Received July 20, 2021, accepted August 1, 2021, date of publication September 3, 2021, date of current version September 15, 2021.
Digital Object Identifier 10.1109/ACCESS.2021.3109972

TaScaaS: A Multi-Tenant Serverless Task Scheduler and Load Balancer as a Service

VICENT GIMÉNEZ-ALVENTOSA, GERMÁN MOLTÓ, AND J. DAMIAN SEGRELLES
Instituto de Instrumentación para Imagen Molecular (I3M), Centro mixto CSIC - Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain
Corresponding author: Vicent Giménez-Alventosa (vicent.gimenez@i3m.upv.es)

This work was supported in part by the Spanish ''Ministerio de Ciencia e Innovación'' for the project Serverless Scientific Computing Across the Hybrid Cloud Continuum (SERCLOCO) under Grant PID2020-113126RB-I00, in part by the program ''Ayudas para la contratación de personal investigador en formación de carácter predoctoral, programa VALi+d'' from the Conselleria d'Educació of the Generalitat Valenciana, Spain, under Grant ACIF/2018/148, in part by the Fondo Social Europeo (FSE), in part by the project ''AI in Secure Privacy-Preserving Computing Continuum (AI-SPRINT)'' through the European Union's Horizon 2020 Research and Innovation Programme under Grant 101016577, in part by the European Regional Development Fund (ERDF) of the Comunitat Valenciana 2014–2020, the regional government of the Comunitat Valenciana, Spain, (High-Performance Algorithms for the Modeling, Simulation and early Detection of diseases in Personalized Medicine), under Project IDIFEDER/2018/032, in part by the European Open Science Cloud - Hub (EOSC-Hub) Project under Grant 777536, and in part by the Helix Nebula Science Cloud (HNSciCloud) Project is also sponsoring the service, allowing users to access the HNSciCloud services pilot for limited scale usage using the voucher schemes provided by the two contractors: T-Systems and Exoscale, under Grant 687614.

ABSTRACT A combination of distributed multi-tenant infrastructures, such as public Clouds and on-premises installations belonging to different organisations, are frequently used for scientific research because of the high computational requirements involved. Although resource sharing maximises their usage, it typically causes undesirable effects such as the *noisy neighbour*, producing unpredictable variations of the infrastructure computing capabilities. These fluctuations affect execution efficiency, even of loosely coupled applications, such as many Monte Carlo based simulation programs. This highlights the need of a service capable to handle workload distribution across multiple infrastructures to mitigate these unpredictable performance fluctuations. With this aim, this work introduces TaScaaS, a highly scalable and completely serverless service deployed on AWS to distribute loosely coupled jobs among several computing infrastructures, and load balance them using a completely asynchronous approach to cope with the performance fluctuations with minimum impact in the execution time. We demonstrate how TaScaaS is not only capable of handling these fluctuations efficiently, achieving reduction in execution times up to 45% in our experiments, but also split the jobs to be computed to meet the user-defined execution time.

INDEX TERMS Cloud computing, heterogeneous computing, load balance, serverless.

I. INTRODUCTION

The use of huge computational power is commonly required in science and engineering to be able to perform computational experiments. Many of these experiments are carried out by loosely coupled algorithms which can be easily parallelized to be executed in a distributed environment. However, the high computational power requirements typically forces the researchers to use several infrastructures belonging to different organisations. For instance, in Monte Carlo simulations of radiation transport applied to the calculus of ionisation chamber correction factors, the work presented by Christian *et al.* [1] required more than 30000 CPU hours to simulate a single case consisting on more than $7 \cdot 10^{11}$ primary particles, and Vicent *et al.* [2] reported approximately 13800 CPU hours to simulate each combination of ionisation chamber and photon beam considered in the study, which results in a total of 745200 CPU hours. As consequence, both works have used several independent infrastructures to cope with the huge computational workload of the studies. These cases, and many others, highlights the need to efficiently handle the execution of loosely coupled applications across several computing infrastructures.

