Received February 17, 2022, accepted March 7, 2022, date of publication March 10, 2022, date of current version March 30, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3158743

Autoscaling Pods on an On-Premise Kubernetes Infrastructure QoS-Aware

LLUÍS MAS RUÍZ, PERE PIÑOL PUEYO, JORDI MATEO-FORNÉS, JORDI VILAPLANA MAYORAL, AND FRANCESC SOLSONA TEHÀS
Department of Computer Science and INSPIRES, University of Lleida, 25001 Lleida, Spain

Corresponding author: Lluís Mas Ruíz (lluis.mas@udl.cat)

This work was supported by the Intelligent Energy Europe (IEE) program and the Ministerio de Economía y Competitividad under contract PID2020-113614RB-C22 funded by MCIN/AEI/10.1H6 network TIN2016-81840-REDT.

ABSTRACT Cloud systems and microservices are becoming powerful tools for businesses. The evidence of the advantages of offering infrastructure, hardware or software as a service (IaaS, PaaS, SaaS) is overwhelming. Microservices and decoupled applications are increasingly popular. These architectures, based on containers, have facilitated the efficient development of complex SaaS applications. A big challenge is to manage and design microservices with a massive range of different facilities, from processing and data storage to computing predictive and prescriptive analytics. Computing providers are mainly based on data centers formed of massive and heterogeneous virtualized systems, which are continuously growing and diversifying over time. Moreover, these systems require integrating into current systems while meeting the Quality of Service (QoS) constraints. The primary purpose of this work is to present an on-premise architecture based on Kubernetes and Docker containers aimed at improving QoS regarding resource usage and service level objectives (SLOs). The main contribution of this proposal is its dynamic autoscaling capabilities to adjust system resources to the current workload while improving QoS.

INDEX TERMS Cloud, microservices, Kubernetes, SLO, QoS.

I. INTRODUCTION

The popularity of cloud usage has grown dramatically in recent times, see [1], [2]. There is an increasing number of companies deploying public clouds and on-premise infrastructure. Almost all consumer services in banking [3], education [4], or health-care [5] rely on online services. These services are real-time and critical application models deployed in clouds. One of the main advantages of cloud computing is resource provision, which offers rapid elasticity and dynamic scalability [6]. IaaS clouds can scale up or down on demand. However, providing scalability is not trivial in SaaS environments. Legacy applications must coexist and cooperate with new applications and services to improve service quality. This way, to integrate all these constraints into the systems, businesses nowadays require a portable and interoperable environment capable of scaling and adapting to users' needs. Furthermore, businesses need an ecosystem where traditional tools and emerging innovation can coexist and cooperate. Many legacy applications have been migrated to cloud infrastructures that only consume and run on a predefined static set of resources [7]. To tackle these requirements, a container-based infrastructure can be used. This infrastructure helps not only to deploy it to the cloud (achieving all the advantages of cloud computing mentioned above) but also to keep logic developed by different firms and integrating and interacting with other services. Applications have evolved from a monolithic development, which encapsulates the entire system, to small decoupled microservices for solving particular tasks. Nevertheless, microservices in the cloud need to mitigate different problems compared to traditional cloud services, such as communication growth.

Regardless of their underlying architecture, cloud services and applications are strictly governed by quality of service (QoS) constraints regarding such metrics as efficiency, availability, reliability, and power awareness. Currently, QoS plays a vital role [8] and is regulated by a service-level agreement (SLA). An SLA is a contract between clients and providers

The associate editor coordinating the review of this manuscript and approving it for publication was Tomas F. Pena.

that expresses the price for a service, the QoS levels required during the service provision and the penalties associated with violations of the SLA. In this context, the service level objective (SLOs) are the specific, measurable characteristics of the SLA such as availability, throughput, frequency, response time, or quality. Quality of service and user satisfaction are directly related. A lower level of QoS due to an SLO violation leads to a decrease in user satisfaction. Hence, service providers must design these contracts very carefully to maintain user confidence and also avoid revenue loss [9]. The main challenge for a service provider is to determine the best trade-off between profit and customer satisfaction. Monitoring QoS constraints is the key to ensuring SLO constraints and service alignment to QoS compliance [10].

One of the challenges in guaranteeing QoS is that real-time performance metrics vary depending on the type of service. For example, in a service based on computation and high CPU usage, the primary metrics are the CPU use and throughput, while for a service based on a database, the most critical metric is response time. In this heterogeneous context, different metrics have to be defined depending on the purpose of the service.

Kubernetes is a container cluster management system that allows controlling the lifecycle of container clusters. It provides tools for automatic deployment and scaling of containerised applications and routines for starting/restarting, scheduling, and rescheduling applications in a dynamic context when some host fails. Thus, Kubernetes allows spanning hosts across the public and hybrid cloud and on-premise infrastructure. One of the main handicaps of Kubernetes is the large waste of resources usage. It is mainly focused on handling the highest peaks of workload, but the system is not always at the peak of workload. Therefore it is necessary to adjust scaling up and down the resources dynamically to improve resource use and ensure the quality of the services (QoS).

Kubernetes proposes two distinct autoscaling methods: vertical and horizontal autoscaling (HA and VA). HA happens when the computer resources of the underlying cluster are modified. Thus, it means adding or removing pods (HPA) or nodes (HNA). On the other hand, VA requires updating the resources associated with pods (VPA) or nodes (VNA). By default, Kubernetes triggers HPA when CPU and memory usage of the pods exceeds a certain static threshold. Moreover, using HPA and VPA simultaneously to manage resources can arise conflicts resulting in a wrong allocation of resources. There are not many works aimed to improve Kubernetes autoscaling methods. In this context, a deep analysis of HPA has been presented in [11]. Regarding VPA, authors in [12] propose container migration aimed to enhance resource utilization minimizing the need to estimate container resources. Furthermore, in [13] LIBRA is presented, a mixed VPA and HPA. The authors proposed controlling the resources and load condition by recalculating the threshold iteratively. Finally, there are different proposals (addons and plugins) that provide a deeper configuration minimizing this static context focused on node-level like Cluster Autoscaler [14] or in pod-level like Vertical Pod Autoscaler [14].

Conventional container orchestration platforms usually only offer limited autoscaling functionalities [15]–[17]. The authors in [18] present a method to establish a CPU use threshold automatically to meet the requirements of a specific application. This work results in a cluster scaling algorithm that converges towards an ideal number of nodes in the Kubernetes Cluster and improves CPU use by 28.9%. Nevertheless, in a real situation, cloud architectures need to deal with a range of applications and services where, *a priori*, estimating the requirements are challenging. That is where Kubernetes' autoscaler starts to show its flaws, as stated in [19]. This work proposes an autoscaler service that deals with a hybrid environment with legacy applications (with general requirements) jointly with a routine creation service (without general requirements).

Another critical point to consider is that while resource saturation possibly causes SLO violations, the exact correlation has yet to be explicitly discovered. An ideal autoscaling strategy is expected to react directly to application-level metric changes, such as increasing SLO violation rates [20]. This work proposes a multi-tier architecture, with a monitor and a QoS service to track workload and resource usage in all tiers and levels and dynamically adapt to meet QoS and SLO agreements.

