Future Generation Computer Systems 155 (2024) 256–271

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Serverless-like platform for container-based YARN clusters

Óscar Castellanos-Rodríguez *, Roberto R. Expósito, Jonatan Enes, Guillermo L. Taboada, Juan Touriño

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

ARTICLE INFO

Keywords:
Serverless computing
Big Data
Hadoop YARN
Resource scaling
Container cluster
Infrastructure as Code

ABSTRACT

Serverless computing is an emerging paradigm that has gained a lot of relevance in recent years, as it allows users to consume computing resources without worrying about the underlying infrastructure and pay only for what they actually use. Most current services that implement this paradigm typically rely on the Function-as-a-Service (FaaS) model, which works perfectly for simple applications based on stateless functions triggered by specific events. However, these services are not designed to run more complex applications with intricate interactions, usually presenting a significant degree of configuration difficulty and/or low ability to customise the execution environment. They also tend to be designed for short and simple workloads, with some services even limiting their maximum runtime to just a few minutes. In this paper, we present a platform based on Hadoop YARN oriented to the execution of Big Data workloads in a containerised and serverless way, so that the resources allocated to such containers are automatically and dynamically scaled according to their actual usage. An experimental evaluation has been carried out to compare our serverless-like platform with a standard YARN deployment when executing Big Data workloads concurrently. Our results have shown experimental evidence of enhancing both performance and overall resource efficiency, providing runtime reductions and resource usage improvements of up to 41% and 50%, respectively.

1. Introduction

Serverless computing [1] is today a highly demanded paradigm due to the growing need for users to run workloads easily, while avoiding the need to explicitly provision or manage a server, and paying only for the specific resources consumed when billing is applied. However, this flexibility requires from the provider: deploying the appropriate infrastructure to offer the service, continuously managing the available resources to allow each client to run its applications seamlessly, and overall minimising the costs during the whole process. This last point is particularly important because, for example, the overestimation of resources can lead to underutilisation, which in turn means a loss of profit as these resources are not billed. Additionally, it is worth considering the FaaS model [2], the most popular and widespread serverless approach today, as it allows users to automatically run an application (or "function") for which they provide just its code. However, FaaS is usually very limited to simple workloads, for example a series of single functions used as event triggers, as well as stateless services that can be easily scaled [3]. From the user's point of view, it would be highly interesting to support a wider variety of use cases, but there is still little research regarding the execution of more complex applications in these serverless environments. Examples are workloads with specific needs such as communication between processes or just with very long execution times, aspects that are usually limited in public serverless services.

In this context, it is worth noting the great relevance of Big Data technologies in many fields, such as healthcare [4] or retail [5]. Currently, there are several frameworks like Hadoop [6], Spark [7] and Flink [8], which allow processing the data in a distributed and scalable manner. One major advantage is their ability to process computationally intensive tasks on commodity hardware by distributing them into smaller tasks that can be efficiently processed independently. However, deploying and configuring such distributed computing frameworks may be complex for inexperienced users. The serverless model can be a viable approach for tackling this issue, while also providing cost benefits since users would only pay for the resources they actually consume. Unfortunately, these Big Data frameworks are examples of the applications previously described as too complex to fit in current serverless platforms. This is because such platforms usually restrict communications between tasks or processes, impose specific runtime and memory limits, and work only with network-based storage, limiting I/O rate

* Corresponding author.
E-mail addresses: oscar.castellanos@udc.es (Ó. Castellanos-Rodríguez), roberto.rey.exposito@udc.es (R.R. Expósito), jonatan.enes@udc.es (J. Enes), taboada@udc.es (G.L. Taboada), juan@udc.es (J. Touriño).

https://doi.org/10.1016/j.future.2024.02.013
Received 15 July 2023; Received in revised form 11 January 2024; Accepted 16 February 2024
Available online 17 February 2024

and slowing down intermediate writes performed by the frameworks. Furthermore, Big Data workloads are usually resource intensive and their performance can be negatively affected if the scaling performed by the serverless platform is not proper.

In this paper we present a platform that automatically deploys serverless clusters on demand oriented for running Big Data workloads. These clusters are based on containers to leverage the fast and light deployment provided by this virtualisation technology. A cluster is deployed for each application, so that different clusters belonging to different applications and users can coexist on the same physical infrastructure at a given time. Furthermore, Infrastructure as Code (IaC) [9] tools are used to further accelerate and automate both the cluster deployments and the platform itself. In order to provide end users with support for a wide range of Big Data workloads, the Hadoop Yet Another Resource Negotiator (YARN) cluster manager [10] is configured on each container cluster. Therefore, each application or user is provided with a private and customised YARN cluster. Our approach is significantly different from a standard YARN deployment, where a single YARN cluster tries to satisfy the needs of all the applications and users. To provide serverless capabilities to the container clusters and thus offer serverless YARN environments, our platform relies on a framework capable of continually accounting for and scaling the resources of each container, thus efficiently adjusting the allocated resources according to the actual usage, while minimising the impact on performance. Finally, the platform also provides a web interface that allows launching workloads and easily managing the platform. The main contributions of this paper to the state of the art are the following:

- Up to our knowledge, this is the first platform that allows to deploy serverless container clusters based on YARN and focused on running Big Data workloads.
- An automated deployment system for the platform itself and the serverless clusters and applications through IaC-based technologies, as well as a web interface to ease the platform management.
- An experimental evaluation using different Big Data applications to prove the benefits of the serverless YARN clusters provided by our platform when compared to simply deploying YARN without any serverless capabilities.

The rest of the paper is organised as follows. Section 2 introduces the technologies relevant to this paper. Related work is discussed in Section 3. Section 4 presents the overall functionalities of the platform, while Section 5 details its implementation. The experimental results are presented in Section 6, comparing them with a standard YARN deployment. Finally, Section 7 concludes the paper.

2. Background

This section introduces the main concepts relevant to this work, such as container-based virtualisation in Section 2.1 and an overview of serverless computing in Section 2.2. Section 2.3 presents the IaC tools upon which the platform deployment is built, and finally Section 2.4 exposes the Big Data technologies that are relevant to this paper.

2.1. OS-level virtualisation

Among the different forms of server virtualisation, OS-level is nowadays the most widely extended. Basically, it consists of creating isolated virtual environments, usually named as containers, which share the OS kernel in which they are hosted, imposing a low overhead and making their deployment very light. In addition, many container technologies support cgroups [11], a Linux kernel feature that allows limiting, accounting for, and isolating the resource usage of a collection of processes. Considering that containers can just be treated as a group of processes from the OS point of view, cgroups allows vertically scaling their resources according to their actual usage, thus providing the underlying foundation for a potential serverless-like environment. The container engine chosen for our platform is Apptainer, formerly known as Singularity [12]. Besides being compatible with cgroups, Apptainer also offers certain advantages over other popular engines, such as Docker [13]. For instance, it follows a daemonless architecture, requiring no additional background processes, and provides rootless support by default. These features make Apptainer a perfect candidate for easily running containers on shared multi-user systems, such as High Performance Computing (HPC) clusters.

```
1 - hosts: all
2   become: yes
3 - name: Install apache
4   package:
5     name: apache2
6     state: present
```

Listing 1: Example of a basic Ansible playbook

2.2. Serverless computing and FaaS

The serverless paradigm is a model that enables users to run applications without needing to define the resource configuration of the underlying servers. Although servers still exist, users are relieved of managing them; instead, providers operate such servers and scale their resources based on demand. There are two key concepts associated with this paradigm:

- Pay only for what you use: this is closely related to cloud providers when billing is applied. Users only pay for the resources consumed while their code is running, as opposed to renting a virtual machine for a period of time, where they would pay even for idle resources. The challenge is to scale resources appropriately, for example by releasing those that are allocated but underutilised by a user to make them available to other users.
- Elastic scalability: users can access all the necessary resources within their limits and/or the provider's capacity. The allocated resources are dynamically and transparently scaled to ensure the required capacity at any time.

This paradigm is strongly related to virtualisation, as the most common implementation approach is to have several physical servers where multiple virtual environments are deployed. The resources of these environments are allocated and changed dynamically, thus achieving greater flexibility when managing them. Fargate [14] is an example of a serverless service from the Amazon Web Services (AWS) cloud provider [15], which allows application environments to be deployed using a container orchestration service, either Elastic Container Service (ECS) [16] or Elastic Kubernetes Service (EKS) [17].

Currently, FaaS is the most widespread implementation of the serverless paradigm. In this model, users only submit the application code to the FaaS service, which will execute it typically when triggered by specific events or requests, making it well suited for simple tasks or single functions commonly used to process a data stream. Other workloads that fit well are stateless services that may need to scale easily, such as a web server. However, this simplicity often limits its potential use cases, especially the type of workloads that can be executed, leaving out complex workloads such as Big Data ones. Furthermore, some FaaS services limit the maximum allowed runtime and/or allocable resources. Examples of FaaS services are Amazon Lambda [18], Azure Functions [19] and Google Cloud Functions [20].

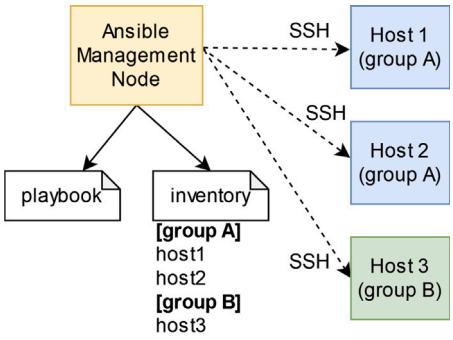


Fig. 1. Ansible architecture.

2.3. Infrastructure as Code

The IaC paradigm allows infrastructure resources to be defined and managed using source code. The main idea is to develop code that describes the infrastructure specification so that it can be provisioned and/or managed in an automated manner. In this way, methodologies and tools commonly used in traditional software development can also be applied (e.g., Git), but managing infrastructure instead of applications. The main purpose of IaC is to dynamically provide and maintain systems that are predictable and whose configurations should be immutable. In other words, the same environment should always be generated from the same code, making it very easy to repeatedly deploy such environment on different machines or to restore it if necessary. There are currently multiple IaC tools that can be classified under different criteria, such as their supported functionalities or their underlying architectures.

Ansible [21] was the IaC tool chosen to automatically deploy our platform, as well as the containers and applications that run on the physical nodes. This tool follows an agentless architecture, so it does not need to be installed on all the servers to be managed, but only on a single server, called the management node, from which remote servers are configured through SSH. Fig. 1 depicts its architecture along with the most important concepts. The nodes to be managed can be grouped, an interesting feature in order to apply the same configuration to an entire group. The definition of the managed nodes and groups is done in the inventory file. Another key concept is the playbook, which is a YAML file that defines the actions to be performed on the nodes. Listing 1 shows a basic playbook where the Apache web server is installed on all the nodes listed in the inventory. Note that Ansible tasks are idempotent: if a task is executed several times, the final result remains the same.

2.4. Big Data technologies

The vast development in many fields has significantly increased the amount of generated data. To address this challenge, new technologies have emerged, as traditional ones struggle to process such large datasets. It was remarkable the emergence of MapReduce [22], a parallel paradigm characterised by dividing data processing into two phases: Map and Reduce. These sub-processes are executed in a distributed manner on a cluster of commodity nodes. To support this processing, MapReduce relies on a distributed data storage and processing model that follows a master/worker architecture. Inspired by this model, other technologies implementing or evolving it came to light, most notably the Apache Hadoop open-source project [6]. Hadoop integrates HDFS [23] as the storage system to distribute data in blocks across the cluster nodes, thus providing the prior data division needed by the Hadoop MapReduce processing engine, in addition to replicating those blocks to ensure fault tolerance. HDFS relies on two main services: the NameNode, a process executed on the master node to manage the distributed file system, and the DataNode, which is executed on each worker node to store the data blocks on local disks. Apache Spark [7] is another open-source framework that emerged as an evolution of Hadoop MapReduce, designed to overcome some of its limitations (e.g., stream processing).