However, distributed infrastructures usually involve heterogeneous computing environments. Therefore, a single infrastructure could exhibit disparate performance among its available computing nodes due to differences in the underlying hardware. Moreover, it is common for computing infrastructures to use a multi-tenancy approach i.e. multiple users share the same underlying physical infrastructure in order to

The associate editor coordinating the review of this manuscript and approving it for publication was Fan-Hsun Tseng.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

optimise resource usage. This technique is used both in local infrastructures and in cloud computing environments. As a consequence, physical resources such as processors, memory, disk or network bandwidth, could be shared between different users or the same user itself, either through virtualization or using a queuing system. Sharing hardware resources causes a non-negligible effect on the whole performance, commonly known as *noisy neighbour* [3]. Notice that the noisy neighbour effect unpredictably affects performance, since it depends on the tenant's workload and the resources being used.

Both effects, *hardware heterogeneity* and *noisy neighbour*, have been widely studied in the literature. For example, Alexandru *et al.* [4] performed a long term study of the performance variability on ten production cloud services. In the same line, Philipp and Jürgen [5] studied the impact of this variability on four cloud environments, showing a different impact in each cloud. Furthermore, in a study performed on Amazon Web Service (AWS) [6], Jörg *et al.* [7] concluded that the performance variability not only differs among cloud providers, but also among Availability Zones (AZs). Recent studies still confirm the existence of this performance variability in IaaS providers [8] and services such as AWS Lambda, as described in our previous work [9]. Indeed, it has even been studied how to reproduce experiments under these changing conditions [10].

To face these problems we present TaScaaS (Task Scheduler as a Service), a completely serverless and highly scalable job scheduler and load balancer service for long-running loosely coupled applications. TaScaaS can be deployed on any FaaS (Functions as a Service) solution, but we have relied on AWS Lambda [1] to exemplify its deployment in a particular cloud provider. Thus, it does not require any previously provisioned infrastructure, it can scale automatically and rapidly according to the workload and it can run at a zero cost if the usage level does not exceed AWS's free tier. In addition, it mitigates the impact on the application performance introduced by the variability in shared and heterogeneous environments. This is done via an asynchronous load balancer system named RUPER-LB [11], which automatically splits the workload to satisfy execution time constraints specified by the user. Finally, TaScaaS has been designed to be easily deployed with the Serverless framework [12] without administrator privileges and it is provided as an open source project available in GitHub.[2]

After the introduction, the remaining sections are organised as follows. First, section II discusses the related work in the area. Then, section III introduces TaScaaS and provides details about the architecture and components. Later, section IV tests and discusses the capabilities of TaScaaS, which are compared to a static split approach on section V. Finally, section VI summarises the conclusions and introduces the future work.

[1] AWS Lambda - https://aws.amazon.com/lambda
[2] TaScaaS - https://github.com/grycap/TaScaaS

The contributions of this work are, first, introducing a service to automatically distribute the workload of iterative long-running applications among several nodes of different infrastructures, each of which may belong to a different organization. Secondly, the work provides a distributed and asynchronous load balance system whose impact on the execution time is negligible. Third, TaScaaS assists the infrastructures in the scaling process to achieve an efficient usage of the available resources, avoiding both, over- and under-provisioning. Finally, an automatic system to partitioning the incoming jobs is provided to achieve execution time constrains set by the user.

II. RELATED WORK

Since this work considers loosely coupled applications, previous works based on task scheduler systems can be used in this regard. In this case, the entire execution could be split in tasks wrapping a portion of the computations.

