Real-time resource scaling platform for Big Data workloads on serverless environments

Jonatan Enes*, Roberto R. Expósito, Juan Touriño

Universidade da Coruña, CITIC, Computer Architecture Group, Campus de A Coruña, Spain

Abstract

The serverless execution paradigm is becoming an increasingly popular option when workloads are to be deployed in an abstracted way, more specifically, without specifying any infrastructure requirements. Currently, such workloads are typically comprised of small programs or even a series of single functions used as event triggers or to process a data stream. Other applications that may also fit on a serverless scenario are stateless services that may need to seamlessly scale in terms of resources, such as a web server. Although several commercial serverless services are available (e.g., Amazon Lambda), their use cases are mostly limited to the execution of functions or scripts that can be adapted to predefined templates or specifications. However, current research efforts point out that it is interesting for the serverless paradigm to evolve from single functions and support more flexible infrastructure units such as operating-system-level virtualization in the form of containers. In this paper we present a novel platform to automatically scale container resources in real time, while they are running, and without any need for reboots. This platform is evaluated using Big Data workloads, both batch and streaming, as representative examples of applications that could be initially regarded as unsuitable for the serverless paradigm considering the currently available services. The results show how our serverless platform can improve the CPU utilization by up to 77% with an execution time overhead of only 6%, while remaining scalable when using a 32-container cluster.

Keywords: Serverless computing, Big Data, resource scaling, operating-system-level virtualization, container cluster

1. Introduction

Among the current novel workload execution paradigms, serverless computing is one of the most studied and expected to grow in use over the next years [1, 2, 3]. This paradigm is especially tailored to the execution of code functions triggered on an event-based manner, as well as for applications and workloads that are mainly stateless and have a potentially time-varying load, like web services and REST APIs. However, the serverless paradigm not only abstracts code execution from the underlying infrastructure, but also allows for a different resource accounting compared to existing services such as IaaS/PaaS. For applications deployed on a serverless platform, it is expected that: 1) they are only given the resources strictly needed and used by them; and 2) if the user is billed for the resource usage like in public cloud scenarios, only the amount actually used should be accounted for. If both conditions apply, the user ideally only pays for the resources consumed by the application, while also avoids suffering any performance penalty from the infrastructure management. Although several options exist to execute easily adaptable applications or plain functions as code on a serverless environment, there is still little research that shows how more complex applications (e.g., Big Data workloads) behave in a scenario where resources could vary according to the workload demands.

*Corresponding author

Email addresses: `jonatan.enes@udc.es` (Jonatan Enes), `rreye@udc.es` (Roberto R. Expósito), `juan@udc.es` (Juan Touriño)

Furthermore, the use of software containers is becoming widespread not only for their lighter virtualization capabilities, but also as a means of easily packaging applications to make them portable and deployable across a wide range of infrastructure providers.

In this paper we bring forward a novel platform that is capable of dynamically and continuously scale the resources used by Big Data applications hosted in clusters of containers, giving the user fully capable instances like in a cloud scenario but with a resource accounting from the serverless paradigm. Such dynamic scaling is possible by relying on real-time, precise container-based resource monitoring and a feedback loop architecture, which is also able to be dynamically configured as needed and scale according to the size of the infrastructure to be managed. Nevertheless, it is important to note that this kind of resource scaling while the application is running and without restarting the instances requires support by the underlying virtualization technology used. For this reason, we rely on operating-system-level virtualization in the form of Linux Containers (LXC) [4]. This technology allows to change the container resources in real time thanks to the use of cgroups, a kernel-backed accounting feature that allows to limit the resources of groups of processes. Using this platform we study the behaviour of representative Big Data workloads and their resource usages when they are applied real-time resource scaling according to their real usage. In addition, we aim to prove that for CPU and memory resources the usage amount is generally independent and loosely coupled from the available or reserved amount. This in turn implies that, from the user's point of view, when a serverless platform is used there should not be significant differences in performance or overall behaviour.

The developed architecture is specifically designed to be scalable and to remain as non-intrusive as possible, aiming to provide the applications with transparent and automatic resource scaling. The platform is thoroughly evaluated using both batch and streaming workloads deployed on two different computing facilities: a Big Data cluster at the Galicia Supercomputing Center (CESGA) [5] and the Grid'5000 infrastructure [6]. The main contributions of this paper are:

- A state-of-the-art survey on the technologies already available to support serverless execution, as well as the latest research works on how to improve such platforms to support more flexible approaches.
- A detailed description of a novel platform as a key-enabling technology to automatically scale the resources of a cluster of containers in real time.
- A thorough evaluation of representative Big Data workloads in terms of resource accounting, execution time overhead and scalability that shows the benefits of the proposed platform.

The rest of the paper is organized as follows: Section 2 introduces the background and the terminology used throughout the paper. The current state of the art along with the related works are presented in Section 3. Section 4 describes both the design and implementation details of the architecture developed for this platform. Section 5 presents the procedure used to perform two resource analysis functionalities: a real-time scaling while the workload is running and a subsequent resource accounting once the workload has finished. All the configuration options are also explained in this section. The experimental configuration and the analysis of results are presented in Section 6. Finally, conclusions are exposed in Section 7.

2. Background

To properly understand the chosen technologies and architecture design that makes possible our resource scaling platform, some specific clarifications have to be made regarding container-based virtualization solutions (Section 2.1) and the serverless paradigm as a whole (Section 2.2).

2.1. Container-based virtualization and monitoring

In order to lighten the performance overhead of virtualization and provide real-time scaling of resources, while still offering some kind of application isolation, LXC containers are used. Such containers combine the use of namespaces and cgroups, which together allow to create isolated environments where processes can be executed and file systems deployed. With LXC in particular, it is possible to deploy containers that replicate any Linux-kernel-based operating system (e.g., Debian, CentOS), being such containers very close from the user's point of view to a hypervisor-based virtual machine. Nevertheless, it is important to note that the platform presented in this paper could be easily integrated with other container technologies also backed by cgroups like Docker [7], as the resource scaling is ultimately done at the kernel level by setting appropriate values to the corresponding cgroup files for CPU and memory limits.

Unfortunately, the fact that software containers share the kernel among all the containers running on the same physical host means that some caution is required for any operation that accesses the kernel or collects system-wide information. An example of this is traditional resource monitoring, which is typically carried out by tools (e.g., top, collectl) that report overall resource usage and optionally per-process metrics. If looking for aggregated container usages in a system-wide manner, it is not advisable to execute such tools inside a container as they most probably report the host's usages rather than the container ones. Nevertheless, these tools can still be used for process-based monitoring and to produce a container resource usage report via metric aggregation. For our platform, we rely on the BDWatchdog monitoring framework [8] which offers this exact feature. With this solution, resource usage metrics for CPU, memory, disk and network are collected and stored as time series in real time and on a per-process manner.

By combining LXC and BDWatchdog we have the basis to execute any application or workload on a virtual infrastructure composed of software containers that imposes a minimum virtualization performance overhead and is accurately monitored.

2.2. Serverless paradigm

The serverless paradigm is currently presented as a novel platform for application execution. Its main difference from already existing paradigms like IaaS (e.g., Amazon EC2 [9], Google GCE [10]) or PaaS (e.g., Amazon ECS [11]) is that the user is abstracted from the underlying infrastructure and has no control over it. While either a hypervisor or a container manager is used in the aforementioned IaaS/PaaS services, in a serverless environment the user may not be given access to the underlying infrastructure as the provider reserves the right to make the resource management in order to maximize resource utilization. As a direct consequence of this, the user typically does not choose the amount of resources allocated for the application and is only charged for the used ones. Although this billing policy is also advertised for IaaS/PaaS services, it is crucial to point out some differences and present some terminology that later used.

On the one hand, with IaaS/PaaS the user asks for a resource amount to be given for an instance, from now on 'reserved' resources. These resource limits do not change while the infrastructure is running and are in the end billed according to the time they were allocated. On the other hand, with serverless the user executes an application and is only billed for the resources that it used, from now on referred to as 'used' resources. Because the underlying infrastructure is abstracted, there are no 'reserved' resources in the serverless scenario. In case of an application that is idle, the user would still be charged for the 'reserved' resources in an IaaS/PaaS service, while would only be charged for the minimally 'used' resources on a serverless platform. As a whole, this difference is the main advantage for the users as it can significantly lower the costs for applications that adapt to the serverless paradigm, as is the case for web applications or event-based workloads. Other advantages are quicker deployments or increased flexibility, taking benefit of the abstracted infrastructure.

However, in the end, even applications deployed on a serverless platform need an underlying infrastructure to be executed, thus it is only abstracted from the user's point of view. So, it is the task of the service provider to allocate the necessary resources for the application, taking into account that the allocated but unused resources do not return any profit as they are not billed. Considering that containers are used in our platform, whose resource limits can be changed at any moment, from now on we will refer to these resources as 'allocated'. In addition, it is worth to mention that while the service provider might look for a high server consolidation to minimize any non-billable resources, at the same time it must abide by the Service Level Agreement (SLA) with the user and guarantee a minimum performance at all times, whether the application is idle or not.

Finally, it is necessary to introduce some additional terminology to describe components or operational processes of our platform. Up to this point we referred to applications as abstracted workloads that are deployed and executed in any form of infrastructure. However, from the point of view of the proposed platform, an ‘application’ is composed of a set of ‘containers’ that jointly execute a workload. Although this abstraction is not actively used for the experimentation, which targets the scaling on each container individually, it is still useful as an aggregated view for any subsequent analysis. In addition, it is important to remember that containers have to be deployed and executed in an underlying infrastructure, a ‘host’, usually a dedicated instance with a large amount of resources. Along with the containers, we consider applications and hosts as ‘structures’ in our serverless platform, as ultimately they all represent some form of infrastructure unit. For example, all of them share the need for an accounting of each resource, whether such information is to point out the allocated or total deployed amount (e.g., a container or application structure, respectively), or the unused amount (e.g., a host structure).

3. Related work

There are several fields of work related to our platform, from the Function-as-a-Service (FaaS) paradigm (Section 3.1) to novel research frameworks and solutions that aim to extend FaaS to support applications deployed as clusters of containers (Section 3.2). Lastly, it is interesting to study more closely the dynamic scaling of applications (Section 3.3), already present in established technologies like the cloud.

3.1. FaaS platforms

Currently, most serverless products offered by major cloud providers tend to focus on specific use cases around the FaaS paradigm, such as Amazon Lambda service [12], Google Cloud Functions [13] or Azure Functions [14]. With these products and the FaaS overall, small programs are executed with properties similar to functions (i.e., the applications do not have an internal or persistent state), but with the difference that they are abstracted from the underlying infrastructure that actually hosts them. Service providers guarantee availability and scalability by automatically resizing the infrastructure as needed, and only charge the user for the resources really used. Unfortunately, this also means that the user may not be given access to the instances or infrastructure used to host the running functions, as by design the use cases do not usually require this. Furthermore, it is not possible to execute complex types of applications (e.g., Big Data workloads) as the platforms require adherence to predefined templates thus making the integration incompatible. Besides commercial platforms, there are also open solutions for the FaaS paradigm currently being developed, like OpenLambda [15] and OpenWhisk [16], but they have similar limitations.

3.2. Containerized applications on serverless scenarios

While the previously described serverless platforms restrict the user to running small tasks following some guidelines, several authors have found the serverless paradigm interesting to be extended to more general-purpose use cases. To do this, applications are commonly bundled using container images as the means to easily package and distribute them, and are then instantiated to be executed under any container manager platform (e.g., Kubernetes [4], LXD [17]). By using the container as the smallest infrastructure component, instead of an abstracted entity that represents a piece of code like in FaaS, it is possible to support more types of applications which can be in turn much more complex. As the literature suggests [1, 18, 19], serverless is one of the trends to follow in the future when it comes to applications that present burst-like behaviour, event-based processing or simply adapt to predictable patterns. The applications already deployed in FaaS solutions are included in such scenarios, but efforts are being made to adapt other kinds of workloads to the serverless paradigm, as described next.

In [20], a scientific workload from high performance computing environments is successfully deployed on FaaS platforms. Interestingly, it is pointed out that in the future a better resource management should be offered by the service provider, considering that with the serverless paradigm the user loses control on this matter. User hints are proposed as a means to guide and modulate such resource management and possibly auto-scaling. Additionally, a cost/performance analysis is presented, showing that for some applications there is neither a performance penalty nor a higher cost. In a similar way to this work, we look forward to adapt Big Data workloads to a serverless scenario where resource scaling is present, assessing how they behave in terms of performance and resource usage, something important in case they had to be billed for.

