementation, they approached it quickly and gave more interest to the collaborative text editor

Etherpad's solution, where they added some functionalities. Instead, our study details each component of the architecture and the role it plays within it. In the same way, the authors did not elaborate on any system for the management of users; that is, they left security aside. It is worth mentioning that Etherpad by itself does not have a management system, which means that anyone can access any shared file. We provide in our solution a management system that is in charge of user management, pad management, and container management. Our management system is new and innovative since, so far, not even the official Etherpad site has implemented one. Making a management system for Etherpad is a difficult task to perform since it is not enough to connect users to a container. However, it must know from which container a user should be redirected because Etherpad session handling is developed with socket.io. Another notable difference is that the authors did not obtain or show any metrics. Their only numerical results stated that 80 users were able to connect, and the text was synchronized correctly. However, there was nothing that quantified the achievements. Therefore, it could not be established whether it improved on something compared to a monolithic architecture. In our research, results are quantified and compared with a monolithic architecture, providing the pros and cons of the proposed architecture.

Quang-Vinh and Claudia-Lavinia [16] performed concurrent tests with collaborative tools such as Google Docs and Etherpad. According to them, Etherpad lacks allowing more than ten users to connect at the same time. In our proposal, however, we procured 47 actual users connected concurrently without any inconvenience. Moreover, the available users limit such numbers at the time of testing instead of Etherpad's performance. As Etherpad became a microservice, its performance improved. Their results demonstrated that this software was not ready for collaboration activities at a significant scale since new users' connections were rejected. Furthermore, delay problems appeared when the system needed to deal with an increasing number of users. For example, text writing and updating speeds were slowed down in the shared document.

Brodahl and Hansen [17] proposed the use of a text editor in academic environments. Google Docs and Etherpad Web 2.0 are tools that provide the opportunity for multiple users to work online in the same document consecutively and concurrently. In this case, they intended to examine the factors or practices that students manifested, in order to appreciate how relevant and useful these tools were, and also how these tools influenced them. Their main conclusions were that Etherpad and Google Docs facilitated new ways of approaching communication for different styles of collaborative writing work, as well as in different environments. However, the configuration in which the tool was used influenced the way students perceived its usefulness. Recommendations derived from students' perceptions of the success factors for using the collaborative writing tool included a variety of factors: The group size should preferably not exceed three persons. Students needed to be prepared for technical difficulties. They also needed to have a contingency plan. At last, they were required to agree in advance about the working method and the rules for commenting and editing each other's work.

Villamizar et al. [18] proposed, beyond a monolithic architecture and microservices, the use of cloud services for deployment purposes. This study was conducted using Amazon Web Services (AWS). They accomplished performance, methodology, implementation, operation, and process adoption tests. Results indicated lower costs in the AWS services based on the microservices architecture, in addition to the control and management of each service independently [18]. They evaluated the performance of a monolithic application versus an application based on microservices. The outcome displayed response time improvement and the use of AWS. In our research, we analyzed CPU and RAM consumption, given a certain number of users, and we implemented in our infrastructure leading the solution to the cloud.

Sunil [19] pursued the very same purpose of the current study. They converted a monolithic application into a microservices application. Nevertheless, they only dealt with theoretical research, lacking a specific software application. In our study, however, we implemented a collaborative tool using Etherpad. Compared with our study, the approach of the article differed almost wholly. In this case, a proposal was made for a "generic" procedure, which could be used to transform a monolithic

application into microservices. The article did not show quantifiable values to determine if there were real advantages or disadvantages. This was because the authors' methodological approach was very attached to literary and theoretical aspects. They mentioned the "agile methodology" as a suggestion. Their architecture was the same as found on official websites. Instead, our proposal shows the interaction of a practical and real case (collaborative systems). All this is thanks to the implementation of a management system that allows adapting and taking advantage of microservices.

Stubbs et al., in Equation [20], proposed a framework to implement a distributed architecture based on microservices using Serfnode and Docker. Nevertheless, there was no evidence of any interface or an additional application as a proof of concept. Brogi et al. [21], suggested a Docker Analyzer as a solution to customize images and configure the architecture according to its needs. The difference from the current study is that we customize the Etherpad image through Docker build and control; also, the deployment of containers is monitored and controlled by our management system. Saha et al., in Equation [22], developed a tool that monitored and controlled Docker containers, "Clusters of Messos," in such a way that users were notified by messaging about any infrastructure issue. This functionality will be implemented in future works of our research. Kuroki et al. [23] proposed the implementation of an orchestrator that dealt with application deployments according to the users' requests.

Qingfeng et al. [24] designed a container-based anomaly detection system (ADS) to detect and diagnose anomalies in microservices and monitor and analyze performance data in real-time from them. The proposed ADS consisted of a monitoring module that collected performance data from containers, a data processing module based on machine learning models, and an integrated fault injection module to train these models using machine learning techniques. Specifically, its ADS relied on performance monitoring data from anomaly detection services for container-based microservices, machine learning algorithms to classify abnormal and healthy behaviors, and the failure injection module to collect performance data on various system conditions. Instead, in our study, we focus on the design of a central user management system, which creates instances and continuously interacts with containers of the Etherpad tool. This is because Etherpad currently works without authentication or user management, in such a way that it works as a service using containers and giving better use to hardware resources.

Gedia and Perigo [25] measured the performance of a network function virtualization (VNF) that ran on container platforms and also on virtual machines (VMs). The authors used the ONOS software defined networks (SDN) driver as a network feature in a container and VM format and compared the performance metrics: CPU, memory consumption, performance, and VNF provisioning time. Identical tests were performed in a bare server environment. The authors hoped that the results of this research would help network operators identify an optimal platform to host VNF services that are agile and profitable. Performance tests conducted revealed that a containerized ONOS VNF outperformed a VM-based ONOS VNF in terms of memory consumption, performance (intra-node), and provisioning time. The framework in this research demonstrated that Grafana, Prometheus, cAdvisor, and Node Exporter could serve as a unified platform (container and VM) for VNF orchestration and monitoring. Compared to our work, we use cAdvisor, which is a Docker container advisor that allows us to monitor in real-time the use of resources and the characteristics of each running container and the system in general. cAdvisor shows complete histories of processor activity, memory consumption, and network usage, among other things, which evidences the efficient performance of our proposal.

All previous studies were extraordinary efforts. However, there are additional features to our proposal. We use microservices, and our management system handles the services' orchestration. This deals with the creation of instances for each user and redirects the requests accordingly. An orchestrator is not implemented per se. Instead, we implement a management system that behaves like a container orchestrator.

3. Theoretical Framework

This section briefly addresses the conceptual elements in the current research. Figure 1 is a general representation of the components. It emphasizes why they were chosen and how each concept contributed to this research.

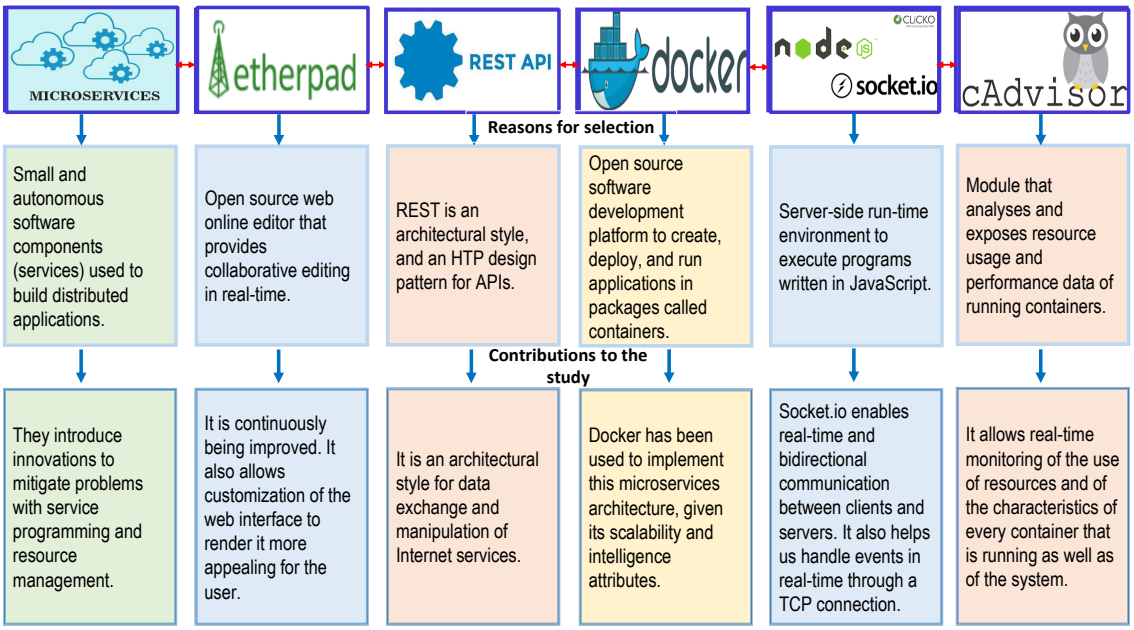

Figure 1. A visual representation of the systematization of the fundamental concepts used during the development of this research.