In this work, the challenge is to design and implement a monitoring service that tracks and integrates monitoring data from all the microservices and datacenter's resources. With this information, the QoS service implements policies and performs actions to guarantee the SLOs regarding resource usage. This way, we propose to improve the Kubernetes cluster autoscaler with custom rules based on the system's current status. Therefore, this work makes the following contributions:

- Corroborate the current state of the art. Showing how to implement a Kubernetes cluster over on-premise infrastructure.
- Propose a Kubernetes on-premise infrastructure that can dynamically scale up and down based on SLO requirements to improve resource usage and guarantee QoS.
- Enhance the performance of pod-level Kubernetes auto scalers.

II. METHODOLOGY

In this section, the platform's requirements are analyzed one by one, and different solutions for each of these are suggested. First of all, preliminary concepts are explained to help understand the decisions adopted. Then these proposals are brought together and the full architecture is described in detail.

A. ARCHITECTURE

The architecture proposed aims to provide a diverse set of functionalities in a cloud environment. In this case, "diverse" means that different services may have completely unrelated dependencies. Thus, the preferable solution would be to isolate each functionality, along with its dependencies, from the rest. In other words, a microservice architecture.

One of the platform's requirements is to allow the integration of legacy applications in the cloud. New functionalities have to coexist with these and might take advantage of their data and services. Containers are useful to separate legacy and new environments. Without using these containers, dependency management to allow for backwards compatibility would become complex.

However, legacy services still require a custom interface to communicate with the outside world, if necessary, with other containers. It may also be useful to copy information to more efficient databases or adapt to the services' requirements. As a result, a middleware that assumes these responsibilities might be required in the architecture.

Secondly, services should have high availability and responsiveness, along with other QoS requirements. This implies the need for performance measurement (through resource usage or more complex calculations) and scaling. At this point, it seems appropriate to use a container orchestrator.

Orchestrating means automating container allocation and management tasks, essentially abstracting away the underlying physical or virtual infrastructure from service developers [21]. In other words, it consists of managing the lifecycle of the containers: provisioning, scaling, resource allocation and health monitoring, among others.

The fact that orchestration will prove a useful addition to the architecture is clear, as it will automate the behavior necessary to achieve responsiveness and availability. However, custom orchestration rules may have to be defined to fully align the orchestrator's actions with the platform's SLOs. As a result, a separate monitor and a QoS service will have to be created.

Finally, a gateway will redirect incoming traffic to the appropriate service. Having only one point of access has various benefits:

- Support for implementation of access-control methods (e.g. authentication).
- User-experience improved by providing a transparent interface.
- Allowing the establishment of load-balancing measures when paired with container orchestration.

Once all the requirements are taken into account, the resulting architecture is represented by Figure 1. An explanation of its components follows:

- Virtual machines, servers and Nodes: the cluster is deployed on virtual machines, which act as Kubernetes Nodes and run on a cloud infrastructure. This layer is optional, and can be avoided if we deploy directly to bare-metal.
- Container orchestrator: as explained above, this layer abstracts the hosts into a Cluster, providing a transparent interface and container management capabilities.
- Containers: these are divided into three categories.
  - Data tier: the files, databases, and other information sources relevant to the services running in the cloud.
  - Core tier: contains the business logic.
  - Composite tier: composite and aggregate services, which use the resources provided by other tiers.

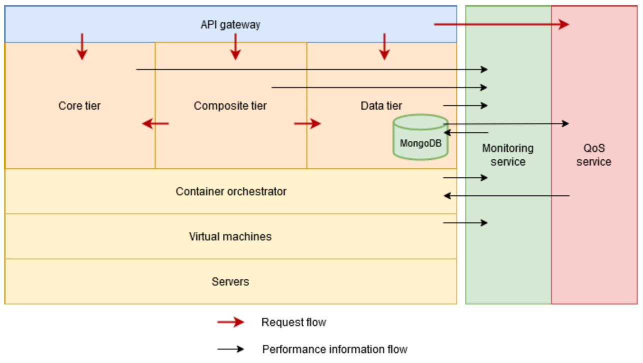

FIGURE 1. Architecture of eHQoS.

Legacy services may belong to any tier, depending on their purpose.

- API gateway: made up of a set of mappings from URIs of containers accessible from the outside. It is also responsible for load balancing and security.
- Monitoring service: calculates usage metrics of each service by communicating with all tiers. The outcomes are relayed to the QoS service.
- QoS service: takes the metrics provided by the monitoring service and directs the container orchestrator in decisions such as scaling and migrating across machines.

III. IMPLEMENTATION

In this section, a case study of the proposed architecture is presented. First, the minimal configuration setup and the experimental setup are described in subsection III-A and subsection III-B. Afterwards, the services deployed on them through a Kubernetes Cluster are presented. The internal services required by the architecture are first presented in subsection III-C and, afterwards, business logic services explore those that directly provide the functionality to the end-user in subsection III-D.

The GitHub repository containing the implementation of the framework can be found at https://github.com/gcd-cloud-research/KAQoS.

On top of the framework implementation, a set of Ansible playbooks [22] can also be found. This can be used to deploy the system on an entirely new cluster quickly. Ansible playbooks allow configuration distribution and easy provisioning in a set of commands that can be run by all users.

A. MINIMAL CONFIGURATION SETUP

The minimum deployment uses seven (virtual or physical) machines located in on-premise infrastructure: a Docker registry, a Master, two workers, a service node and a monitor node. The registry stores Docker images related to the services provided. The Master and workers are named after their Kubernetes roles and run a precooked image that includes all the necessary dependencies to set up a Kubernetes Cluster with minimal configuration. The service and monitor node is also in the Kubernetes cluster but will not process any work that a user might submit. The purpose of the service node is to run pods related to the functioning of the architecture. The monitor node, on the other hand, hosts the pods related to the performance monitoring.

B. EXPERIMENTAL SETUPS

We set up the architecture proposed under on-premise infrastructure managed by OpenNebula. This infrastructure was made by 6 nodes with a total capacity of 164 cores, 656 GB of main memory, and approximately 12TB disk storage. The deployment uses the minimal configuration setting presented above. Their specifications are given in Table 1.

TABLE 1. Specifications of the virtual machines in the private cloud.

Name	Replicas	CPU	Memory (GB)	Disk (GB)
Docker registry	1	1	2	8
Master	1	2	16	15
Service	1	2	16	15
Monitor	1	2	16	15
Worker	2	2	8	15

The disk is required by the Kubernetes' and Docker's internal operations. All persistent data regarding the services is stored in two external virtual hard disks. One of the disks stores sensitive information and is encrypted with Linux Unified Key Setup (LUKS) [23], the most commonly used standard for encryption in Linux. The other keeps relevant but non-sensitive data and is not encrypted. Both are 30GB in size, mounted on the Master and shared via NFS.