Other works present frameworks that aim to run applications in clusters of containers on a serverless environment using cloud infrastructures. In [21] the authors describe SCAR, a solution to deploy several kinds of applications through Docker containers in the Amazon Lambda service in order to benefit from its scalability and fault-tolerance features. It is also further proved that containers can be integrated into an existing serverless platform and used to better suit the user's needs, as they are more flexible and richer than standalone pieces of code. Nonetheless, we consider that although this approach is interesting, there should not be a need to integrate an already mature and standalone technology like Docker with a serverless platform like Amazon Lambda. In our solution, containers are deployed and monitored natively by the serverless platform, with the resources being scaled dynamically to cater for the application's requirements, without the need for any integration or rebooting process.

Finally, it is also worth noting that while Amazon Lambda service imposes limits, like a maximum timeout of 300 seconds for the functions to finish, our platform does not impose any limit and the containers can be maintained with minimum resource usage when idle while waiting for work.

3.3. Automatic resource scaling

Previous work regarding automatic resource scaling has already been explored extensively for virtual machines backed by hypervisor-based virtualization, proving that it is interesting both for users and service providers due to lower costs and higher resource utilization ratios. In [22, 23] the authors present a system that is able to automatically scale applications running in several virtual machines by either scaling the number of instances or the amount of resources. This scaling is done by using several algorithms, policies and thresholds to scale up or down, in a similar way to our platform.

Newer studies focus on using containers instead of virtual machines due to their advantages including the possibility of real-time resource scaling. Dhalion [24] is a framework capable of automatically scaling applications running on top of Apache Heron [25]. With this solution, the workload's health is monitored through a series of metrics in order to detect several symptoms that may be the cause of performance losses. Among the issues monitored for are both the resource overuse or underuse, which can then be treated to achieve a higher resource utilization without incurring any performance penalty. Similarly, Trevor [26] expands on the Dhalion solution using models created and trained using metric measurements from applications to automatically and continuously search for a configuration that optimizes their execution, even while running. As with our platform, Trevor is able to deal with changing resource demands by either increasing or decreasing the amount of resources. However, due to the fact that Trevor is implemented on top of Apache Heron, the focus is put more on the number of instances (horizontal scaling) than on the size of them (vertical scaling). As opposed to Dhalion or Trevor, our platform aims for a more flexible implementation that supports the dynamic and continuous scaling of containers with no specific restriction for any underlying solution or technology, thanks to relying on the use of cgroups. By using the container as the minimal infrastructure instance unit, our platform provides the user with the freedom to deploy a broad range of applications ranging from a few containers to a large cluster.

When it comes to commercial products, there are several available solutions in the cloud environment to automatically adjust the user's infrastructure to meet the application's demands at any time. Amazon Auto Scaling [27] is a representative example that fully integrates with several other Amazon services like EC2/ECS and with Amazon CloudWatch [28], which acts as the monitoring daemon that feeds usage metrics used for the scaling policies. Unfortunately, only horizontal scaling is possible by adding more instances. Note that this combination of resource usage information and the Auto Scaling service accomplishes the same purpose as combining BDWatchdog and our resource scaling platform, as previously mentioned in Section 2.1.

It is also interesting to further study Amazon ECS [11], a service that allows to run containers. ECS supports two modes for container deployment and management: EC2 and Amazon Fargate [29]. On the one hand, with EC2 the user chooses an instance type to host the containers, being the access granted to such instance. Once the instance is deployed, it is possible to automatically scale the application relying on the Auto Scaling service by adding or removing containers according to predefined user policies. However, the resources of an EC2 instance cannot be changed without a reboot. So, if there is a need to further increase the number of containers hosted, either an additional EC2 instance has to be spawned or the one in use has

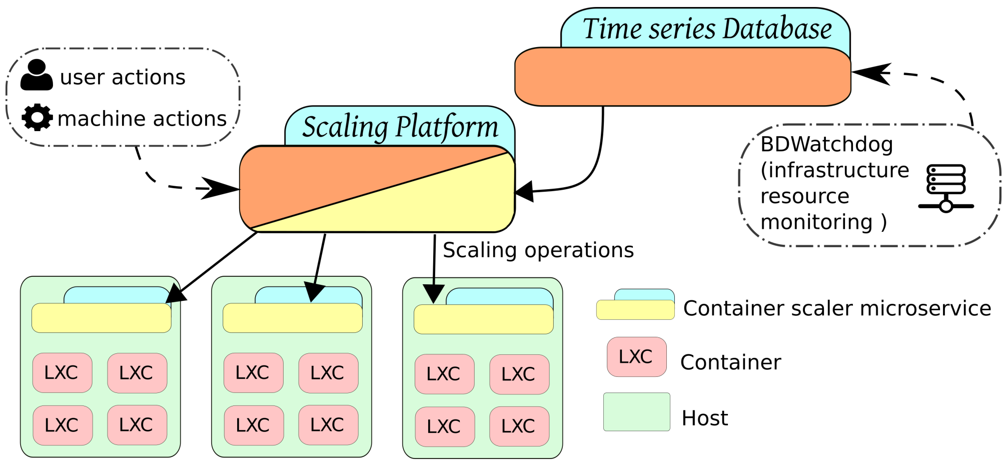


Figure 1: High-level overview of the resource scaling platform

to be stopped, scaled and restarted. Regarding the billing policy, the EC2 instances are charged according to their type. On the other hand, Fargate takes over the task of choosing an EC2 instance type to run the containers and further abstracts any infrastructure management. With Fargate it is possible to deploy an application specifying only the initial resources it needs and the number of containers to use. In contrast with EC2, only the used resources are billed, although at an incremental rate. Unfortunately, with Fargate the user does not have access to the underlying instances or infrastructure used. Although the difference between both modes mainly lies in the billing policy and the access to any underlying instance, the first difference is of particular interest for the user and was previously exposed in Section 2.2. In contrast, our platform implements automatic scaling of resources without any need of rebooting, while preserving the access to the underlying containers. If billing is to be applied, it is possible to account for the used resources as well as for the allocated ones.

4. Design and architecture of the resource scaling platform

The objective of the proposed platform is to create a serverless environment where Big Data applications are executed with resources being allocated on demand according to their real usage. It is also desirable that the platform does not impose a significant overhead while being scalable and remains overall autonomous.

Figure 1 represents a high-level overview of the system and the environment on which it works. Lying at the center is the resource scaling platform, whose only inputs are the control actions taken by either a user or another system and the information about resource usage, as provided by external monitoring tools. For this requirement, the monitoring features of BDWatchdog [8] are used to feed such information in the form of time series, which after being stored in the *Time series Database* can be later retrieved by any service that needs it. In this case, OpenTSDB [30] is used to persist and expose the time series. The output of the platform is represented by scaling operations sent to each infrastructure host, on which a container scaler microservice forwards such operations to the container manager or to the containers themselves via cgroups. The specific details of the architecture and technologies used to implement this platform are described in Section 4.1. An in-depth analysis of the feedback loop that lies at the core of the platform is presented in Section 4.2. Finally, a scalability discussion is provided in Section 4.3.

4.1. Microservice-based architecture

The platform architecture (see Figure 2) is designed around the idea of being able to process events in real time and acting quickly enough to respond accordingly. To do so, a microservice-based architecture is chosen along with a feedback loop arrangement for the services. The platform developed from this architecture design is able to self-regulate, thanks to the loop created, and to exploit the inherent parallelism

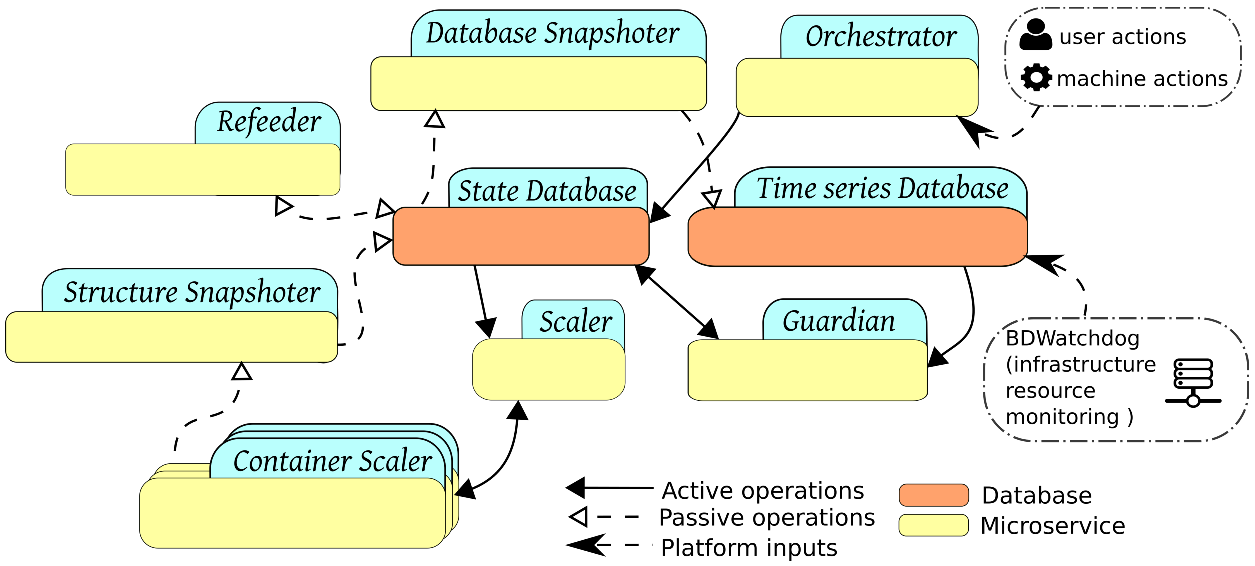


Figure 2: Microservice-based architecture

of the microservice division, which in turn allows to work in real time. Furthermore, thanks to having the possibility of isolating the services and to the use of the REST protocol for inter-service communication, the development and testing of the individual services is much simpler.

In order to perform the service domain division and thus break down a complex loop into smaller systems, which can then be easily managed or configured, a global intermediary service is used (see *State Database* in the figure). A document database, CouchDB [31], is chosen to act as the agent to keep the whole container and microservice metadata and thus act as such intermediary service. Using this light, document-based database, all the platform state information is kept in a single point, allowing for an easier supervision as well. This centralized configuration of the services means that they can be implemented fully stateless and with a simpler, daemon-oriented design. It has to be noted that although the *State Database* represents a single point of failure, this can be mitigated by taking into account the different features implemented in CouchDB, such as load balancers or data replication, which readily grant high availability. It must also be considered that such database only acts as a temporary means of holding the platform state, which is usually reduced to a low amount of information even if a large infrastructure were to be deployed, and thus a recovery and startup process could be easily carried out with a new database.

All the microservices shown in Figure 2 are implemented with Python following the REST philosophy, using JSON as the information format to exchange data. By using such format it is also possible to directly store the information in the *State Database* as JSON documents. In addition, thanks to their more human-readable characteristics, the administrator can inspect and modify such documents (e.g., a service configuration document) if necessary.

4.2. Feedback-based loop control

In order to maintain stability and to avoid high overheads to the running applications, the platform must minimize the time ranging from the moment a resource usage event is detected (e.g., a bottleneck, when the usage is close to the allocated amount) to the moment a request is generated to tackle such situation (e.g., a scaling-up operation, which increases the allocated amount). Benefitting from the domain division mentioned in Section 4.1, which causes the services to be highly specialized, the loop behaviour can be explained by analysing both the services deployed and the information that is generated and exchanged.