3.1. Microservices

Currently, many web applications provide dynamic experiences to users. A clear example is when users are capable of editing texts collaboratively in real-time from several devices. For this purpose, architectures have evolved into microservices, in part due to the low levels of coupling and decoupling they offer. Furthermore, through microservices, several users are enabled to co-edit a document simultaneously without the need for the presence of very robust hardware. The principle of using microservices within this research is to have small components, which work autonomously, with the characteristic of being able to communicate with each other [15].

In the context of this study, we selected a microservices architecture adaptation since it significantly reduces development time and adapts to market changes, which allows customers to deliver products with adequate quality. Furthermore, it introduces innovations to mitigate problems with service programming and resource management. This architecture is based on small services that individually are focused on achieving a specific goal and have a unique responsibility. These services are highly decoupled and are configured in such a way that they fulfil user requirements [19].

Therefore, the containerization was performed employing Etherpads by separating the database from the containers to the database engine. For this, the administration application performs the requests for user creation, execution, and container administration. All these were created according to containers on demand.

On the other hand, the reason why we chose microservices is that they significantly reduce the use of hardware, since they allow reducing the number of nodes that will execute the collaborative tools. Hence, the number of nodes corresponds to the number of servers that have an image of the operating system created for the Etherpad application. Etherpad is executed via REST Web Service commands from the administration service to the application servers (nodes), thus starting the image of the application. All these nodes are independent and perform specific tasks, and a management

system controls them. This system, per se, is also a microservice. Another remarkable contribution that microservices provide is the scalability and availability within any proposed architecture [26].

We propose the adaptation of a monolithic architecture (see Figure 2) to a microservices architecture (see Figure 3) based on the following arguments: (1) It is an architecture style that defines and creates systems through the use of small, independent, and autonomous services aligned closely with business logic [19]; (2) its features include: speed, quality, agility, and scalability; (3) its exceptional ability to be executed in the cloud; (4) its ability to use each resource appropriately, thus avoiding capacity problems; (5) each service can run speedily, with high fault tolerance and less complexity; and (6) the Docker platform allows the creation of several containers, each taking place with a specific service. These benefits will make this architecture the perfect fit to replace the monolithic architecture [27].

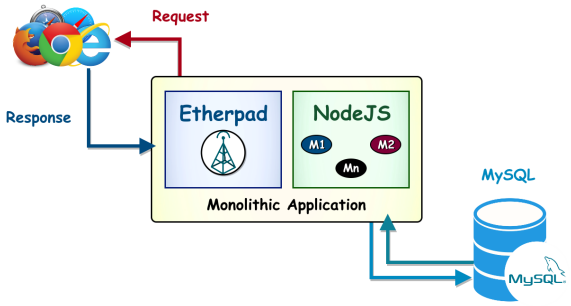

Figure 2. Schematic representation of the monolithic-based architecture. Adapted from Reference [19].

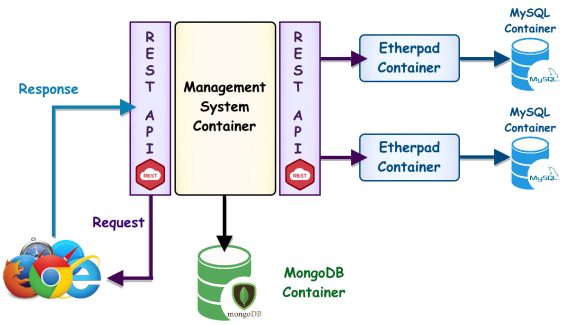

Figure 3. Schematic representation of the microservices-based architecture. An adaptation from Reference [19].

3.2. Etherpad

Etherpad is a collaborative real-time, open-source text editing web tool that is continuously being improved. In the scope of this research, we see that Etherpad allows researchers to make modifications depending on their needs. It also lets customization of the web interface to render it more appealing for the user. Etherpad is developed in JavaScript.

Etherpad allows several users to write text in real-time. If a user modifies or adds the text in a file within Etherpad, the other users can see the changes and contribute to the editing process. Etherpad owns an online chat that allows users to share their ideas with each other. It handles file versioning and enables us to view all the changes from the file creation [28]. In this research, we propose to use Etherpad in a different way, that is a Docker-based microservices architecture, which is customizable and can modify the specific connection and operation properties.

From the academic point of view, the use of online and collaborative tools gives enhanced feedback to students. It is intended to improve academic performance, and it may be used as an option in pedagogical design [17,29].

3.3. Rest API

Representational state transfer (REST), also known as the REST API architecture, is an architectural style for data exchange and manipulation through web services. Its interaction is usually based on the HTTP protocol, mainly when it is used for web APIs [30], i.e., developers do not need to install libraries or additional software to adapt to a REST API design. A REST application can be any inter-system interface that uses HTTP to obtain data or execute operations on that data in formats such as plain text, XML, and JSON [31]. It is also an alternative to other standard data exchange protocols such as the simple object access protocol (SOAP). Additionally, REST handles a more straightforward data manipulation solution.

In our research, the REST API is used to allow external users to access the description of images stored locally due to its microservices architecture. The leading research service enables users to add, delete, update, and search for descriptions of pictures stored in the database through a REST API. Therefore, there is no need to be in the same device of the database. However, with a request, it is able to perform any service that it provides, and thus, it is capable of visualizing the information [32].

3.4. Docker Containers

The use of Docker is helping in different types of applications and architectures. Thanks to Docker, it is possible to improve costs, performance, and capacity, among other issues. Its implementation has a database image repository, operating systems, and several applications. It also allows the elimination, modification, and creation of new images, reducing hereby the time of use of each service required in different containers with their respective IP addresses and several open ports [33].

Docker was used for implementing this microservices architecture because of its scalability and intelligence characteristics [34]. If it detects a running node that stops, it can start it up immediately, and the end-user is not affected by errors in the system. This is possible thanks to the implementation of a microservices cluster. In this way, it is evident that vulnerabilities have nothing to do with the number of connected users, highlighting that the user may create one or more documents within the same container independent of the sharing they may perform. In addition, Docker is able to run in the cloud. However, for Docker to work in this way, it is necessary to have an orchestrator that can create services according to the need of the architecture or the supported workload.

This current study focuses on providing each user a container that runs the Etherpad application, so all their pads are inside their container. This would provide independence from other users' containers, just as described in the principle of a microservices-based architecture. Therefore, if there is an issue with the container of a specific user, it will not affect other users, and they are able to continue editing their text regularly. On the other hand, if it is performed in the traditional way (i.e., a monolithic architecture), it would generate a severe problem that would affect the whole system and possibly cause its complete failure.

3.5. Socket IO

The Socket IO is a Node.js library that enables real-time and bidirectional communication between clients and servers. It also allows handling events in real-time through TCP connections. It helps avoid compatibility problems between computers. This library uses sessions. Thus, every user who is in that session can interact and exchange information [35].