C. INTERNAL SERVICES

Figure 2 depicts all the services implemented and their relationships.

Internaldb acts as the storage for all information related to the Cluster's internal data on Jobs run on the platform. MongoDB was the document-based NoSQL database [24] chosen.

Monitordb is responsible for storing all the performance log records. As data is written with high frequency, read periodically and never updated or deleted, the NoSQL database chosen has been Elasticsearch [25].

Most of the content in the storage is not accessed externally (except by system administrators), so access to the database should be restricted. This can be done via access control or a proxy. The second option was selected, resulting in the deployment of **mongoapi**, because it also provides ease of access and transparency, allowing the database's specifics to be modified as necessary. On the other hand, it adds overhead and an additional failure point, so access control would also be a viable option.

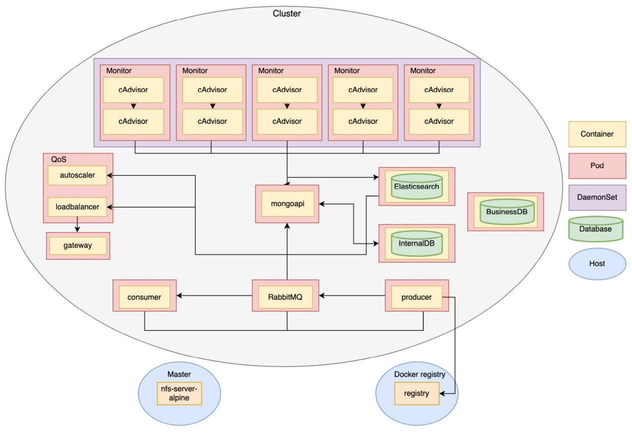

FIGURE 2. Services included in the implementation and their relationships. Arrows indicate information flow.

The monitoring service was implemented as a DaemonSet — that is, an instance of it is running in each Worker Node. Its function is to collect performance information from hosts and their containers and save it in **monitordb**. The QoS service will then extract data from there.

The tool used for compiling performance metrics was **cAdvisor**, a daemon that collects, aggregates and exports information about hosts and running containers [26]. A custom aggregator queries **cAdvisor** and writes into **internaldb** through **mongoapi**.

CPU usage per core for each host is calculated as the increment in the number of nanoseconds of busy CPU over the increment in time, taking the previous measurement as a reference. This number is then divided by the number of cores to obtain total CPU usage. Regarding containers, the measurement is the same but only considering the nanoseconds of CPU that a specific container has used.

Memory usage is calculated by dividing the memory used at the time of the measurement by the amount of Memory available to the host or container.

The usual technique to set up a gateway with a Kubernetes Cluster involves two elements. First, a proxy inside the Cluster receives traffic from outside and routes it to the appropriate service. Second, a load balancer outside the Cluster shows it to the public through a static IP address and sends all requests to the proxy.

Typically, Kubernetes Clusters are deployed on cloud provider platforms such as AWS or GKE. These already provide their own load balancers, which developers can use to make their Clusters externally accessible.

On the other hand, given that the Kubernetes instance is running on a bare-metal Cluster, there are no available Load Balancers. Some alternatives exist, such as combining MetalLB [27] as a load balancer and Ambassador [28] as a router. However, this requires modifying the cloud's (OpenNebula) configuration, which was not contemplated during this work. The main reason for this decision was that the authors prioritize the minimum adoption cost of introducing a new layer on top of Openebula without reconfiguring the legacy cloud system. Kubernetes' Load Balancer was also discarded as it is deemed as insufficient in some cases, like our study, as stated in [29] and [30].

As a workaround, a custom solution was coded. A Flask API running outside the Cluster, in the Kubernetes Master, acts as an entry point, checking the user's permissions and routing requests to the best available worker. The **loadbalancer** (described in the next section) is queried to provide the best replica of the desired Deployment, to which the request is forwarded.

The QoS Deployment contains two services: **autoscaler** and **loadbalancer**.

The **autoscaler** service is designed to allow the combination of different scaling criteria. These norms are implemented as classes and can be plugged into the **autoscaler**. The result is computed as a weighted average of the output of each of these. The default plugin, *loadtracker*, uses upper and lower thresholds to determine if a Deployment needs to be scaled up or down.

The service contains two processes that communicate through a pipe. The first one monitors load, as explained above, and writes the desired replicas of each service into the pipe. The second one reads from it and compares the desired replicas with the currently available ones, scaling up or down as necessary.

This service also takes into account the replicas of a Deployment desired by the system administrator through a specific tag in the Kubernetes *.yaml* file (*io.Kubernetes.replicas* in *metadata.labels*). This can be modified without the need to restart the **autoscaler**.

The script running in this Deployment accepts a JSON configuration file that allows the system administrator to set scaling and descaling criteria. All fields are optional; the *autoscaler* will resort to default values when necessary.

- **scaling**: Settings for the *loadtracker* plugin.
  - **min_load**: If the CPU or Memory usage of a Pod is under this value for a certain amount of time and with some degree of tolerance, the Deployment will be descaled.
  - **max_load**: Same specification as *min_load*, but being the upper threshold
  - **max_load_nowait**: If the CPU or Memory usage go over this value, the Pod is immediately scaled up.
  - **wait_seconds**: elapsed time until an under- or overloaded Pod is deployed.
  - **tolerance**: Number of times a Pod can be under- or overloaded.
  - **grace_period** The number of seconds during which a Pod should not be tracked after scaling it.
- **update_seconds**: The period between two consecutive updates in the scaler, in seconds.
- **exclude**: Deployments which should not be monitored.
- **over_threshold**: From zero to one, the weighted sum of the outputs of the plugins has to be greater than this value to consider that a Pod needs to be scaled.
- **under_threshold**: The same specification as *over_threshold*, except that the weighted sum has to be smaller than this value to descale.

The **loadbalancer** is responsible for choosing the best replica of service to route a request to. An initial version selects the Pod which is using less average resources (CPU and Memory). The selection criterion used is a key parameter, as it determines the Pods that will receive additional workload. Even when using a production-ready load balancer, the method for worker selection is usually a configurable parameter. In consequence, finding the best configuration for each specific Deployment is essential for achieving a well-balanced application.

Another endpoint can be used to see if Jobs can be created. It is used by the **consumer** to prevent scheduling when the platform is overloaded. The system's average resource usage determines if this is under a certain threshold (in the current case, 80%).

D. BUSINESS LOGIC

businessdb is another Deployment of MongoDB that contains information relative to the platform's intended use. End users could run data analysis jobs on the data, providing new insights into the vast amounts of intelligence available.

Ideally, the end-users would keep using the preexisting user interface with which they are familiar. Thus, an Extract, Transform, Load procedure (ETL) is necessary to keep the **businessdb** updated, either running as a periodic job or whenever changes are detected.

Additionally, the database acts as a backup for the data contained in the legacy storage. When a failure occurs, the regular updates would only allow for minimal or null loss of data, which is a key requirement in online services.