When running applications using these Big Data frameworks, it is necessary to schedule and assign the processing tasks to be executed on the available cluster nodes, as well as to manage the resources allocated to such tasks, which is the responsibility of the cluster manager. Hadoop provides YARN [10] as a built-in manager, also compatible with many other frameworks, including Spark. YARN is divided into three main components:

- ResourceManager (RM), a process running on the master node responsible for managing the cluster resources. The RM maintains a list of active nodes and their available resources, and accepts requests for application execution.
- NodeManager (NM), which runs on each worker node and provides the resources required by the applications following the instructions from the RM.
- Application Master (AM), which is launched by each application when its execution starts. Its main function is to negotiate the resources with the RM and monitor the status and progress of the application.

The scheduling of the application execution submitted to the RM depends on the scheduler. Currently, YARN provides three different schedulers: FIFO, Capacity and Fair. FIFO, the simplest, executes the applications in order of arrival and does not allow concurrent execution. So, long-running applications can potentially block shorter ones that may only require a small portion of the resources. The Capacity scheduler allows defining multiple queues with a percentage of the cluster resources allocated, so that each queue is guaranteed a minimum amount of resources and applications can run concurrently if submitted to different queues. Applications in the same queue may also run concurrently, depending on the queue policy. Finally, the Fair scheduler is similar to Capacity, sharing the feature of queues and their minimum resources, but instead of partitioning the resources statically they are dynamically balanced among the submitted jobs. When an application is already running and a new one is submitted, resources will be allocated to it as soon as any of the tasks of the running application finishes.

3. Related work

Our work is related to multiple topics, such as the FaaS and serverless paradigms (Section 3.1), the dynamic scaling of applications (Section 3.2), and the containerised deployment of Big Data applications (Section 3.3). Next, each of these topics is studied, with a focus on their most common approaches.

3.1. FaaS and serverless platforms

Currently, most of the serverless products offered by major cloud providers are based on the FaaS paradigm. Of particular note are the services mentioned in Section 2.2, AWS Lambda, Azure Functions and Google Cloud Functions, which are oriented towards the transparent execution of single functions. There are also open-source solutions, most notably OpenWhisk [24]. These types of services focus on the simplicity for the user to execute applications, although this indirectly limits the use cases to relatively simple workloads. Inspired by these services, a body of work is emerging that seeks to exploit their capabilities, but overcoming such limitations and thus allowing the execution of more complex applications and frameworks. For example, there is a prototype [25] that combines the HyperFlow engine [26] with AWS Lambda and Google Cloud Functions in order to execute

scientific workflows. SWEEP [27] is another workflow-oriented system that supports Lambda together with AWS Fargate as serverless engines. Moreover, specifically for Big Data, there is Flint [28], a Spark-based engine implemented with AWS Lambda, and MARLA [29], a framework aimed at executing MapReduce jobs in Python also through Lambda. However, despite their interesting features, they have a common disadvantage in comparison to our proposal, which is the set of restrictions imposed by the underlying services on which they depend. This causes them to be limited in terms of maximum allowed execution time or allocable resources, or to experience some performance degradation when implementing workarounds to overcome such limitations.

Other works focus on solving some of the intrinsic problems of these platforms. For example, SCAR [30] combines FaaS with container engines such as Docker, allowing to transparently launch custom containerised environments on serverless platforms. This provides greater flexibility in the applications to be executed because they are not tied to developing functions in the programming languages supported by the serverless platforms. However, they are still constrained by the runtime and resource boundaries of such platforms. Others solutions include SOCK [31], designed for the rapid deployment of containers reducing the overhead usually present in serverless platforms, and Fifer [32], a framework that not only reduces such overhead but also improves the resource usage efficiency of the cluster where containers are deployed. Although these solutions aim to address certain performance issues of FaaS platforms, they are still limited to their common use cases.

3.2. Dynamic resource scaling

There are several solutions capable of scaling applications by increasing or reducing the number of virtual machine instances (horizontal scaling), as well as their resources such as CPU or memory (vertical scaling) [33]. Newer approaches focus on scaling containers, taking advantage of their faster allocation/deallocation. For instance, there are container orchestrators such as Docker Swarm [34], which only allows horizontal scaling, and Kubernetes [35], a more complex platform that also supports vertical scaling. However, Kubernetes presents some limitations, such as conflicts occurring when both scalers are used [36], as they may respond to the same scaling need without considering each other, which can lead to unknown side effects. Furthermore, the vertical scaler requires restarting a container to modify its allocated resources, which can interrupt the execution flow of an application. In addition, both are non-specific container orchestrators, which may complicate the deployment and scaling of applications with particular needs (e.g., Big Data workloads, which require dedicated disks and communication between workers). Overall, all these limitations mean that there is no straightforward solution to create container-based serverless environments with these technologies as of today.

There are other solutions such as Dhalion [37], a framework capable of automatically scaling applications running on Twitter Heron [38]. It works by monitoring the health of the workload to detect symptoms that could point to performance degradations, triggering as a result the scaling up or down of applications. However, Dhalion is limited to streaming applications built upon Heron. Another interesting framework is RUBAS [39], capable of dynamically adjusting the allocation of containers deployed on Kubernetes to match the needs of the running application. Finally, there is also Serverless Containers [40], which focuses on fine-grained resource scaling (e.g., CPU, memory) of containers in real time via cgroups and, unlike Kubernetes, without restarting them, while providing the resource accounting that characterises the serverless paradigm. Both RUBAS and Serverless Containers are designed to run standalone workloads, but have the drawback that running a more complex execution model (such as MapReduce) makes it very difficult to configure and set up the environment, as opposed to our platform designed to simplify the execution of Big Data applications on top of YARN. Even so, due to its interesting vertical scaling features, the Serverless Containers framework was used to perform the underlying resource scaling of the serverless YARN clusters provided by our platform.

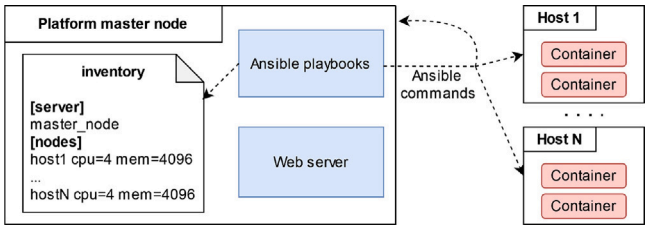


Fig. 2. High-level platform architecture.

3.3. Containerised deployment of Big Data applications

Regarding this topic, there are some interesting works to be mentioned. The system presented in [41] is able to deploy Docker containers in different clouds for Big Data analysis. Another research work [42] proposes a MapReduce-oriented HPC system based on C++, which can run on Docker or Singularity. The work [43] proposes a deployment scheme for Big Data applications through Singularity containers, and also describes the network configuration required for their communication. Despite the interesting contributions of these previous works, they do not provide serverless capabilities to adjust the allocated resources to the actual usage. Finally, it is worth mentioning JellyFish [44], a performance tuning system for Hadoop YARN that implements elastic containers where tasks are executed and can dynamically expand and shrink according to their usage. However, it is restricted to running Hadoop MapReduce, and it lacks support for multi-tenant environments where concurrent application execution occurs.

4. High-level overview of the serverless platform

Our platform has been designed with the following main objectives in mind: (1) ease of use and deployment for both users and system administrators; (2) improvement of usage efficiency so that users only pay for what they actually use, while freeing up resources for other clients and allowing providers to increase their revenue; and (3) focus on Big Data workloads through Hadoop YARN. The platform is built on top of two fundamental components: an automated deployment system, and a web interface for managing the platform and launching applications within container-based YARN clusters. It is worth mentioning that, although our focus is on Big Data workloads, any application or framework compatible with YARN can also be deployed. The platform follows the classic master/worker architecture, as depicted in Fig. 2, where the microservices that make up the core of the platform are deployed on a master node, and the virtual YARN clusters run on N worker nodes, or "hosts" from now on. The hosts and the master node itself are managed through Ansible, so the master node acts as the management node from the point of view of the Ansible architecture (see Fig. 1). Therefore, the inventory file specifies the hosts to be managed and their available computational resources in terms of CPU, memory and disk. If the platform is deployed on a homogeneous cluster, the inventory can be loaded automatically through a configuration file, defining variables such as the number of cores per host.

The automated deployment of the entire platform is performed by a set of Ansible playbooks, which are all executed through a provided script to achieve a higher level of automation. The tasks executed in those playbooks range from the installation of required packages and dependencies to the startup and configuration of all the microservices that make up the platform. In addition, the platform can be deployed on a physical cluster infrastructure or, for testing purposes, on a virtual cluster executed on a single physical computer. The latter is done by relying on Vagrant [45], an IaC tool for managing virtual environments, so that this deployment is also fully automated.

Once the platform is deployed, its main functionalities can be accessed through the web server. Some of them include to directly

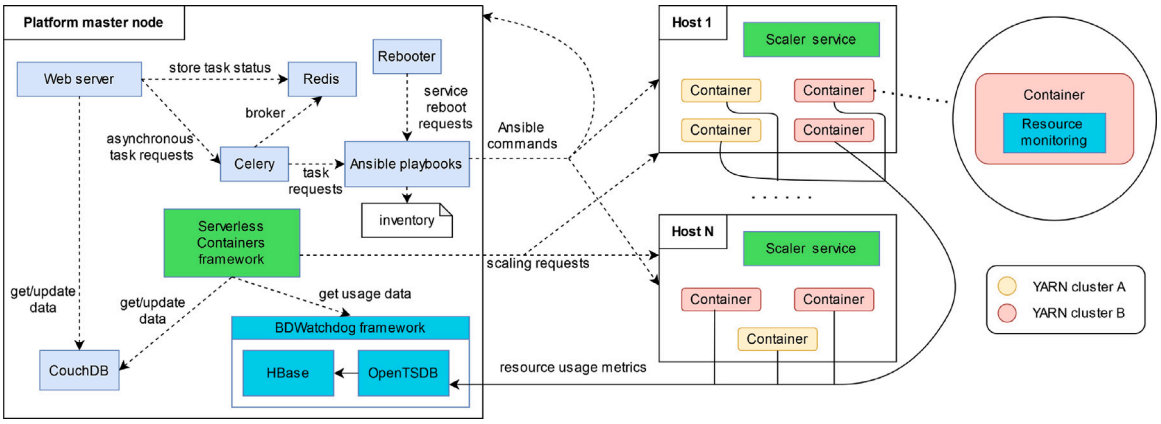


Fig. 3. Fine-grained platform architecture.

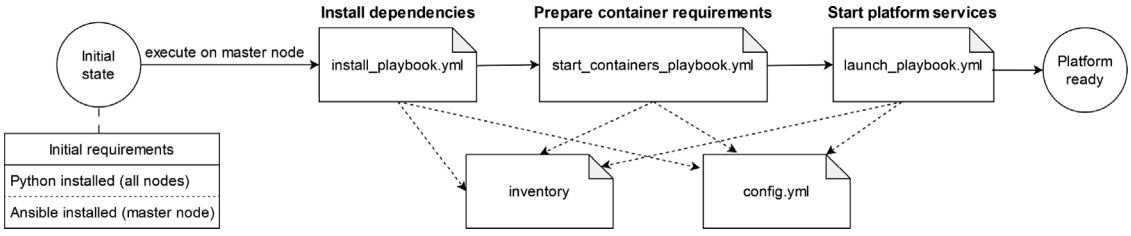


Fig. 4. Platform deployment sequence diagram.

deploy and manage containers for testing purposes, launch virtual YARN clusters for application execution, and modulate the underlying vertical scaling of the containers.