Regarding the services (see Figure 2), they can be classified into those that are active, meaning that they take actions to change the platform state, and those that are passive, only generating information that refeeds the system constantly. On the one hand, the active services are: 1) *Guardian*, the core service

```
1  type: structure
2  subtype: container,
3  scale: true,
4  host: c10-10,
5  host_scaler_ip: c10-10,
6  host_scaler_port: 8000,
7  name: cont2,
8  resources: {
9      cpu:  { allocated: 300,  scale: true,  max: 300,   min: 50   },
10     mem:  { allocated: 8192, scale: true,  max: 10240, min: 1024 },
11     disk: { allocated: 100,  scale: false, max: 100,   min: 20   },
12     net:  { allocated: 200,  scale: false, max: 200,   min: 100  }
13 }
```

Listing 1: Example of a structure document

responsible for the time series analysis so as to generate events and requests; 2) *Scaler*, which processes such requests and then forwards them to the hosts and their corresponding *Container Scaler* services; 3) *Container Scaler*, the service that runs on each physical host and exposes a REST API through which requests are applied to the containers; and 4) *Orchestrator*, which provides a REST API that allows users or other external services to make changes to the platform configuration. To implement both REST APIs, Flask [32] is used along with Gunicorn [33] to expose them via a WSGI-compliant server [34]. On the other hand, the passive services are: 1) *Structure* and 2) *Database Snapshoters*, which when combined persist the container metadata both in the *State Database* to refeed the loop and in the *Time series Database* for later analysis; and 3) *Refeeder*, the service responsible for creating the aggregated metrics that represent an application from its containers. Further details about the functionalities of both active and passive services are explained in Section 5.1.

Regarding the information used and generated throughout the platform, it can be classified into time series and documents. On the one hand, the most important time series are the application resource usages generated by BDWatchdog, as mentioned before. These act as the main source of information of the platform as they are needed to overall detect the events that will later trigger the requests to scale the resources. Secondly, the platform also generates time series by periodically persisting relevant documents from the *State Database* to the *Time series Database*. Although not so important, this information is used for later analysis and reporting. On the other hand, documents are used throughout the platform, mainly by the active services, as the basic means of interchanging information, although are not persisted over time. Documents are classified into: 1) 'structures', which represent the metadata of either a host, a container or an application, mainly storing the amount of allocated resources; 2) 'services', the per-service, uniquely stored document containing its configuration; 3) 'rules', which store the policies to be applied for time series analysis; 4) 'limits', having for each container or application structure the resource values for the upper and lower thresholds, as well as the boundary to be kept between them; 5) 'events' and 6) 'requests', the resulting documents after the continuous resource analysis, as explained in Section 5.1.

Structure documents (see an example in Listing 1) store crucial information such as the maximum and minimum values for each resource, which correspond to the amount of resources that a structure (e.g., an individual container or an application) never surpasses or is always given, respectively. They also store the allocated amount for each resource (see lines 9-12). Regarding rule documents, it is interesting to mention that although stored as JSON files, they contain the logic ultimately used to define the scaling policies. Thanks to the use of JsonLogic [35] it is possible to inject complex boolean rules and dynamically evaluate them in the continuously running loop. This feature grants the system flexibility when it comes to changing the configuration in real time. The rest of documents are more precisely explained in Section 5 by using an example.

4.3. Scalability discussion

After having discussed the design and inner workings of the platform, it is interesting to address potential scalability limitations and improvements. This is even more important when containers are deployed, which can typically be smaller in terms of resources when compared with virtual machines, and thus a significantly larger number of containers could be running at any moment. In addition, if Big Data workloads are considered to be deployed on such virtual infrastructure based on containers, it is increasingly possible that a large number of them have to be managed.

From the point of view of the design, it is possible to analyse some features that enhance the scalability of the platform. First, the use of a microservice approach makes it possible to have not only the inherent parallelism of several services working simultaneously, which mitigates the effect of any local bottleneck, but also the replication of such services as needed to exploit the parallelism of their operations. For example, as can be seen in Figure 2, the active operations performed by the *Scaler* can be potentially parallelized by splitting them either across several threads inside an instance or even across multiple instances of the service. This behaviour also applies to other critical services such as the *Guardian* or the *Snapshoters*. Second, and related to the parallelism of services and operations, the metadata database used to store the documents could also benefit from a scenario with several database instances, with each instance storing the documents of a part of the managed infrastructure, thus creating some sort of domains which would be independent.

However, if a specific scalability study had to be carried out some tools could be used to test the degree up to which the platform could work. In [36] the authors present PatternMiner, which allows exploiting the tracking and logging of RPC calls on many common systems with central components, such as HDFS with its NameNode, to detect if a bottleneck will arise once the system is scaled to many instances that communicate with each other and the central component. Considering that our platform relies on a central metadata database, and that communications play a key role, it could be possible to use PatternMiner to further test if scalability would be an issue once the number of containers managed and the platform grow significantly. Nevertheless, this kind of tools can be challenging to deploy and are mainly used when there is the need to know beforehand the scalability limit of a platform (e.g., as part of the technical requirements for a production deployment). As our scenario is experimental, in this paper the scalability of the platform is further analysed with empirical data and the use of specific experimentation, as presented in Section 6.5.

Finally, it should also be considered that the platform could benefit from better hardware specifications or newer technologies in order to improve its internal response time to events, which would help it to further scale. Previous studies like [37] show how improving communications can enhance the performance of the application that use them in a seamless and unobtrusive way.

5. Resource time series analysis

This analysis is at the core of the platform, having a two-fold role. First, while the applications are running, the platform continuously monitors the infrastructure to determine if there are scaling actions to be performed in real time according to the policies set by the user. Secondly, it also gives more insight into how resources are used in any past time window. Considering that the system is continuously persisting the most important information, such time windows can be really close to present, which in turn allows to perform any analysis or detect performance issues quickly after they arise. Finally, after an application is executed, a subsequent resource analysis can provide fine-grained detail regarding how resources were used.

Both functionalities, real-time resource scaling and subsequent resource analysis are respectively described in Sections 5.1 and 5.2. In addition, these functionalities are extensively referred to in Sections 6.3 and 6.4, respectively, where they are used to assess the platform efficiency and to study the behaviour of the resource usage for different Big Data workloads. Finally, all the configuration aspects of the platform are described in Section 5.3.

5.1. Real-time resource scaling

A continuous analysis is performed while the application is running, requiring all the platform services to implement the feedback loop. To start any scaling operation, the *Guardian* service performs a two-step

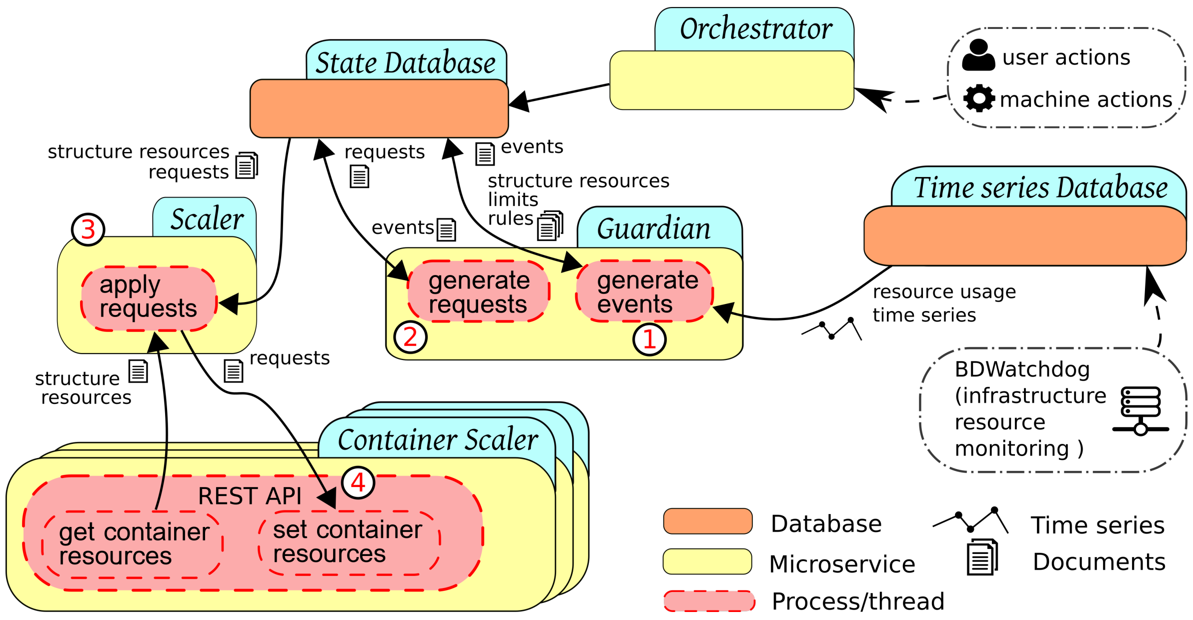


Figure 3: Active services

analysis of the resource usage metrics. To do so, first this service retrieves the resource time series from the *Time series Database*, which were previously collected by BDWatchdog, and matches them against a set of event-triggering rules. This may generate events (see number 1 in Figure 3), which are stored in the *State Database*. Second, all the events are retrieved and, after filtering out the expired ones, they are equally matched against a set of request-triggering rules, which generates in turn requests (see number 2), also stored in the *State Database*. Meanwhile, the *Scaler* is continuously polling the *State Database* for any request (number 3). Once new requests are found and after filtering out any duplicated or expired ones, they are processed and applied to the structure resources before being actually sent to the *Container Scaler*, which ultimately applies the request to the container (number 4). This additional processing carried out by the *Scaler* is needed as an intermediary check to ensure that no request, if applied, would cause the resource amount to exceed a maximum or drop below a minimum value. Requests can even be discarded if the underlying host of the container has run out of resources to be allocated.

In parallel to the active services and in no particular order, the passive ones are continuously propagating the actually allocated structure resources (see number 1 in Figure 4), generating the aggregated application resource metrics (number 2) and ultimately persisting the structure metadata (resources and limits) in the *Time series Database* for later analysis (number 3).

The described two-step analysis to generate requests is designed and implemented in order to mitigate 'hysteresis' and provide the option to modulate the 'responsiveness' in the platform configuration. Hysteresis, which appears when a resource usage is constantly oscillating around a fixed range, could severely affect the platform if scaling operations took place once any limit was crossed. If such limits are not wide enough, the oscillating usage would constantly exceed them and cause constant scaling operations, and overall an unstable state. With this two-step analysis, hysteresis is mitigated thanks to the fact that rules that trigger requests can be configured to trigger only when a configured number of events of a certain type (e.g., a bottleneck) are detected, and none or a configured number of opposite events (e.g., an underuse) arise. Regarding responsiveness, considering it as the time it takes the platform to respond and adapt to the application resource usage, it can be modulated by changing in the rules the number of events needed to trigger the corresponding requests. This configuration feature is further described in Section 5.3.

To better understand the function of the active services and overall the real-time resource scaling, Figure 5 shows an example where CPU scaling requests are applied to a container represented by the structure shown

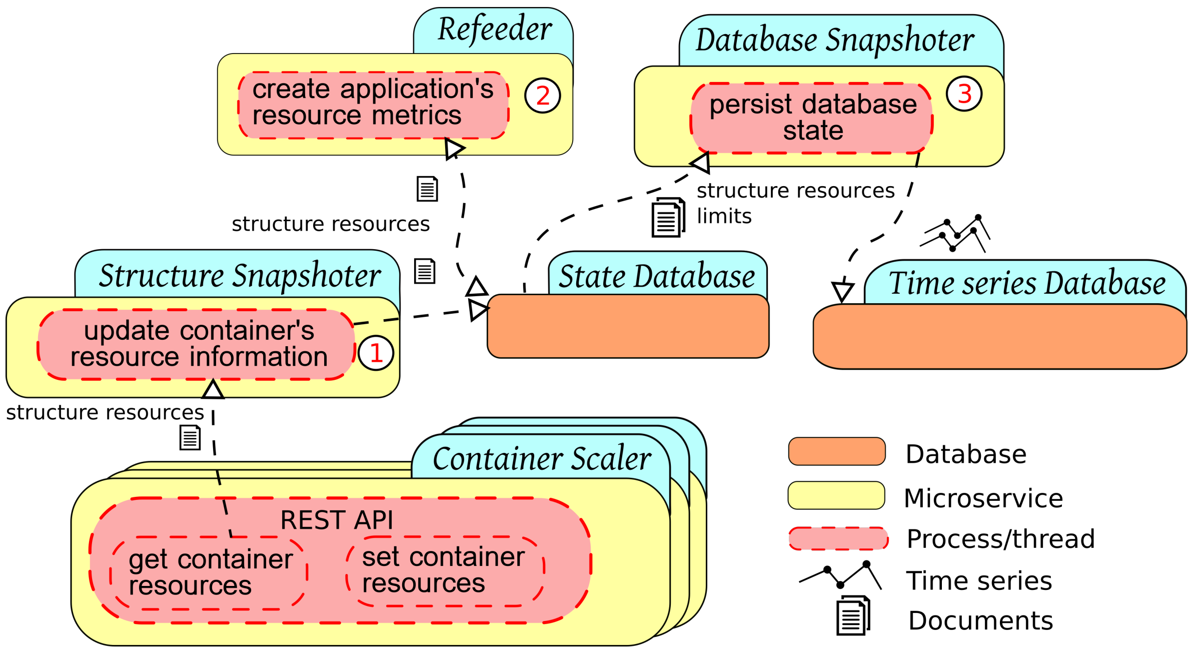


Figure 4: Passive services

in Listing 1 (CPU maximum/minimum values: 300/50 shares). While from roughly second 60 to 220 the CPU usage fits into the normal scenario (i.e., between the lower and upper limits) and nothing is done, from 220 to 270 the CPU usage exceeds the upper limit. At this point, the rule shown in Listing 2, which detects CPU bottlenecks, is triggered. Specifically, this is done when both the CPU usage is higher than the upper limit (see the first condition of the 'and' logic in line 8) and the allocated CPU amount is lower than the maximum allowed (see the second condition in line 9). This rule is activated several times before ultimately triggering the rule presented in Listing 3. This latter rule is specifically triggered after 2 CPU bottleneck events (see the first condition in line 10) and less or equal to 2 CPU underuse events are detected (see the second condition in line 11), generating in turn a request to scale CPU shares upwards. However, after this resizing, from seconds 280 to 420, CPU shares are not fully consumed incurring thus in an underuse scenario. As in the previous bottleneck scenario, this triggers a rule that detects underused resources and after a certain number of such events, a CPU scaling-down operation is performed. From second 420 onwards the container is given the minimum resources as the lower limit is set to the minimum value (50 CPU shares) and thus no underuse events are possible anymore.