Focusing on serverless schedulers, there are systems to distribute the workload according to the application characteristics and/or the underlying hardware performance. For example, focusing on applications running on serverless environments, we can find Wukong [13], which provides a serverless parallel computing framework to handle executions on AWS Lambda using a decentralised scheduler. However, this approach does not allow to combine executions from other infrastructures. On the same topic, the SDBCS algorithm, proposed by Pawlik *et al.* [14], plans serverless executions according to budget and time constraints, but it is restricted to serverless executions. These approaches, assume that the application can be represented as a direct acyclic graph (DAG) composed of several short fine-grained tasks. Thus, these tasks are well suited to run directly on a serverless environment. However, computing intensive tasks are less suitable to run on serverless environments, due to the execution time and storage limitations [15], [16]. In addition, depending on the prices, the cost could be no longer competitive [17].

Another scheduler for DAG applications, but not executed on serverless environments, is described in the work by Weiling *et al.* [18], where the authors consider the performance fluctuations of the Virtual Machine (VM) carrying out the computation to predict its future performance and schedule the tasks accordingly. However, their approach is used on tasks with short executions of less than a minute. As the expected performance fluctuations are slower than the mean execution time of a single task, the predictions done by their scheduler are stable enough during the task execution. Nevertheless, on long tasks, this prediction will be less accurate. Although the execution could be divided in smaller tasks, this approach will require a huge amount of unnecessarily communications. Furthermore, the application could require a initialisation step whose execution time is, usually, negligible considering the whole execution. However, splitting the computation too much could force

the initialisation to be repeated on each task, and produce a non-negligible overhead in the whole execution.

The same applies to other workflow DAG scheduling systems such as the ones proposed in [19]–[21]. These schedulers attempt to predict the performance of the computing nodes. Due to the short life of each task, these kind of algorithms could be adapted to fluctuating environments regularly measuring the performance of the system, and assuming that the performance will not fluctuate significantly during the task execution. However, for applications which are not composed of short fine-grained tasks, but by long-running independent ones, the performance can strongly fluctuate during the computation, thus rendering the performance prediction wrong. Thus, for the considered applications, a static assignment is not well suited for fluctuating environments.

HTCondor [22] is a useful tool to maximise the usage of the available resources in a distributed infrastructure, even if the user does not own the resources. However, it does not provide a system to minimise the effect of performance fluctuations on long executions or to satisfy execution time constraints.

To summarise, first, in previous works the strategy followed to distribute the jobs consists on measuring and predicting the infrastructure performance and assume that it remains stable during the mean execution time of a single task. Therefore, the tasks are assumed to be short enough. Thus, those approaches are not suitable for long-running tasks. To solve this limitation, TaScaaS will use a load balance system during the job execution, allowing to reassign the workload among the available resources during the job execution.

Secondly, most works are unable to handle executions on multiple infrastructures. Moreover, some works can handle only executions on a specific service in a specific provider, such as the work of Carver *et al.* [13], which is focused in executions in the AWS Lambda environment. Instead, TaScaaS is agnostic to the infrastructures where the executions take place, allowing to combine resources from different providers and organizations.

Finally, the studied previous works use a predefined partitioning of the job to be executed. For example, in the DAG based schedulers, each task is identified by a graph node. This procedure makes the user responsible for correctly partitioning the work. Thus, it is necessary a previous knowledge about the execution cost of each task. However, the behaviour of many applications strongly depends on the input parameters, making it difficult to predict the execution time of each possible task for each resource type. Alternatively, TaScaaS handles the job partitioning automatically according to the execution time constraint specified by the user. Moreover, the partitioning is updated during the execution to be adapted to the fluctuating performance.