5. Implementation

The details of the automated deployment system are presented in Section 5.1. The implementation of the web interface is described in Section 5.2, whereas the mechanisms for automatically deploying serverless YARN clusters to execute applications are discussed in Section 5.3. Throughout this implementation section, the platform architecture and its components are described in detail. Fig. 3 shows a fine-grained platform diagram that zooms in on the high-level one shown in Fig. 2.

5.1. Automated deployment

The deployment of the platform from scratch on a cluster of nodes must meet some minimum software requirements, as shown in Fig. 4, and have SSH connectivity enabled from the node that acts as master to the nodes with the role of hosts. These roles and the resources of the hosts must be configured in the Ansible inventory, as explained in the previous section (see Fig. 2). From this state, the administrator can deploy the platform with a single command executed on the master node. This process consists of three main phases, as shown in Fig. 4, each phase corresponding to an Ansible playbook. These playbooks are in turn divided into several files that group together related tasks. This approach improves code modularity and allows tasks to be reused across multiple playbooks. It is worth noting that due to the idempotency of Ansible, as stated in Section 2.3, only the necessary tasks are carried out.

More specifically, the first phase installs all the software and dependencies of the platform with the *install_playbook.yml* file. It is important to note the installation of BDWatchdog [46], a monitoring framework that provides detailed information about the resource usage of containers. As a requirement for BDWatchdog, both an HBase [47] database and an OpenTSDB [48] service must be deployed, the latter being

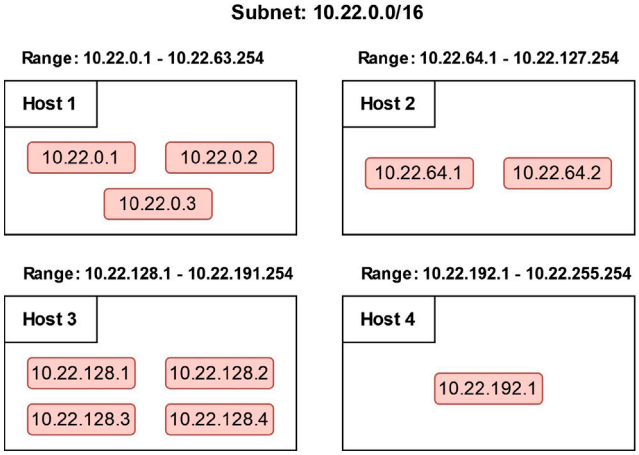


Fig. 5. Example of IP address assignment to containers.

a time series management service that uses HBase as the underlying storage. The resource monitoring tool of BDWatchdog runs within the containers and collects usage metrics that are sent to OpenTSDB for their subsequent management by the Serverless Containers framework (see Fig. 3). Moreover, this first playbook configures the network connections between containers. For this purpose, our platform relies on the Container Network Interface (CNI) [49] that consists of a specification and libraries for writing plugins to configure container networking, along with different supported network plugins. In our platform, a range of IP addresses is assigned to each host, where all ranges are under the same subnet with no address overlap, so that all containers across the hosts are reachable within the same network. Fig. 5 shows an example of address assignment to containers using four hosts. To implement such networking, our platform allows selecting a network plugin, currently supporting different options such as bridge, macvlan or ipvlan.

The second phase of the deployment prepares all the requirements needed for the proper execution of the containers, carried out by *start_containers_playbook.yml*. Regarding the containers themselves, a key Ansible task consists of creating a base image using a Singularity definition file that holds the required software for BDWatchdog's monitoring (see container bubble in Fig. 3). This base image is customised later before cluster startup (i.e., "offline") depending on the Big Data framework to be deployed. Therefore, users simply choose the required framework for their applications (e.g., Hadoop, Spark) and the corresponding YARN clusters are built using an image with the requested framework installed. Regarding the hosts, the Scaler service responsible for scaling the container resources is deployed on them. Once both the container image and hosts are prepared, this playbook starts the containers requested by the users, either in this phase just for testing, or later through a request via the web interface. A final task consists of binding one directory of each container to one of the physical disks available on the host in order to distribute disks among containers, trying to minimise disk contention.

The third phase (*launch_playbook.yml*) is responsible for launching all the microservices that make up the platform on the master node (e.g., Redis), including the web server for its management. Moreover, this playbook launches the BDWatchdog and Serverless Containers frameworks, including a CouchDB database [50] that stores all the information about hosts and containers, as well as the Serverless Containers' microservices and their scaling configuration. This information is displayed in the web interface and is constantly accessed and updated, as it is critical to later provide the serverless capabilities and modulate their behaviour. Additionally, the "Rebooter" service is responsible for monitoring the rest of the microservices and detecting those that are unresponsive or malfunctioning, in order to restart them if necessary. Fig. 3 shows all these components and their interactions within the platform architecture.

Finally, another key piece is the configuration file (*config.yml*, see Fig. 4), partially shown in Listing 2. Most options are self-explanatory or include a brief description, but some clarification may be necessary. The "hosts" configuration parameters (lines 2–8) refer to the worker nodes on which containers are executed. These parameters are used to automatically generate the Ansible inventory when deploying the platform on homogeneous clusters. Thus, it is possible to set the number of hosts, their CPU cores, memory and disks, even distinguishing between SSD and HDD disks. The "containers" configuration parameters (lines 11–15) can be used to start some test containers during the platform deployment.

5.2. Web interface

The interface was implemented using the Django web framework [51]. When opening any page, the required information is retrieved from the CouchDB database at runtime, and then displayed to the user. This visualisation is done through HTML templates using variables whose values are specified with such retrieved data. Most of these items are provided through drop-down boxes, which allows the user to select the amount of information to be displayed at any time. To process user requests such as launching a new YARN cluster, a web form with the corresponding fields is displayed (see Fig. 6).

When it comes to these requests, some of them are very simple and only require interaction with the database, but others also need interaction with the underlying Ansible deployment system, such as when starting the containers for an application or provisioning a new host. Considering that such requests can take longer than just a few seconds, a task queuing system has been used to enable their asynchronous execution, thus avoiding blocking the interface. To implement such queuing system, Celery [52] was chosen, mainly because of its easy integration with Django. Additionally, Redis [53] is used as a broker to manage the requests and control their status (pending, finished or failed), after having been launched and had their execution handed over to Celery. The web interface displays the pending requests, and when they are finished, it shows whether they were successfully completed or not. In the case of running applications, the user can also view the execution time as soon as they are finished. The web server together with its related services (i.e., Redis and Celery) are shown in Fig. 3.

```
1  ## Hosts
2  number_of_hosts: 4
3  cpus_per_host: 8
4  memory_per_host: 32768
5  hdd_disks_per_host: 1
6  hdd_disks_path_list: /scratch/hdd
7  ssd_disks_per_host: 1
8  ssd_disks_path_list: /scratch/ssd
9
10 ## Containers initial values
11 number_of_containers_per_host: 2
12 max_cpu_percentage_per_container: 200
13 min_cpu_percentage_per_container: 50
14 max_memory_per_container: 8192
15 min_memory_per_container: 512
16
17 ## Container network configuration
18 # interface for inter-host
       communication
19 iface: eth1
20 # modes: bridge, ptp, macvlan, ipvlan
21 mode: ipvlan
22 subnet: 10.22.0.0/16
```

Listing 2: Excerpt of the platform configuration file (*config.yml*)

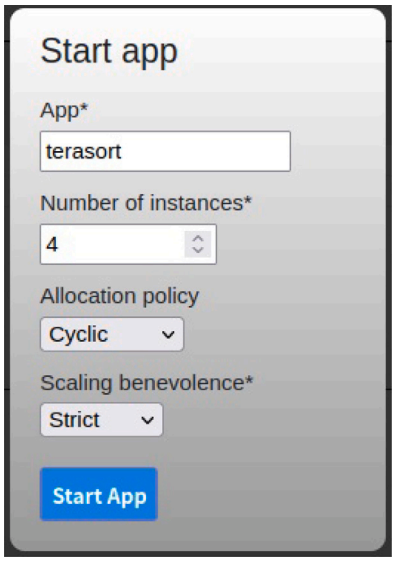


Fig. 6. Example of a web form to launch a YARN cluster.

5.3. Serverless YARN clusters for Big Data workloads

Our platform enables the user to deploy a serverless container cluster to run YARN applications. Typically, these applications are Big Data workloads, but any application running on top of YARN may actually be executed. The deployment of a YARN cluster is initiated through the web interface, where the first step is to define the application (or applications) to execute on such cluster. This definition includes information such as the application(s) startup script, the required Big Data framework, and the maximum and minimum resources (CPU and memory) in order to set the upper and lower limits that the cluster will

have later on. Frameworks currently offered include Hadoop MapReduce and Spark, although adding more YARN-compatible frameworks like Flink is straightforward. The first time an application is defined, a Singularity image is automatically built from the base container image, which was generated during the platform deployment as explained earlier (see Section 5.1). This new image is bundled with a full Hadoop distribution including YARN and HDFS. This image, in turn, may be used as a new base image to create another bundled with Spark or, in the future, with other YARN-compatible frameworks. Once the final container image is built, it is automatically distributed to the available hosts through Ansible so that they can run the corresponding containers when requested. After this step, a YARN cluster to execute the application can be launched by just clicking a single button that displays a web form like the one shown in Fig. 6. The first two fields specify the application name and the number of instances, which in turn sets the number of containers to be deployed as workers for the newly created YARN cluster. Therefore, each worker runs the YARN NodeManager process together with the corresponding DataNode to provide support for HDFS within the YARN cluster. A small portion of the resources defined for the application are used to create an additional container that will act as the master container within the YARN cluster, running the less intensive ResourceManager and NameNode processes for YARN and HDFS, respectively. The remaining resources are distributed evenly among the worker containers. It is worth mentioning that this is a temporary cluster that will be deleted once the application(s) are finished. This means that the HDFS data will be removed, so it is necessary to upload the input data after cluster startup, and retrieve any results to be preserved before cluster deletion.

5.3.1. Allocation policies

The allocation of the container instances of a YARN cluster on the available platform hosts can be controlled by three different policies: fill-up, cyclic and best-effort (see the third field of the form in Fig. 6). Note that these policies manage the allocation of newly created containers, as live migration of containers between hosts is not currently supported. The fill-up policy tries to allocate all the containers on the first host with enough free resources available. As soon as such host runs out of capacity, it moves to the next one and retries to allocate the remaining containers. This operation is repeated until no containers are pending. The cyclic policy seeks to allocate containers in a distributed manner, so that each time it allocates a container on a certain host, it moves to the next one even if that first host still has enough resources. After allocating a container on the last host, this policy returns to the first one to continue the distribution. Finally, the best-effort policy is similar to cyclic, but instead of going through the hosts in numerical order, it goes from lowest to highest disk contention, seeking to occupy the hosts with unallocated disks first. This policy allows for less contention on each disk, which improves performance for I/O-intensive applications such as many Big Data workloads.

To illustrate these policies, a simplified example is shown in Fig. 7. The base scenario is presented in Fig. 7(a), where there are four hosts, each with two disks and eight cores (i.e., maximum capacity of 800 CPU shares). In this base scenario, two initial containers are already deployed on the infrastructure, consuming 400 shares and both disks on the first host. If a YARN cluster is launched with six worker containers (and one master container) and 1300 CPU shares, the result would be the allocation of 200 shares for each worker container and 100 shares for the master. Considering the three policies, three scenarios are possible. Fig. 7(b) shows the result of applying the fill-up policy, where the first two hosts are currently at full CPU capacity, and their two disks are used by two containers each. The third host has only one container deployed with 100 shares allocated and using one disk, while the last host is completely idle. Fig. 7(c) shows the result of the cyclic policy. The first host is in the same situation as before (i.e., fully used), while the second and third hosts have 400 and 300 shares consumed, respectively, and their two disks are used by only one container each. The difference in the number of consumed shares is because the master container has a different resource allocation (100 shares less). In this scenario, the last host has one container deployed that consumes 200 shares and one disk. Finally, Fig. 7(d) shows the best-effort scenario. All the worker containers have been distributed along the second, third and fourth hosts, since the disks of the first host were already assigned to the two initial containers (coloured red). The last container (master) is then assigned to the first host as a direct result of the cyclical distribution. Therefore, the first host has a total of 500 shares consumed, one disk used by two containers and the other by only one, while the remaining hosts have 400 shares consumed and each disk used by only one container. Note that the disks may be marked as unavailable for new containers until older containers finish, if their contention is too high, in order to minimise I/O bottlenecks. Master containers are excluded from this process since they are not considered to be I/O intensive. As discussed in Section 5.1 and shown in Listing 2, the platform distinguishes between SSD and HDD disks. Therefore, SSD disks are prioritised when new containers are allocated and also allow a higher ratio of assigned containers per disk.

