TREBALL FINAL DE MÀSTER

ESCOLA
POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA
INSPIRING THE FUTURE

Estudiant: **Lluís Mas Ruiz**

Titulació: Màster en Enginyeria Informàtica

Títol de Treball Final de Màster: A new proposal to extend a private cloud to a QoS-aware container-based architecture

Director/a: Jordi Mateo, Francesc Solsona

Presentació

Mes: Juliol

Any: 2021

Abstract

Cloud systems and microservices are becoming a powerful tool 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 rising in popularity. 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. Moreover, these systems require the capacity to integrate into current systems while meeting the Quality of Service (QoS) constraints. Computing providers are mainly based on data centers formed of huge and heterogeneous virtualized systems, which are continuously growing and diversifying with time. The primary purpose of this work is to present a cloud architecture based on containers aimed at guaranteeing a defined level of QoS regarding cost, resource usage, and service level agreement. The main contribution of this novel architecture is its adaptability to underlying virtualized systems without having to reinstall it in each framework forming the datacenters. The results obtained provide useful guidelines for cloud architects.

1 Introduction

The popularity of cloud usage has grown dramatically in recent times, see [1,2]. There is an increasing number of companies deploying public and private clouds. 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 the 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. In an effort 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 helps not only in 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 aimed at solving particular tasks. Nevertheless, the usage of microservices in the cloud needs to mitigate different problems compared to traditional cloud services, such as communication growth.

With the rise in the adoption of the cloud, one of the main issues about delivering services in the public cloud is trust [8]. Companies and clients do not rely on public clouds when they want guarantees about data privacy or data security. Moreover, government regulations of cloud providers make it more difficult to increase trust. Therefore, cloud architects are suggesting hybrid clouds as the best solution. This way, hybrid clouds take advantage of locally trusted resources for sensitive applications, and the promise of the virtually infinite resources that could be offered by public clouds.

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 [9], and is regulated by service-level agreements (SLAs). A SLA is a contract between clients and providers that expresses the price for a service, the QoS levels required during the provision of the service, and the penalties associated with violations of the SLA. Hence, service providers must design these contracts very carefully to maintain user confidence and avoid revenue loss [10]. The quality of service and user satisfaction are directly related. A lower level of QoS due to a SLA violation leads to a decrease in user satisfaction. The main challenge for a service provider is to determine the best trade-off between profit and customer satisfaction. Monitoring QoS constraints is key to ensuring SLA constraints and service alignment to QoS compliance [11].

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 metric is the CPU utilisation 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. In this work, the challenge is to design and implement a monitoring service that tracks and integrates monitoring data from all microservices and datacenter resources. With this information, the QoS service implements policies and performs actions to guarantee the SLAs and decreases the impact of run-time such uncertainties as service failures, load-balancing efficiency, or overloading.

To address these requirements regarding **QoS**, this work makes the following contributions:

- Designing and implementing a QoS-aware cloud architecture able to scale up and down according to IaaS and SaaS constraints.
- Implementing a Kubernetes cluster on top of a virtual machine to migrate a current private cloud based on OpenNebula to achieve the benefit of both virtual machines and containers with minimum adaptation cost. It could, however, be implemented on any other private cloud regardless of the platform.
- Fitting new emerging technology in controlled and elastic architectures aimed at guaranteeing SLAs.

- Highlighting the advantages of using container-based technology in hybrid clouds focused on business contexts.
- Providing a simple way for users to submit tasks to a cloud without needing to be knowledgeable in cloud technologies.

1.1 Related Work

Cloud computing research faces many challenges to design cloud architectures and services, such as load balancing, hot migration, autoscaling, orchestration, security, user QoS satisfaction, and resource utilisation [6].

In recent years, much research has focused on improving the elastic allocation of cloud resources to meet SLA requirements. In this context, one of the crucial issues is load prediction [12]. The scheduling policies of most SaaS lack predictive load models [13];; they are based on the current workload and are not prepared for a sudden increase in load traffic.

Previous work focuses primarily on the hardware level (IaaS) and proposes load-balancing algorithms that incorporate most metrics, such as throughput, response time, fault tolerance or energy consumption among others, to provide higher resource usage and reduced response time [14–16]. Less research has focused on taking advantage of business requirements and work on the application level (SaaS) [17]. Although many studies deal with load balancing, a critical challenge in the current state of the art is considering it at the hardware and software levels simultaneously. In this work, an architecture is proposed that deals with the virtual machines (hardware level) together with microservices concerning business requirements (software level).

Container-based virtualisation technologies and microservices have become very popular due to their lightweight nature, scalability, and flexibility, among others aspects [18]. Conventional container orchestration platforms usually only offer limited autoscaling functionalities [19–21]. where only resource metrics such as CPU usage are considered. The authors in [18] present a method to establish a CPU utilisation 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 utilisation 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 is challenging and that is where Kubernetes' autoscaler starts to show their flaws as stated in []. This work proposes an autoscaler service that deals with a hybrid environment with legacy applications (with known requirements) jointly with routine creation service (without known requirements).

Another critical point to consider is that while resource saturation possibly causes SLA violations, their exact correlation is yet to be explicitly discovered. An ideal autoscaling strategy is expected to react directly to application-level metric changes, such as increasing SLA violation rates [17]. 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 keep QoS and SLA agreements.

The Horizontal Pod Autoscaler [21] of Kubernetes automatically scales the number of Pods in a replication controller, deployment, replica set, or stateful set based on observed computing resource utilisation. Here, it is essential to highlight that the Kubernetes' Autoscaler is minimally configured and must be customised to overcome the issues commented on above. This work explores how to improve and take advantage of the standard metrics to design custom rules. This is staged in the proposed QoS service.

2 Methodology

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

2.1 Preliminaries

In recent times, cloud computing has revolutionised society and the IT industry. According to the National Institute of Standards and Technology (NIST) [22], cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can rapidly be provisioned and released with minimal management effort or service provider interaction.