It is important to mention that the behaviour of scaling requests is different according to the request type. If the resources are to be reduced, there are several options, as it is possible to know beforehand the amount of resources the structure has recently been using up to that point. Several simple policies are supported such as always reducing by a fixed value or reducing by a percentage of the unused amount. Nevertheless, in order to fit the allocated resources closely to the used ones, a more complex 'fit to usage' policy is implemented. This policy tries to set a specific allocated resource value with the aim of making the application usage stay between the lower and upper limits (see this effect from second 60 to 220 in Figure 5). If this is achieved and the usage remains more or less stable, no further operations are likely to be needed. When it comes to increasing the resources, it may not be as straightforward considering that if the usage is in a bottleneck or close to it, it is uncertain how many resources the application actually needs. For this scenario there is only one policy implemented for scaling up, increasing the allocated resources by a fixed amount, and it is left to the user to configure such amount (e.g., see label 'amount' in Listing 3). Both configuration options of changing the resource amount when scaling up and the policy used when scaling down can be used to modulate how benevolent the platform is with the applications in terms of vertical

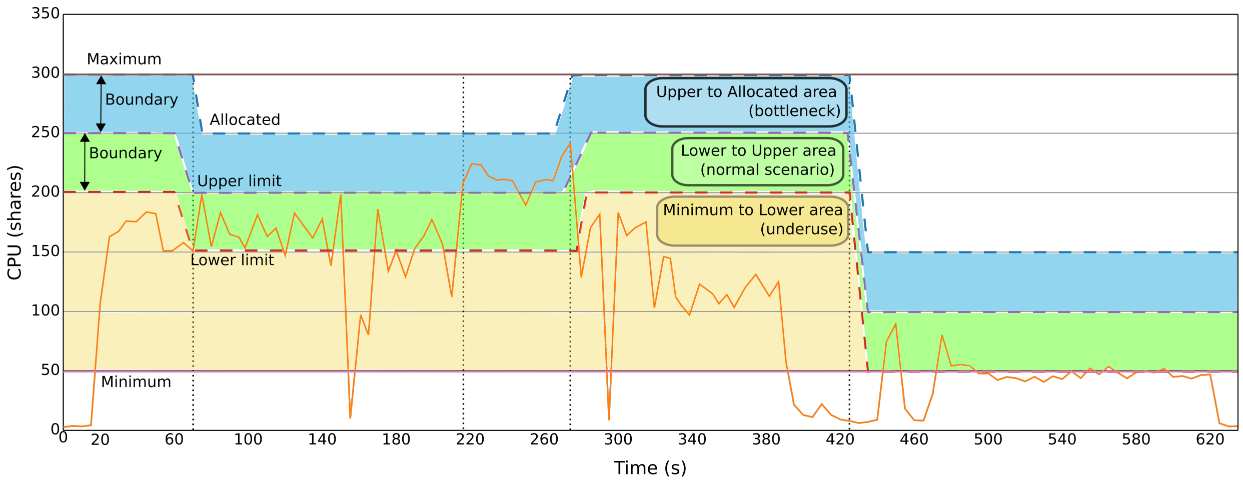


Figure 5: Continuous resource analysis with areas for scaling operations

```
action: {
    events: {scale: {up: 1}}
},
generates: events,
name: cpu_exceeded_upper,
resource: cpu,
rule: {
    and: [{>: [{var: structure.cpu.used}, {var: limits.cpu.upper}]},
          {<: [{var: structure.cpu.allocated}, {var: structure.cpu.max}]}]
}
```

Listing 2: Rule to generate CPU bottleneck events

scaling.

Finally, it can be noted that the time it took to trigger the scaling-up operation (50 seconds, from second 220 to 270) is significantly lower than the one needed to trigger a scaling-down (150 seconds, from second 290 to 440). This behaviour corresponds to the responsiveness previously described, choosing in this case a more friendly configuration that acts sooner when a bottleneck is detected (3 times faster) and then gives the structure more time before reclaiming resources.

5.2. *Subsequent resource analysis*

Thanks to persisting the structure and limits documents from the *State Database* to the *Time series Database* (performed by the *Database Snapshoter* service), it is possible to carry out different kinds of analyses for any past time window. However, considering that the continuous analysis may change both kinds of documents in a short time span, the snapshots have to be performed with a frequency equal to or higher than the time windows used by the active services. Fortunately, this in turn also means that any a posteriori resource analysis can be performed from any moment in the past to close to present, even when the application is still running.

Although there are many ways to analyse a time series, we introduce some terminology and concepts that can be used to evaluate the differences between our serverless platform and more traditional IaaS/PaaS models with instances that have reserved resources. Using the resource time series shown in Figure 5, Figure 6 defines three areas: 1) the area under the time series itself, representing the amount of resources currently in use (i.e., the used area); 2) the area under the allocated resources (the allocated area); and 3) the area under the maximum allocatable resources (the reserved area). The allocated area is only applicable to the serverless scenario as it represents the amount of resources that are available to the instance at any

```
action: {
    requests: [CpuScaleUp]
},
amount: 50,
generates: requests,
name: CpuScaleUp,
scale_by: amount,
resource: cpu,
rule: {
    and: [{>=: [{var: events.scale.up}, 2]},
          {<=: [{var: events.scale.down}, 2]}]}
}
```

Listing 3: Rule to generate CPU scaling-up requests

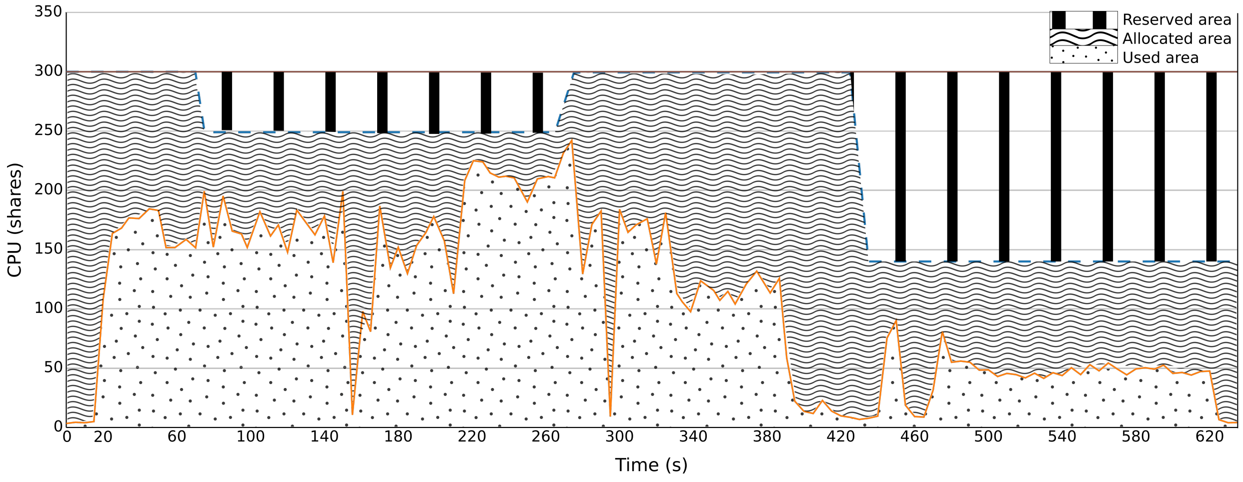


Figure 6: Subsequent resource analysis marking the different areas for accounting

precise moment, as enforced by cgroups. The reserved area is applicable both for serverless and cloud-like instances. On a serverless scenario, this area represents the maximum allocatable amount of resources (i.e., allocated when a resource is fully used). On a cloud scenario, it represents the amount of resources initially assigned to such instance.

For each area, an integral can be performed to add up the total amount of resources, thus excluding the time variable. Furthermore, thanks to the discrete nature of these time series, it is possible to perform the total integral as the sum of the partial integrals for each subinterval using the trapezoidal rule, having a perfect fit and thus no error in the end result. With these integrals, we can in turn define the resource utilization ratio as the quotient between the resources used and the resources allocated/reserved in the serverless/cloud scenario. This ratio is useful to measure how many of the resources given to an application are actually used in the end, and is employed for the experiments of Section 6.4 to compare different configurations and scenarios.

5.3. Platform tweaking and configuration

Configuration parameters are crucial if there is a need to adapt the platform to a specific workload pattern that proves to be unpredictable or too resource demanding. As previously mentioned, our platform configuration can be tuned to modulate either its ‘benevolence’ or ‘responsiveness’.

Regarding benevolence, the parameters seek to configure how the platform adapts itself to the application in terms of the resource amounts applied to scaling operations. This configuration modulates the platform’s response in the vertical scaling. To configure benevolence, parameters in the structure and rule documents can be modified. For the container structure, there is a value for the maximum and minimum amount for each resource (e.g., CPU). The maximum value is used to set a limit on the amount of resources potentially available, thus making the platform equivalent to running a container on an IaaS/PaaS if the resources are fully used. The minimum value represents the amount that is ensured to be allocated. This is a crucial parameter because while a high value may result in unused resources when the application is idle, a very low value may cause the application to fail due to simply not having enough resources or even cause the container to freeze. Additionally, container structures also have a boundary parameter which specifies the amount to be maintained between the lower-to-upper and upper-to-allocated areas (see Figure 5). This parameter is the one that most contributes to the benevolence of the platform as it ultimately determines how wide is the range where resource usage is considered as normal (see 'normal scenario' in the figure), and thus how much the platform tries to adjust the allocated amount to the real usage. While both the maximum and minimum parameters are expected to remain fixed, the boundary can be changed if needed, as shown in the experiments of Section 6. When it comes to rules, specifically those that generate requests, they can be modified to change the amount of resources to be increased when scaling up, or the policy used when scaling down.

Regarding responsiveness, it seeks to modulate the platform's response in terms of the time it takes to adapt the allocated resources to the real usage. This configuration option is based on the concept of time window, defined as the time duration used to isolate services or measure intervals. To configure responsiveness, parameters in the service and rule documents can be modified. In service documents, the time window duration for each service is specified in seconds. Rule documents can be modified to specify the number and type of events needed to trigger each kind of request. As mentioned in Section 5.1, responsiveness allows to mitigate hysteresis, as the rules can be configured to trigger only after a certain event is consistently detected, and the time to respond can be modulated by specifying the number of events needed for a request to be triggered. All the parameters involved in the benevolence and responsiveness aspects are summarized in Table 1.