In Figure 2 there are no information transfers from **businessdb** because all services deployed are related to the architecture, and the Deployment only contains test data. Its potential is fulfilled when other business-oriented services appear in the platform, for example, an anomaly-detection Deployment could be analyzing records in the database in real-time, or custom aggregators could show end-user information relevant to their tasks.

The producer-consumer relationship allows the user to run jobs (also called routines) remotely. Its purpose is to provide the necessary resources for performing data analysis tasks in the platform, using the data contained therein. It supports the submission of Python and R scripts. This section focuses on the flow of a routine Python submission request.

Firstly, the end-user submits a request to the **producer** containing a script and, optionally, a requirements file. This Deployment stores the request's contents initializes a *task* instance in **internaldb** (which returns an ID) and uses a routine template to build a Docker image, which contains:

- The script to be run.
- A wrapper for the script that changes the routine's status as necessary and writes the results to **internaldb**.

- The modules required both by the wrapper and the end-user.

This image is built and stored in the cloud's Docker registry using the host's Docker daemon. The routine's identifier is sent to a queue. After that, all data related to the routine (local Docker image and files saved) are removed.

The **consumer** polls for items in the queue. Whenever an item is received, this Deployment queries **loadbalancer** to see if the system's load is under a certain threshold. If the condition is not met, **consumer** returns the message to the queue and waits some time before repeating the process. When **loadbalancer** approves, a new job is submitted to Kubernetes using the routine ID.

Each step in this process is represented in the database through its status in the object representing the routine. The possible states and transitions are specified in Figure 3.

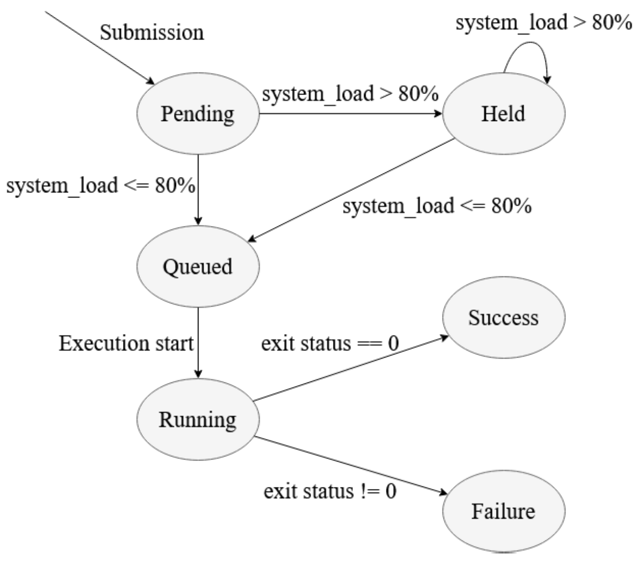

FIGURE 3. State diagram of a routine.

Communication between the **producer** and the **consumer** is achieved through RabbitMQ [31], a queuing messaging system.

IV. RESULTS

The tests were performed to focus on stressing the platform and examining its behavior. First of all, an initial analysis to check the specific architecture mechanisms in regards to SLO compliance (scaling and job queuing) is specifically examined. Afterwards, an overall performance analysis is conducted, to shed light on the platform's general behavior and highlight the situations where the platform fits better than a traditional Kubernetes cluster. Thus, we aim to prove in which situations the proposed QoS method expands the conventional tools, such as default QoS methods in Kubernetes and obtains better performance regarding the usage of computational resources overcoming some of the issues found in traditional Kubernetes clusters.

Given that with the current implementation, CPU and Memory usage are the monitored metrics, this results section also revolves around these metrics. A context with legacy applications, APIs, databases and other tools was deployed and, to simulate stress conditions, CPU and Memory wasters have been employed. This section shows how the system responded. Ideally, all these systems would coexist and share available resources.

We set up the architecture described in subsection III-B. However, in our experimentation, we play with different sizes to deploy the Kubernetes cluster, and also we change the number of resources provisioned to each node (virtual machine in our case) to better assess the efficacy of the proposed system.

A. JOB CREATION WITH OVERLOADED PLATFORM

In this test, the behaviour of the system in job creation is examined. To simplify the results and help the reader understand the plotting, we run this test using a Kubernetes cluster made up of only 3 Workers replicas. The resource provision is the same as the ones described in subsection III-B.

First of all, we increase the CPU and Memory usage of all hosts (Workers) to around 90% through a DaemonSet, **cpuwaster**[1] and **memorywaster**.[2] A job is then submitted. The job is a simple python script that performs some calculations. After that, we stop both wasters. So, the primary purpose is to ensure the correctness of the QoS policies implemented in the platform, which must prevent the execution of this job until the system resource occupancy is over 80%. The job should be put on hold by the **consumer**, and then, once the **waster** is deleted, be executed in the Kubernetes Cluster.

Figure 4 depicts the different phases in routine creation and highlights the correctness of the QoS policies implemented. First of all, when the job is submitted, the CPU usage of all the hosts (Workers) is higher than 80% (the green dashed line with the tag Overload). In this situation, job execution is prevented, and its status is changed to HOLD. When a decrease in the overall usage of CPU resources is detected (when the **cpuwaster** ends), the job is added to the queue, and then executed. This situation is depicted around $t = 70s$ when the load decreases drastically from 80% to 40%. Execution takes place in Worker 3, as it is the only host that increases CPU usage after the job is queued. These results are proof of concept and demonstrate that QoS performs well and follows the rules we implemented.

B. SCALING BASED ON POD LOAD

This test examines the possibilities of autoscaling the proposed system by creating a job to stress **businessdb** to the point where **autoscaler** creates additional replicas. This test is set up with a Kubernetes cluster using five nodes.

[1]https://github.com/gcd-cloud-research/KAQoS/tree/main/images/cpuwaster

[2]https://github.com/gcd-cloud-research/KAQoS/tree/main/images/ memorywaster

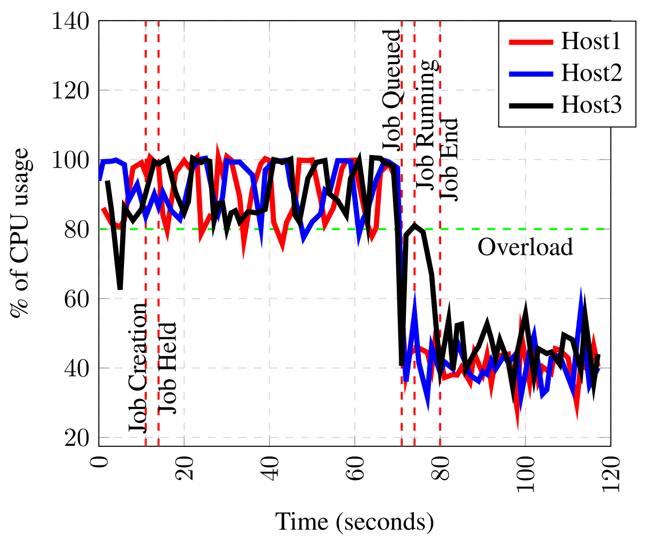

FIGURE 4. Events in job creation related to the Cluster's CPU usage.