In this context, services and resources are offered to users as a service (XaaS). The most common are Infrastructure as a service (IaaS), Platform as a service (PaaS), and Software as a service (SaaS). One of the main advantages of clouds is elasticity: the capacity to adapt to workload changes by providing computing resources on demand.

Virtualisation is a vital concept for achieving elasticity in clouds and obtaining scalable environments able to adapt to users' needs [6].

Virtual machines (VMs) aim to provide an infrastructure to offer services in the cloud. A virtual machine usually contains an operating system, allowing the execution of multiple resource-intensive functions at once, while a container usually does not package anything more significant than an application and all the files necessary to run it, and these are commonly used to deploy single functions that perform specific tasks (microservices).

Microservices are small, independent running modules that are often deployed separately and developed by different teams that need to interact to satisfy specific needs [23]. The main feature of a microservice architecture is the loose coupling of components, which run autonomously and use message-passing communication to exchange information. Microservices adopt open standards such as JSON and XML, allowing the integration and cooperation of different solutions programmed in any language.

Consequently, microservices minimise the risk and impact of introducing new technologies or components into the road-map of existing processes [24], which enables an extremely efficient way of developing applications as many independent teams can work in a service without relying on the rest.

The popularisation of container deployment environments (such as Docker, Ranger, or Kubernetes) and microservice architecture has helped to transform the Infrastructure as a Service into a Software as a Service, alleviating some common SaaS problems at the same time. Consequently, users can pre-build containers and deploy them in the cloud abstracting them from the underlying configuration of the machine where it will be deployed.

The cloud platform chosen to develop this work is OpenNebula — see [25]. This is an enterprise-ready platform that helps build an Elastic Private Cloud. It avoids risks and vendor lock-in by choosing a powerful, but easy-to-use, open-source solution. OpenNebula is based on virtual machines but also allows containerised applications from Kubernetes or Docker to be run.

OpenNebula offers an open and transparent means to build private clouds. See [26]. The Stormy server [27] is a legacy private cloud platform supported by public funding, developed and maintained by the author's research group aimed at assisting researchers and deployed with OpenNebula and KVM. There are several manuscripts, such as [28] or [29], that use the Stormy service as a support platform to achieve their goals.

The main aim of this research is focused on developing a software API (Application Programming Interface), on top of any private cloud, to allow the execution of tasks without requiring a high technical background. On logical levels, this API can be replicated or extended to implement hybrid clouds by communicating with an additional external virtualization layer, such as Amazon Web Services (AWS), Google Kubernetes Engine (GKE), or Microsoft Azure. As mentioned before, a hybrid cloud is composed of two parts: a mixture of a public and a private cloud. In this work, we assume that the public cloud is externalized and we focus on improving the private cloud related aspects.

We propose, as our main contribution, the use of any private cloud with KVM over the bare-metal physical servers (i.e. the Stormy platform), and adding another virtualisation layer using container virtualisation (Google Kubernetes) to implement the private part of a hybrid cloud by improving isolation, resource usage, portability and system management tasks. Furthermore, the addition of virtual machines as middleware between physical resources and containers allows the latter to benefit from the security provided by hypervisors and the virtual machines themselves. The main drawback of this design resides in the performance and energy overhead of an additional virtualisation layer.

Docker [30] is a tool-set for developers to run and manage containers. It allows deploying and runnning *Docker images*, which contain all necessary resources for an application, on containers. These images allow Docker to provide scalability and ease of sharing, among other advantages.

Kubernetes [31] implements a traditional Master/slave model. A Master manages Pods, which deploy Docker containers across multiple Kubernetes workers (VMs). In other words, applications are deployed in Docker contain-

ers with the assistance of the Kubernetes Master. Moreover, applications are divided into one or more tasks executed in one or more containers.

The Kubernetes terminology used in this project is:

- **Node**: a physical or virtual machine available to Kubernetes. Also called **host**.
- **Cluster**: a set of Nodes that pool their resources to run applications. It performs as a transparent interface for the programmer. Kubernetes performs almost all the Cluster-to-host logic.
- **Pod**: a group of containers sharing storage and network. It is the smallest deployable unit in a Kubernetes Cluster.
- **Volume**: an abstraction of a directory in which to keep data. When a container restarts, the files it contains are lost. Volumes are needed to provide persistent storage to Pods. The Volume's type determines how that is achieved.
- **Deployment**: specifies a Pod to be deployed, and how many replicas of it are needed.
- **Service**: defines how a set of Pods (e.g. those in a single Deployment) is to be accessed. In other words, it shows that set.
- **Job**: a Pod or group of Pods that run to complete a task.
- **DaemonSet**: a mechanism to create a replica of the same Pod in each Node.

Throughout this work, all references to any of these concepts are capitalised to assist interpretation, as some of the expressions are also used to convey non-Kubernetes concepts.

2.2 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 integration of legacy applications in the cloud. New functionalities have to coexist with them, and might take advantage of their data and services. Containers are useful for the separation of legacy and new environments. Without using them, 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 — and with other containers. It may also be useful to copy information to more efficient databases or ones more adapted

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 [23]. In other words, it consists of managing the containers' life-cycle: 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 behaviour that is 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 SLAs. 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 improvement by providing a transparent interface.
- Allowing 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 is presented:

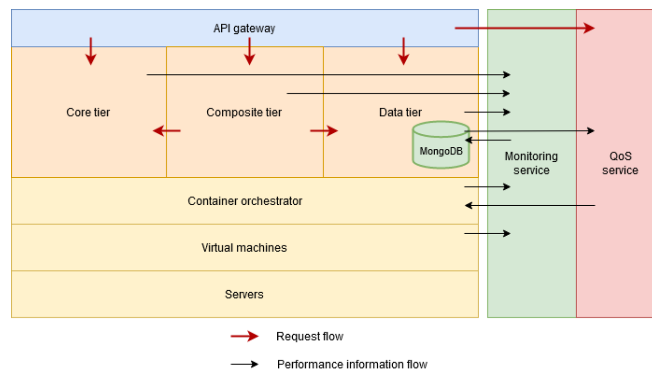

Figure 1: Architecture of eHQoS

- Virtual machines, servers and Nodes: the Cluster is deployed on virtual machines, which act as Kubernetes Nodes and run on a cloud infrastructure.