For responsiveness, it is important to define an appropriate time window configuration across all the services. Figure 7 describes in a timeline the chain of operations carried out from the moment an event is detected to the moment when the allocated resources are finally scaled and the new information fed back into the platform. Before any analysis can take place, the resource usage time series must be generated by BDWatchdog and properly persisted. To allow for this time margin and ensure that these data are available, the *Guardian* is configured with a window delay parameter (see Table 1), needed to set back the time window used to retrieve the time series. In addition, the duration of the analysis window that the *Guardian* works with is configured via the window duration parameter. In the example shown in Figure 7, which considers both a window delay and a window duration of 10 seconds, the *Guardian* performs from second 20 to 30 the two-step analysis (i.e., event detection and request generation) described in Section 5.1 using the resource usage metrics previously persisted by BDWatchdog from second 0 to 10. This 10-second analysis is repeated a total of three times in a pipelined fashion (see second 20 to 50 for the *Guardian* and 0 to 30 for BDWatchdog), and after enough events are detected, a scaling-up request is generated by the *Guardian* around second 50. This request is in turn retrieved by the *Scaler* in its next polling (second 50 to 55). Although such polling frequency can also be configured, it should always have a lower value than the *Guardian* window duration in order to process requests as soon as possible. Once the request is processed, it is sent to the corresponding *Container Scaler* (second 55). As this service is a REST API waiting for requests, no delay is expected. Finally, after the scaling is properly performed, the snapshoter services feed back the changes into the system in their next polling (second 60 to 65).

6. Experimental results

The main objective of our scaling platform is to adjust the resources given to Big Data applications to their real usage while also minimizing the overhead imposed and maintaining scalability overall. On the one hand, to measure the fitness and gains provided by the platform, as well as to study the overhead, both batch and streaming workloads are deployed on a first testbed cluster using 12 containers. On the other

Table 1: Parameters to modulate benevolence and responsiveness

	Dimension	Documents		
		Structures	Limits	Rules
Benevolence	vertical (resource amount)	maximum amount minimum amount	boundary	amount (scaling up) policy (scaling down)
		Services		Rules
Responsiveness	horizontal (time to respond)	window delay (*Guardian*) window duration (*Guardian*) polling frequency (*Scaler*)		#events for scaling up #events for scaling down

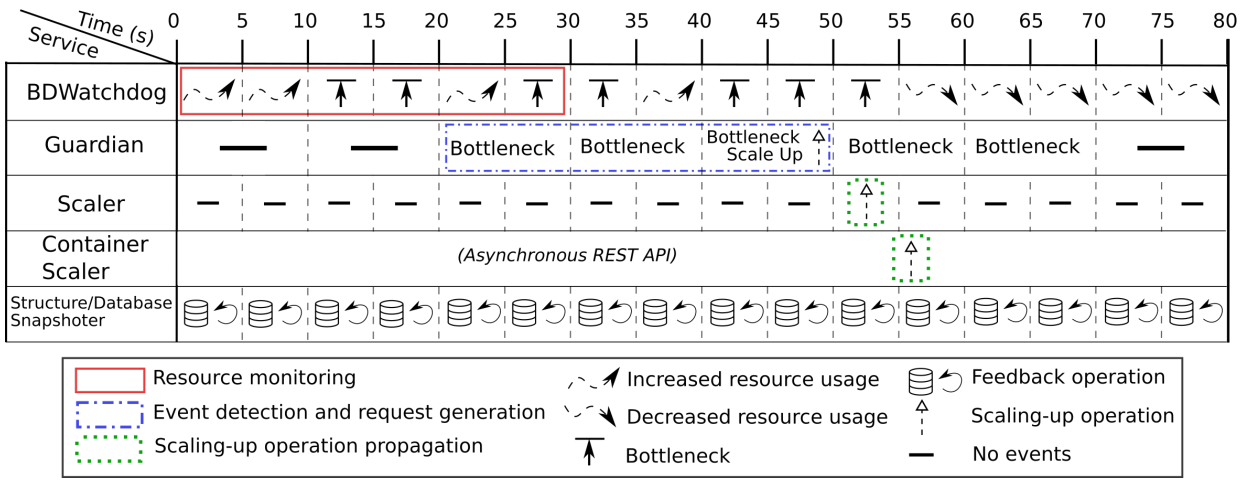


Figure 7: Time window analysis (window delay and duration: 10s, polling frequency: 5s)

hand, to analyse the scalability of the platform as well as its flexibility in comparison with other commercial services, the same workloads are deployed on a larger second testbed using 32 containers.

The experimental and platform configurations for both testbeds are detailed in Sections 6.1 and 6.2, respectively. Sections 6.3 and 6.4 present real-time scaling results and a subsequent resource analysis of the workloads, respectively, for the first testbed, while Section 6.5 presents the scalability results for the experiments conducted on the second testbed.

6.1. Experimental configuration

The experiments are carried out on container clusters based on LXC with LXD 3.6 as container manager. Each LXC container runs Ubuntu 16.04. The first testbed consists of a 12-container cluster deployed on the Big Data infrastructure at CESGA [5]. Each container is configured with a maximum of 2 cores (i.e., 200 CPU shares) and 10 GiB of memory (24 cores and 2400 CPU shares in total). The second testbed consists of a 32-container cluster deployed on the Grid'5000 infrastructure [6]. In this case, each container provides up to 6 cores and 45 GiB of memory (192 cores and 19200 CPU shares in total). In both testbeds the containers have a dedicated disk mounted from the physical host. Resources are guaranteed to be reserved across containers at all times, even when they are all scaled up to the maximum.

Regarding the workloads, TeraSort and PageRank are selected as representative batch workloads, while FixWindow is deployed as an example of a streaming application (see Table 2 for their specific configuration for each testbed). In order to present results using different Big Data processing frameworks, the first testbed uses Hadoop 2.9.0 [38] to execute the TeraSort workload, and Spark 2.3.0 [39] for PageRank and FixWindow. The second testbed uses Spark for all workloads. These workloads are selected because they exhibit very different resource usage patterns, particularly for the CPU. On the one hand, TeraSort is an I/O-bound

Table 2: Workload configuration

Batch workloads			
		Testbed 1	Testbed 2
TeraSort	Dataset size	50 GiB	300 GiB (hybrid 1st use) 200 GiB (concurrent) 130 GiB (hybrid 2nd use)
PageRank	Number of pages	5 million	60 million
	Number of iterations	5	2
Streaming workloads			
FixWindow	Stream data size	stage 1: 66 MiB stage 2: 167 MiB stage 3: 28 MiB	stage 1: 3.72 GiB stage 2: 5.58 GiB stage 3: 1.49 GiB
	Stream window size	5 seconds	10 seconds

workload that shows a CPU usage with low variation over time. PageRank, on the other hand, is an iterative and CPU-bound workload showing a high degree of variability in CPU usage. This different behaviour is to be taken into account when scaling policies are applied in order to minimize the execution time overhead. Regarding FixWindow, its streaming nature requires to process a constant input of data, which implies a very low variation of resource usage when compared to batch applications. Unlike batch workloads, the stream processed by FixWindow is endless as long as the processing remains stable (i.e., processing times are lower than the incoming data interval span). For this reason, the execution of FixWindow is restricted to 15 minutes in the experiments. Specifically, in order to simulate a variation in the resources needed to process the stream, the 15-minute execution is divided in 3 consecutive 5-minute stages with different stream sizes (see Table 2).

To deploy Hadoop and Spark, the Big Data Evaluator (BDEv) tool [40] is used. For the FixWindow streaming workload, HiBench 7.0 [41] is also required to produce the data source to be processed. In addition, a Kafka 2.1.0 broker [42] is needed to receive the data from the HiBench producers and temporarily store it before being offered to the FixWindow consumers. For these streaming experiments, the containers have to be split accordingly to serve the roles of producers, brokers and consumers. The first testbed uses a configuration of 1+2+9 (12) containers for Kafka, HiBench and FixWindow, respectively, while the second testbed uses 4+8+20 (32) containers. In these experiments the broker and producer containers are ignored as their resource usage patterns are not interesting.

6.2. Platform configuration

Considering the different objectives and characteristics of both testbeds, two platform configurations are used for the experimentation. The objective of the first testbed is to present a study of the resources according to different configuration scenarios. For this matter, Table 3 presents two serverless scenarios in terms of the level of benevolence and responsiveness used by the platform, configured using the rule documents. Although close in terms of responsiveness as the number of events to trigger the rules are similar for both scenarios, the difference regarding benevolence is to be noted. On the one hand, for TeraSort and FixWindow, which show a more constant resource usage, the amounts to be scaled when a bottleneck is present are 75 CPU shares and 2 GiB of memory. On the other hand, for PageRank, which shows a more variable resource usage, the values are 100 CPU shares (i.e., one core) and 3 GiB of memory. Table 4 presents different configurations for the boundary parameter that are set up by using the limits documents. In the same way as before, two serverless scenarios are laid out. For the first scenario, used for TeraSort and FixWindow, a total of four configurations are defined (e.g., cpu_MEM means a low boundary value for CPU and a high value for memory). For the second scenario, used for PageRank, only one configuration is used. Furthermore, a baseline configuration is used for all the workloads, which runs them without any scaling, thus representing any cloud-like instance with reserved resources.

Table 3: Configuration of the scaling platform (Testbed 1)

	Rule configuration			
	CPU underuse	CPU bottleneck	Memory underuse	Memory bottleneck
TeraSort FixWindow	#events up = 0 #events down >= 8	#events up >= 2 #events down <= 2 amount: 75 shares	#events up = 0 #events down >= 8	#events up >= 2 #events down <= 2 amount: 2 GiB
PageRank	#events up = 0 #events down >= 6	#events up >= 2 #events down <= 3 amount: 100 shares	#events up = 0 #events down >= 5	#events up >= 2 #events down <= 2 amount: 3 GiB

Table 4: Boundary values for different serverless configurations (Testbed 1)

	TeraSort & FixWindow				**PageRank**
	cpu_mem	*CPU_mem*	*cpu_MEM*	*CPU_MEM*	*CPU_MEM'*
CPU (shares)	20	40	20	40	50
Memory (GiB)	2	2	3	3	3

Table 5: Configuration of the scaling platform (Testbed 2)

Rule configuration		
CPU underuse	CPU bottleneck	CPU boundary
#events up = 0 #events down = 5	#events up = 2 #events down = 3 amount: 225 shares	150 shares

The second testbed aims to show the scalability as well as some specific flexibility features of the platform, and because of this there is just one platform configuration scenario common to all the workloads (detailed in Table 5). In addition, only the CPU is studied in this case. This configuration is considered to be in a middle point in terms of both benevolence and responsiveness.

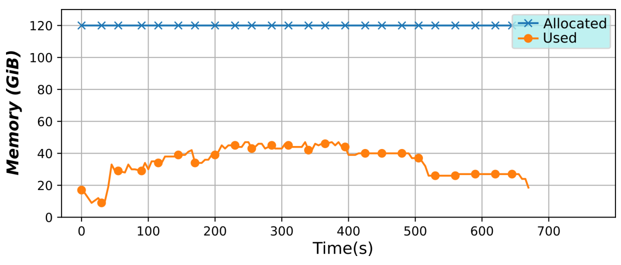
Finally, the remaining configuration parameters, mainly those to tune the services, are the same across both testbeds: a polling frequency of 5 seconds for the *Scaler* and the *Structure/Database Snapshoters*, and a 10-second window delay and window duration for the *Guardian*.

6.3. Real-time resource scaling

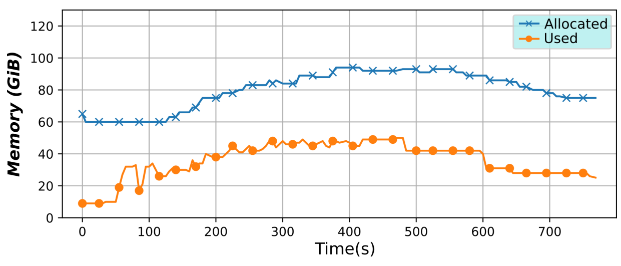
The experiments presented in this section show the resource time series of workload executions for both the baseline and the serverless configurations, where dynamic resource scaling is applied by the platform in real time. The first testbed is used to execute them.

Figure 8 shows the aggregated, application-level memory usage for TeraSort. From top to bottom, the plots represent the baseline configuration (Figure 8a) and the four serverless configurations described in Table 4. Due to the high-level nature of the plots, which aggregate the memory usage of all containers, upper and lower limits cannot be shown as these only apply to individual containers. It is worth noting that although the allocated amount of memory varies between configurations, the used amount follows mostly the same pattern, showing that both metrics actually independent. Regarding the allocated amount, it remains constant in the baseline, while the boundary parameter causes it to be closer to or farther from the used amount in the serverless configurations. The middle plots (Figures 8b and 8c) represent the configurations where the memory boundary per container is set to 2 GiB (see Table 4), thus having a lower 'benevolence', while the bottom ones (Figures 8d and 8e) have a boundary of 3 GiB.