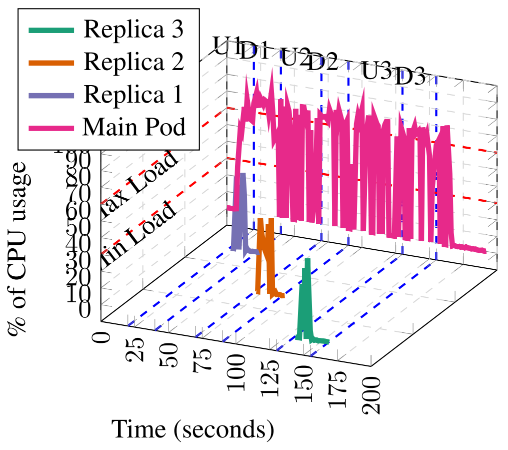

FIGURE 5. CPU load and scaling/descaling events while a test job is stressing businessdb.

The job, *mongo-flood*,[3] queries **mongoapi** to set the load in **businessdb** to 60%. It then waits for a minute before completing.

The fact that the containers' resource usage is calculated using the resources available to the host (as opposed to some limit set to the container) causes the minimum load required to scale up to be lower to avoid overloading the whole system. As a result, the configuration used in **autoscaler** for this test is as follows (details of the parameters are explained in section III):

```
{
  "scaling": {
    "min_load": 30,
    "max_load": 60,
    "max_load_nowait": 90,
    "wait_seconds": 10,
    "tolerance": 5
  },
  "update_seconds": 5,
  "exclude": ["monitor"],
  "over_threshold": 0.5,
  "under_threshold": 0.5,
  "grace_period": 20
}
```

As can be seen in Figure 5, the Deployment (**businessdb**) scaled up and down three times (purple, orange and green lines - Replica 1, Replica 2 and Replica 3 respectively). As a result, four Pods were involved in the test, but only two at most co-occurred at any given point in time.

This information, combined with the performance metrics obtained from the monitoring service about **businessdb**'s Pods' CPU consumption over time, yields Figure 5. It depicts the CPU usage of the different replicas (Pods) in the Deployment, the low (min_load) and upper (max_load) load thresholds, set to 30 and 60 respectively, and the scaling events, represented by lines (U1, U2, U3, D1, D2, D3; where U and D are used to indicate scaling up and down events, respectively).

This first scaling event seen in Figure 5 (denoted by U1, at $t = 20s$) is caused by a continuous increase in CPU usage over the maximum threshold (*max_load*). Note that the **Main Pod** is using a percentage superior to the *max_load* between $t = 10s$ and $t = 20s$. As a result, the autoscaling service scales up the Deployment (generating the Pod **Replica 1**). After this event, both Pods are ready to serve incoming requests.

Afterwards, the mean load decreases, leading the Pod to exceed its tolerance threshold, after which it is considered to have a normal load, and the QoS service eliminates **Replica 1**. This is illustrated by D1, at $t = 40s$. This situation, where the Deployment is scaled up and down, happens twice more (**Replica 2** and **Replica 3**), denoted by (U2, D2, U3, D3). This behavior highlights the capacity of the proposed system to adapt to an uncertain workload regarding resource usage.

Finally, after the events discussed, the number of replicas necessary is one. Thus, the **autoscaler** scales down until the desired value is reached, only maintaining the initial (**Main Pod**).

C. PERFORMANCE DURING JOB EXECUTION

The tests performed consist of the submission of 1000 identical jobs that train and test supervised models on two samples stored in **businessdb** under 2 different situations. First of all, an ideal situation is tested where the hosts are close and reserved to the experiment and will only execute these 1000 test tasks. Then, we test another situation where the hosts are not reserved and needs to share the resources. Note that this situation can be similar to a platform that contains other tasks and legacy applications that consume resources, but also new tasks can also arrive at the system and must be executed. We perform this experiment with the experimental setup presented in subsection III-B.

[3]https://github.com/gcd-cloud-research/KAQoS/blob/main/test-routines/mongo-flood/mongoflood.py

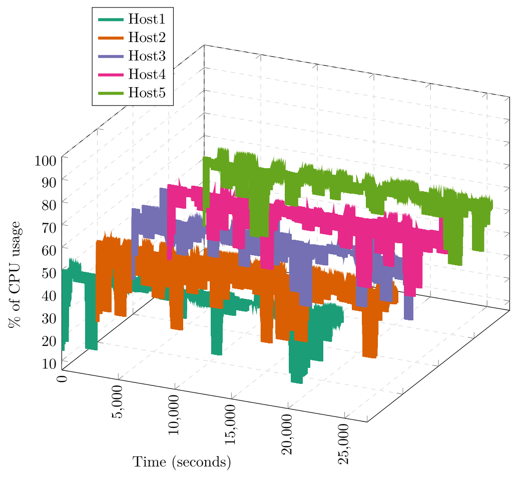

FIGURE 6. Kubernetes without waster.

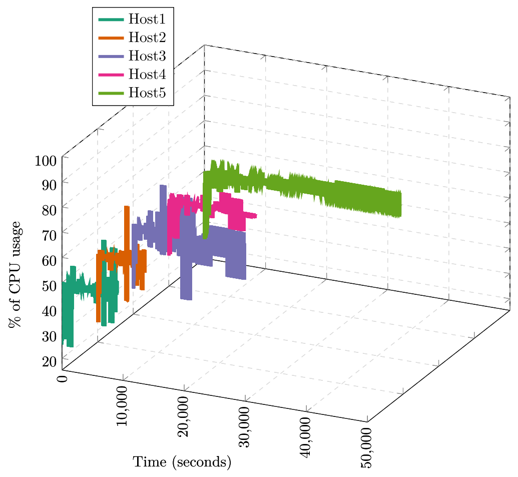

FIGURE 7. Kubernetes with waster each worker.

The first experiment is launched in the ideal situation. The QoS proposed services are stopped for this experiment and Kubernetes is allowed to manage the situation with the default configuration. Figure IV-C shows the performance concerning usage of CPU (y-axis) over time (y-axis) for each host in the Cluster. This figure highlights that the default configuration on the Kubernetes cluster can manage the resources in this situation very well, and our proposal is not required. From these results, it is clear that all the tasks are scheduled and executed homogeneously among the hosts keeping the resource usage under 80%. Therefore, we can conclude the excellent behavior of the Kubernetes cluster under this situation, as other authors in the literature point out.

The second experiment is performed in the non-ideal situation (stressed system) and using only default Kubernetes. Figure 7 depicts the overall performance over time for the 1000 tasks. Note that the waster processes are not depicted. This figure points out that Kubernetes cannot handle this situation where the hosts are stressed and performing other tasks. Compared with Figure IV-C, the task is not now homogeneously distributed (hosts 5 is overloaded) while other hosts are free and need double of time to finish the same experiment.