In this study, Etherpad operations are based on the socket.io library. A significant difference emerges, as for each pad, Etherpad creates connection rooms, and within such rooms, it provides sessions. This guarantees the execution of several editing pads at the same time. It also prevents users from losing control and staying in the pad they want to edit. We applied rooms and sessions with socket.io themes. However, this resulted in a limitation, since it is improbable to run Etherpads to perform in a distributed style. While this happens, the program loses control of the sessions and stops working as a collaborative tool, becoming a simple online individual text editor.

3.6. cAdvisor

The containers advisor (cAdvisor) runs in a container, and it uses the Docker Remote API to obtain statistic values [36]. It allows real-time monitoring of resources use and of every running container characteristics, as well as of the system. It displays a complete history of processor activity, memory consumption, network usage, etc. Google developed this container and used it as the primary tool to monitor architecture resources (see Figure 4).

Figure 4. The cAdvisor interface management system provides users with an understanding of the use of resources and the performance features of their running containers. It is a running daemon that collects, accumulates, processes, and exports information about container execution.

Within the context of the current research, we developed a management system that allows a high-level visualization of computer network nodes, Docker containers, Kubernetes, and Linux Containers that are executed. This enables controlling containers, as well as deploying and migrating applications in real-time without losing connectivity, obtaining thereby complete control. This results in improved portability, security, and automation [37]. Additionally, our container management system reduces operating costs, providing agility in their life cycle service management. This system offers options for creating, repairing, and eliminating a network service by employing a user-friendly interface. It also helps in reducing the time of execution of each task with automatic processing. All the characteristics mentioned above are additional issues, which are beyond the main scope of the current study [23].

4. Materials and Methods

This section describes the implementation model of the proposed microservices architecture. Subsequently, it represents both the design and the implementation of the central management system, as well as the algorithms for its operation. It also describes the creation process of the container image and the deployment of the algorithms. It finalizes with the proofs of concepts that were applied to this architecture.

4.1. Proposed Microservices Architecture Design

According to [38,39], a microservices architecture is an approach for building a server application that consists of a collection of small autonomous services. In our case, there was a unique database and several containers that accessed that database, to obtain a microservice (logic + data). Figure 5 illustrates the architecture layers, the components, and tools. This model was based on the microservices architecture style [39]. It consisted of a collection of small and independent services. Each service was self-contained and should perform a single business logic action. A brief description of every layer and its components will follow further below.

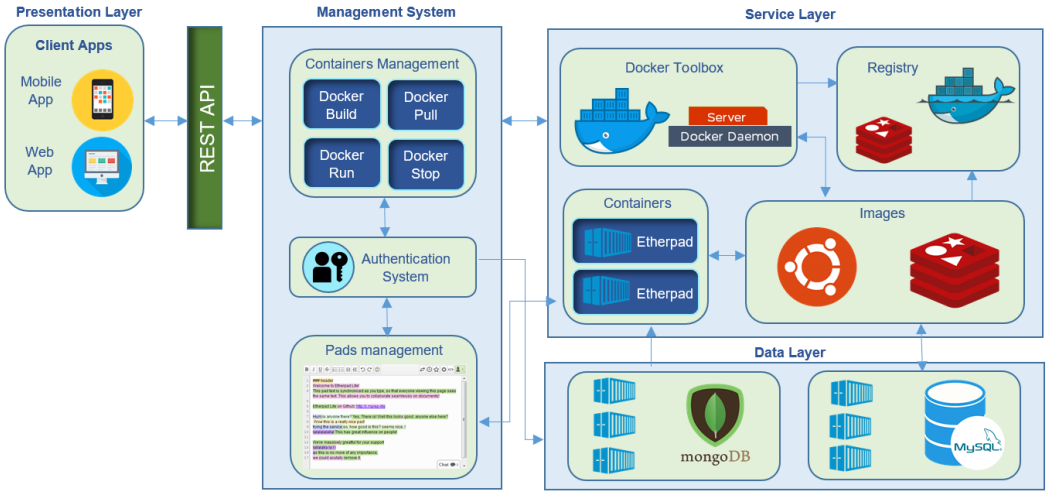

Figure 5. The proposed microservices-based architecture [39].

Figure 5 presents the elements of our proposed architecture. In the presentation layer, there is a responsive web interface that can be visualized from any device or equipment (computer or phone) with a web browser. These devices are connected to the management system through a REST API using HTTP methods. The management system is in charge of performing all the actions for the correct operation of our software, and this, in turn, has three modules:

- Container management module: This module is in charge of carrying out the requests to unload, build, execute, and stop containers. It is in charge of deploying a container when a user connects to manage a pad. This module has a direct connection to the service layer, which executes the instructions sent by it;
- Authentication module: This module is in charge of the logical creation, edition, and elimination of users. It is also responsible for managing users' sessions;
- Pad management module: This module allows users to edit, create, delete, and share pads with other users.

The service layer is responsible for providing the Etherpad containers that the container management module requires. In this layer, basic components of Docker are handled. In addition, the custom images of the Etherpad containers are hosted. The data layer allows the information to be stored persistently. This data layer sits on top of containers in order to facilitate deployment and minimize server resource consumption.

Since this architecture is running on containers, scalability is conducted using Docker Swarm. Docker Swarm is a native Docker application that allows easily and quickly creating container clusters. Basically, the number of containers is configured for the cluster, and replication is enabled. In this way, Docker Swarm automatically delegates to a master node to orchestrate the work of the slave nodes. Similarly, it is responsible for load balancing within the worker nodes. The potential advantage of this Docker application is that it is able to scale an application in 2, 10, 50, or more containers. Based on the aforementioned functionality, we can tell that the scalability and high availability characteristic of our architecture is covered.

Concerning availability, the management system was developed in NodeJS, where many instances of running threads use a single process [40]. To make the most of the hardware resources, we used PM2. PM2 is a process manager that allows a system always to be online, and we used it in cluster mode so that the application scales to all the available CPUs, without the need to modify code. A NodeJS cluster ensures that multiple child processes run multiple instances of the server; it also provides a load balancer embedded within the cluster mode. In this way, scalability was achieved at the CPU level according to the workload administered by the management system. Furthermore, it contributes to

error handling and the relaunch of the applications in the event of a crash, i.e., it provides availability for the users.

4.2. Design of the Etherpad Instances Management System

The management system is located in the Service Layer. It implements the Observer, the Proxy, and the Factory method patterns from the GoFDesign patterns book [41]. We also applied the Service Layer pattern [42,43] throughout the full architecture of our solution. The management system is responsible for creating Etherpad nodes accordingly: Observer + Factory Method, orchestrating existing and new nodes: Proxy, identifying faults: Observer, rebalancing services: Proxy, editing documents: Service Layer, and managing user authentication and sessions: Observer + Proxy + Service Layer [38]. Every resource is a microservice running on a Docker instance. The interaction between the management system and the Etherpads is achieved through the HTTP protocol. Clients do not call services directly; instead, they call the API GATEWAY: Service Layer, which forwards the call to the appropriate service in the backend. After a set of responses from several functions, it is possible to return an aggregate response. In other words, the API GATEWAY allows the calls to execute tasks such as authentication, registration, and user logging, as well as the creation of documents, everything through one service. When Docker resources are full, the management system will create a new Docker instance for balancing existing and new Etherpad nodes resources. In a further study, the architecture will be optimized by utilizing a load balancer for adjusting the number of concurrent sessions to the system [38]. The management system stores user information and the container port assigned to the user in a MongoDB database microservice. MongoDB does not execute replicas. Instead, replication is accomplished by Docker Swarm, allowing programmers/users to manage instances and delegate tasks according to the needs of the application being executed [39]. As stated before, when a set of Etherpads is executed in Docker instances, and the current container is filling its capacity, the management system creates a new container and rebalances the load in such a way that it splits current services (Etherpads) among the previous and the new docker instance. Concluding, our software architecture design implements a combination of the Observer, the Proxy, the Factory Method, and the Service Layer design patterns, breaking the undesirable coupling and initializing objects when needed. The advantage of this design is that the API GATEWAY manages client requests and abstracts every functionality; it also monitors and manages RAM and CPU consumption for allocating new recourse nodes (Etherpads) or new resource containers (Dockers).