III. ARCHITECTURE

As discussed in the introduction, TaScaaS provides a complete serverless service to schedule and distribute iterative jobs among the different infrastructures. To describe the TaScaaS architecture and functioning we define the following set of terms and assumptions:

First, we will define the processes to be executed in the available infrastructures, hereinafter named *jobs*. The characteristics of these *jobs* are described as follows and summarised in Table 3. The *jobs* are supposed to be iterative processes where each iteration is independent of each other. Therefore, the job with N_i iterations can be split in N_p *job partitions* where each one processes, independently, a subset of the total number of iterations. The easiest way to perform the split consists on dividing equally the number of iterations among all *partitions*, i.e. each one processes N_i/N_p iterations. In addition, we suppose that the number of iterations assigned to each partition can be changed at runtime. Also, as the iterations are independent, the *job partitions* require no communication among them to compute their partial results. Although the assumptions seems to be restrictive, a wide variety of applications satisfy these conditions, for example, many Monte Carlo simulations algorithms. Furthermore, considering only the field of radiation transport simulations, based on Monte Carlo techniques, there are a wide variety of simulation programs, such as PENELOPE [23], GEANT4 [24], FLUKA [25], EGS [26], MCNP [27] among others. In addition, several programs have been developed based on the previous ones, such as PenEasy [28], PenRed [29] or GATE [30]. This codes are widely used for several applications, and are considered as the gold standard methods to perform calculations for clinical radiation based treatments, according to the Task Group report number 186 [31] of the American Association of Physicists in Medicine (AAPM), among other international protocols. Moreover, these characteristics are met not only by Monte Carlo simulations, but also for other applications, such as multiparametric explorations, like neural network design exploration, or file processing, like image recognition processes. With these assumptions, the execution speed can be measured in iterations per second, where the meaning of *iteration* depends on each application. Following the same examples, an iteration could be, each simulated primary particle, for the radiation transport simulation case, each set of parameters that define a neural network or a set of images to process.

Second, each *job partition* will be processed by *workers*, which can be physical nodes, CPUs, vCPUs, a computing cluster, etc., since our framework is agnostic to the underlying computing infrastructure being used for processing. Notice that the number of iterations assigned to each partition will not be a fixed amount. Instead, the number of iterations assigned to each partition will be recalculated according to the processing speed of each one, assigning more iterations to the partitions which are processed faster. Therefore, TaScaaS maximises the usage of faster resources to achieve reduced *job* execution times.

Third, each *worker* belongs to a single *worker infrastructure*, which are composed of one or more *workers*, such as a computing cluster, a set of deployed nodes in a public or on-premises cloud, a single computer, etc. TaScaaS

TABLE 1. Summary of the required job characteristics to be handled by TaScaaS.

Job characteristics
Iterative process
Independent iterations
Divisible in partitions
No communication required for partial results
Measurable processing speed at runtime

will consider each worker as a slot to compute a single *job partition*, i.e. a single *worker infrastructure* can process concurrently as many *job partitions* as the number of its workers. In addition, each *worker infrastructure* has a *frontend* process which performs periodic communications with the TaScaaS service to control the infrastructure live cycle and request more *jobs partitions* to process. TaScaaS is intended to distribute *job partitions* among several *worker infrastructures*, as shown in Figure 1, and to balance the individual *jobs* workload among all their *partitions* according to their speeds. To achieve this purpose, each *worker* performs communications with the load balancer system during the *job partition* processing.

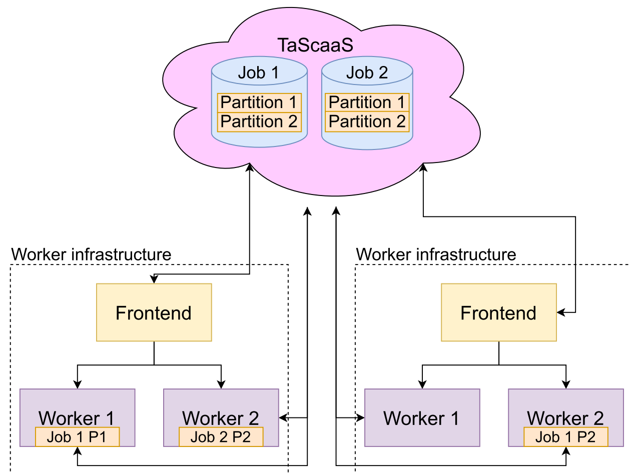


FIGURE 1. TaScaaS functioning diagram.