- Container orchestrator: as explained previously, 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.

  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. Choices are taken based on the cloud's SLAs by using prediction models which aim to maintain the agreed-upon QoS by anticipating changes in the services' demand for resources.

3 Implementation

In this section, a case study of the proposed architecture is presented. First, the virtual machines used are described in subsection 3.1. Afterwards, the services deployed on them through a Kubernetes Cluster are presented. The internal services required by the architecture are first presented in subsection 3.2 and, afterwards, business logic services explore those that directly provide functionality to the end user in subsection 3.3.

The GitHub repository containing the implementation of the framework can be found at
https://github.com/LluisMas/EQoS/.

On top of the framework implementation, it can also be found a set of Ansible see [32] playbooks that can be used to easily deploy the system on a completely new cluster. Ansible is an open-source software provisioning, configuration management and application-deployment tool. It allows to deploy Infrastructure as Code. An Ansible playbook is a set of commands that can be run by anyone. Among Ansible's strengths the most important ones are that it allows the distribution of configuration and provisioning in an easy way on top of their push approach where Ansible only needs to be installed in a single

machine rather than all the cluster like other technologies such as Puppet [33].

The created playbooks and their purpose are the following:

- **prep:** Installs all the dependencies and updates the core files such as /etc/hosts with all the host names from the cluster.
- **registry:** Deploys and configures the node that will act as Docker registry.
- **master:** Configures the Kubernetes master node and creates the cluster.
- **node:** Configures the Kubernetes nodes and joins them to the cluster.
- **deployehqos:** Clones the EHQoS repository, changes all the environment variables and creates the needed configuration files.
- **service:** Deploys all the needed pods for the system to run smoothly.
- **monitor:** Deploys the pods related to the monitoring (Kibana and Elasticsreach).

3.1 Virtual machine setup

The deployment uses nine virtual machines located in a private cloud: a Docker registry, a Master, five 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 are 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 that are related to the functioning of the architecture. The monitor node, on the other hand, hosts the pods related to the performance monitoring.

Their specifications are given in Table 1.

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	3	2	8	15

Table 1: Specifications of the virtual machines in the private cloud

The disk is only used by Kubernetes' and Docker's internal operation. All persistent data regarding the services in the cloud is stored in two external virtual hard disks. One stores sensitive information and is encrypted with Linux Unified Key Setup (LUKS) [34], 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, and are mounted on the Master and shared via NFS.

3.2 Internal Services

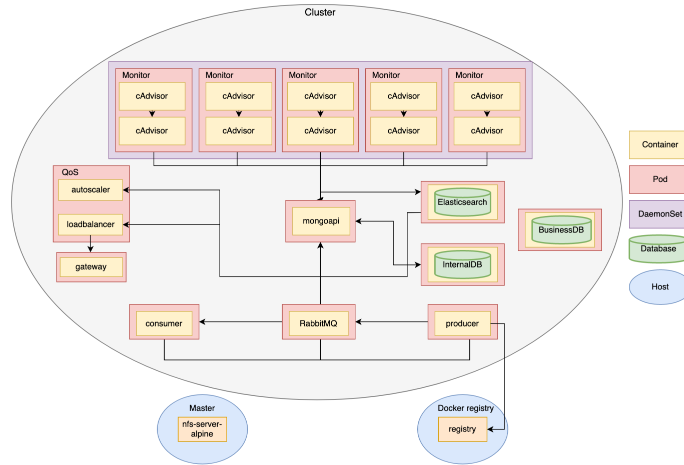

Figure 2: Services included in the implementation and their relationships. Arrows indicate information flow.

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 [35] chosen.

Monitordb is the responsible for storing all the performance log records. As data is written with in high frequency, read periodically and never updated or deleted, the NoSQL database chosen has been Elasticsearch [36] as it is incredibly powerful in this scenario where data is only created or accesses but seldom updated or deleted.

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.

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 performance metric compilation was **cAdvisor**, which is a daemon that collects, aggregates and exports information about hosts and

running containers [37]. 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. With regards to 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, as seen in Figure 3. First, a proxy inside the Cluster which receives traffic from outside and routes it to the appropriate service, which in our case it is the Gateway. Second, a load balancer, outside the Cluster, that shows it to the public through a static IP address and sends all requests to the proxy.

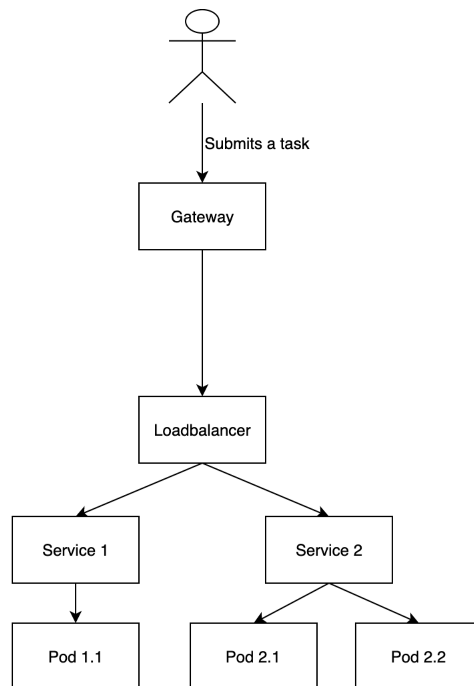

Figure 3: Exposing the Cluster through a gateway

Typically, Kubernetes Clusters are deployed on cloud provider platforms such as AWS or GKE. These already provide their own load balancers, which can be used by developers 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 [38] as load balancer, and Ambassador [39] as 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 the top of open nebula without reconfiguring the legacy cloud system. Kubernetes' Load Balancer was also discarded as it is deemed as not enough in some cases, like our study, as stated in several research papers.