Figure 6 illustrates a not actual, but a representative class diagram of the management system. Even though we are not using classes, we consider that the following explanation can be mapped to the components and functions that our management system implements: The REST API receives the request from the client application. This request reaches the "Service Proxy", where depending on the type of request, it redirects it to the appropriate services. The services, in turn, are connected to "Create", which works as an object factory. These objects can be Containers, Pads, and Users. A user can have multiple pads, while a container has various pads on it. The "Observer" design pattern is used to implement the monitoring of container and pad states. By monitoring the containers, it is possible to determine when a container stopped executing a service in an unscheduled manner and to deploy immediately another container. By monitoring the pads with the "Observer", it is possible to determine when a user accesses a pad and the total number of users connected to that pad. Likewise, when an unscheduled stop happens, it immediately deploys a new pad. On the other hand, the management to access the system is given by the "SessionService", which is responsible for authenticating the user and creating a session when the user enters the system. Furthermore, it terminates a session when the user leaves the system.

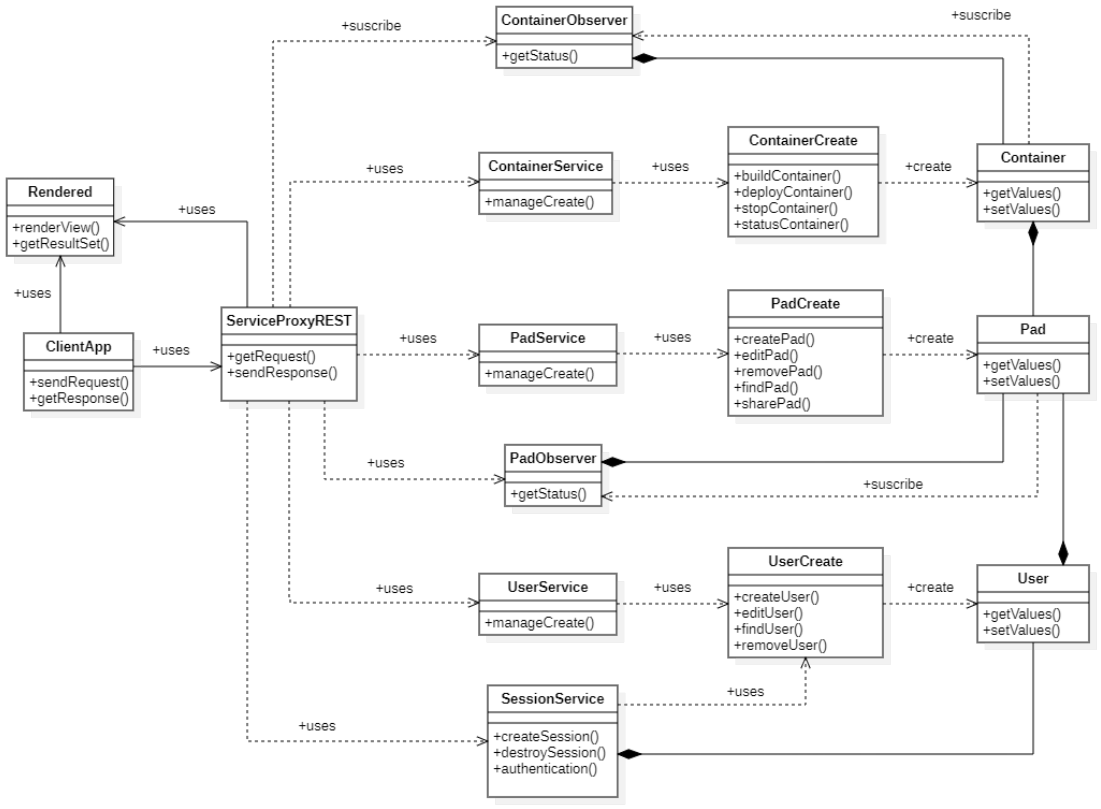

Figure 6. Class diagram of the management system.

4.3. Implementation of the Etherpad Instances Management System

The management system is responsible for establishing Etherpad instances according to users' creation and their logins into the system. Users' data and ports assigned to use the correspondent Etherpad container are stored in a MongoDB (a NoSQL database) [7]. We used Pug for the graphical user interfaces. Pug is a template engine, the middleware and routing solution for Node.js [44]. The management system runs once the creation of users is requested and executed since the Etherpad container deployment is performed for the specific created user. The container configuration is the same for all users, except for their container ports. Port numbers are determined consecutively. Instead of redirecting when the user wants to enter a pad, this interface of Etherpad is embedded in the management system layout (scheme), since Etherpad has its graphical interface for document management. Hence, users may edit their documents without leaving the management system domain. Nonetheless, Pug is designed for the administration layer, and subsequently, it is linked to the Etherpad interface and is not part of that layer (Pug). The following algorithm (see Algorithm 1) demonstrates how the edition of pads was implemented within the management system. The management system layout has a menu bar that allows users to access all their pads, configure their data, and close the current session.

Algorithm 1 The edition of pads within the management system.

```
extends layout
 block title
       title Etherpad ESPE - Pads
       -statusDocker = 'stopped'
if config.running
       - statusDocker = 'running'
 block content
       style.
       # content {
       padding: 0px; }
       }
       iframe (style="border:none; width:100 %; height : calc(100vh-10px)"
       src='http://$config.host : $cofig.port /p/$pad')
```

The main advantage of our management system is its integration with Etherpad, which does not include a user management module. This provides users privacy over their documents, overcoming Etherpad's problem of free/open use of the pad's URL. In this way, users can share the pads with the users they want. It should be noted that the entire management system works through REST. For example, POST is used to change user's passwords, while PUT is used to enter the system, and GET is used to respond to the menu bar options selected by the user (see Algorithm 2). Furthermore, another source of scalability is the safe operations (GET), whose responses can be cached by the intermediary proxies to enhance performance [45].

Algorithm 2 Operation of the management system.

```
doctype html
  html
head
  include head
  block title
body
  #page
  #na
  #navigator
       p.title ESPEpad
       p.small.text-center # {user.name }
       nav
       a (href='/' class=navPads) # [i.fa.fa-book] Pads
       a (href='/config' class=navConfig) # [i.fa.fa-cog] Config
       a (href='/logout') # [i.fa.fa-sign-out] Close session
  #content
  block content
  block scripts
```

Figure 7 is a visual representation of a set of rules and actions that involve the Managementoperating system. The idea focuses on being able to provide each user with a container, which contains the user's documents. If the document is not of their own, the system will search the document's author container and redirect it to the specific port. It is also responsible for user authentication and management, creating instances, and regularly interacting with Etherpad containers. The whole process works on Docker containers.

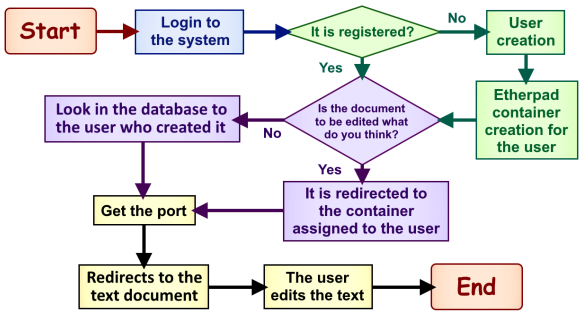

Figure 7. Management system flowchart.

The process starts when the user enters the management system. If the user is already registered, he/she must log in with their credentials. Otherwise, he/she needs to register first, and the

management system will automatically assign a port number. Once the user is logged into the system, the user may create an editing space (pad). That is, if the user wants to create a new pad, the management system will search for it in the database. When it finds a match by name, this means that another user is running it in another container. If not, it will redirect to the container port that was assigned when registered. If the user wants other users to edit the text concurrently, the user needs to share the link containing the designated port, i.e., when it receives an API call and the access URL to the created pad is generated and thus may be shared, the Gateway executes this. This procedure also explains how the management system processes its elements and how they perform, i.e., inputs, requirements, conditions, outputs, and their relationships.