5.3.2. Serverless capabilities

Up to this point, the mechanisms to deploy virtual YARN clusters and their resources have been explained, but it is still necessary to describe how to provide serverless capabilities to each cluster. This is done through the BDWatchdog and Serverless Containers frameworks, which work in conjunction. As can be seen in Fig. 3, BDWatchdog reports resource usage metrics for each container and stores them in OpenTSDB. BDWatchdog is of great interest for this purpose because monitoring container consumption is not a trivial task and system-wide monitoring tools do not usually support it, as they do not provide the process-based monitoring plus the time series aggregation that BDWatchdog does. Serverless Containers is responsible for performing the actual resource scaling on each container. To do so, it assigns, on the one hand, maximum and minimum resource limits when creating each container, based on the limits defined for the application. On the other hand, it also assigns upper and lower thresholds to each container resource, which will be below the amount actually allocated. Serverless Containers constantly reads the usage metrics reported by BDWatchdog and compares them with the associated thresholds. If the consumption of a specific resource falls below the lower threshold, an underutilisation scenario is considered (i.e., usage is far from the allocated amount), so it will try to decrease its allocated resources and thus free them for other containers provided that its minimum allocation is maintained. If the consumption exceeds its upper threshold, a bottleneck scenario is considered (i.e., it is close to the allocated amount), so it will try to increase such allocation as long as it is possible considering its maximum limit and the available resources of the underlying host. Both thresholds are recalculated when a scaling operation occurs. These scaling operations are carried out by modifying the amount of usable resources for each container via cgroups on each host, since containers are groups of processes for the operating system. This is done “on-the fly”, that is, without restarting any container or running application.

Once a new YARN cluster is deployed, the scaling applied to it can be modulated. The last field of the form shown in Fig. 6 refers to how benevolent the scaling of the container resources should be, with three possible values: “lax”, “strict” and “medium”. This option involves changing the upper and lower thresholds of Serverless Containers mentioned earlier, tuning the gap between the thresholds and thus how narrow is the band of what is considered regular usage. With lax, the minimum capacity that the containers have at any time is very permissive, staying around half of the maximum capacity, and also issuing scaling up operations with enough time margin to avoid actually reaching a bottleneck, or at least shorten its duration. Strict intends to ensure that the allocated resources are as close as possible to the actual ones, only scaling up when usage is very close to a bottleneck, or scaling down as soon as some underutilisation is detected. Finally, medium is

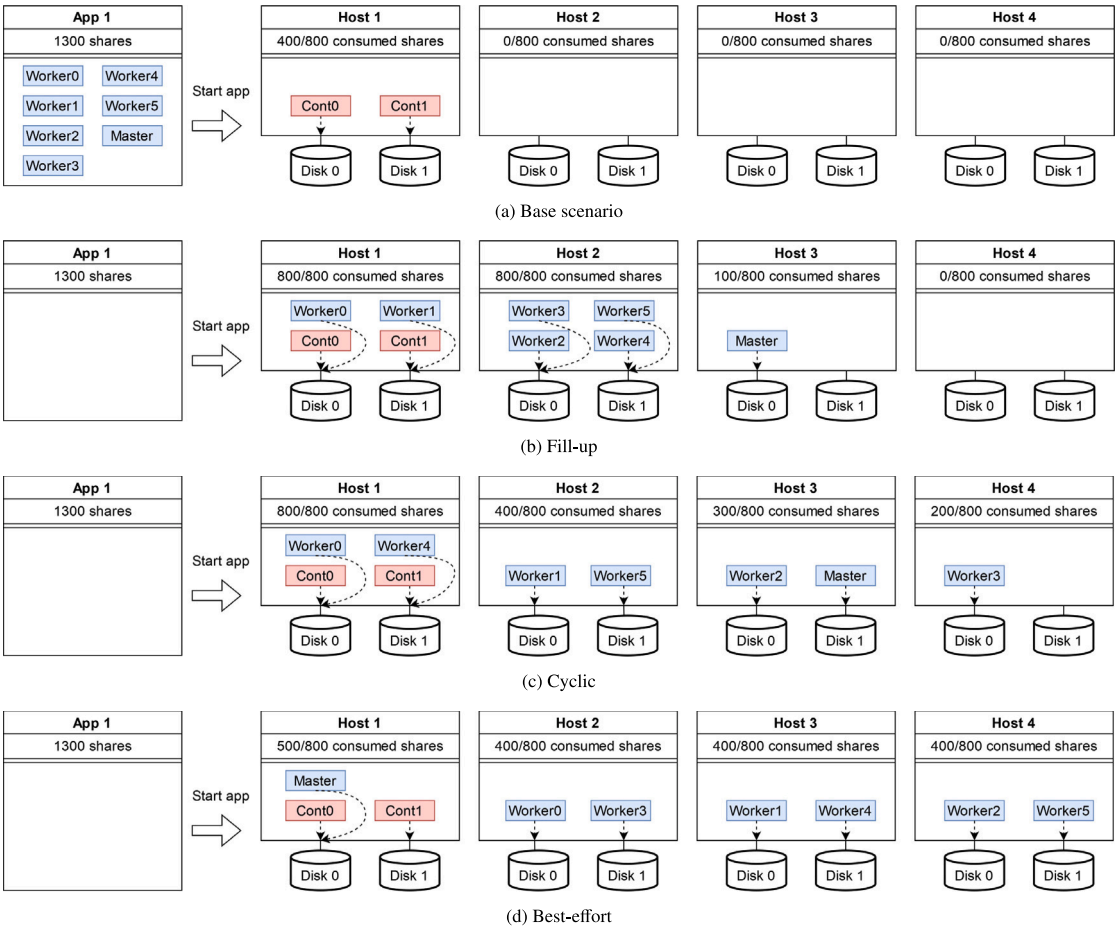


Fig. 7. Illustration of the different container allocation policies.

an intermediate point between the two previous settings. The idea is to modulate the serverless capabilities of the platform according to the user's needs, either prioritising the speed of scaling up operations with a lax policy at the potential cost of allocating more resources, or improve resource efficiency with a stricter policy.

5.3.3. YARN cluster and application startup

Once the form in Fig. 6 has been filled in to define the parameters of the YARN cluster on which the application(s) will be executed, and also to configure its serverless behaviour, the request to launch them can finally be submitted. This request is sent to the Celery queuing system, which in turn automatically creates the cluster according to the previously defined configuration and runs the application(s) through Ansible. Two playbooks are used for this purpose: (1) the containers are first deployed by *start_containers_playbook.yml*, as already discussed in Section 5.1; and (2) a new playbook is then required to further configure such containers in order to enable them to operate as a YARN cluster. To do so, this playbook gets the IP addresses of the deployed containers and properly configures the name resolution between them. Then, the HDFS and YARN configuration files, as well as those of Hadoop or Spark depending on the framework, are copied to each container. Note that these files are tailored for each YARN cluster. For example, taking into account the maximum allocable resources, the resources available to YARN for task allocation are properly defined. After this, the passwordless SSH connection from the master container of the YARN cluster to the workers is set up for ease of management. Finally, both HDFS and YARN services are started, all files associated with the application(s) to run are copied to the master container (e.g., a startup script), and the execution starts. Upon completion, the YARN cluster is deleted, and the runtime is displayed if the execution was successful, otherwise the corresponding error is shown. The results generated by the application(s) can be stored in a user-defined directory, which remains accessible after the cluster has been deleted.

6. Performance evaluation

The main goal of the experimental evaluation is to compare the performance of our platform when executing Big Data workloads separately, each on its own serverless YARN cluster, as opposed to running them on a standard YARN cluster. Note that with "standard YARN" we refer to the direct deployment of vanilla YARN over the same physical infrastructure, thus without serverless capabilities. For comparison, both execution time and resource efficiency are evaluated. In terms of runtime, the platform is expected to be faster when executing multiple serverless clusters concurrently, each running a Big Data workload, as it can dynamically reallocate underutilised resources between them. However, the potential overhead caused by the scaling can have a negative impact and must be carefully considered. Regarding resource efficiency, the serverless platform should be more efficient by continually adjusting the allocated resources to their actual usage.

The experimental configuration is presented in Section 6.1, including the environment used for deployment and the executed workloads, while Sections 6.2, 6.3 and 6.4 describe the experiments and analyse their results. Finally, Section 6.5 studies some overheads that can arise when using the platform.

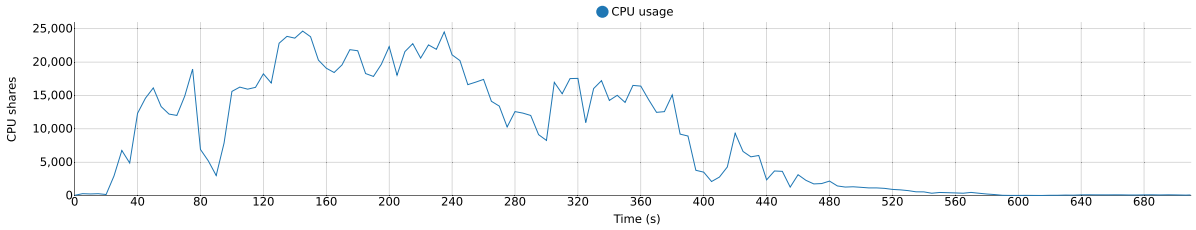


Fig. 8. CPU usage of TeraSort executed by standard YARN (without concurrency).

Table 1
Hardware and software specifications of the physical cluster nodes.

Hardware		
CPU Model		2 × Intel Xeon Silver 4216
#Cores per node		32
Memory		256 GiB DDR4 2933 MHz
Disk		1 × SSD 240 GiB SATA3
Network		Gigabit Ethernet (1 Gbps)
Software		
OS/Kernel		CentOS Linux 7.9.2009/5.4.233
Java		OpenJDK 1.8.0_322
Apptainer		1.1.6
Container OS		Ubuntu 20.04
Spark	Version	3.4.0
	Executors per node	16
	Executor heap size	13 GiB
	Executor cores	2
Hadoop	Version	3.3.5
HDFS	Block size	128 MiB
	Replication factor	2

6.1. Experimental configuration

The experiments have been carried out on a homogeneous physical cluster. Each node has two Intel Xeon CPUs with 16 cores each (i.e., 32 cores per node), 256 GiB of memory, one local SSD disk and a Gigabit Ethernet network (see Table 1). Our serverless platform and standard YARN were both deployed, one at a time, on this same infrastructure.

Table 1 also presents the software configuration, with details such as the OS and kernel of the cluster nodes. The versions of the Java runtime, Hadoop and Spark used for the experiments are also shown, as well as some relevant configuration parameters used for these Big Data frameworks, such as the number of cores for Spark executors or the HDFS block size. The container engine for the serverless YARN clusters was Apptainer 1.1.6 and the containers use Ubuntu 20.04 for the OS template.