As a workaround, a custom solution has been coded. A Flask API running outside the Cluster, in the Kubernetes Master, acts as entrypoint, 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 **autoscaler**.

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

```
1    {
2      "scaling": {
3        "min_load": 30,
4        "max_load": 60,
5        "max_load_nowait": 90,
6        "wait_seconds": 10,
7        "tolerance": 5
8      },
9      "update_seconds": 5,
10     "exclude": ["monitor"],
11     "over_threshold": 0.5,
12     "under_threshold": 0.5,
13     "grace_period": 20
```

```
14            }
```

- **scaling**: Settings for the *loadtracker* plugin.
  - **min_load**: If the CPU or Memory usage of a Pod are 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 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 a 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**: 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 a 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. This is determined by the system's average resource usage being under a certain threshold (in the current case, 80%). It is used by the **consumer** to prevent scheduling when the platform is overloaded.

In this second version, there has been several improvements and multiple bug fixes. The implementation of **monitorDB** and separating it from **internalDB** is the biggest jump on performance. It was noticed that after a few hours of having the system running, it ran slower, the cause of it was the way the Database was structured, that is why we decided to migrate it to a more efficient technology (Elasticsearch). Also, regarding to performance, some nodes have

been tainted so only certain deployments could be applied to these nodes, this has proved useful to deploy all the service related pods used for the internal functioning of the cluster to a separate node that can scale separately and relief the worker nodes from this task and new queries have also been made to improve the performance of the existing tasks. Using a much bigger test bed allowed us to detect more bug in the implementation, for example, most of the petitions were not retrying after a timeout or not successful answer, this has been fixed, the consumer's connection to RabbitMQ was getting closed making the cluster enter in a critical state, it has also been fixed. Parameter forwarding in **mongoapi** has also been fixed. Some miscellaneous have been carried out as well, some deployments remained as Docker deployments instead of Kubernetes, all of them have been migrated to a Kubernetes approach, more logs have also been added and the correct task status flow has also been implemented, it is a purely informational record as it is not queried by the system but useful for debugging and future monitoring tools nonetheless. A script that launches and automatically retrieves both result and performance while the execution has been implemented.

3.3 Business Logic

businessdb is another Deployment of MongoDB that will contain 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 analysing 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 cloud, using the data contained therein. It supports 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, initialises 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. After that, all data related to the routine — local Docker image and files saved — are removed. The routine's identifier is sent to a queue.

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.

The results are saved in a file named *results.json* and the logs in *log.log* — or printed to standard output/error, which redirect to the same file —, but both are optional. Those are then automatically uploaded to the previously mentioned MongoDB.

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 4.

Communication between the **producer** and the **consumer** is achieved through RabbitMQ [40], a queue messaging system.

4 Results

4.1 Computational Benchmarks

The tests performed focus on stressing the platform and examining its behaviour. First, an overall performance analysis is conducted, which will shed light on the general behaviour of the platform and expose some important issues to be considered. Afterwards, the specific mechanisms of the architecture in regards to SLA compliance (scaling and job queuing) are specifically examined.

Given that, with the current implementation, the monitored metrics are CPU and Memory usage, testing also revolves around these. Specifically, the metric exploited to stress the platform is both Memory and CPU usage, as it is fairly simple to create Deployments or Jobs that waste cycles on irrelevant operations. A cause and effect relationship is then established between the stress and the consequent events produced in the platform.

4.1.1 Performance during job execution

The first test consists of the submission of ten identical jobs that train and test supervised models on two samples stored in **businessdb**. The aim of this test is

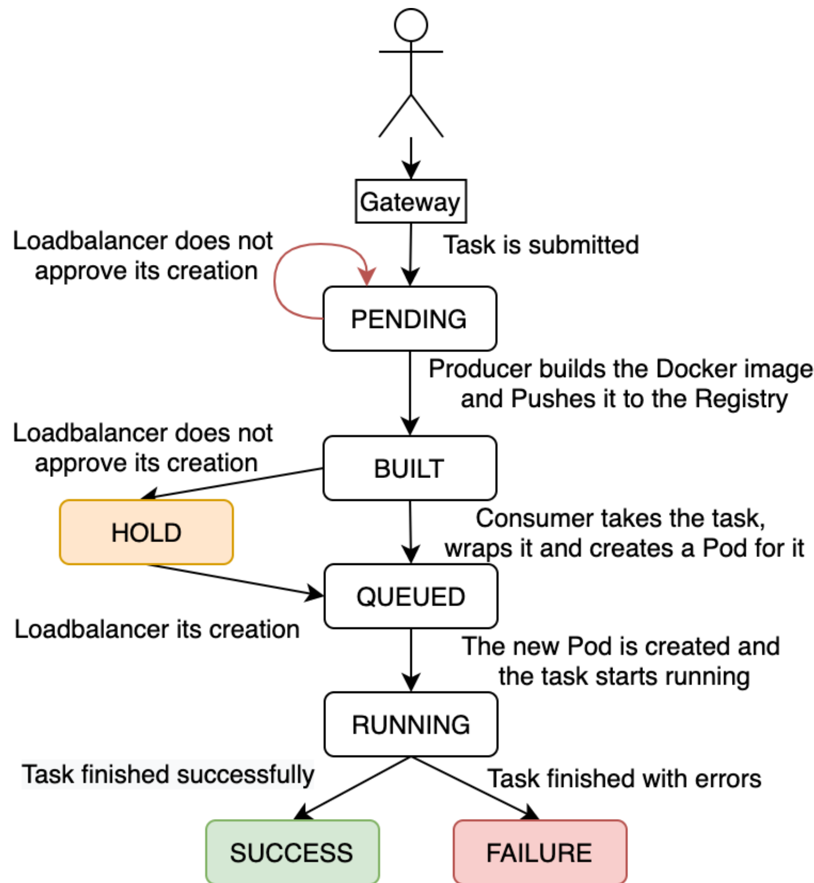

Figure 4: State diagram of a routine.

to stress the platform and analyse its behaviour. The script used can be found in GitHub[1].