4.4. Creating an Image from a Container

We used a Docker command-based script to create the Etherpad container image. Once the image is ready, it is executed in the container. We used a Debian 9.0 image since it has network connectivity characteristics and allows the interconnection with the rest of the nodes that are part of the proposed architecture [46]. In addition, the script contains an entry point that is able to modify specific characteristics of the container before it is created; for example, the port and information needed for the database connection.

Docker built is a tool that aims to create custom images. It is able to load components that the developer wants, and it also configures the network, storage, and user permissions. In addition, the user can execute commands that allow the container to be seen as a service. The following script (see Algorithm 3) contains some tools that are installed for the execution of the Etherpad container such as Debian OS, Node.js, Git, curl, MySQL-client, and software properties. Finally, the Etherpad that has been modified for this project is downloaded.

Algorithm 3 Etherpad container creation.

```
1:  FROM debian
2:  RUN apt-get update && apt-get –y install git && apt-get –y install curl && apt-get –y install nano curt unzip mysql –client
3:  RUN apt install –y curl software-properties-common gnupg
4:  RUN curl –sL https://deb.nodesource.com/setup _6.x | bash -&& apt-get install –y nodejs
5:  WORKDIR /etherpad/etherpad-lite
6:  RUN bin/installDeps.sh && rm settings.json
7:  COPY  entrypoint.sh / entrypoint.sh
8:  RUN sed –i's/^ node &exec node/' bin/run.sh
9:  RUN chmod g+rwx, o+rwx –R
10: VOLUME  /opt/etherpadlite/var
11: RUN  ln –s var/settings.json settings.json
12: ENTRYPOINT ["/entrypoint.sh"]
13: CMD ("bin/run.sh", "==root")
```

4.5. Deployment of Containers

The containers are deployed by the management system. Once a new user is registered, the user is granted a container for his/her documents (see Algorithm 4). As shown in this code, the deployment is performed when a new user is registered, and a new container is assigned. The default variables used by Etherpad are those starting with ETHERPAD_. In this case, the credentials of the database in which the text of each pad of that user is stored are sent as a parameter to the container. The default internal port of the container is 9001, while the external port uses a function that increases its value (sequencer) by one each time a new container is created. Finally, the container is executed.

Algorithm 4 Deployment of Etherpad containers from the management system.

```
1:  debug ('creating Etherpad docker')
2:  a let dockerConfiguration = {
3:      Image: 'etherpad-lite',
4:      Env: [
5:      'ETHERPAD_DB_HOST=$ {etherpadConf.url}',
6:      'ETHERPAD_DB_PASSWORD=$'{etherpadConf.password}',
7:      ETHERPAD_DB_NAME=$ {user.username}',
8:          ],
9:      ExposedPorts: {9001/tcp': {}},
10: HostConfig: {
11: PortBindings: {
12: '9001/tcp : [{ HostIP: ' ', HostPort : nextPort . toString() }
13:          }
14:      },
15:      name: user,name
16: }
17: then (container => {
18: debug ('starting container')
19: return container.start()
20: }]
```

4.6. Proofs of Concept

Proofs of the concept were performed on an Ubuntu 16.04 cloud server. Users accessed the service from Windows10 desktop computers using Google Chrome and Mozilla Firefox. Additionally, users needed to be registered prior to entering the system. The tests were performed when traffic was similar to the daily average. The download speed was 5.75 Mbps, and the upload speed was about 5.01 Mbps. These tests were performed using the virtual servers' provisions from the Cloud Computing infrastructure of the Universidad de las Fuerzas Armadas ESPEin Ecuador, i.e., Infrastructure as a Service (IAAS). This allowed researches to install and deploy different computer services that were accessed through the University's LAN/WLAN/WAN. These resources were used for data processing, storage, and backup in an HP Blade Infrastructure, a Citrix XenServer Hypervisor, and an EMCbackup system. This layer was used for the deployment of the created containers that executed the Etherpad collaboration tool, i.e., Software as a Service (SAAS). We describe in more detail the different proofs of concept:

- User's creation: We used actual users who created accounts so that we were able to observe the behavior of container creation for different users.
- Collaborative text edition: We obtained some metric values related to server behavior. These metric values were collected when the users were editing their collaborative text.
- Edition of a shared document: In this test, we captured data related to the edition of one document accessed by several users simultaneously. Therefore, it was possible to determine whether the proposed architecture supported concurrent users without long delays.
- CPU and memory consumption: The performance and consumption of server resources and containers were monitored in real-time using cAdvisor. The cAdvisor container ran on the same servers where the entire microservices-based architecture was deployed.

At the end of the proofs of concept, it was possible to get interesting outcomes of the correct operation in the automation of the container's creation and its collaborative text edition. These results are shown in the next section.

5. Results

5.1. Results Obtained When Running Etherpad on a Microservices Architecture

As explained in Section 3.6, we used cAdvisor to collect our experiment metric values. This software analyzes and exposes the use of resources and performance data of running containers [47]. As a result, we obtained the necessary parameters that were used for a detailed analysis. This information was collected from a total of 47 users. The tests were performed using desktop computers, with an Intel Core i7 processor and 4GB of RAM. It is worth noting that all computers were connected physically to the University. We used Google Chrome to perform the tests. We performed two tests,

the first test in a scenario focused on the creation of one pad, while 47 users accessed that single pad. The second scenario was based on the creation of eleven pads with different users, so it revealed the consumption of CPU and RAM. Table 1 lists the percentage of the initial and final use of CPU, RAM, and file system consumption in the first scenario.

Containers ran within the same infrastructure comprising two virtual servers with the following characteristics: Intel Xeon Family VI 2.55 GHz OctaCore processor, 12 GB RAM, and 20 GB HDD storage. The service management layer managed the execution of containers.

Table 1. CPU, memory, and file system consumption, before and after the tests for 47 users in one only pad.

Resource	Initial Consumption	Final consumption
CPU	1%	2.2%
Memory	56 MB	160 MB
File System	39	40

Figure 8 illustrates the CPU consumption when 47 users accessed a single pad. The initial condition increased by 1.2% of CPU, 104 MB of RAM, and 1% of the file system. Further, Figure 9 illustrates the network use when 47 users accessed a single pad. This was evidence of the fact that, even though we had 47 users connected through sessions, the exchange of messages for the connection remained constant.

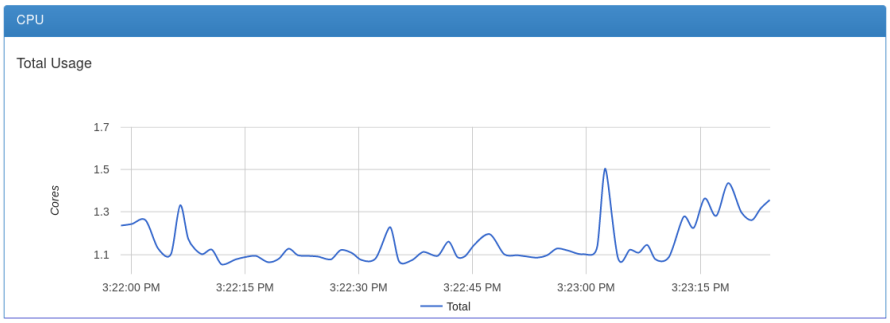

Figure 8. CPU consumption of 47 users on one pad.