Once clarified the previous concepts, the TaScaaS architecture explanation follows. The provided TaScaaS service package is deployed on AWS, however, as the used services are commonly found in other public cloud providers, a equivalent approach can be followed on other providers.

TaScaaS does not require any pre-provisioned active computing infrastructure. Instead, it is executed in an event-driven approach, avoiding executing costs when is unused. In addition, the scale capabilities of all the services used by TaScaaS can be handled automatically by the cloud provider.

Notice that TaScaaS does not handle the infrastructure deployment, since there exist several tools for this purpose, such as Infrastructure Manager (IM) [32], Terraform [33] or services offered by cloud providers to automate the deployment, such as AWS Batch, Amazon EMR, etc. Therefore, resource provisioning is out of the scope of this work, as TaScaaS uses computing infrastructures already deployed by the user or where the user has access to perform computations. The motivation for this choice is to take advantage of combining resources of different organizations in which scientists have access. Nevertheles, as we will see, TaScaaS sends information to the *worker infrastructures* to assist them on the scaling process, requesting more or less slots according to the total workload.

TABLE 2. Required services to implement TaScaaS in different Cloud providers.

Provider	Storage	NoSQL DB	FaaS	REST API
AWS	S3	DynamoDB	Lambda	API Gateway
Azure	Blob	CosmosDB	Functions	API Management
GCP	Storage	Datastore	Functions	API Gateway

The components and functionality of TaScaaS are described in Figure 2, which can be split in two main parts: *jobs* and *worker infrastructures* life cycles. The implementation of TaScaaS involves four services: A high-performance object store system, which corresponds to Amazon S3 [34] in our implementation, a managed NoSQL database service based on tables, corresponding to Amazon DynamoDB [35], a Function as a Service (FaaS) serverless service, provided by AWS Lambda [36], and, finally, a REST API service to handle the communications with the *workers* and *worker infrastructures*, corresponding to Amazon API Gateway [37]. The motivation to use a REST API is to allow both *workers* and *worker infrastructures* to communicate with TaScaaS with standard HTTPS requests through a REST API, avoiding the requirement of using a specific Software Development Kit (SDK).

Notice that the choice of AWS as the cloud platform to implement TaScaaS is a mere implementation detail, since other cloud providers such as Microsoft Azure [38] or Google Cloud Platform (GCP) [39] provide similar services. Indeed, an equivalent set of services for Microsoft Azure and GCP providers is shown in Table 2.

The S3 service is used to store the input job files and the results of the executions in a bucket configured by the user. These buckets are the S3 storage units where the objects are stored. DynamoDB stores all the required information about running *jobs* and the available *worker infrastructures*. To handle the system workflow, four Lambda functions have been created, the *Configuration*, *Dispatcher*, *LB* and the *Data* functions. The functionality of each one will be discussed in the following sections. Finally, as mentioned, the API Gateway handles the communication between TaScaaS, the *worker infrastructures* and their *workers*, redirecting the requests to the appropriate Lambda function.

A. JOBS LIFE CYCLE

To start the *jobs* life cycle, first, the job creation and initialisation must be discussed. To simplify visualisation of the description, the involved steps are summarised in the