On the one hand, the standard YARN cluster has been deployed using five nodes, one for Hadoop master services and four as workers. On the other hand, the serverless platform has been deployed on six nodes, one for the platform master itself, while the others act as hosts to deploy the serverless YARN clusters. One of the hosts is used for containers that run master services, and four for containers that deploy workers. Therefore, there are 128 physical cores available for processing tasks, 256 logical cores due to Intel's HyperThreading technology (i.e., 25,600 CPU shares), and 1 TiB of memory. Although master containers could also be deployed on the same host as worker containers, we avoid this scenario in order to have a more similar experimental environment compared to standard YARN. Following this idea, each serverless YARN cluster requests four worker containers assigned cyclically in order to have one worker per host, in the same way as standard YARN. It should also be mentioned that a strict scaling benevolence level was used with the serverless platform.

Regarding the Big Data workloads evaluated in the experiments, two popular and widely used benchmarks were chosen: TeraSort and PageRank. They were executed using Spark and Hadoop MapReduce, respectively. These workloads were selected because of their different CPU usage patterns, the resource on which the experiments are focused. On the one hand, TeraSort sorts 100-byte key–value tuples, where there is usually a first phase of high CPU consumption and a second phase in which it decreases significantly. On the other hand, PageRank is an iterative graph algorithm which ranks elements by counting the number and quality of the links to each one. Each iteration runs two MapReduce jobs, with each job having a high CPU usage peak. In order to keep runtimes within reasonable limits, an appropriate problem size was previously explored, eventually choosing 130 GiB of input data for TeraSort and 20 million pages with two iterations for PageRank.

The experiments in Sections 6.2, 6.3 and 6.4 have been divided according to both the degree of concurrency of the workloads executed and the amount of resources they can potentially use from the YARN clusters and the infrastructure overall. The results obtained are analysed mainly in terms of runtime and CPU usage efficiency. More specifically, for the runtime it is analysed both the relative time of each workload (i.e., the time perceived by the user until it finishes), and the total time (i.e., the time from the start of the first workload to the end of the second). The CPU usage efficiency is measured by calculating the ratio between used and allocated resources. The experiments in Section 6.5, however, are focused on measuring the YARN clusters startup time, their memory footprint and the impact produced by concurrent disk sharing.

6.2. Experiments without concurrency

The first group of experiments consists of executing both Big Data workloads in sequence and without overlap, using all available resources of the underlying infrastructure. For comparison purposes, the experiments are executed on standard YARN and using two serverless YARN clusters on our platform. This testbed represents a non-contention scenario to analyse the results of each workload separately.

Figs. 8 and 9 show the CPU usage plots of TeraSort and PageRank, respectively, when executed by standard YARN. Regarding their usage patterns, TeraSort shows high consumption for a long time, reaching up to 25,000 CPU shares, until it drops significantly at a certain point around second 380. PageRank presents lower overall CPU usage with four peaks exceeding 15,000 shares, which correspond to the two MapReduce jobs from each of the two iterations performed by this workload. Figs. 10 and 11 show the analogous plots for the serverless YARN clusters executed on our platform, where the orange line represents the amount of CPU allocated to each cluster over time, and the blue line is the actual CPU usage by the workloads. It can be observed that the usage patterns are similar to the previous ones obtained with standard YARN, but slightly "deformed", which is a direct consequence of the underlying transient bottlenecks and resource scaling operations performed by the serverless platform. Note that the CPU usage shown is automatically calculated by the platform as the sum of the usage of each container within the serverless cluster.

Table 2 shows the runtime and efficiency results for each scenario and workload. In terms of runtime, the serverless platform introduces a negligible overhead (<3%) due to the scaling, which can cause short bottlenecks when scaling upwards. More specifically, it can be observed that both TeraSort and PageRank just take a few seconds longer, slightly increasing their runtimes by only 1% and 3%, respectively. The lower

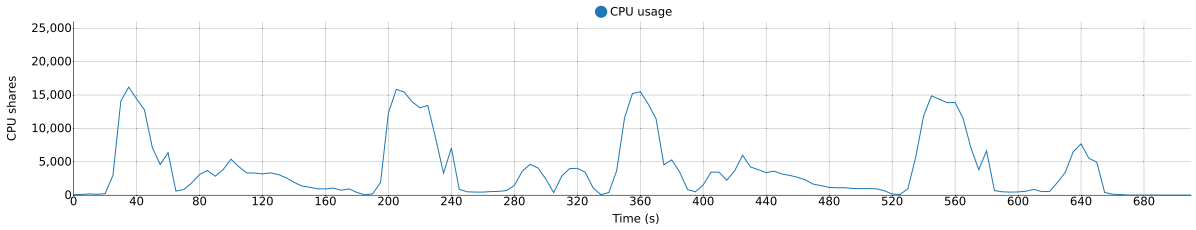


Fig. 9. CPU usage of PageRank executed by standard YARN (without concurrency).

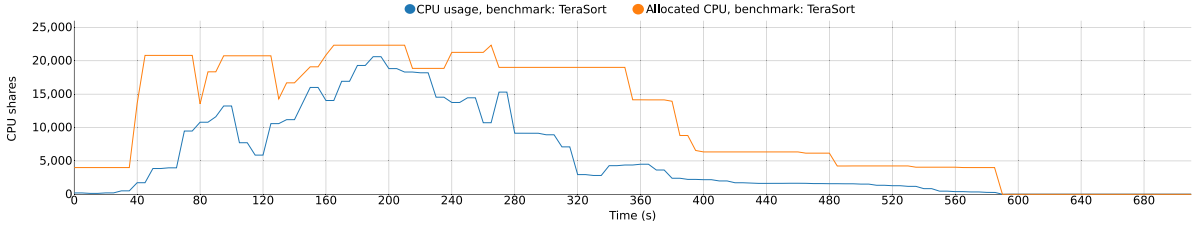


Fig. 10. CPU usage of TeraSort executed by the serverless platform (without concurrency).

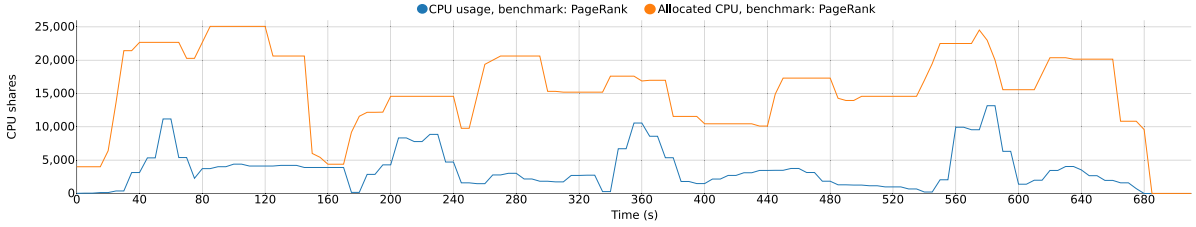


Fig. 11. CPU usage of PageRank executed by the serverless platform (without concurrency).

Table 2
Runtimes and resource efficiency of standard YARN and the serverless platform (without concurrency).

Scenario	Workload	Runtime (s)	Allocation integral (core-s)	Usage integral (core-s)	CPU usage ratio (%)
Standard YARN	TeraSort	583	149,248	62,415	42
	PageRank	662	169,472	27,989	17
	Total	1245	318,720	90,404	28
Serverless YARN	TeraSort	589 (+1%)	79,377	39,333	50 (+19%)
	PageRank	679 (+3%)	111,714	24,851	22 (+29%)
	Total	1268 (+2%)	191,091	64,184	34 (+21%)

variation of TeraSort is probably because it suffers less from platform overhead due to a fairly stable CPU usage. In any case, the goal of the platform is not to reduce runtimes for single workloads, so it is very positive that it barely introduces any overhead.

In terms of resource efficiency, it should be explained how the results were calculated (fourth and fifth columns in Table 2). To begin with, two areas are defined in each experiment: the area under the allocated resources and the area under the consumed resources. The first one refers to either all resources available for standard YARN (no line shown in the plots), or to the resources allocated to each serverless YARN cluster at any given time (i.e., the area under the orange lines). The second area is self-explanatory, corresponding to the areas under the blue lines for the plots with standard YARN and the serverless YARN clusters. Their integrals are calculated to obtain the numerical value of the corresponding areas measured in "CPU core-seconds" (abbreviated as "core-s" in the table). Finally, the CPU usage ratio (last column) is the quotient between the resources consumed and those allocated.

The efficiency results in Table 2 show that standard YARN obtains usage ratios of 42% and 17% for TeraSort and PageRank, respectively, and 28% for the whole execution. PageRank obtains a significantly lower result compared to TeraSort, because it presents only brief high CPU peaks. On the other hand, our platform increases the usage ratios by 19% and 29% for TeraSort and PageRank, respectively, and 21% for the whole execution. The better the scaling is performed, the better the ratio obtained, since the resources allocated are closer to those actually used. It is interesting to note the difference in resource efficiency between both workloads when executed on the serverless platform. As mentioned above in the runtime analysis, TeraSort has a more stable consumption, thus being easier to scale by the platform, whereas the CPU peaks of PageRank become much more difficult to manage, as the platform tends to overestimate and keep allocated resources that are no longer used just a few seconds later.

6.3. Experiments with concurrency at 100% oversubscription

The second group of experiments executes both workloads simultaneously, each trying again to use all available resources. These experiments aim to create a contention scenario with 100% resource oversubscription. This means that, even though the physical resources remain the same as before, both standard YARN and the platform advertise twice as many resources to the workloads (i.e., they are deployed with potentially full resources). Considering this concurrency, standard YARN is configured using the Fair scheduler with a single queue in order to execute both workloads trying to maximise resource

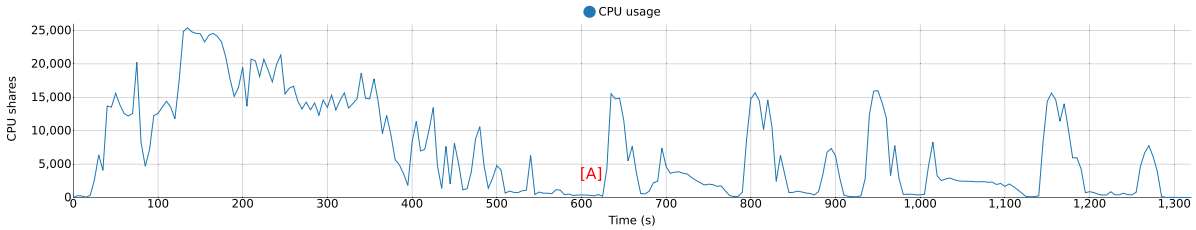


Fig. 12. CPU usage of both workloads at 100% oversubscription executed concurrently by standard YARN (1-minute delay).

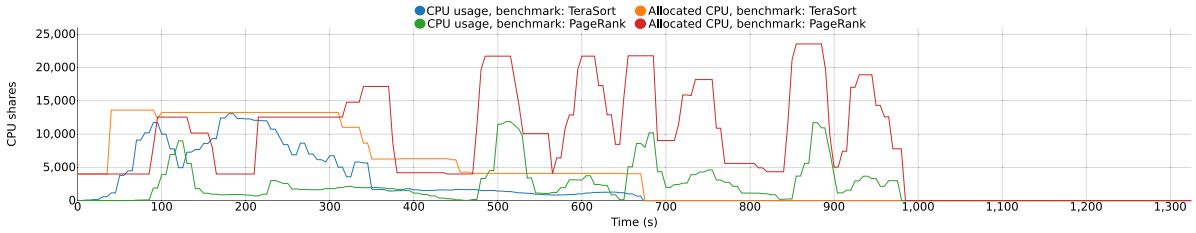


Fig. 13. CPU usage of both workloads at 100% oversubscription executed concurrently by the serverless platform with high contention (1-minute delay).