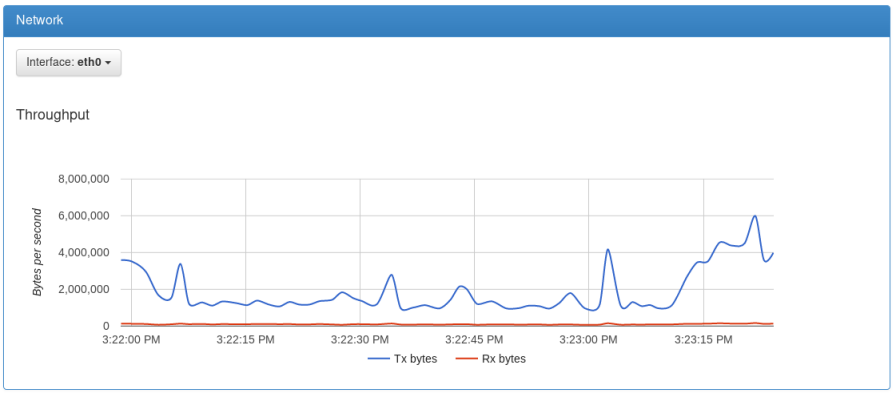

Figure 9. Network throughput of 47 users on one pad.

For the second scenario, where one pad was created for every user, i.e., 47 pads running on 11 containers independently, we collected CPU, RAM, and storage consumption values per created user, as listed in Table 2.

Table 2. Consumption per created user.

Resources	Consumption
CPU	1.2%
Memory	104 MB
Storage	1048 MB

When sizing a new service, it is also essential to be clear about the expected performance of the file system. File systems are designed to optimize data availability, efficiently manage scalability for large volumes of data, and ensure their integrity. Figure 10 demonstrates the analysis of the performance and resource use of file systems using microservices. As noted in the third section, cAdvisor was integrated into Kubernetes to calculate the metrics related to the file system information.

Finally, we could tell that, as CPU and memory consumption are central factors in the performance of a system, the results revealed how a container may be able to group the information of both resources. In this case, after automatically discovering all the used containers and sub-containers, as well as collecting the data, cAdvisor presented the best CPU (2.2% and 1.2%, respectively) usage and the lowest memory consumption of the sub-containers (104 MB, 104 MB, respectively). Therefore, Docker was able to provide a better performance using fewer memory resources. Its best performance was when the application ran without virtualization.

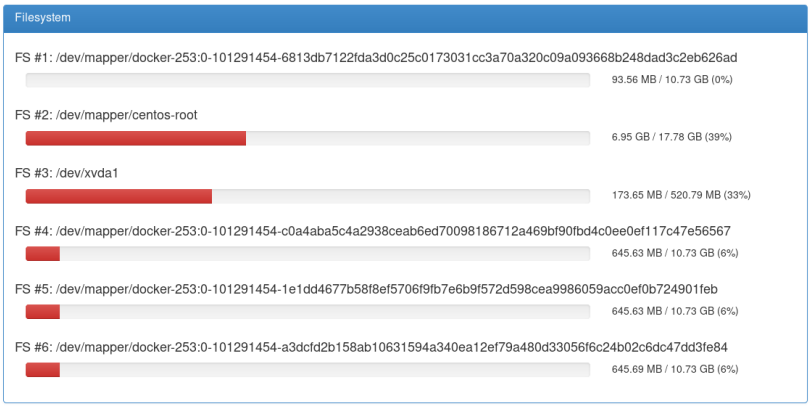

Figure 10. File system performance of 47 users on one pad each.

5.2. Comparison of Results When Running Etherpad on a Monolithic Architecture

To compare the results from our research, a proof of concept was performed by running the collaborative Etherpad tool on a monolithic architecture. In particular, Etherpad was executed directly on a virtual machine running the NodeJS development environment. We used Dynatrace [48] to capture the Etherpad performance data running on the monolithic architecture; this software allows the management and monitoring of infrastructure, as well as applications in the cloud. It is ideal for measuring the performance of applications developed in NodeJS. Despite being proprietary software, it provides an optional free 14 day trial. We only ran the scripts found in the official software documentation, and we were able to see the captured data immediately.

It is essential to note that the tests performed were the same as those used with Etherpad in the microservices-based architecture. Table 3 shows the results. In the first scenario that consisted of the access of 47 users to the same pad, a memory consumption of 174 MB of RAM was achieved compared to some 104 MB of consumption of the architecture based on microservices. This meant that with the proposed architecture, we obtained a drop of about 40.22% of the RAM consumption. In terms of CPU consumption, the monolithic architecture used 0.65% compared to the 1.2% used by the microservices-based architecture (see Figure 11).

Table 3. Comparison of the results obtained by executing Etherpad on the monolithic and microservices platforms.

Resource	Scenario1 (47 Users in Only One Pad)		Scenario 2 (Consumption per Created User)	
	Monolithic	Microservices	Monolithic	Microservices
CPU	0.65%	1.2%	2.33%	56.4%
RAM	174 MB	104 MB	126 MB	4888 MB

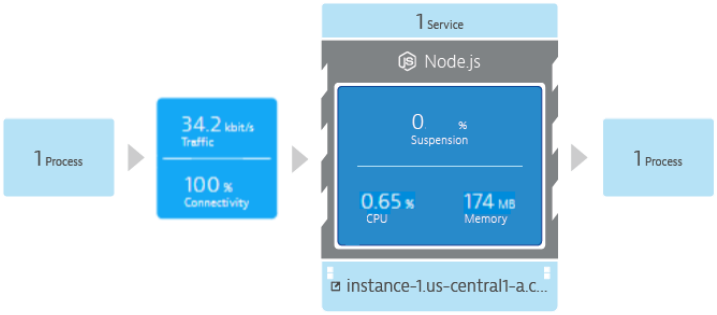

Figure 11. Monolithic performance of 47 users on one pad each.

In the second scenario, which consisted of creating eleven pads and allowing each user to access the pad he/she created, the monolithic architecture consumed 126 MB of RAM, while the proposed architecture with microservices consumed 4888 MB of RAM. In terms of CPU consumption, the monolithic architecture used 2.33% of the CPU, while the microservices-based architecture used about 56.4% (see Figure 12). This notable difference in the second scenario as based on the fact that not only a pad was being created, but also a Docker container, which guaranteed independence between one container from another. This led to greater security for the proposed solution. In Etherpad, any user is able to access any pad that is created in the monolithic architecture, avoiding users from gaining privacy on their pads.

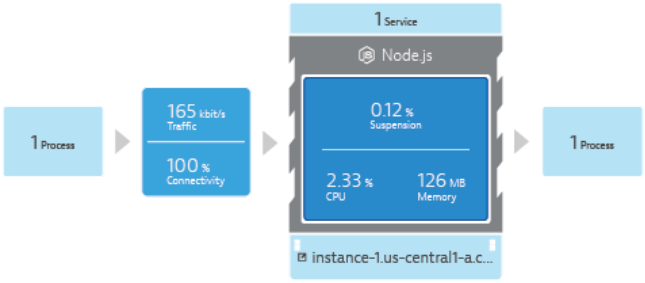

Figure 12. Monolithic performance in the second scenario.

However, it may be considered that the memory consumption in the proposed architecture could be reduced if the way to avoid creating a container per user is researched. In future work, and based on the current research, it is planned to implement a load balancer that deploys containers as needed, without losing the privacy and independence of the containers.

6. Discussion

Our proposed architecture was intended for the implementation of Etherpad based on microservices and a user management system to minimize the use of hardware resources, through vertical and horizontal scalability in any of its N layers. The design and implementation of a management system are essential when conducting studies related to microservices. Therefore, our software had the function of an administrator that allowed establishing a link between those services. Thereby, an end-user may control it, regardless of where it may be located. Likewise, it needed to

manage an internal process for navigation and use of an application, since each service, database, and applications may be deployed in different containers.

Our architecture was not different from other architectures. It was clearly established that we followed [38,39] designs. Our architecture, hence our design, moved from the monolithic instantiation and handling of Etherpads towards a services architecture, where every Etherpad offered a microservice. Our management system followed (implemented) three popular design patterns that together solve the following problems. These patterns can be found on the GoF Design patterns book [41]: (1) Observer pattern: "several components depend on a subject components' state, a dependent component should be informed about state changes of the subjects, loose coupling between dependent and subject component required, and keeping related objects consistent without tightly coupling them", (2) Factory Method pattern: "A class needs to instantiate a derivation of another class but does not know which one". (3) Proxy pattern: "creation and initialization of objects are expensive tasks, some objects do not have appropriate access rights, and we need different access rights to different objects, when an object is accessed by a pointer, additional action is performed by that pointer, it is difficult to create an object in a different space." [43]. (4) Service Later pattern: "The interactions may be complex, involving transactions across multiple resources and the coordination of several responses to an action. Encoding the logic of the interactions separately in each interface causes a lot of duplication." On the other had, the monitoring tools were only based on the Observer design pattern [41], and a monitor system does not make any decision; it only informs other applications about monitored objects' states.