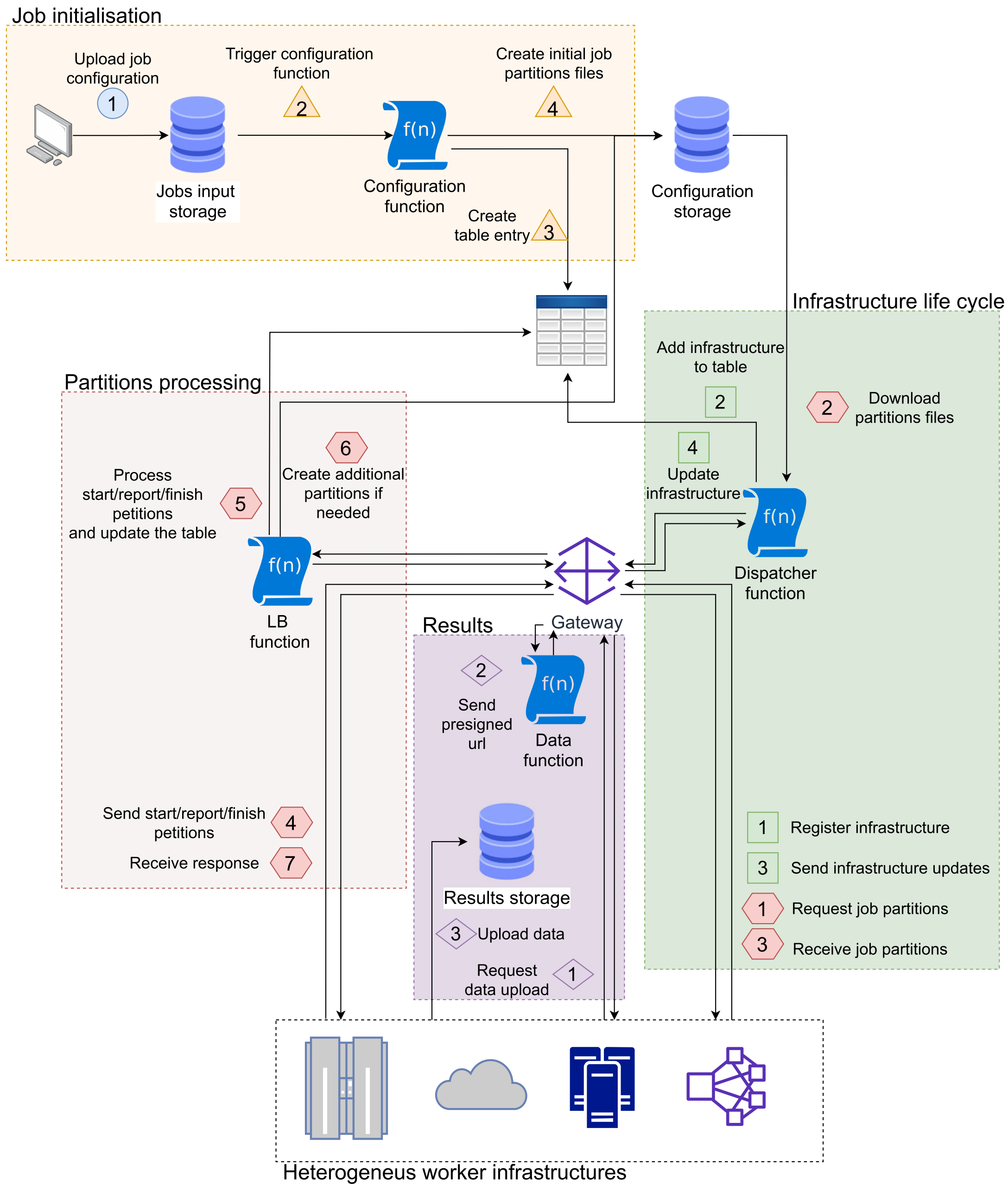


FIGURE 2. TaScaaS component interaction diagram. The blue (circle) number represents the only step done by the user, the job upload. The orange numbers (triangles) correspond to the steps done to prepare the new incoming jobs from the user. On the infrastructures side, first, the green numbers (squares) describe the steps to register each infrastructure in the TaScaaS service and mantain the infrastructure information up to date. Then, the red numbers (hexagons) represent the procedure to request and process job partitions. Finally, the purple numbers (diamonds) describe how data is stored using the TaScaaS service. Also, the diagram has been split in zones to facilitate the description.