The last experiment is performed in the non-ideal situation (stressed system) but now with our QoS proposal activated to manage the cluster. Figure 8 shows better usage of the resource compared with Kubernetes alone. See Figure 7. When comparing both results, it must be pointed out that QoS provides better resources utilization keeping the hosts busy by scaling the pods up and down on demand.

V. BENCHMARK

To assess the efficacy of the proposed platform, it has been compared against Kubernetes' default configuration and with community-developed autoscalers [14].

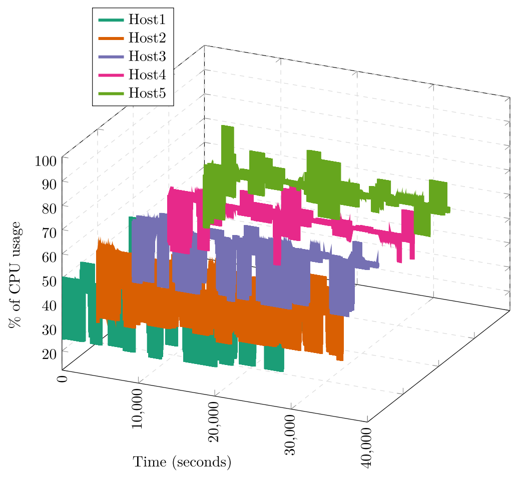

FIGURE 8. QoS with waster each worker.

A. PERFORMANCE AND SCALABILITY AGAINST KUBERNETES DEFAULT CONFIGURATION

We perform this experiment with a Kubernetes cluster made up of three nodes. We provision the nodes with different ranges of resources assigned to make up the cluster:

- **case 1:** 3 nodes of 2 CPUs each node
- **case 2:** 3 nodes of 4 CPUs each node
- **case 3:** 3 nodes of 8 CPUs each node
- **case 4:** 3 nodes of 10 CPUs each node

The experiment consisted in submitting 500 tasks. The task is a Logistic Regression model, which is CPU-consuming and does several databases calls as well [32]. To set up this experiment, we first deploy a default Kubernetes cluster and monitor the main performance metrics (response time, execution time and throughput). Then, we deploy our proposed QoS aware architecture and resubmit the same tasks again.

We analyze four scenarios where we configure the nodes with different resource provisioning related to CPU resources. The results show a reduction in response time and a better throughput between the proposed architecture and the default configuration of a Kubernetes cluster. Table 2 shows

TABLE 2. Performance results between Kubernetes default configuration environment and proposed infrastructures based on a Kubernetes QoS awareness under normal workload conditions.

Case	RT (s)	ET (s)	Throughput
1	167.30	1018.90	3.738
1	207.41	1177.99	2.453
2	58.91	565.89	5.591
2	83.51	701.69	4.211
3	22.50	237.46	7.304
3	29.40	671.84	6.518
4	16.48	195.01	10.302
4	33.50	251.05	9.394

the average response time measured in seconds (RT (s), average execution time measured in second (ET (s) and the throughput of each experiment. These results were captured in a non-saturated environment, simulating a normal workload. We have highlighted in green the rows with the results of our proposal with QoS enabled. Finally, the results also show that our proposal is scalable and does not depend on nodes' resources.

Table 3 shows the average response time measured in seconds (RT (s), average execution time measured in second (ET (s) and the throughput of each experiment. These results were captured in a saturated environment, simulating a stressed workload. The results show the same trend as the ones gather from normal conditions. However, the differences between our proposal and default configuration are even better under saturated conditions.

TABLE 3. Performance results between Kubernetes default configuration environment and proposed infrastructures based on a Kubernetes QoS awareness under saturated context.

Case	RT (s)	ET (s)	Throughput
1	238.83	1463.02	1.36
1	371.63	1670.07	1.053
2	70.01	675.55	4.30
2	110.33	917.80	3.4
3	35.83	596.38	4.458
3	53.43	712.61	3.502
4	24.59	475.31	7.406
4	39.88	575.53	5.41

B. PERFORMANCE AGAINST COMMUNITY-DEVELOPED AUTOSCALERS

Kubernetes provides some community-based autoscalers. Down below is the list of them and whether they are useful for assessing the proposed work.

- **Cluster Autoscaler:** It uses node-level scaling and we focus on a pod level in this paper. However, it will be useful for our future work regarding this paper, which will be a study of how the node-level autoscaling fares when considering costs too and whether it can be improved.
- **Vertical Pod Autoscaler:** Scales pod instances up or down based on the amount of requests and resource usage.
- **Addon Resizer:** It scales resources up or down based on some parameters. The resource amount needs to be specified and only works for a singleton.
- **Charts:** Likewise Cluster Autoscaler, focuses on node scaling while we are interested in pod scaling in this paper.

To sum up, only the VPA (Vertical Pod Autoscaler) fits the needs of the project in terms of scope. We perform the same experiment of submitting 500 tasks under a non-saturated and stressed environment. The only difference is that we execute this experiment under only one node configuration (case 3).

The results with the VPA have been compared and shown in Table 4. As it can be seen, the VPA plugin does in fact improve the performance results without the QoS enabled (default Kubernetes configuration). However, it still falls behind the performance obtained using our proposal.

TABLE 4. Performance results between Kubernetes default configuration, community VPA and proposed infrastructure based on a Kubernetes QoS awareness.

Autoscaler	Environment	RT (s)	ET (s)	Throughput
our proposal	non-saturated	22.50	237.46	7.304
VPA	non-saturated	27.96	459.59	6.706
default	non-saturated	29.40	671.84	6.518
our proposal	saturated	35.83	596.38	4.458
VPA	saturated	41.87	664.87	3.954
default	saturated	53.43	712.61	3.502

These results point out that our autoscaler performs better than the competitors in on-premise infrastructure regarding performance metrics related to Quality of the service and SLO awareness like response time and throughput under normal and saturated conditions.

VI. DISCUSSION

The findings of this work are similar to [10] and demonstrate that monitoring QoS constraints is the key to ensuring that SLOs are satisfied.

The results obtained on the implementation described in section III confirm that the architecture proposed is viable in regards to maintaining SLO compliance at both the hardware and software levels. Although the metrics used, focusing on CPU and Memory usage, do not reflect all the nuances of QoS in a production environment, the platform's performance in terms of standard SLO requirements is encouraging.

These results go beyond previous reports [20], [33]–[35], showing that it is complex but viable to deal with both levels of abstraction in the design and implementation of QoS regarding SLO.

Furthermore, this work ties nicely with previous studies where the authors conclude that it is possible to adjust the Kubernetes Cluster [18] if the requirements are known beforehand. However, the results highlight the issues that must be faced when these requirements are stochastic.

On the other hand, this implementation is far from being complete and marketable. Several issues that need to be addressed before applying the architecture on a commercial level were revealed by the tests. These are described below and proposed as future work in section VIII.

One issue found is that Alpine Docker images require additional Linux packages. This to light an important issue about the routine template's design. At the same time, it is not possible to provide an image flexible enough to allow the execution of jobs with differing requirements (e.g. programming language, libraries) while following the principles behind containerization (e.g. minimum dependencies and weight).