The performance tests yielded that when 47 users were connected simultaneously to the same Etherpad, the memory consumption did not exceed 104 MB per user. However, CPU consumption increased by 16%. This represented that it did not affect the number of containers since they all accessed the same pad. In the second scenario, where every user had a pad, it increased by 1.2% CPU. In conclusion, with 82 containers, the CPU would become saturated, which was a limitation, unless more nodes were incorporated.

Similarly, the consumption of RAM limited the use of the system to a maximum of 54 concurrent users connected simultaneously to the same pad in real-time. In normal conditions, RAM consumption was about 56%. Not implementing a load balancer in our architecture was a limitation in improving the performance and optimization of resources use. This happened when Etherpad handled sessions within rooms, forcing all users to connect to the container in which the pad was created, causing the duplication of processes.

In addition, the proposed architecture allowed easy scaling, which was directly proportional to the number of connected users. This may become a disadvantage since a connected user did not necessarily need to represent an instantiated and running container. This was not optimal or viable when executing the proposal, taking into consideration the limitations of the Etherpad. This issue may be solved in future research, where the creation of containers will be optimized to the reduction of the consumption of both CPU and RAM by more efficient container management with the support of a load balancer that adapts to the operation of socket.io. The latter is another issue of conception and adaptability.

Finally, the current study evidenced how advisable it was to use this type of tool and what its benefits and limitations may be. It also produced information about the levels of satisfaction and usability, as well as the technical difficulties that appeared while correcting them. Furthermore, it aimed to identify the percentage of students who were enthusiastic about using collaborative tools, as well as those who may be frustrated by the technical difficulties that potentially arose. The main objective of this study was to implement a collaborative architecture based on microservices through the use of containers. The application of this prototype was tested in the laboratories of the Universidad de las Fuerzas Armadas ESPE.

7. Conclusions

In this study, we designed and implemented a three-layer-based microservices architecture that supported Etherpad, a collaborative tool. In the services layer, we developed a central management system, using containers assembled in the cloud so that we were able to deploy Etherpad instances in an efficient, scalable, and available way. This allowed several editors to edit a text document simultaneously and monitor all participants in real-time. The management system was responsible for establishing Etherpad instances according to created users that accessed the system. User data and ports of every Etherpad container were stored in MongoDB. For the graphical user interfaces, we used Pug. Instead of using re-addressing at the time the user desired to enter a pad, the Etherpad graphical interface was embedded in the management system layout. This allowed the users to enter the edition of their documents without leaving the domain of the management system. Besides, we were able to share the address of the pad so that other users may edit files simultaneously during online editing. The results demonstrated the functionality of the architecture, although with a non-negligible consumption of CPU and memory when adding new collaborative users.

As future work, we plan to improve our current architecture scalability in two ways: (1) using a NoSQL database for the management system, for user data storage, and the users' pads' names. Containers will be used more efficiently since it will not only cover pads, but also registered users and stored data. The microservices will run containers in the form of clusters employing Docker Swarm. The management system pages' design and style will be standardized using embedded JavaScript templates (EJS). The management system will also incorporate basic persistence storage operations (CRUD), as well as user authentication using the Facebook login API and SSL valid certificates through HTTPS. (2) We plan to implement a load balancer that will adapt to socket.io. This will allow an improvement of text editing in real-time. The load balancer will automatically deploy a new instance of Etherpad when the total of containers' CPU or RAM consumption exceeds 80%. Lastly, it will balance the load among the new number of available Etherpad instances. This will be performed for every new user request and for every time that a user leaves a pad, which will ensure scalability in the Etherpad instances' microservices layer.

Author Contributions: F.T. was the initiator of the project. Conceptualization: F.T., W.F., and J.E.L. identified the theoretical constructs and the different elements that represented the phenomenon, in academic terms. Validation: M.Á.M. and J.E.L. conducted the verification and validation such that the system complied with the requirements and specifications according to the intended purpose. Formal analysis: F.T. and M.Á.M. developed such analysis. Resources and funding: W.F. and T.T. obtained technological, 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: J.E.L. and 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 read and agreed to the proposed version of the manuscript.

Funding: The funding of this research is provided by the Mobility Regulation of the Universidad de las Fuerzas Armadas ESPE, from Sangolquí, Ecuador.

Acknowledgments: The authors would like to thank the technological support of the specialists of the Technologies of Information and Communication Unit (UTIC) of ESPE and also the scientific and collaborative contribution by the Universidad Autónoma de Madrid in Spain.

Conflicts of Interest: We ensure that our manuscript has not been submitted simultaneously for publication anywhere else, that it contains original data, and that the content of the paper has not been 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 obtaining permission to reproduce copyrighted material from other sources. There is no conflict of interest with any party; there is no problem with ethical standards of any kind.

References

1. Baškarada, S.; Nguyen, V.; Koronios, A. Architecting Microservices: Practical Opportunities and Challenges. *J. Comput. Inf. Syst.* **2018**, 1–9, doi: 10.1080/08874417.2018.1520056.
2. Cavallari,M.; Tornieri, F. Information systems architecture and organization in the era of microservices. In *Network, Smart and Open*; Springer: Cham, Switzerland, 2018; pp. 165–177.

3. Kwan A.; Jacobsen, H.-A.; Chan, A.; Samoojh, S. Microservices in the modern software world. In Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, Toronto, ON, Canada, 31 October–2 November 2016. Available online: https://dl.acm.org/doi/proceedings/10.5555/3049877?tocHeading=heading4 (accessed on 15 January 2020).
4. Di Francesco, P.; Malavolta, I.; Lago, P. Research on architecting microservices: trends, focus, and potential for industrial adoption. In Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden, 5–7 April 2017.
5. Al-Debagy, O.; Martinek, P. A Comparative Review of Microservices and Monolithic Architectures. *arXiv* **2019**, arXiv:1905.07997.
6. Amaral M.; Polo, J.; Carrera, D.; Mohomed, I.; Unuvar, M.; Steinder, M. Performance evaluation of microservices architectures using containers. In Proceedings of the 2015 IEEE 14th International Symposium on Network Computing and Applications (NCA), Cambridge, MA, USA, 28–30 September 2015 .
7. Ueda, T.; Nakaike, T.; Ohara, M. Workload characterization for microservices. In Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC), Providence, RI, USA, 25–27 September 2016; pp. 1–10.
8. Traifeh H.; Staubitz, T.; Meinel, C. Towards More Human-Centered openHPI Collab Spaces. In *Design Thinking Research* ; Springer: Cham, Switzerland, 2020; pp. 273–288.
9. Doberstein D.; Hecking T.; Hoppe H. Sequence patterns in small group work within a large online course. In Proceedings of the CYTED-RITOS International Workshop on Groupware, Saskatoon, SK, Canada, 9–11 August 2017; Springer: Cham, Switzerland, 2017; pp. 104–117.
10. Lascano, J. A Pattern Language for Designing Application-Level Communication Protocols and the Improvement of Computer Science Education through Cloud Computing. Ph.D. Thesis Dissertations, Utah State University, Logan, UT 84322, United States, May 2017. Available online: https://digitalcommons.usu.edu/etd/6547 (accessed on 1 April 2020).
11. Coulouris, G.; Dollimore, J.; Kindberg, T.; Blair, G. *Distributed Systems: Concepts and Design*, 5th ed.; Person Education Limted: London, UK, 2012; ISBN 13: 978-0-273-76059-7.
12. Knister, M.J.; Prakash A. DistEdit: A distributed toolkit for supporting multiple group editors. In Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work, Los Angeles, CA, USA, 7–10 October 1990.
13. Gianoutsos, S.; Grundy, J. Collaborative work with the World Wide Web: Adding CSCW support to a Web browser. In Proceedings of the Oz-CSCW 96, Brisbane, Australia, 30 August 1996; DSTV Technical Workshop Series; University of Queensland: Brisbane, Australia, 1996.
14. Preeth, E.N.; Mulerickal, F.J.P.; Biju, P.; Yedhu, S. Evaluation of Docker containers based on hardware utilization. In Proceedings of the International Conference on Control Communication and Computing India (ICCC), Trivandrum, India, 19–21 November 2015; pp. 697–700.
15. Gadea, C.; Trifan, M.; Ionescu, D. A Microservices Architecture for Collaborativ. In Proceedings of the IEEE International Symposium on Applied Computational Intelligence and Informatics, Timisoara, Romania, 12–14 May 2016; pp. 441–446.
16. Quang-Vinh, D.; Claudia-Lavinia, I. Performance of real-time collaborative editors at large scale: User perspective. In Proceedings of the 2016 IFIP Networking Conference (IFIP Networking) and Workshops, Vienna, Austria, 17–19 May 2016; pp. 548–553.
17. Brodahl, C.; Hansen, N.K. *Education Students' Use of Collaborative Writing Tools in Collectively Reflective Essay Papers*; ERIC United States, 2014; p. 30. Available online: https://www.semanticscholar.org/paper/Education-Students'-Use-of-Collaborative-Writing-in-Brodahl-Hansen/c24242c69566fd08d8418b5ba665993b9598f0e1 (accessed on 27 September 2019).
18. 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 10th Computing Colombian Conference (10CCC) and Workshops, Bogotá, Colombia, 21–25 September 2015; pp. 583–590.
19. Sunil S. Transform Monolith into Microservices using Docker. In Proceedings of the 2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA), Pune, India, 17–18 August 2017.