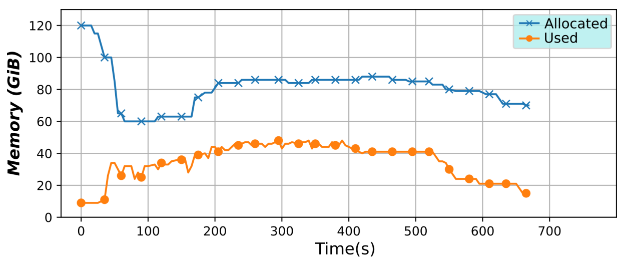
In Figure 9, the aggregated, application-level CPU usage is presented for PageRank. For this case only one serverless configuration is tested (right plot) along with the baseline (left plot). It can be observed that the pattern for the allocated amount roughly mimics the pattern of the used one, albeit with a slight delay. More precisely, this delay is the side effect to be tackled and modulated via the 'responsiveness' configuration



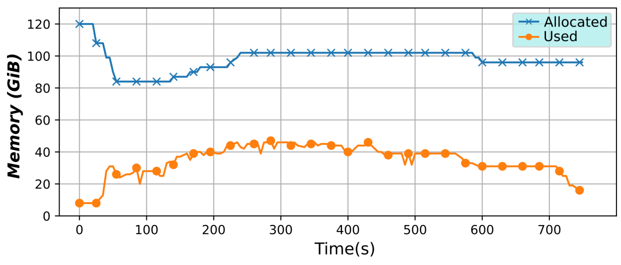
(a) Baseline



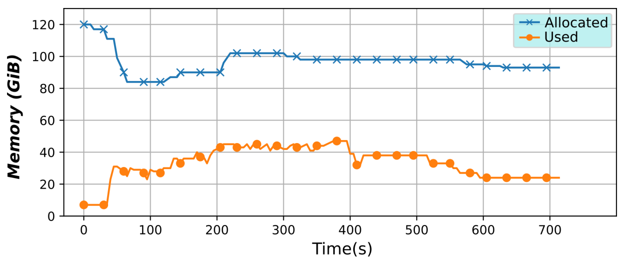
(b) cpu_mem



(c) CPU_mem

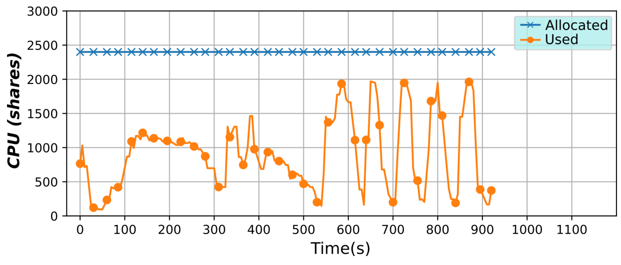


(d) cpu_MEM

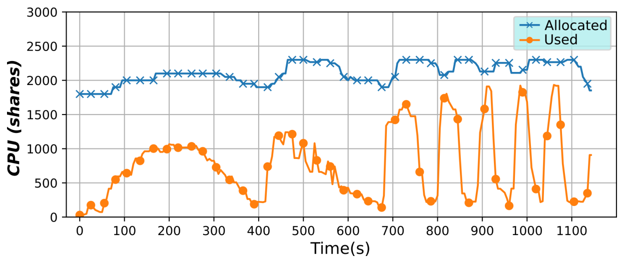


(e) CPU_MEM

Figure 8: Aggregated memory usage of TeraSort



(a) Baseline

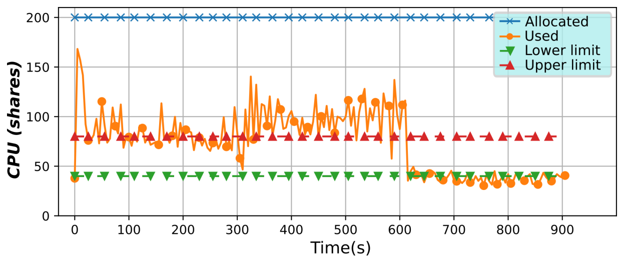


(b) CPU_MEM'

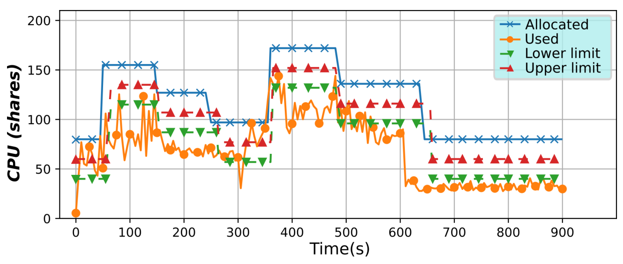
Figure 9: Aggregated CPU usage of PageRank

discussed in Section 5.3. In addition, it is worth noting how the CPU usage peaks, which correspond to iterative phases of the algorithm, have different values for both configurations. In the baseline, such peaks consistently reach around 2000 shares (i.e., 20 cores). In the serverless configuration, they range from 1600 to 2000 shares. So, our platform adapts to the iterative nature of PageRank, as CPU peaks increase with each iteration.

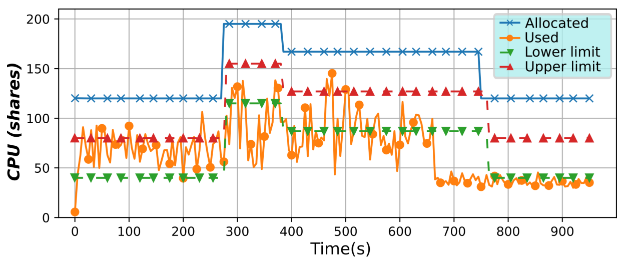
Finally, Figure 10 presents the CPU usage of FixWindow for an individual container following the same configuration order as in Figure 8. Unlike previous application-level plots, it is now possible to show the



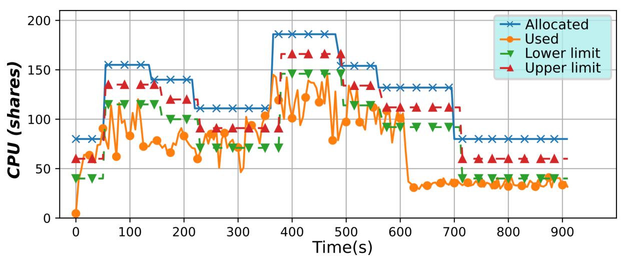
(a) Baseline



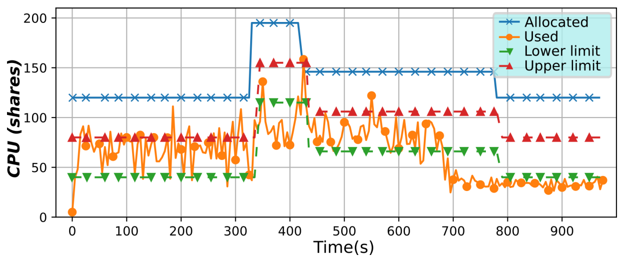
(b) cpu_mem



(c) CPU_mem



(d) cpu_MEM



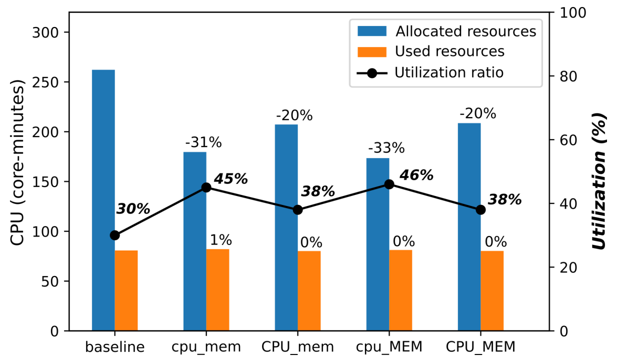
(e) CPU_MEM

Figure 10: CPU usage and scaling of FixWindow

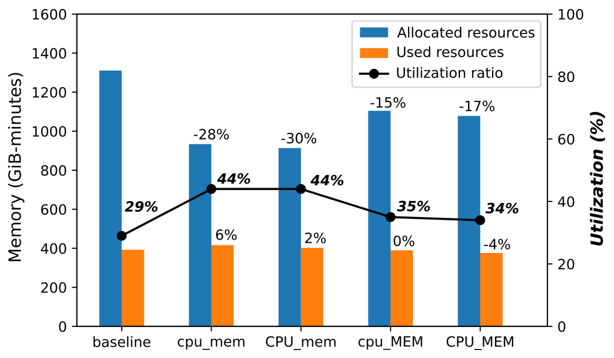
upper and lower limits as they are only present when analysing container resource usages. Moreover, these container-level plots allow to analyse the scaling operations that took place, why they were triggered, and how it is possible that the allocated resources may closely fit the used ones. For CPU, this is particularly the case with the *cpu_mem* and *cpu_MEM* configurations, as they set a lower boundary value. Regarding the platform behaviour to adapt to the stream and considering the three 5-minute stages previously described in Table 2, it is easily identifiable that the four serverless configurations must perform a scaling-up operation around second 300, as the second stage processes a larger stream. Later, around second 600, a scaling-down operation is performed as a low amount of CPU is required, considering that the third stage starts with the smallest stream (see Table 2). In addition, it is interesting to note how the left-hand plots (Figures 10b and 10d), which have a lower CPU boundary, perform initial scaling-up operations around second 50. This is due to a cold-start effect, as all the executions begin with the lowest possible allocated amount, considering that the minimum amount is set to 40 CPU shares for all configurations. The fact that the right-hand plots (Figures 10c and 10e) do not show this behaviour is because the allocated CPU amount was already large enough thanks to the higher boundary (see the effect of the boundary on the allocated amount in Figure 5).

6.4. Subsequent resource analysis

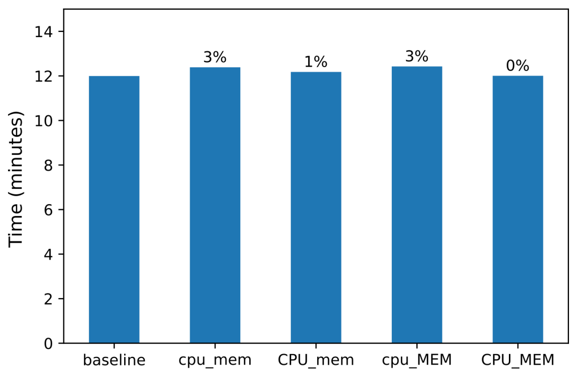
The subsequent analysis presents the aggregated amounts for used and allocated resources (i.e., the area values shown in Figure 6), as well as the resource utilization ratio as previously defined in Section 5.2. These results are presented for each workload using the baseline and the serverless configurations. The execution



(a) CPU usage



(b) Memory usage



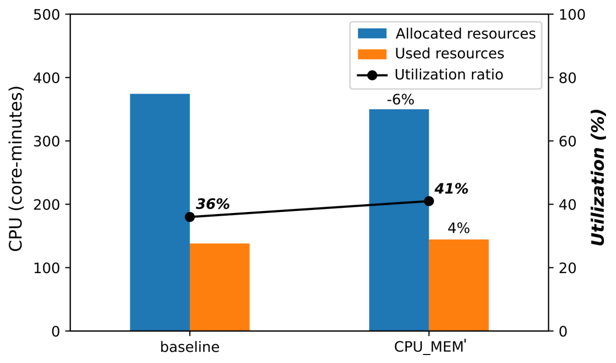
(c) Execution time

Figure 11: Resource usage and execution time of TeraSort

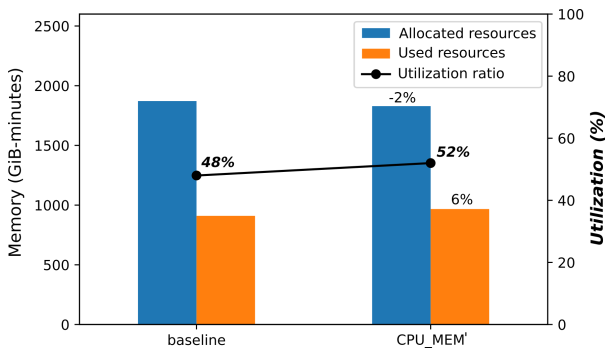
times of the workloads are also included to account for the platform overhead. We show the average value of a minimum of 10 executions for each workload and configuration.