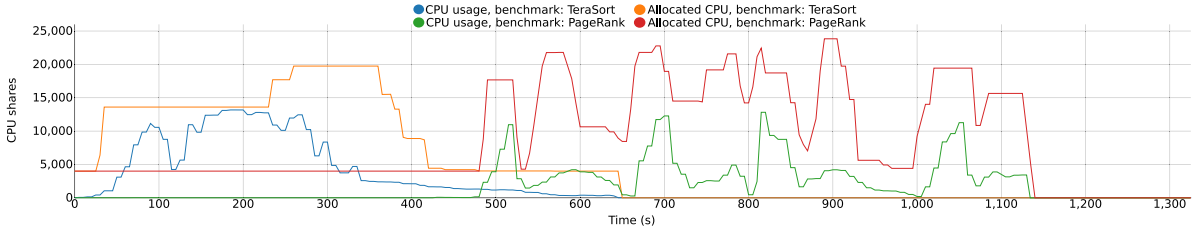


Fig. 14. CPU usage of both workloads at 100% oversubscription executed concurrently by the serverless platform with low contention (7-minute delay).

efficiency (see Section 2.4). The workloads are submitted to the YARN scheduler with a one-minute delay between them. Regarding the platform, two serverless YARN clusters will now coexist, with the same resource configuration as in Section 6.2, that is, each serverless cluster requesting one master and four worker containers, all distributed across the available hosts, and with each worker potentially using all host resources. This is done to accommodate each workload in isolation and, similar to the standard YARN scenario, trying to improve resource efficiency. However, this in turn means that two containers from different serverless YARN clusters will now coexist on each host. As mentioned in Section 6.1, master containers do not coexist with workers in order to create an environment more similar to standard YARN. Taking into account the two serverless YARN clusters and the concurrent execution, two scenarios are analysed for our platform: (1) a high-contention scenario where there is a one-minute delay between the execution of both workloads (the same delay as with standard YARN), causing an overlap that lasts for a long time; and (2) a low-contention scenario, where the delay is around seven minutes, resulting in a short overlap. It is expected that the 100% oversubscription may lead to differences depending on the contention, considering that in the high-contention scenario resources will shift between both serverless clusters but will eventually be exhausted, whereas in the low-contention case resources will be able to move from one cluster to another according to demand.

Fig. 12 shows the CPU usage when executing both workloads concurrently with standard YARN, launching TeraSort first and then PageRank with a one-minute delay. In this scenario, it is expected that TeraSort will request all available resources, forcing PageRank to wait until TeraSort finishes. In this case, the YARN scheduler is simply unable to do anything. This can be observed in the figure, since the representative CPU usage peaks of PageRank are not appreciated until the label "[A]", which also signals the end of the TeraSort execution at around second 615. This is the main issue of standard YARN, even with the Fair scheduler, as it does not balance the available resources directly. Instead, it balances the processing tasks that have the resources allocated, regardless of whether they use them or not (see Section 2.4). This means that if an application requests all resources, several tasks will be created with those allocated resources, but YARN is unable to redistribute them when they are underutilised. So, resources are reserved for those processing tasks until they are finished.

Fig. 13 shows the results using the serverless YARN clusters on the high-contention scenario, where the orange and red lines represent the CPU allocated by our platform to TeraSort and PageRank, respectively, and the blue and green ones represent their actual usage. In this case, both workloads can actually be executed concurrently as they are deployed on different and isolated YARN clusters, which is a major advantage over standard YARN. However, as the resources are eventually shared transparently between both clusters, each one can use fewer resources, as expected. Nevertheless, this overlap only occurs during a part of the execution, as the first iteration of PageRank takes longer than before, which causes the second iteration to run when TeraSort has already finished. This behaviour is related to the network, as both workloads are now using it concurrently and intensively. During the first job of PageRank there is a Map and a Reduce phase, where the latter exchanges a lot of data between workers. At the same time, TeraSort is writing its output data to HDFS, which also involves network activity. As a result, there is a high network load due to the HDFS data replication. Therefore, the concurrent execution of the workloads is mainly limited by the underlying network speed.

Fig. 14 presents the results when executing the workloads on our platform with a delay of seven minutes (i.e., the low-contention scenario explained earlier). At first glance, it can be seen that each workload individually now takes less time than before because they do not have to share resources intensively. However, it is important to note a limitation of the experiments carried out with the platform, which is

Table 3
Runtimes and resource efficiency of standard YARN and the serverless platform (workloads executed concurrently at 100% oversubscription).

Scenario	Workload	Runtime (s)	Allocation integral (core-s)	Usage integral (core-s)	CPU usage ratio (%)
Standard YARN	TeraSort	615	157,440	64,306	41
	PageRank	1219	171,776	28,068	16
	Total	1286	329,216	92,374	28
Serverless YARN (high contention)	TeraSort	673 (+9%)	58,180	30,779	53 (+29%)
	PageRank	922 (−24%)	112,249	26,834	24 (+50%)
	Total	982 (−24%)	170,429	57,613	34 (+21%)
Serverless YARN (low contention)	TeraSort	646 (+5%)	69,726	32,772	47 (+15%)
	PageRank	716 (−41%)	116,376	26,077	22 (+38%)
	Total	1136 (−12%)	186,103	58,849	32 (+14%)

particularly noticeable in this case: TeraSort is running alone at the very beginning, but its maximum resources are actually limited. The reason for this behaviour is that there is a minimum capacity already reserved for PageRank, even though it has not been started yet. This is because each serverless YARN cluster actually starts earlier than shown to precisely execute each workload with the intended delay, so that a minimum reserved capacity is maintained from that moment on. Future work to address this limitation is to detect situations of near-zero usage in order to reallocate resources from one serverless cluster with usage well below the minimum to another cluster.

Table 3 shows the runtime and efficiency results for each scenario. For standard YARN, the total runtime is similar to the previous case without concurrency (see Table 2), considering that no concurrent execution is actually achieved when using the Fair scheduler. Regarding relative times, TeraSort obtains similar performance to the case without concurrency, but PageRank increases it significantly by 84%. Even though PageRank is submitted just one minute after TeraSort, it cannot actually begin until TeraSort finishes, so its relative time (1219 s) is close to the total (1286 s). For serverless YARN clusters in the high-contention scenario, TeraSort runtime increases by only 9% when compared to standard YARN, while both PageRank and the total time decrease significantly by 24%. Therefore, although users may or may not see their applications running faster (relative time), they benefit globally from potentially faster times (total time) and, very importantly, from avoiding queue waiting times. In other words, this scenario provides a fairer experience for all users, without prioritising some workloads over others because of their submission order. Compared to the serverless scenario without concurrency (see Table 2), TeraSort and PageRank runtimes increase by 14% and 36%, respectively, but the total time decreases by 23%. Thus, the increase in relative times is balanced by the overall decrease. For the low-contention scenario, the relative times obtained are lower than for the high-contention counterpart, which makes sense given the lower resource contention, but the downside is a higher total time. PageRank presents a significant improvement (22%), explained by the fact that it was the workload most affected by the network bottleneck. This experiment shows that the platform in a low-contention scenario is able to execute both workloads concurrently without significantly impacting their performance compared to their individual execution (see serverless runtimes in Table 2), while achieving a lower overall time than standard YARN (−12%), even with a seven-minute delay.

Regarding resource efficiency, the results for standard YARN are very similar to the case without concurrency, as expected (see Tables 2 and 3). The high-contention scenario on the serverless platform obtains similar, though slightly higher, usage ratios than without concurrency: 53%, 24% and 34% for TeraSort, PageRank and total, respectively. This translates into an improvement of 29%, 50% and 21% over standard YARN. Finally, the low-contention scenario is slightly worse than without concurrency, especially for TeraSort due to the aforementioned minimum capacity associated with the PageRank execution. Although this scenario obtains slightly lower efficiencies than its high-contention counterpart, it still benefits from serverless execution, providing higher ratios than standard YARN.

6.4. Experiments with concurrency at 50% oversubscription

The third group of experiments again execute both workloads concurrently, but now limiting the amount of resources each workload can use to 75%, implying a resource oversubscription of 50%. For standard YARN, this is done at submission time by limiting the resources that the workloads request to the scheduler. For serverless YARN clusters, this limit is instead applied to the maximum amount of resources each cluster can potentially have allocated, setting it to 75% of the underlying host resources, which also directly limits the maximum amount that a container can use. The same high- and low-contention scenarios are explored for the serverless platform. The goal of these experiments is to analyse the behaviour of standard YARN and the serverless platform in a scenario with less demanding workloads, and therefore, less resource contention overall. This situation makes it easier to share resources between workloads for standard YARN, which should reduce the differences with serverless YARN to some extent.

Fig. 15 shows the CPU usage for standard YARN. In this case, TeraSort starts and reserves 75% of the available resources, so that PageRank is now able to use the remaining 25% while running concurrently. Three of the four characteristic CPU peaks in PageRank can be clearly seen in the graph, as the CPU usage for the first peak “merges” with the TeraSort execution. What happens is that during the first job of PageRank, it only has a small portion of the available resources. This situation, together with the network limitation mentioned earlier, causes PageRank to run slowly. Upon completion of TeraSort, PageRank can now run at full speed. Figs. 16 and 17 present the high- and low-contention plots for our platform, respectively. The graphs show CPU usage patterns similar to using the platform at 100% oversubscription, with the most noticeable difference being that each workload now has a lower limit of allocated resources. This similarity is expected since the main bottleneck for concurrency is the network, not the CPU.

Table 4 shows the runtime and efficiency results. On the one hand, TeraSort executed by standard YARN obtains a slightly higher time than the previous two experiments, but is not severely affected by concurrency. On the other hand, PageRank runtime increases by 64% with respect to YARN without concurrency (see Table 2), but it is reduced by 11% compared to the 100% oversubscription scenario (see Table 3), as it no longer has to wait for TeraSort to finish in order to start. The total time is slightly reduced by 8% compared to YARN without concurrency. Overall, this experiment is the best for standard YARN, but the improvement achieved is not particularly significant. In the high-contention scenario with the platform and compared to standard YARN, TeraSort runtime increases by 10% and both PageRank and the total time decrease by 8%. The low-contention scenario obtains lower relative times and higher total time than the high-contention counterpart, as was the case in the analogous serverless scenarios shown in Section 6.3. Particularly significant is the difference in the reduction of the relative time of PageRank in the two contention scenarios (8% vs. 24%), mainly due to the reduced overlap of the workloads. The results achieved by the serverless platform in both contention scenarios are closer to those obtained with standard YARN

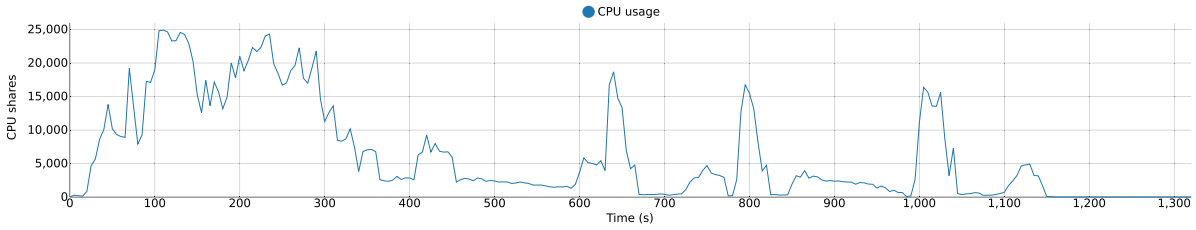


Fig. 15. CPU usage of both workloads at 50% oversubscription executed concurrently by standard YARN (1-minute delay).