20. Stubbs, J.; Moreira, W.; Dooley, R. Distributed systems of microservices using Docker and serfnode. In Proceedings of the 2015 7th International Workshop on Science Gateways, Budapest, Hungary, 3 June 2015.
21. Brogi, A.; Neri, D.; Soldani, J. A microservices-based architecture for (customisable) analyses of Docker images. *Software: Pract. Exp.* **2018**, *48*, 1461–1474.
22. Saha, P.; Govindaraju, M.; Marru, S.; Pierce, M. Integrating apache airavata with Docker, marathon, and mesos. *Concurr. Comput. Pract. Exp.* **2016**, *28*, 1952–1959.
23. Kuroki, K.; Fukushima, M.; Hayashi, M. Framework of network service orchestrator for responsive service lifecycle management. In Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), Ottawa, ON, Canada, 11–15 May 2015; pp. 960–965.
24. Qingfeng, D.; Tiandi, X.; Yu, H. Anomaly Detection and Diagnosis for Container-Based Microservices with Performance Monitoring. *Algorithms and Architectures for Parallel Processing*; Springer International Publishing: Cham, Switzerland, 2018; pp. 560–572, ISBN 978-3-030-05063-4.
25. Gedia, D.; Perigo, L. Performance Evaluation of SDN-VNF in Virtual Machine and Container. In Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Verona, Italy, 27–29 November 2018; pp. 1–7.
26. Taibi, D.; Systä, K. From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining. In Proceedings of the 8th International Conference on Cloud Computing and Services Science, Heraklion, Greece, 2–4 May 2019.
27. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Softw.* **2016**, *33*, 42–52.
28. Papp, E. *The Design and Implementation of Key Applications for a Live, Collaborative Online Environment*; University of California, Santa Cruz, United States, 2015. Available online: https://escholarship.org/uc/item/03n1t6vs (accessed on 20 December 2019).
29. Baneres, D.; Serra, M.; Rodríguez, M.E. Collaborative Tool to Enhance Quality Evaluation of Higher Education Programmers. In Proceedings of the International Conference on Intelligent Networking and Collaborative Systems, Taipei, Taiwan, 2–4 September 2015; pp. 14–20.
30. Bakshi, K. Microservices-based software architecture and approaches. In Proceedings of the 2017 IEEE Aerospace Conference, Big Sky, MT, USA, 4–11 March 2017; pp. 1–8.
31. Salah, T.; Zemerly, M.J.; Yeun, C. Y., Al-Qutayri, M., and Al-Hammadi, Y., The evolution of distributed systems towards microservices architecture. In Proceedings of the 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), Barcelona, Spain, 5–7 December 2016; pp. 318–325.
32. Pawlik, R.; Werewka, J. Recreation of Containers for High Availability Architecture and Container-Based Applications. In Proceedings of the International Conference on Computer Networks, Gliwice, Poland, June 2019. 2019; pp. 287–298. Available online: https://www.springerprofessional.de/en/recreation-of-containers-for-high-availability-architecture-and-/16820430 (accessed on 20 November 2019).
33. B. Krämer; Hupfer, M.; Zobel, A. Time to Redesign Learning Spaces. In Proceedings of the Conference on Transformative Science and Engineering, Business and Social Innovation, 2015. Available online: https://www.researchgate.net/publication/282666337_Time_to_Redesign_Learning_Spaces (accessed on 20 December 2019).
34. Liu, X.; Shen, W.; Liu, B.; Li, Q.; Deng, R.; Ding, X. Research on Large Screen Visualization Based on Docker. *J. Phys.* **2019**, *1169*, 012052.
35. Mehmood, N.Q.; Culmone, R. A Data Acquisition and Document Oriented Storage Methodology for ANT+. In Proceedings of the 30th International Conference on Advanced Information Networking and Applications Workshops, Crans-Montana, Switzerland, 23–25 March 2016; pp. 312–318.
36. Casalicchio, E.; Perciballi, V. Measuring Docker performance: What a mess!!! In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, L'Aquila, Italy, 22–26 April 2017; pp. 11–16.
37. Elliott, D.; Otero, C.; Ridley, M.; Merino, X. A Cloud-Agnostic Container Orchestrator for Improving Interoperability. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2–7 July 2018; pp. 958–961.
38. McElhiney, P.R. Scalable Web Service Development with Amazon Web Services. Master's Thesis, University of New Hampshire, Durham, United States, 2018.

39. Wasson,M. and Celarier, S. Microsoft Azure: Microservices Architecture Style. Available online: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices (accessed on 12 June 2019).
40. Maatouki, A.; Meyer, J.; Szuba, M.; Streit, A. A Horizontally-Scalable Multiprocessing Platform Based on Node.js. In Proceedings of the 2015 IEEE Trustcom, BigDataSE, ISPA, Helsinki, Finland, 20–22 August 2015; pp. 100–107.
41. Hunt, J. Gang of Four Design Patterns. *Scala Design Patterns: Patterns for Practical Reuse and Design*; Springer International Publishing: Cham, Switzerland, 2013; pp. 135–136, ISBN 978-3-319-02192-8.
42. Fowler, M. *Patterns of Enterprise Application Architecture*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002; ISBN 0321127420, .
43. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley: Boston, MA, USA, 1995; Volume 2, pp. 369–378.
44. Krause, J. Introduction to Pug. In *Programming Web Applications with Node, Express and Pug*; IApress: Berkeley, CA, USA, 2017; pp. 81–87.
45. Li, L.; Chou, W. Design Patterns for RESTful Communication Web Services. In Proceedings of the 2010 IEEE International Conference on Web Services (ICWS), , Miami, FL, USA, 5–10 July 2010; pp. 512–519.
46. Mouat, A. *Using Docker: Developing and Deploying Software with Containers*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2015.
47. Promethus, Monitoring Docker Container Metrics Using CAdvisor. Available online: https://prometheus.io/docs/guides/cadvisor/ (accessed on 10 March 2020).
48. Dynatrace, Oficial Web Site. Available online: https://www.dynatrace.com/support/help/ (accessed on 7 April 2020).