For the TeraSort workload in Figure 11, the CPU plot (see Figure 11a) shows how the used amount is constant across all the executions, whether any scaling is applied or not. However, the allocated amount varies from the baseline to each configuration. Taking into account the boundary parameter, for *cpu_mem* and *cpu_MEM* configurations, which have a lower CPU value, the allocated amount is reduced by around 32%, whereas for *CPU_mem* and *CPU_MEM*, where such value is higher, the allocated amount is reduced by 20%. The CPU utilization ratio increases from 30% in the baseline to 46% and 38% for the lower and higher boundary configurations, respectively (which means an improvement of 53% and 27%). When it comes to memory (see Figure 11b), the results are analogous. The used amount varies across the different configurations with a higher amount for *cpu_mem* and a lower amount for *CPU_MEM*. Nevertheless, this difference can be ignored as memory amount varies slightly even for different executions with the same configuration due to the activity of the garbage collector in these Java-based workloads. For the lower memory boundary configurations (*cpu_mem* and *CPU_mem*), the amount of allocated memory is around 29% lower than the baseline, whereas for the higher memory configurations (*cpu_MEM* and *CPU_MEM*) it is around 16%. In turn, the memory utilization ratio increases from 29% of the baseline to around 44% and 34% for the lower and higher boundary configurations, respectively (which means an improvement of 52% and 17%). Figure 11c shows, for each configuration, the execution time and the corresponding overhead compared to the baseline. It is interesting to note how the overhead changes slightly according to the CPU boundary configuration while for memory it seems to have no effect. For the lower CPU boundaries an overhead of only 3% is imposed, being negligible for those with high boundaries.

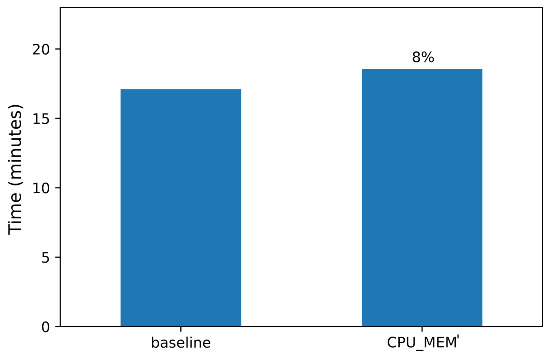
Figure 12 shows the results for PageRank. Both CPU and memory (see Figures 12a and 12b) present similar used values with only a 4% and a 6% increase, respectively, and a slightly lower allocated amount (6%



(a) CPU usage



(b) Memory usage

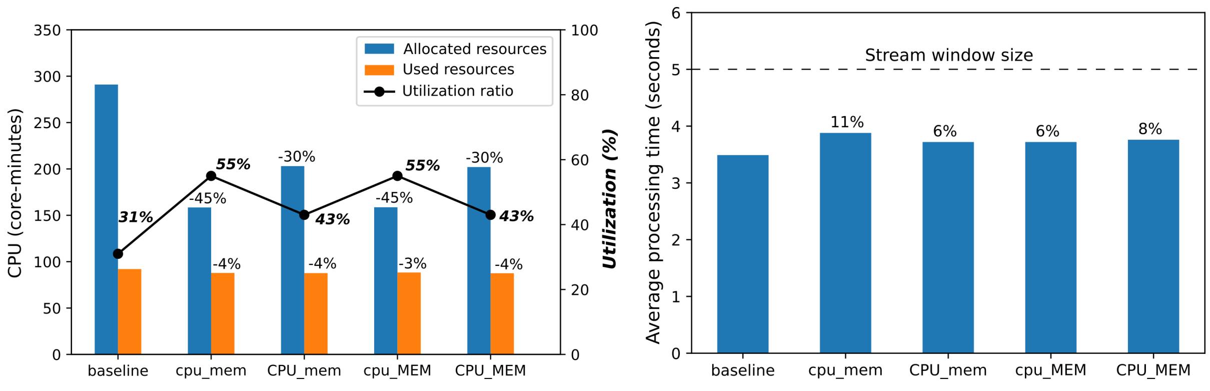


(c) Execution time

Figure 12: Resource usage and execution time of PageRank

and 2%). The utilization ratio raises slightly from 36% to 41% for CPU and from 48% to 52% for memory. In this workload, the platform has more difficulties to achieve better results due to the previously commented oscillating behaviour of the iterative phases of PageRank, having overall an execution time overhead of 8% (see Figure 12c). As with TeraSort, PageRank also shows to be more sensitive to CPU than to memory scaling, which requires to have a specific scenario with a more benevolent and responsive configuration (see Tables 3 and 4). Nevertheless, these experiments prove that in a worst-case scenario with an application that shows a highly variable resource usage pattern, our platform does not have a big impact on the workload and merely steers the serverless environment towards a traditional cloud instance with reserved resources. If needed, the platform can also be dynamically configured to further minimize this overhead by increasing benevolence even more, as the rule configuration can also be changed in real time to increase the amount of resources given to the application when scaling up.

Finally, the results for FixWindow are shown in Figure 13. The CPU usage (see Figure 13a) follows a similar behaviour as TeraSort. While the used amount is almost equal for all the configurations (around 4% lower than the baseline), the allocated amount is greatly reduced. According to the boundary, when a lower CPU value is used (*cpu_mem* and *cpu_MEM* configurations) the allocated CPU is reduced by 45%, while with a higher boundary it is reduced by 30%. As a consequence, the utilization ratio is greatly increased from 31% of the baseline to 55% and 43% for the lower and higher boundary configurations, respectively (which means an improvement of 77% and 39%). The execution time overhead is more or less constant for all configurations, ranging from 6% to 11% (see Figure 13b). It is important to note that this execution time corresponds to the average processing time for a window and not to the workload duration, as the stream is endless. Thus, the stream processing does not suffer any penalty as long as the average processing time remains lower than the stream window size (see this parameter in Table 2 and Figure 13b). Overall, FixWindow represents a best-case scenario for our platform as, unlike PageRank, the resource usage patterns



(a) CPU usage (b) Average processing time

Figure 13: CPU usage and average processing time of FixWindow

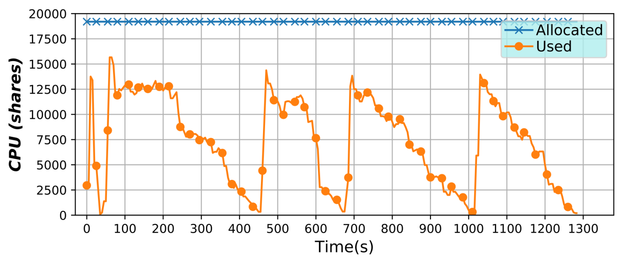
are more stable.

6.5. Scalability and hybrid experiments

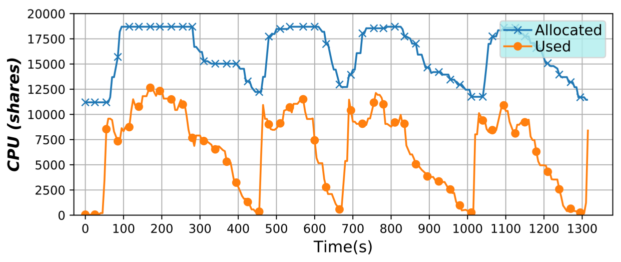
Considering that this platform aims to support Big Data workloads or, overall, any application that requires a lot of containers to be deployed, it is important to test the scalability of the platform on a realistic scenario. In addition to scalability experiments based on increasing both the size of the cluster and the datasets, other experiments referred to as 'hybrid' are also presented. Hybrid experiments try to show the flexibility of the platform when it comes to dealing with unexpected resource patterns that arise when multiple workloads are executed, even in a concurrent way, as well as with the use case of the need to change the resource limits within the execution. All the experiments presented in this section use the second testbed described in Section 6.1, which increases the number of cores from 24 to 192 (8x). Furthermore, the size of the datasets also increases (e.g. 12x for PageRank, see Table 2). For each workload and experiment, the baseline and the serverless scenario with the configuration of Table 5 are presented.

Regarding scalability experiments, Figure 14 shows the aggregated, application-level CPU usage for PageRank (top graphs) and FixWindow (bottom graphs) using the 32-container cluster. It can be seen how on the serverless scenario the limit is scaled accordingly and follows closely the usage pattern. For these experiments the CPU utilization ratio increases from 37% to 41% (11% improvement) for PageRank and from 28% to 39% (39% improvement) for FixWindow. The overhead in execution time is around 2% for PageRank, and 25% for the window average processing time of FixWindow. However, in this latter case the average processing time still keeps below the stream window size of 10 seconds, and thus it does not affect the stream processing overall.

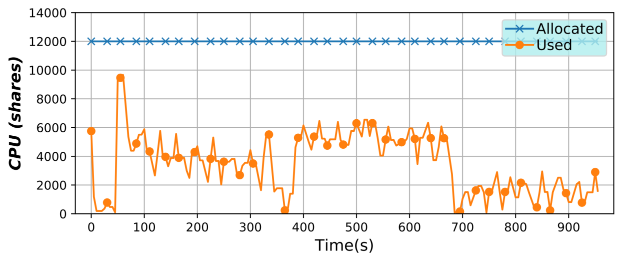
Figure 15 presents the execution of two hybrid workloads, where TeraSort and PageRank are executed using different combinations. The workload of Figures 15a and 15b consists of executing TeraSort (300 GiB), followed by PageRank (60 million pages), and finally another TeraSort (130 GiB). Note that this workload uses two different CPU limits: a maximum of 19200 shares (i.e., 32 containers with 6 cores each) for the first TeraSort and PageRank, while such limit is halved to 9600 shares for the second TeraSort at approximately second 2700. This change of resources is done dynamically, without any reboot, and with no inherent cost for our platform (see Figure 15b), only requiring a specific scaling operation and some minor changes in the structure documents to enforce the new limit. If compared with other solutions such as traditional cloud instances (see Figure 15a) a reboot would be needed to change the instance resources, thus having an undetermined delay. For this workload the overhead imposed by our platform is only around 5%, while the CPU utilization increases from 32% to 38% (19% improvement). The hybrid workload of Figures 15c and 15d consists of executing two TeraSort (200 GiB) concurrently with a 3-minute delay between both. This workload differs from the previous one in the fact that concurrency causes it to be significantly less deterministic, which serves to show how the platform adapts in real time to the unpredictable behaviour of



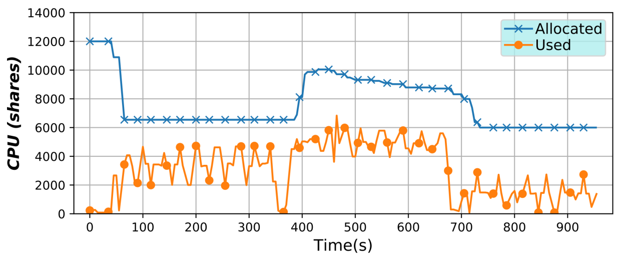
(a) Baseline (PageRank)



(b) Serverless (PageRank)

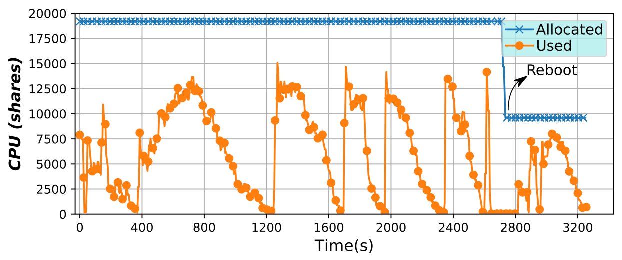


(c) Baseline (FixWindow)

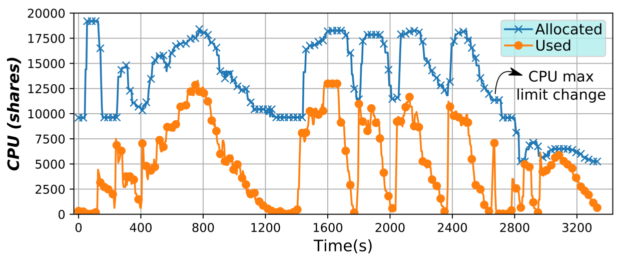


(d) Serverless (FixWindow)

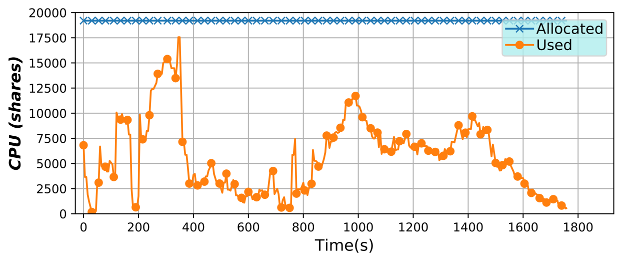
Figure 14: Aggregated CPU usage of PageRank (top) and FixWindow (bottom)



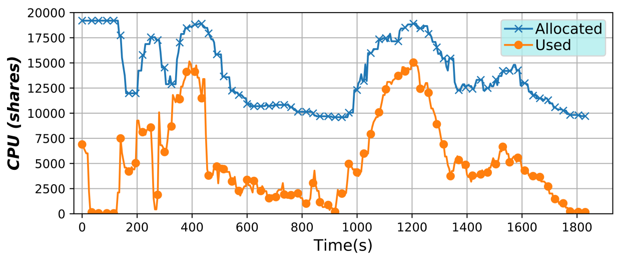
(a) Baseline (TeraSort+PageRank+TeraSort)



(b) Serverless (TeraSort+PageRank+TeraSort)



(c) Baseline (two concurrent TeraSort)



(d) Serverless (two concurrent TeraSort)

Figure 15: Aggregated CPU usage of the hybrid workloads

the underlying infrastructure. In this case, there is an execution overhead of only 4% (ignoring the reboot delay of the baseline) with a CPU utilization increase from 29% to 37% (28% improvement).