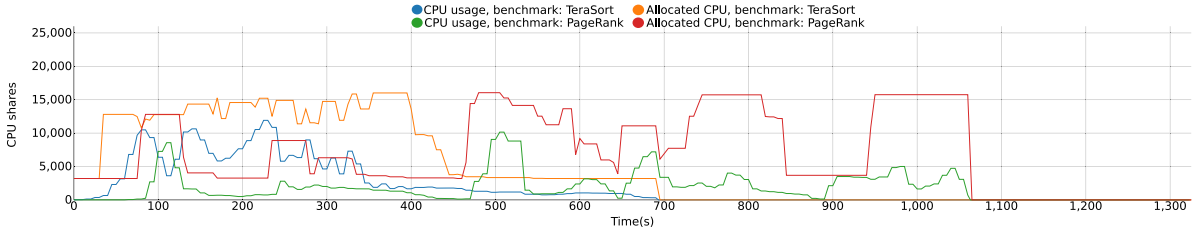


Fig. 16. CPU usage of both workloads at 50% oversubscription executed concurrently by the serverless platform with high contention (1-minute delay).

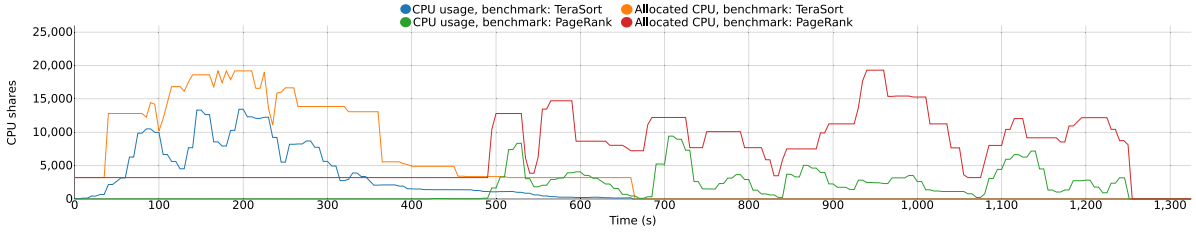


Fig. 17. CPU usage of both workloads at 50% oversubscription executed concurrently by the serverless platform with low contention (7-minute delay).

Table 4
Runtimes and resource efficiency of standard YARN and the serverless platform (workloads executed concurrently at 50% oversubscription).

Scenario	Workload	Runtime (s)	Allocation integral (core-s)	Usage integral (core-s)	CPU usage ratio (%)
Standard YARN	TeraSort	630	–	–	–
	PageRank	1088	–	–	–
	Total	1151	294,656	82,183	28
Serverless YARN (high contention)	TeraSort	694 (+10%)	64,818	27,591	43
	PageRank	1003 (−8%)	93,704	24,113	26
	Total	1063 (−8%)	158,523	51,704	33 (+18%)
Serverless YARN (low contention)	TeraSort	664 (+5%)	62,153	28,589	46
	PageRank	831 (−24%)	94,434	22,501	24
	Total	1251 (+9%)	156,588	51,090	33 (+18%)

in these experiments, as the network seems to be the main bottleneck, rather than the CPU. The main difference is that our platform is fairer because it does not prioritise the TeraSort workload just for being the first to be submitted.

Regarding resource efficiency, only the total value can be provided for standard YARN, as there is no clear distinction between the areas of the different workloads due to the concurrent execution of TeraSort and the first job of PageRank. The total usage ratio only reached 28%, the same value as in the case of 100% oversubscription. Therefore, standard YARN is unable to use the resources more efficiently despite having a part with actual concurrency. The results for the high-contention scenario show that, compared to the previous experiments, the serverless platform obtains the lowest efficiency for TeraSort (43%) and the highest for PageRank (26%), with the total usage ratio being very similar (33%) to the case of 100% oversubscription (see Table 3). This scenario obtains an improvement of 18% over standard YARN. The low-contention counterpart achieves similar results, with the same improvement over standard YARN. This similar efficiency between the high- and low-contention scenarios reinforces the consistency of the platform despite the significant differences in runtimes. Overall, the results obtained by our platform are still better than its counterpart with standard YARN. Compared to the 100% oversubscription experiments, the results of the platform are very similar, which is expected because of the small change in the usage pattern in the end. The largest difference is observed in the high-contention scenario, where TeraSort efficiency decreases from 53% to 43%. This is due to the fact that the workload is more unstable with fewer resources, showing more CPU drops and spikes.

6.5. Platform overheads

In order to analyse the overheads associated with using the serverless platform, additional experiments were performed to evaluate the startup time of the YARN clusters and their memory footprint, as well as the impact of disk sharing.

Table 5
Serverless YARN clusters startup overhead.

Workers	1	2	4	8	16
Startup time (s)	42	43 (+2%)	48 (+14%)	55 (+31%)	71 (+69%)

6.5.1. Startup overhead

The startup time of a serverless YARN cluster was measured from the moment the user presses the button on the web interface to start it, until it is finally deployed and ready to run the desired application(s). Several measurements were taken by varying the number of worker containers to analyse how the times evolve as the cluster size increases. Table 5 presents the results obtained, including the relative increase based on the time of one worker. It shows relatively low times considering that, once the cluster is deployed, Big Data workloads are typically long-running applications. This overhead is mainly due to the deployment process and not to the container technology, and can therefore be further optimised in the future.

6.5.2. Memory footprint

The memory overhead caused by the concurrent execution of two applications on different YARN clusters is now analysed, since each cluster deployed on the platform requires its own YARN/HDFS processes (see Section 2.4). To do so, two TeraSort workloads processing 80 GiB of data were run concurrently using both standard YARN and the serverless platform. The deployment was done as described in Section 6.1, using four workers for standard YARN and four worker containers for each of the two corresponding serverless YARN clusters. Additionally, the HDFS replication factor was set to 1 to reduce network contention and focus on memory consumption. Regarding concurrency, a 100% oversubscription and a one-minute delay between workloads was used.

Table 6 shows the average and aggregate memory consumption over time of YARN and HDFS processes in both scenarios. As can be seen, the average values obtained in our platform are higher, with a maximum increase of 69% over standard YARN, which is reasonable since twice as many processes can be running simultaneously. However, this difference is significantly reduced in terms of aggregate consumption, reaching up to 40%. In other words, although the number of YARN and HDFS services started is proportional to the number of serverless clusters deployed, memory usage does not necessarily scale linearly because it depends on the concurrency of the executions. There will be periods of higher memory consumption when there are clusters deployed simultaneously, and periods of lower consumption since clusters will be deleted as they finish (i.e., their services are stopped). This is noticeable in the average usage, as it does not reach twice the consumption despite running twice the number of processes. This is even more significant in the aggregate usage, reinforced by the fact that concurrency is fully exploited with our platform by decreasing runtimes, which also reduces the time that YARN/HDFS services are active. This even leads to an overhead as low as 15% in one experiment. Nevertheless, these processes do not consume excessive memory on average, as shown in Table 6, where master and worker services do not reach 1 and 3 GiB, respectively, on standard YARN. So, it is feasible to run multiple clusters concurrently without noticing a severe impact.

6.5.3. Impact of disk sharing

The last set of experiments analyses the impact of disk sharing on the execution of concurrent I/O-intensive workloads using the serverless platform. To do so, two concurrent TeraSort were executed in a high-contention scenario (one-minute delay) at 100% oversubscription to compare their corresponding runtimes with the execution of a single TeraSort. Additionally, the total time was compared to the non-concurrent execution of two TeraSort. These experiments were performed by varying the data size, processing 80 and 160 GiB of input data, and using an HDFS replication factor of 1 to reduce network contention and focus on disk load.

Table 7 shows the results obtained. As can be seen, the concurrent execution increases the relative time of both workloads compared to a single execution, which is expected since they have to share resources, especially disk. It is also noticeable how it affects the second TeraSort to a greater extent, increasing its runtime by 80% and 57% for 80 and 160 GiB, respectively. Nevertheless, neither workload has to wait for the other, which would result in a runtime increase of 100%, twice that of a single TeraSort. Regarding the whole execution, the total time when processing 80 GiB concurrently (490 s) is only slightly higher (3%) than the non-concurrent execution of both TeraSort. Therefore, disk sharing had a negative impact, but barely noticeable. For 160 GiB, however, the total time is reduced by 15%, so disk sharing leads to an improvement. So, on the one hand, disk contention increases the relative time of each workload, as expected, although overall it is still fairer to them than a non-concurrent execution. On the other hand, disk sharing may not have that much of an impact on total time, and it can even lead to benefits in some cases. Anyway, I/O efficiency is a crucial aspect to consider with Big Data. Note that in these experiments, there was only one disk available on each physical node (see Table 1), but if there were more, the platform could distribute the available disks among the containers as explained in Section 5.3.1, which should further reduce the impact of disk sharing.

7. Conclusions

Serverless computing is a very interesting paradigm today for the execution of applications in a simple and profitable way, both from a monetary point of view for the user and from a resource efficiency point of view for the infrastructure provider. Within serverless, FaaS is the most widely extended approach, allowing the execution of simple functions directly, without the need to specify any type of configuration for the underlying server on which they are executed. However, FaaS has not been designed to run more complex applications that may have an intricate interaction between many different components.

In this paper, we have presented a platform that is capable of deploying serverless YARN clusters oriented to the execution of Big Data workloads. These clusters are made up of containers whose resources are dynamically scaled in real time according to their actual usage. When an application is submitted, the platform automatically deploys a new YARN cluster with serverless capabilities, placing its containers on the underlying hardware infrastructure according to different supported policies, so that multiple YARN clusters can coexist at the same time. In addition, the deployment of both the platform and the container clusters is automated through IaC tools, with a web interface that eases the management of the platform and allows it to be used by a wider range of users. By relying on the popular YARN cluster manager for the serverless clusters, the platform can be easily extended to support multiple Big Data processing frameworks, currently allowing to execute both Hadoop MapReduce and Spark workloads.

A comprehensive experimental evaluation has been carried out using Big Data workloads and analysing several scenarios to compare the runtime and resource usage efficiency of our platform with a standard YARN deployment. These results have revealed the following: (1) the overhead caused by the scaling performed by the platform is not significant (<3%); (2) the platform obtains better results in the concurrent execution of workloads than standard YARN, providing up to a 24% reduction in total time and a 50% improvement in total CPU usage ratio; and (3) workloads with bursty usage peaks become much more difficult to manage than more stable loads. The overhead caused in other aspects has also been evaluated, such as the startup time of serverless YARN clusters or their memory footprint. The source code of the platform presented in this paper is available at https://github.com/UDC-GAC/ServerlessYARN.

Table 6
Memory usage of YARN/HDFS processes of standard YARN and the serverless platform (workloads executed concurrently at 100% oversubscription).

Scenario	Total time (s)	NameNode		ResourceManager		DataNode		NodeManager	
		Avg usage (GiB)	Agg usage (GiB-s)	Avg usage (GiB)	Agg usage (GiB-s)	Avg usage (GiB)	Agg usage (GiB-s)	Avg usage (GiB)	Agg usage (GiB-s)
Standard YARN	587	0.52	305.13	0.55	324.58	2.26	1,320.65	2.43	1,419.44
Serverless YARN (high contention)	487 (−17%)	0.88 (+69%)	428.18 (+40%)	0.88 (+60%)	428.75 (+32%)	3.51 (+55%)	1,707.91 (+29%)	3.36 (+38%)	1,632.82 (+15%)

Table 7
Disk overhead of the serverless platform with high contention (1-minute delay and workloads executed concurrently at 100% oversubscription).

Data size	Non-concurrent execution (s)		Concurrent execution (s)		
	Each single workload	Total	First workload	Second workload	Total
80 GiB	237	474	309 (+30%)	427 (+80%)	490 (+3%)
160 GiB	436	872	542 (+24%)	686 (+57%)	739 (−15%)

Future work includes adapting the platform to other container engines (e.g., Docker) and extending it to other YARN-compatible Big Data frameworks (e.g., Apache Flink). Another interesting line would be to support horizontal scaling to increase or decrease the number of container instances within the serverless clusters at runtime.