In fact, implementation gives the user the option to submit a Pip (Python Package Installer [36]) requirements file, which is meant to reduce dependencies in the routine template. An option would be to extend this behavior to allow users to submit their own Dockerfile, although this would go against the transparency of job management and execution. Additionally, the routine wrapper is written in Python, so support for the language and certain packages would still be required. Alternatively, separate templates could be added for each supported language.

The test (subsection IV-B) justifies the need to measure container load using the container's own resource limit as a reference, instead of using the hosts' (VMs) own resource limits. Otherwise, values for the **autoscaler** configuration need to be smaller and closer to each other, which would be detrimental in situations with high variability in resource consumption.

Nonetheless, selecting an appropriate limit *a priori* for a Deployment is complicated and depends both on the platform's own state and external factors. For example, **internaldb**'s usage is a function of the size of its collections, while **businessdb** depends on external submission of data. Although Deployments scale up if the load is excessive, minimizing replication operations is useful for keeping the architecture-related overhead to a minimum.

In the graph for this same test (Figure 5) a certain desynchronization between peak loads and scaling events can be observed. This has two causes:

1) The **monitor** DaemonSet and the **autoscaler** are not synchronized.
2) The scaling process has to obtain data from Kubernetes API before scaling.

The *update_time* of business is five seconds. As a result, the maximum time between the two events (performance submission in the database and scaling) should be just over five seconds. However, the delay between **cAdvisor**'s measurement and the **monitor**'s upload should also be taken into account.

Finally, during daily usage of the platform, it was established that resource usage increased with time. This can be easily explained by the accumulation of data in **internaldb**. Given that performance metrics are recorded each second for every host and container, the amount of data stored in the database leads to a great rise in resource usage. That is why we migrated from an initial approach in MongoDB to more efficient storage in Elasticsearch, which induces a much lower overhead over time. However, these old performance logs should either be purged or moved to another database or files to keep the main database lighter.

Similarly, cleanup should also be implemented in the Docker registry. This stores images for all jobs and Deployments, and the Kubernetes API server keeps records of executed Jobs. A reasonable way would be to clean all the records as mentioned above after a few days of not being used.

VII. CONCLUSION

In this study, a system based on the Kubernetes cluster and built over on-premise infrastructure was proposed to fulfil QoS requirements and be transparent to final users and automatically capable of executing tasks on demand and coexisting with legacy applications.

This paper presents a generic system to dynamically monitor and adjust the computer resource regarding QoS and SLO constraints. The architecture proposed contains such services as a monitor, QoS module, an executing service and scaling routines. These findings provide a potential mechanism for monitoring, adjusting, and autoscaling services, overcoming some challenges in the traditional Kubernetes environment.

The proposed system can improve resources management and QoS in a Kubernetes cluster. The QoS and the scaling routines can be configured to specific resource usage thresholds according to SLO rules, which can ensure QoS in the Cluster. The QoS module is responsible for the resource scaling, and then the executing module carries out pod scaling accordingly. Therefore, the data yielded by this study provide substantial evidence that the proposed system is capable of ensuring QoS regarding performance metrics such as Response Time and Throughput.

We build the system on top of on-premise legacy OpenNebula. This way, we tested that the proposed platform can be adopted as a new layer without reconfiguring the system, thus minimizing the impact of adoption. The underlying evidence from the proposed architecture states that it can be easily deployed in any other on-premise infrastructure as it does not use any OpenNebula specific configuration. Furthermore, there is no problem in deploying it directly on bare-metal nodes.

In conclusion, all these findings encourage continuing this work. They indicate that this is the path for evolving current cloud architectures into more robust, automatic, elastic and QoS-aware regarding the SLOs. We believe these findings have generalizable research value in the field and can be applied either in on-premise/hybrid clouds or for migrating traditional non-cloud servers/architectures to the cloud to achieve all their benefits.

VIII. FUTURE WORK

As mentioned in section VI, this project still requires several key features before achieving its potential. For example, Cron Jobs to remove old performance records, other database objects such as old tasks, Kubernetes API's Job data and Registry images should be scheduled. The implementation of authentication and authorization through roles, possibly using Lightweight Directory Access Protocol (LDAP), must also be taken into consideration.

If these requirements were met, the platform could be considered to be in its minimal marketable state. On the other hand, several additions are desirable to facilitate deployment and improve performance and thus, user experience. Regarding the former, a global configuration that includes parameters for all services in the architecture could be used to minimize in-script modifications. Next, routine submission should be tailored separately to each supported programming language. This would ensure dependency minimisation, isolation and transparency.

Additionally, tests could be performed to assess the effectiveness of executing Machine Learning algorithms to maintain QoS. These include preemptive actions based on load prediction, anomaly detection and visualization for system administrators or optimization of **autoscaler**'s configuration values. Additional performance metrics should be added to capture facets of execution that are not taken into account as of now.

After implementing a better scaling and load balancing solution on a pod level, the clear next step would be to develop a solution that addresses node-level scaling. That would ideally use historical performance data to predict using AI algorithms such as Machine Learning or Reinforcement Learning. That would allow the system to grow and reduce its size optimally, thus reducing infrastructure costs and power consumption.

Finally, an interesting addition regarding user experience would be to develop a front-end for the application. Many of the users will not have the necessary knowledge to call a bare HTTP API, and cannot be expected to acquire that knowledge on top of their other responsibilities. As such, a user interface that is friendly, easy to understand and tailored to the services offered by the specific deployment would be necessary.

REFERENCES

[1] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Gener. Comput. Syst.*, vol. 79, pp. 849–861, Feb. 2018.

[2] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, Jun. 2009.

[3] S. Asadi, M. Nilashi, A. R. C. Husin, and E. Yadegaridehkordi, "Customers perspectives on adoption of cloud computing in banking sector," *Inf. Technol. Manage.*, vol. 18, no. 4, pp. 305–330, Dec. 2017. [Online]. Available: https://link.springer.com/article/10.1007/s10799-016-0270-8

[4] F. Koch, M. D. Assunção, C. Cardonha, and M. A. S. Netto, "Optimising resource costs of cloud computing for education," *Future Gener. Comput. Syst.*, vol. 55, pp. 473–479, Feb. 2016.

[5] V. Casola, A. Castiglione, K. K. R. Choo, and C. Esposito, "Healthcare-related data in the cloud: Challenges and opportunities," *IEEE Cloud Comput.*, vol. 3, no. 6, pp. 10–14, Nov. 2016.

[6] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 430–447, Mar. 2018.

[7] T. Kiss, P. Kacsuk, J. Kovacs, B. Rakoczi, A. Hajnal, A. Farkas, G. Gesmier, and G. Terstyanszky, "MiCADO-microservice-based cloud application-level dynamic orchestrator," *Future Gener. Comput. Syst.*, vol. 94, pp. 937–946, May 2019.