When submitting a job, Alpine's Docker image [41] (responsible for executing jobs) requires several additional Linux packages: SciPy [42], NumPy [43] and Pandas [44], which are mandatory for running the jobs used in this test and also the vast majority of data analysis scripts that users can write in python. Thus, the Dockerfile was re-implemented to use a native image of Python:3.7, which is lighter and more specific for executing python-based jobs. Note that this is not a mandatory choice, and making this decision has an important drawback which must be mentioned: the architecture loses the capability to run jobs written in other languages, such as R. Possibilities that would increase the flexibility of job submission and execution are discussed in section 5.

[1] https://github.com/perepinol/DataSciencePython/blob/master/logistic_regression_updated.py

Figure 5 depicts the execution of the 1000 identical jobs in the architecture implemented formed by 5 different hosts. The jobs used in this experiment were CPU intensive, so the Memory monitoring metric was discarded from the plot. The lines represent host saturation regarding CPU usage over time.

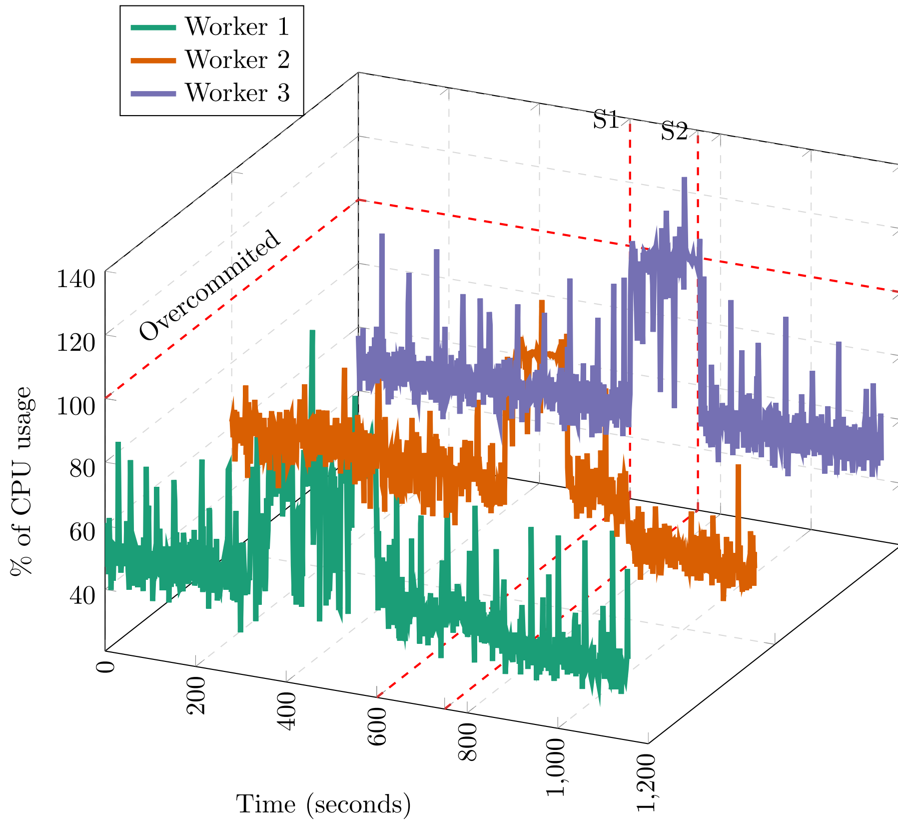

Figure 5: CPU and Memory performance over time on submission of ten identical jobs

Several interesting facts can be drawn from Figure 5. First, there are several spikes between *t=600s* and *t=750s* (denoted by dashed lines S1 and S2), exceeding 100% CPU usage in one of the hosts (Worker 3). This situation is explained because the OpenNebula legacy cloud is configured with the option to exceed the maximum amount of resources reserved for a virtual machine, if and only if the overall system is not saturated. This allows a spike in usage over the expected resources available, providing a certain additional margin in case a specific machine is being overloaded. However, the QoS service should not rely on it. This situation should be prevented as much as possible while ensuring the SLAs.

Secondly, it can also be seen that when jobs are executed in the platform governed by the Kubernetes Cluster, they are mostly scheduled to run on the same host. *A priori*, the platform does not know the amount of resources required to complete the job, so it is launched whenever the host is not saturated. This can lead to situations such as the one shown around *t=600s*, labelled *S1*, where the job submitted saturates the CPU resources of the host. This may affect the correct execution of other services or Jobs located in the same host. If Deployments are not correctly managed (*e.g.* **internaldb**, **mongoapi**), the whole system will become unavailable, stressing the importance of hot migration of Pods between hosts.

4.2 QoS evaluation

4.2.1 Job creation with overloaded platform

In this test, the behaviour of the system in job creation is examined. First, the CPU and Memory usage of all hosts (Workers) is increased to around 90% through a DaemonSet, **waster**[2]. Then, a job is submitted. Its main purpose is to check the correctness of the QoS policies implemented in the platform, which must prevent the execution of this job while the system resource occupancy is over 80%. The job should be put on hold by the **consumer**, and then, once **waster** is deleted, be executed in the Kubernetes Cluster.

Figure 6 clearly 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 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 it is 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.

4.2.2 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.