CRediT authorship contribution statement

Óscar Castellanos-Rodríguez: Conceptualization, Investigation, Software, Writing – original draft. **Roberto R. Expósito:** Conceptualization, Resources, Supervision, Writing – review & editing. **Jonatan Enes:** Resources, Supervision, Writing – review & editing. **Guillermo L. Taboada:** Funding acquisition, Supervision. **Juan Touriño:** Funding acquisition, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

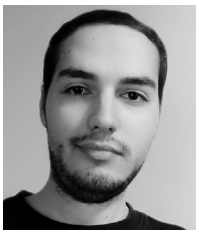
Acknowledgements

This work was supported by grants PID2019-104184RB-I00, PDC2021-121309-I00, TED2021-129177B-I00 and PID2022-136435NB-I00, funded by the Ministry of Science and Innovation of Spain, MCIN/AEI/10.13039/501100011033 (PDC2021 and TED2021 also funded by "NextGenerationEU"/PRTR, and PID2022 by "ERDF A way of making Europe", EU). It was also supported by Xunta de Galicia, Spain (predoctoral fellowship ED481A 2022/067). Funding for open access charge: Universidade da Coruña/CISUG.

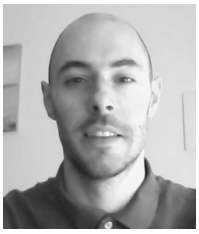
References

[1] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski, The rise of serverless computing, Commun. ACM 62 (12) (2019) 44–54.
[2] T. Lynn, P. Rosati, A. Lejeune, V. Emeakaroha, A preliminary review of enterprise serverless cloud computing (Function-as-a-Service) platforms, in: 2017 IEEE International Conference on Cloud Computing Technology and Science, CloudCom'17, Hong Kong, China, 2017, pp. 162–169.
[3] I. Müller, R.F. Bruno, A. Klimovic, G. Alonso, J. Wilkes, E. Sedlar, Serverless clusters: The missing piece for interactive batch applications? in: 10th Workshop on Systems for Post-Moore Architectures, SPMA'20, Heraklion, Greece, 2020, pp. 1–3.
[4] S. Dash, S. Shakyawar, M. Sharma, S. Kaushik, Big Data in healthcare: Management, analysis and future prospects, J. Big Data 6 (2019) 1–25, Article no. 54.
[5] E. Aktas, Y. Meng, An exploration of Big Data practices in retail sector, Logistics 1 (2) (2017) 1–12.
[6] The Apache Software Foundation, Apache Hadoop, 2024, hadoop.apache.org, [Visited 2024].
[7] M. Zaharia, et al., Apache Spark: A unified engine for Big Data processing, Commun. ACM 59 (11) (2016) 56–65.
[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache Flink: Stream and batch processing in a single engine, Bull. IEEE Comput. Soc. Tech. Comm. Data Eng. 36 (4) (2015) 28–38.
[9] K. Morris, Infrastructure as Code, O'Reilly Media, 2020.
[10] V.K. Vavilapalli, et al., Apache Hadoop YARN: Yet Another Resource Negotiator, in: 4th Annual Symposium on Cloud Computing, SoCC'13, Santa Clara, CA, USA, 2013, pp. 5:1–5:16.
[11] The Linux Kernel Organization, Control groups, 2024, www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html. [Visited 2024].
[12] G.M. Kurtzer, V. Sochat, M.W. Bauer, Singularity: Scientific containers for mobility of compute, PLoS ONE 12 (5) (2017) 1–20, e0177459.
[13] D. Merkel, Docker: Lightweight Linux containers for consistent development and deployment, Linux J. 239 (2) (2014) 76–91.
[14] AWS Fargate service. aws.amazon.com/fargate. [Visited 2024].
[15] Amazon Web Services (AWS). aws.amazon.com. [Visited 2024].
[16] AWS Elastic Container Service (ECS). aws.amazon.com/ecs. [Visited 2024].
[17] AWS Elastic Kubernetes Service (EKS), aws.amazon.com/eks, [Visited 2024].
[18] AWS Lambda. aws.amazon.com/lambda. [Visited 2024].
[19] Microsoft's Azure Functions, azure.microsoft.com/en-us/products/functions. [Visited 2024].
[20] Google Cloud Functions, cloud.google.com/functions. [Visited 2024].
[21] L. Hochstein, R. Moser, Ansible: Up and Running, O'Reilly Media, 2017.
[22] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[23] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop Distributed File System, in: 26th IEEE Symposium on Mass Storage Systems and Technologies, MSST'2010, Incline Village, NV, USA, 2010, pp. 1–10.
[24] K. Djemame, M. Parker, D. Datsev, Open-source serverless architectures: An evaluation of Apache OpenWhisk, in: 13th IEEE/ACM International Conference on Utility and Cloud Computing, UCC'13, Leicester, UK, 2020, pp. 329–335.
[25] 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 Gener. Comput. Syst. 110 (2020) 502–514.
[26] B. Balis, HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows, Future Gener. Comput. Syst. 55 (2016) 147–162.
[27] A. John, K. Ausmees, K. Muenzen, C. Kuhn, A. Tan, SWEEP: Accelerating scientific research through scalable serverless workflows, in: 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC'19 Companion Auckland, New Zealand, 2019, pp. 43–50.
[28] Y. Kim, J. Lin, Serverless data analytics with Flint, in: 11th IEEE International Conference on Cloud Computing, CLOUD'18, San Francisco, CA, USA, 2018, pp. 451–455.
[29] V. Giménez-Alventosa, G. Moltó, M. Caballer, A framework and a performance assessment for serverless MapReduce on AWS Lambda, Future Gener. Comput. Syst. 97 (2019) 259–274.
[30] A. Pérez, G. Moltó, M. Caballer, A. Calatrava, Serverless computing for container-based architectures, Future Gener. Comput. Syst. 83 (2018) 50–59.
[31] E. Oakes, et al., SOCK: Rapid task provisioning with serverless-optimized containers, in: 2018 USENIX Annual Technical Conference, USENIX ATC'18, Boston, MA, USA, 2018, pp. 57–70.

[32] J.R. Gunasekaran, P. Thinakaran, N.C. Nachiappan, M.T. Kandemir, C.R. Das, Fifer: Tackling resource underutilization in the serverless era, in: 21st International Middleware Conference, Middleware'20, Delft, Netherlands, 2020, pp. 280–295.
[33] 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.
[34] F. Soppelsa, C. Kaewkasi, Native Docker Clustering with Swarm, Packt Publishing Ltd, 2016.
[35] E.A. Brewer, Kubernetes and the path to cloud native, in: 6th ACM Symposium on Cloud Computing, SoCC'15, Kohala Coast, HI, USA, 2015, p. 167.
[36] L.M. Ruíz, P.P. Pueyo, J. Mateo-Fornés, J.V. Mayoral, F.S. Tehàs, Autoscaling pods on an on-premise Kubernetes infrastructure QoS-aware, IEEE Access 10 (2022) 33083–33094.
[37] A. Floratou, A. Agrawal, B. Graham, S. Rao, K. Ramasamy, Dhalion: Self-regulating stream processing in Heron, Proc. VLDB Endow. 10 (12) (2017) 1825–1836.
[38] S. Kulkarni, et al., Twitter Heron: Stream processing at scale, in: 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD'15, Melbourne, Australia, 2015, pp. 239–250.
[39] G. Rattihalli, M. Govindaraju, H. Lu, D. Tiwari, Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes, in: 12th IEEE International Conference on Cloud Computing, CLOUD'19, Milan, Italy, 2019, pp. 33–40.
[40] J. Enes, R.R. Expósito, J. Touriño, Real-time resource scaling platform for Big Data workloads on serverless environments, Future Gener. Comput. Syst. 105 (2020) 361–379.
[41] N. Naik, Docker container-based Big Data processing system in multiple clouds for everyone, in: 2017 IEEE International Systems Engineering Symposium, ISSE'17, Vienna, Austria, 2017, pp. 1–7.
[42] V. Srinivasakumar, M. Vanamoorthy, S. Sairaj, S. Ganesh, An alternative C++-based HPC system for Hadoop MapReduce, Open Comput. Sci. 12 (1) (2022) 238–247.
[43] C. Sauvanaud, A. Dholakia, J. Guitart, C. Kim, P. Mayes, Big Data deployment in containerized infrastructures through the interconnection of network namespaces, Softw. Pract. Exp. 50 (7) (2020) 1087–1113.
[44] X. Ding, Y. Liu, D. Qian, JellyFish: Online performance tuning with adaptive configuration and elastic container in Hadoop YARN, in: 21st IEEE International Conference on Parallel and Distributed Systems, ICPADS'15, Melbourne, Australia, 2015, pp. 831–836.
[45] M. Hashimoto, Vagrant: Up and Running, O'Reilly Media, 2013.
[46] J. Enes, R.R. Expósito, J. Touriño, BDWatchdog: Real-time monitoring and profiling of Big Data applications and frameworks, Future Gener. Comput. Syst. 87 (2018) 420–437.
[47] L. George, HBase: The Definitive Guide, O'Reilly Media, 2011.
[48] S. Prasad, S.B. Avinash, Smart meter data analytics using OpenTSDB and Hadoop, in: 2013 IEEE Innovative Smart Grid Technologies-Asia, ISGT Asia'13, Bangalore, India, 2013, pp. 1–6.
[49] The Linux Foundation, The Container Network Interface, 2024, www.cni.dev. [Visited 2024].
[50] J.C. Anderson, J. Lehnardt, N. Slater, CouchDB: The Definitive Guide, O'Reilly Media, 2010.
[51] J. Forcier, P. Bissex, W.J. Chun, Python Web Development with Django, Addison-Wesley Professional, 2008.
[52] Celery - Distributed task queue, docs.celeryq.dev. [Visited 2024].
[53] J. Carlson, Redis in Action, Simon and Schuster, 2013.



Óscar Castellanos-Rodríguez received the B.S. (2021) and M.S. (2022) degrees in computer science from the Universidade da Coruña (UDC), Spain. Currently, he is a Ph.D. student in the Department of Computer Engineering at UDC. His research interests are focused on providing efficient and elastic Big Data processing in serverless environments. His homepage is https://gac.udc.es/~oscar.castellanos.



Roberto R. Expósito received the B.S. (2010), M.S. (2011) and Ph.D. (2014) degrees in computer science from the Universidade da Coruña (UDC), Spain, where he is currently an Associate Professor in the Department of Computer Engineering at UDC. His main research interests are in the areas of HPC, Big Data and serverless computing, focused on the performance optimisation of distributed processing models in both cluster and cloud infrastructures, and the parallelisation of bioinformatics and data mining applications. His homepage is https://gac.udc.es/~rober.

Jonatan Enes received the B.S. (2015) degree in computer science from the Universidade da Coruña (UDC), Spain, the M.S. (2016) degree in Big Data from the University of Santiago de Compostela, Spain, and the Ph.D. (2020) degree in computer science from UDC. Currently, he is an Assistant Professor in the Department of Computer Engineering at UDC. His main research interests are focused on the performance characterisation of Big Data and HPC applications running in containerised serverless environments. His homepage is https://gac.udc.es/~jonatan.

Guillermo L. Taboada received the B.S. (2002), M.S. (2004) and Ph.D. (2009) degrees in computer science from the Universidade da Coruña (UDC), Spain, where he is currently Full Professor in the Department of Computer Engineering at UDC. His main research interests are in the area of HPC and Big Data computing, focused on parallel and distributed applications in HPC and cloud computing environments, both provisioned and serverless. His homepage is https://gac.udc.es/~gltaboada.

Juan Touriño received the B.S. (1993), M.S. (1993) and Ph.D. (1998) degrees in Computer Science from the Universidade da Coruña (UDC), Spain. In 1993 he joined the Department of Computer Engineering at UDC, where he is currently a Full Professor and Head of the Computer Architecture Group. He has extensively published in the area of parallel and distributed computing, currently focusing on the convergence of HPC and Big Data. He is coauthor of more than 170 technical papers in this area. He has also served in the PC of 80 international conferences. His homepage is https://gac.udc.es/~juan.