7. Conclusions

Newer paradigms for application execution are constantly appearing and evolving, always looking for more cost-effective ways of executing workloads that may be increasingly large as is the case of Big Data.

One of such novel paradigms is serverless computing, which evolved from the cloud adoption in recent years. Benefitting from the resource abstraction introduced by the cloud, where the user is able to deploy instances that can be dynamically configurable in terms of resources, serverless goes a step further and removes from the user the need to specify the resource allocation. By doing so, the user only needs a way of properly packaging and deploying the application to be executed. This in turn makes use of the recently adopted operating-system-level virtualization in the form of containers, which allows to easily package an application with its whole dependencies and configuration in one modular unit that may be deployed in any system supported by the corresponding container manager. Regarding resource management, it is worth noting that, on the one hand, the user is billed only for the used resources, while on the other hand the service provider must dynamically adjust and allocate enough resources for the application.

In this paper we presented a novel platform that can be used to deploy a serverless environment on a cluster of containers that host an application. In this environment, the resources available to the containers (and thus to the application) are dynamically and automatically scaled according to their real usage, without any need for user intervention. Unlike the most popular serverless platforms and services like Amazon Lambda, where applications must adapt to a specific template and configuration to be executed, in our platform users are given a virtual environment with a fully functional operating system by relying on containers, being overall closely similar to a traditional cloud instance or virtual machine. Furthermore, thanks to the ability to limit resources with cgroups, these container resources can be dynamically scaled without having to be rebooted unlike other types of instances like Amazon EC2 or most hypervisor-based virtual machines.

The presented platform was deployed on two different infrastructures to execute several Big Data applications in order to show: 1) how they adapt to a serverless environment, where the available resources dynamically change according to their real usage; and 2) the scalability and flexibility of the platform. Regarding the first point, the results were presented in two ways: the real-time resource scaling and the subsequent analysis of the resource usage patterns. The real-time scaling showed how the platform can be adapted to any application by using an appropriate configuration, although the degree of adaptation varies according to the resource usage of each application. The subsequent resource analysis proved why a serverless platform does not impose any penalty as long as it responds quickly enough to the application's needs. Such analysis also showed how the amount of used resources does not vary from experiments executed with a constant or reserved amount of resources (e.g., a virtual machine) to those with resource limits changed dynamically as in our platform. In turn this implies that the resource utilization ratio can be improved from a non-serverless environment to a scenario where the used and allocated resources are closer, while maintaining virtualized environments like containers. Such scenario is possible for workloads with a stable resource usage such as the FixWindow streaming application, where CPU utilization is improved by up to 77% with a window processing time overhead of only 6%. For a batch application like TeraSort, which presents a more variable use of resources than FixWindow, such improvement is up to around 53% for both CPU and memory utilization, with a negligible execution time overhead. Regarding the scalability of the platform, it was shown both from a design and theoretical point of view and experimentally how it can scale to encompass Big Data workloads without appreciating any impact on both the imposed overhead and the resource utilization ratio.

Finally, although the presented work and experiments only take into account CPU and memory resources, it is worth mentioning that our platform could also scale disk and network resources in the same manner. As BDWatchdog already supports the per-process and per-container monitoring of such resources, as future work we aim to limit and dynamically scale them by using cgroups for disks and the Traffic Control (TC) tool for network interfaces. For any other new resource, a few basic requirements could be laid out in order to consider if it could be potentially supported. First, it should be possible to monitor such resource at any moment, both its current usage as well as its current limit, and preferably from inside the container. Second, a limit should be possible to be set and enforced, whether via a control system such as cgroups (e.g., CPU) or any other way (e.g., network with TC).

The source code of the platform is publicly available at `http://bdwatchdog.dec.udc.es/serverless/`.

Acknowledgements

This work was supported by the Ministry of Economy, Industry and Competitiveness of Spain and FEDER funds of the European Union (project TIN2016-75845-P, AEI/FEDER/EU), the FPU Program of the Ministry of Education (grant FPU15/03381) and by Xunta de Galicia (Centro Singular de Investigación de Galicia accreditation 2016-2019, ref. ED431G/01). We also gratefully acknowledge CESGA for providing access to the Big Data infrastructure, and also sincerely thank Dr. Javier López Cacheiro for his technical support to perform some of the experiments. Other experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations.

References

[1] B. Varghese, R. Buyya, Next generation cloud computing: new trends and research directions, Future Generation Computer Systems 79 (2018) 849–861.
[2] R. Buyya, et al., A manifesto for future generation cloud computing: research directions for the next decade, ACM Computer Surveys 51 (5) (2018) 105:1–105:38.
[3] E. van Eyk, A. Iosup, S. Seif, M. Thömmes, The SPEC cloud group's research vision on FaaS and serverless architectures, in: 2nd International Workshop on Serverless Computing, WoSC '17, Las Vegas, NV, USA, 2017, pp. 1–4.
[4] D. Bernstein, Containers and cloud: from LXC to Docker to Kubernetes, IEEE Cloud Computing 1 (3) (2014) 81–84.
[5] CESGA Supercomputing Center, `http://www.cesga.es/`, Last visited: October 2019.
[6] GRID'5000 testbed for experiment-driven research, `https://www.grid5000.fr`, Last visited: October 2019.
[7] D. Merkel, Docker: lightweight Linux containers for consistent development and deployment, Linux Journal 239 (2014) 76–91.
[8] J. Enes, R. R. Expósito, J. Touriño, BDWatchdog: real-time monitoring and profiling of Big Data applications and frameworks, Future Generation Computer Systems 87 (2018) 420–437.
[9] Amazon EC2 service, `https://aws.amazon.com/ec2/`, Last visited: October 2019.
[10] Google Compute Engine (GCE), `https://cloud.google.com/compute/`, Last visited: October 2019.
[11] Amazon ECS service, `https://aws.amazon.com/ecs/`, Last visited: October 2019.
[12] Amazon Lambda service, `https://aws.amazon.com/lambda/`, Last visited: October 2019.
[13] Google Cloud Functions, `https://cloud.google.com/functions/`, Last visited: October 2019.
[14] Microsoft's Azure Functions service, `https://azure.microsoft.com/en-us/services/functions/`, Last visited: October 2019.
[15] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Serverless computation with OpenLambda, in: 8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'16, Denver, CO, USA, 2016, pp. 33–39.
[16] I. Baldini, et al., Cloud-native, event-based programming for mobile applications, in: International Conference on Mobile Software Engineering and Systems, MOBILESoft'16, Austin, TX, USA, 2016, pp. 287–288.
[17] LXD - the Linux container daemon, `https://help.ubuntu.com/lts/serverguide/lxd.html.en`, Last visited: October 2019.
[18] N. Savage, Going serverless, Communications of the ACM 61 (2) (2018) 15–16.
[19] I. Baldini, et al., Serverless computing: current trends and open problems, in: Research Advances in Cloud Computing, Springer Singapore, 2017, pp. 1–20.
[20] M. Malawski, A. Gajek, A. Zima, B. Balis, K. Figiela, Serverless execution of scientific workflows: experiments with HyperFlow, AWS Lambda and Google Cloud Functions, Future Generation Computer Systems (in press) `doi:10.1016/j.future.2017.10.029`.
[21] A. Pérez, G. Moltó, M. Caballer, A. Calatrava, Serverless computing for container-based architectures, Future Generation Computer Systems 83 (2018) 50–59.
[22] R. Han, L. Guo, M. M. Ghanem, Y. Guo, Lightweight resource scaling for cloud applications, in: 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID '12, Ottawa, ON, Canada, 2012, pp. 644–651.
[23] R. Han, L. Guo, Y. Guo, S. He, A deployment platform for dynamically scaling applications in the cloud, in: 3rd International Conference on Cloud Computing Technology and Science, CloudCom'11, Athens, Greece, 2011, pp. 506–510.
[24] A. Floratou, A. Agrawal, B. Graham, S. Rao, K. Ramasamy, Dhalion: self-regulating stream processing in Heron, Proceedings of the VLDB Endowment 10 (12) (2017) 1825–1836.
[25] S. Kulkarni, et al., Twitter Heron: stream processing at scale, in: International Conference on Management of Data, SIGMOD'15, Melbourne, Australia, 2015, pp. 239–250.
[26] M. Bansal, E. Cidon, A. Balasingam, A. Gudipati, C. Kozyrakis, S. Katti, Trevor: automatic configuration and scaling of stream processing pipelines, arXiv:1812.09442 (2018).
[27] Amazon Auto Scaling service, `https://aws.amazon.com/autoscaling/`, Last visited: October 2019.
[28] Amazon CloudWatch service, `https://aws.amazon.com/cloudwatch/`, Last visited: October 2019.
[29] Amazon Fargate service, `https://aws.amazon.com/fargate/`, Last visited: October 2019.

[30] T. W. Wlodarczyk, Overview of time series storage and processing in a cloud environment, in: 4th IEEE International Conference on Cloud Computing Technology and Science, CloudCom'12, Taipei, Taiwan, 2012, pp. 625–628.

[31] J. C. Anderson, J. Lehnardt, N. Slater, CouchDB: the definitive guide: time to relax, O'Reilly Media, Inc., 2010.

[32] M. Grinberg, Flask web development: developing web applications with Python, O'Reilly Media, Inc., 2014.

[33] Gunicorn: a Python WSGI HTTP server for UNIX, `http://gunicorn.org/`, Last visited: October 2019.

[34] J. Gardner, The Web Server Gateway Interface (WSGI), in: The definitive guide to Pylons, Springer, 2009, pp. 369–388.

[35] JsonLogic, `http://jsonlogic.com/`, Last visited: October 2019.

[36] R. Shi, Y. Gan, Y. Wang, Evaluating scalability bottlenecks by workload extrapolation, in: 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS'18, Milwaukee, WI, USA, 2018, pp. 333–347.

[37] J. Zhang, X. Lu, J. Jose, M. Li, R. Shi, D. K. Panda, High performance MPI library over SR-IOV enabled InfiniBand clusters, in: 21st IEEE International Conference on High Performance Computing, HiPC'14, Goa, India, 2014, pp. 1–10.

[38] The Apache Software Foundation, Apache Hadoop, `http://hadoop.apache.org/`, Last visited: October 2019.

[39] M. Zaharia, et al., Apache Spark: a unified engine for Big Data processing, Communications of the ACM 59 (11) (2016) 56–65.

[40] J. Veiga, J. Enes, R. R. Expósito, J. Touriño, BDEv 3.0: energy efficiency and microarchitectural characterization of Big Data processing frameworks, Future Generation Computer Systems 86 (2018) 565–581.

[41] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The HiBench benchmark suite: characterization of the MapReduce-based data analysis, in: 26th IEEE International Conference on Data Engineering Workshops, ICDEW'26, Long Beach, CA, USA, 2010, pp. 41–51.

[42] N. Garg, Apache Kafka, Packt Publishing Ltd., 2013.