[8] A. Abdelmaboud, D. N. A. Jawawi, I. Ghani, A. Elsafi, and B. Kitchenham, "Quality of service approaches in cloud computing: A systematic mapping study," *J. Syst. Softw.*, vol. 101, pp. 159–179, Mar. 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121214002830#bbib0066

[9] M. H. Ghahramani, M. Zhou, and C. T. Hon, "Toward cloud computing QoS architecture: Analysis of cloud systems and cloud services," *IEEE/CAA J. Autom. Sinica*, vol. 4, no. 1, pp. 6–18, Jan. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7815547/

[10] I. M. A. Jawarneh, P. Bellavista, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli, and F. Bosi, "QoS and performance metrics for container-based virtualization in cloud environments," in *Proc. 20th Int. Conf. Distrib. Comput. Netw.*, Jan. 2019, pp. 178–182, doi: 10.1145/3288599.3288631.

[11] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, Aug. 2020. [Online]. Available: https://www.mdpi.com/ 1424-8220/20/16/4621

[12] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2019, pp. 33–40.

[13] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of Kubernetes pods," in *Proc. NOMS - IEEE/IFIP Netw. Operations Manage. Symp.*, Apr. 2020, pp. 1–5.

[14] Kubernetes. (2022). *Autoscaler*. [Online]. Available: https://github.com/kubernetes/autoscaler

[15] AWS. *Auto Scaling Documentation*. Accessed: Mar. 14, 2022. [Online]. Available: https://docs.aws.amazon.com/autoscaling/index.htm

[16] Microsoft Azure. *No Azure Autoscale*. Accessed: Mar. 14, 2022. [Online]. Available: https://azure.microsoft.com/en-us/features/autoscale

[17] Kubernetes. *Horizontal Pod Autoscaler*. Accessed: Mar. 14, 2022. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[18] Q. Wu, J. Yu, L. Lu, S. Qian, and G. Xue, "Dynamically adjusting scale of a Kubernetes cluster under QOS guarantee," in *Proc. IEEE 25th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2019, pp. 193–200.

[19] S. Taherizadeh and M. Grobelnik, "Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications," *Adv. Eng. Softw.*, vol. 140, Apr. 2020, Art. no. 102734. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997819304375

[20] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, and X. Liu, "SmartVM: A SLA-aware microservice deployment framework," *World Wide Web*, vol. 22, pp. 275–293, Jan. 2019, doi: 10.1007/s11280-018-0562-5.

[21] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, May 2018.

[22] Ansible. (2021). *Ansible, Simple, Agentless IT Automation*. [Online]. Available: https://www.ansible.com/

[23] Cryptsetup. (2016). *cryptsetup/cryptsetup GitLab*. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup

[24] MongoDB. (2020). *The Most Popular Database for Modern Apps|MongoDB*. [Online]. Available: https://www.mongodb.com/

[25] Elasticsearch. (2021). *Elasticsearch is a Distributed, RESTful Search and Analytics Engine Capable of Addressing a Growing Number of Use Cases*. [Online]. Available: https://www.elastic.co/elasticsearch/

[26] Google. (2014). *GitHub—Google/Cadvisor: Analyzes Resource Usage and Performance Characteristics of Running Containers*. [Online]. Available: https://github.com/google/cadvisor

[27] MetalLB. *MetalLB, Bare Metal Load-Balancer for Kubernetes*. Accessed: Mar. 14, 2022. [Online]. Available: https://metallb.universe.tf/
[28] Ambassador. *Self-Service Comprehensive Edge Stack for Kubernetes: Ambassador*. Accessed: Mar. 14, 2022. [Online]. Available: https://www.getambassador.io
[29] K. Takahashi, K. Aida, T. Tanjo, and J. Sun, "A portable load balancer for Kubernetes cluster," in *Proc. Int. Conf. Proc. Series. Assoc. Comput. Mach.*, Jan. 2018, pp. 222–231.
[30] H. V. Netto, L. C. Lung, M. Correia, and A. F. Luiz, "State machine replication in containers managed by kubernetes," *J. Syst. Archit.*, vol. 73, pp. 53–59, 2017, [Online]. Available: https://www.sciencedirect.com/ science/article/pii/S1383762116302752
[31] Pivotal Software. (2019). *Messaging That Just Works—RabbitMQ*. [Online]. Available: https://www.rabbitmq.com/
[32] P. Pièol. (2020). *Data Science Python*. [Online]. Available: https://github.com/perepinol/ DataSciencePython
[33] E. J. Ghomi, A. M. Rahmani, and N. N. Qader, "Load-balancing algorithms in cloud computing: A survey," *J. Netw. Comput. Appl.*, vol. 88, pp. 50–71, Jun. 2017.
[34] J. O. Gutierrez-Garcia and A. Ramirez-Nafarrate, "Agent-based load balancing in Cloud data centers," *Cluster Comput.*, vol. 18, no. 3, pp. 1041–1062, Sep. 2015.
[35] A. Bala and I. Chana, "Prediction-based proactive load balancing approach through VM migration," *Eng. Comput.*, vol. 32, no. 4, pp. 581–592, Oct. 2016.
[36] Python Software Foundation. (2019). *PIP—The Python Package Installer*. [Online]. Available: https://pip.pypa.io/en/stable/

LLUÍS MAS RUÍZ received the B.Sc. and M.Sc. degrees in computer science from the University of Lleida, in 2021. His research interests include web application and API development, as well as cloud computing and microservice architectures.

PERE PIÑOL PUEYO received the B.Sc. degree in computer science from the University of Lleida, in 2020. His research interests include web application and API development, as well as cloud computing and microservice architectures.

JORDI MATEO-FORNÉS received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the Escola Politècnica Superior, Universitat de Lleida (UdL), in 2012, 2013, and 2019, respectively. He is currently a Lecturer at the University of Lleida and a member of the Distributed Computing Group. His research interests include different topics related to data science fields such as operations research (stochastic optimization), high-performance computing (parallelization of algorithms), cloud computing, big data, the Internet of Things (IoT), decision support systems, e-health, and agriculture 4.0.

JORDI VILAPLANA MAYORAL received the Ph.D. degree in computer science. He is currently a Professor at the University of Lleida and a member of the Distributed Computing Group. His research interests include cloud computing, e-health and m-health, big data, and machine learning applied to the health field. He works in multiple interdisciplinary projects alongside clinicians, psychologists, statisticians, and mathematicians. He uses telemedicine techniques, web-based applications, smartphone applications, image recognition, data entry, big data storage, processing and visualization, machine learning algorithms, and deep learning techniques.

FRANCESC SOLSONA TEHÀS received the B.S., M.S., and Ph.D. degrees in computer science from the Universitat Autònoma de Barcelona, Spain, in 1991, 1994, and 2002, respectively. Currently, he is a Full Professor with the Department of Computer Science, University of Lleida, Spain. His research interests include parallelization of algorithms, coscheduling, cluster computing, multicluster computing, P2P computing, cloud computing, desktop grid computing, scheduling, optimization, multi-stage linear and stochastic programming, reconstruction of metabolic pathways, bioinformatics, ecosystems simulation, e-health, big data, and the IoT.