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 so as to avoid overloading

[2] `https://github.com/LluisMas/EQoS/tree/main/images/memorywaster`

[3] `https://gitlab.com/perepinol/pharmacological-data-analysis/-/blob/master/test-routines/mongo-flood/mongoflood.py`

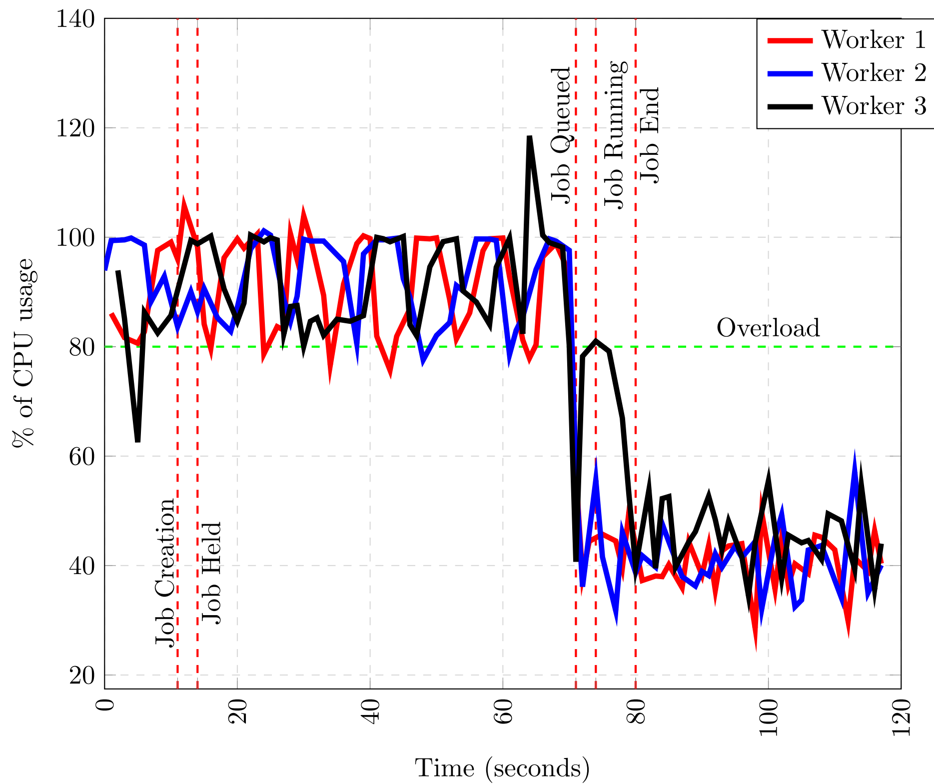

Figure 6: Events in job creation related to the Cluster's CPU usage

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 3):

```
{
  "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
}
```

Figure 7 provides information on the scaling events during the test. This logging information is essential for understanding the behaviour of the system under the test conditions, and also to comprehend the results and conclusions achieved from this test. Note that the last number in each line indicates the number of replicas (Pods executing the same Deployment).

```
2020-06-13 12:09:56,076 - INFO - Scaling businessdb to 2
2020-06-13 12:10:21,081 - INFO - Scaling businessdb to 1
2020-06-13 12:10:45,973 - INFO - Scaling businessdb to 2
2020-06-13 12:11:10,991 - INFO - Scaling businessdb to 1
2020-06-13 12:11:46,112 - INFO - Scaling businessdb to 2
2020-06-13 12:12:16,053 - INFO - Scaling businessdb to 1
```

Figure 7: Logs of **autoscaler** during the test

As can be seen in the figure, the Deployment (**businessdb**) has scaled up and down three times. As a result, four Pods have been involved in the test, but only two at most occurred simultaneously 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 8. 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 illustrated in the log (Figure 7), represented by lines (U1, U2, U3, D1, D2, D3; where letters U and D are used to indicate scaling up and down events, respectively).

This first scaling event seen in Figure 8 (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 the **Replica 1**. This is illustrated by D1, at *t=40s*. This situation, where the Deployment is scaled up and down, happens two more times (**Replica 2** and **Replica 3**), denoted by (U2,D2,U3,D3). This behaviour 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 **autoscaler** scales down until the desired value is reached, only maintaining the initial (**Main Pod**).

5 Discussion

The findings of this work are similar to [11] and demonstrate that monitoring QoS constraints is the key to ensuring that SLAs are satisfied.

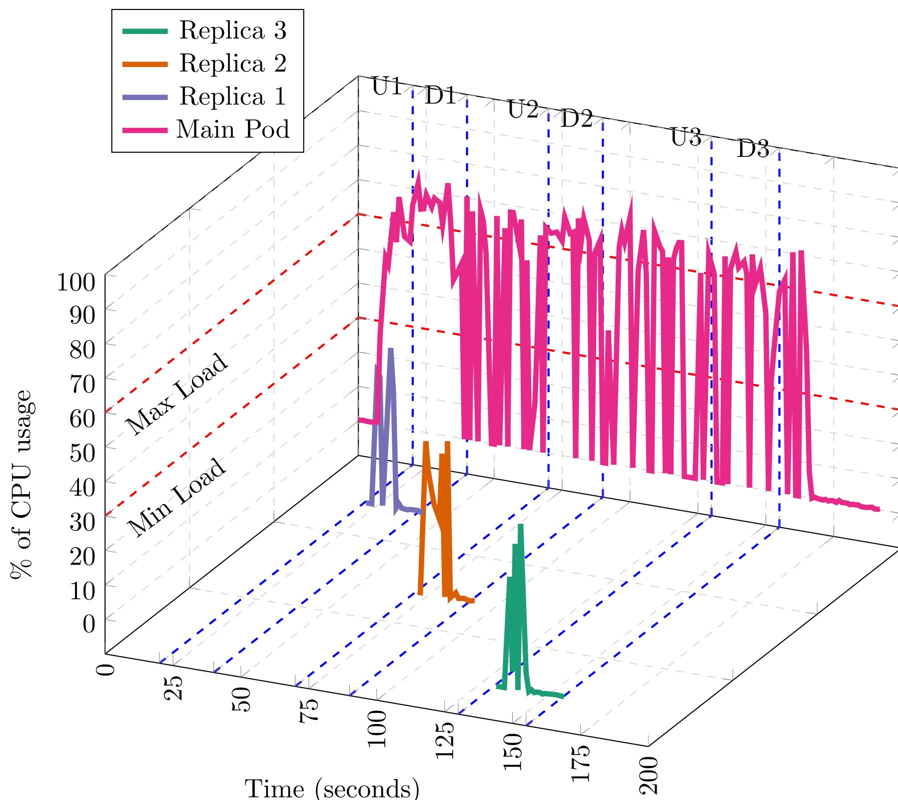

Figure 8: CPU load and scaling/descaling events while a test job is stressing businessdb

The results obtained on the implementation described in section 3 confirm that the architecture proposed is viable in regards to maintaining SLA compliance both at a hardware and software levels. Even though 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 common SLA requirements has been encouraging.

These results go beyond previous reports [14–17], showing that it is complex but viable to deal with both levels of abstraction in the design and implementation of QoS regarding SLA.

Furthermore, this work ties well with previous studies where the authors conclude that it is possible to adjust the Kubernetes Cluster [18] if the requirements are known *a priori*. 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. Throughout the different tests, several issues have been revealed

that need to be addressed before applying the architecture on a commercial level. These are described below, and proposed as future work in section 7.

In relation to the first test (subsubsection 4.1.1), the fact that Alpine Docker images require additional Linux packages brings to light an important issue about the routine template's design. It is not possible, at the same time, 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 containerisation (e.g. minimum dependencies and weight).

In fact, the implementation gives the user the option to submit a Pip (Python Package Installer [45]) requirements file, which was meant to reduce dependencies in the routine template. An option would be to extend this behaviour to allow users to submit their own Dockerfile, although this would go against 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 third test (subsubsection 4.2.2) justifies the need to measure container load using the container's own resource limit as a reference, instead of using the hosts (VMs) own resources 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 load is excessive, minimising replication operations is useful for keeping the architecture-related overhead to a minimum.

In the graph for this same test (Figure 8) a certain desynchronisation between peak loads and scaling events can be observed. This has two causes:

1. The **monitor** DaemonSet and the **autoscaler** are not synchronised.

2. The scaling process has to obtain data from Kubernetes API before scaling.

The *update_time* of businessdb 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 causes a great rise in resource usage. That is why we migrated from an initial approach in MongoDB to a 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 in order to keep the main database

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

6 Conclusions

In this study, a cloud architecture aimed at improving performance in business was proposed. This is one capable of fulfilling QoS requirements automatically and transparent to stakeholders. The results highlight the computational benefits of monitoring, adjusting, and autoscaling services.

The question under discussion in this study was the applicability of a container-based layer on the top of a private legacy cloud. The results of this research provide confirmatory evidence of the benefits of implementing a new virtualization layer based on containers to ensure QoS in both IaaS and SaaS levels.

The data yielded by this study provide substantial evidence that the proposed architecture is capable of ensuring QoS.

Nevertheless, migrating traditional environments requires an initial effort and advanced code skills to prepare and administrate the new infrastructure. It also means increasing the complexity of implementation in regards to developing the Pods, managing the communication methods between these, and also integrating API developments.

By using OpenNebula, we tested the hypothesis that the proposed platform can be adopted as a new layer without reconfiguring the system thus minimizing the impact of adoption and increasing the benefits of both virtualization layers. The underlying evidence from the architecture proposed states that it can be easily deployed in any other private cloud as it does not use any OpenNebula specific configuration.

In this context, it is crucial to know whether migrating to this type of architecture is worthwhile. Although the adoption of the proposed architecture has higher initial costs due to migration and system reconfiguration, these costs are compensated for in the long term, mainly due to the reduction in maintenance cost and ease of integration of new components into the system.

The architecture proposed provides a simple way to change components without spreading problems throughout the platform and to adapt real-time needs to increase customer revenue by reducing customer dissatisfaction due to poor service quality, while also reducing lost opportunities due to an inability to deliver features.

The complex environment of business makes the adoption of this kind of architecture worthwhile. The maintenance cost and revised cost in the introductions of new features or small updates in monolithic applications can interrupt their stability, increasing such business risks as in customer satisfaction and service quality. Moreover, uncertain demand, load, and system failures also contribute to increasing these business risks.

In conclusion, all these findings are encouraging to continue this work and

confirm that there is much work to do. However, this is the path to follow to evolve current cloud architectures into more robust, automatic, elastic, and QoS-aware regarding the SLAs. We believe that these findings have generalizable research value to the field and can be applied either in private/hybrid clouds or in moving traditional non-cloud servers/architectures to the cloud so as to achieve all their benefits.

7 Future work

As mentioned in section 5, this project still requires several key features before achieving its potential. For example, Cron Jobs to remove performance records, other database objects, 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 two 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. In regards to the former, a global configuration that includes parameters for all services in the architecture could be used to minimise 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 visualisation for system administrators or optimisation 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.

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 work 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 Generation Computer Systems*, 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 deliv-

ering computing as the 5th utility," *Future Generation Computer Systems*, 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," *Information Technology and Management*, 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. Netto, "Optimising resource costs of cloud computing for education," *Future Generation Computer Systems*, 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 Computing*, 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 Transactions on Services Computing*, 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 Generation Computer Systems*, vol. 94, pp. 937–946, may 2019.

[8] L. van der Werff, G. Fox, I. Masevic, V. C. Emeakaroha, J. P. Morrison, and T. Lynn, "Building consumer trust in the cloud: an experimental analysis of the cloud trust label approach," *Journal of Cloud Computing*, vol. 8, no. 1, pp. 1–17, dec 2019.

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

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

[11] I. M. Al Jawarneh, G. Martuscelli, P. Bellavista, R. Montanari, F. Bosi, L. Foschini, and A. Palopoli, "QoS and performance metrics for container-based virtualization in cloud environments," in *ACM International Conference Proceeding Series*, 2019, pp. 178–182. [Online]. Available: https://doi.org/10.1145/3288599.3288631

[12] P. Kumar and R. Kumar, “Issues and challenges of load balancing techniques in cloud computing: A survey,” *ACM Computing Surveys*, vol. 51, no. 6, feb 2019.

[13] W. T. Tsai, X. Y. Bai, and Y. Huang, “Software-as-a-service (SaaS): Perspectives and challenges,” *Science China Information Sciences*, vol. 57, no. 5, pp. 1–15, 2014.

[14] E. Jafarnejad Ghomi, A. Masoud Rahmani, and N. Nasih Qader, “Load-balancing algorithms in cloud computing: A survey,” pp. 50–71, jun 2017.

[15] J. O. Gutierrez-Garcia and A. Ramirez-Nafarrate, “Agent-based load balancing in Cloud data centers,” *Cluster Computing*, vol. 18, no. 3, pp. 1041–1062, sep 2015.

[16] A. Bala and I. Chana, “Prediction-based proactive load balancing approach through VM migration,” *Engineering with Computers*, vol. 32, no. 4, pp. 581–592, oct 2016.

[17] 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, no. 1, pp. 275–293, 2019. [Online]. Available: https://doi.org/10.1007/s11280-018-0562-5

[18] Q. Wu, J. Yu, L. Lu, S. Qian, and G. Xue, “Dynamically adjusting scale of a kubernetes cluster under QoS guarantee,” in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2019-Decem. IEEE Computer Society, dec 2019, pp. 193–200.

[19] AWS, “Auto Scaling Documentation.” [Online]. Available: https://docs.aws.amazon.com/autoscaling/index.html

[20] Microsoft Azure, “No Azure Autoscale.” [Online]. Available: https://azure.microsoft.com/en-us/features/autoscale/

[21] Kubernetes, “Horizontal Pod Autoscaler.” [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[22] P. M. Mell and T. Grance, “The NIST definition of cloud computing,” National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., 2011. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[23] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead,” pp. 24–35, may 2018.

[24] H. Knoche and W. Hasselbring, “Using Microservices for Legacy Software Modernization,” *IEEE Software*, vol. 35, no. 3, pp. 44–49, may 2018.

[25] OpenNebula, “OpenNebula.” [Online]. Available: http://opennebula.org

[26] K. Kostantos, A. Kapsalis, D. Kyriazis, M. Themistocleous, and P. Rupino Da Cunha, "OPEN-SOURCE IAAS FIT FOR PURPOSE: A COMPARISON BETWEEN OPENNEBULA AND OPENSTACK," Tech. Rep. 3, 2013.

[27] Stormy, "Stormy — Stormy support site." [Online]. Available: https://stormy.udl.cat/

[28] J. Mateo-Fornes, F. Solsona-Tehas, J. Vilaplana-Mayoral, I. Teixido-Torrelles, and J. Rius-Torrento, "CART, a Decision SLA Model for SaaS Providers to Keep QoS Regarding Availability and Performance," *IEEE Access*, vol. 7, pp. 38 195–38 204, 2019.

[29] J. Mateo, L. M. Pla, F. Solsona, and A. Pagès, "A production planning model considering uncertain demand using two-stage stochastic programming in a fresh vegetable supply chain context," *SpringerPlus*, vol. 5, no. 1, pp. 1–16, dec 2016. [Online]. Available: https://link.springer.com/articles/10.1186/s40064-016-2556-zhttps://link.springer.com/article/10.1186/s40064-016-2556-z

[30] Docker Inc., "Docker Documentation," *Docker Docs*, 2018. [Online]. Available: https://docs.docker.com/

[31] The Linux Foundation, "Production-Grade Container Orchestration - Kubernetes," 2020. [Online]. Available: https://kubernetes.io/

[32] Ansible, "Ansible, Simple, agentless IT automation." [Online]. Available: https://www.ansible.com/

[33] Puppet, "Puppet is an open-source software configuration management tool." 2021. [Online]. Available: httpshttps://puppet.com/

[34] Cryptsetup, "cryptsetup / cryptsetup · GitLab," 2016. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup

[35] MongoDB, "The most popular database for modern apps — MongoDB," 2020. [Online]. Available: https://www.mongodb.com/

[36] Elasticsearch, "Elasticsearch is a distributed, RESTful search and analytics engine capable of addressing a growing number of use cases. — Elasticsearch," 2021. [Online]. Available: https://www.elastic.co/elasticsearch/

[37] Google, "GitHub - google/cadvisor: Analyzes resource usage and performance characteristics of running containers." 2014. [Online]. Available: https://github.com/google/cadvisor

[38] MetalLB, "MetalLB, bare metal load-balancer for Kubernetes." [Online]. Available: https://metallb.universe.tf/

[39] Ambassador, “Self-service comprehensive edge stack for Kubernetes: Ambassador.” [Online]. Available: https://www.getambassador.io/

[40] Pivotal Software Inc., “Messaging that just works – RabbitMQ,” 2019. [Online]. Available: https://www.rabbitmq.com/

[41] Alpine, “alpine - Docker Hub.” [Online]. Available: https://hub.docker.com/_/alpine

[42] S. developers, “SciPy.org — SciPy.org,” p. consulted on: 07/06/2017, 2019. [Online]. Available: https://www.scipy.org/

[43] Numpy, “NumPy.” [Online]. Available: https://numpy.org/

[44] W. A. McKinney, “Pandas : a Python Data Analysis Library,” *New York*, pp. 1–26, 2009. [Online]. Available: https://pandas.pydata.org/

[45] Python Software Foundation, “pip - The Python Package Installer,” 2019. [Online]. Available: https://pip.pypa.io/en/stable/