Figure 2, which has been split in zones to facilitate the description. To create a new job, the steps involved are located in the *Job initialisation* or yellow zone. First, a job configuration file is uploaded to the jobs storage (blue or circle number 1). These configuration files specify the required *job* parameters, which include the number of iterations to perform, in how many *job partitions* the *job* must be initially split in, the required input data, and a time constrain to finish the execution.

Once uploaded, the *configuration* serverless function will be triggered (orange or triangle number 2). This will create an UUID for the *job* and assign it to all its *partitions* to be able to identify the job they belong to. Also, a configuration file for each initial *partition* (orange or triangle number 4) and the corresponding set of entries in the TaScaaS database table (orange or triangle number 3) will be created to track all the running *jobs* information. A single entry is created for each *job partition* and another one to store the *job* configuration

information. After that, the initial *job partitions* are created and ready to be requested and processed by the *worker infrastructures*.

When the initial *job partitions* have been created, they will be queued in the *configuration storage*. This is a folder in the TaScaaS file storage used to store the *partitions* configuration files until the *frontend* of a *worker infrastructure* requests them to be processed. Once requested, the partition configuration files will be dispatched following a first in first out (FIFO) approach.

First, to request *job partitions*, a *worker infrastructure* must be registered in the TaScaaS service, which steps are summarised by the 1 and 2 green *squares* in the green or *Infrastructure life cycle* zone of the Figure 2. First, the *worker infrastructure* sends a register request (green or square number 1) to the REST API. This request must include the number of available slots, which is the actual number of *job partitions* that can process concurrently, and the maximum number of achievable slots, i.e, the maximum number of slots that the infrastructure can allocate increasing the number of nodes, CPUs, etc, with a scaling process. Then, the request is handled by the *Dispatcher function*, which will store that information in the table (green or square number 2).

Then, the *worker infrastructure* is allowed to request *job partitions* to process, which steps are summarised by the red or hexagon steps in the green or *Infrastructure life cycle* zone of the diagram. As before, the procedure begins with a request to the REST API, (red or hexagon number 1). The number of requested *partitions* is set according to the number of free slots. Then, the request is redirected by the Gateway to the *Dispatcher function*, which will process the configuration files stored in the *Configuration storage* (red or hexagon number 2) and send them to the *worker infrastructure* as response (red or hexagon number 3). The input data to perform the execution is provided via a presigned URL, which grants a limited time permission to download the file from the file storage with a simple HTTPS request. These presigned URLs expire after the specified time, and avoids the requirement to provide authentication credentials to the *workers*.

At this point, the *worker infrastructure* has received a set of *job partitions* to fill some, or all, of its available computing slots. Then, the job execution life cycle starts, which is represented described in the red or *Partitions processing* zone. When the processing starts, the *worker* must inform the TaScaaS load balancer system (red or hexagon number 4, start). This request is handled by the *LB* function and is used to inform the load balancer system that this *partition* is being processed. By default, TaScaaS uses RUPER-LB [11] as load balancer, implemented in the *LB* function, because it is precisely designed to be used on unpredictable fluctuating environments. Nevertheless, it can be changed following the procedure explained in the GitHub repository. All the information regarding the load balancer system for the *job partition*, is stored in the corresponding TaScaaS table entry (red or hexagon number 5, start).

During the *job partition* execution, the load balancer system expects periodic reports to balance the workload of each *job partition* belonging the same job. These is done according to their relative processing speeds. These reports are handled by the *LB* function (red or hexagon number 4, report) and the period between requests depends on the specified time constrain and the TaScaaS configuration. During the report, the *job partition* information is updated in the table (red or hexagon number 5, report), and the number of assigned iterations is recalculated and sent back as response (red or hexagon number 7). Also, an estimation of the remaining time to complete the whole *job* is obtained. If the estimated execution time is greater than the specified in the *job* configuration, the *LB* function will create a set of new *partitions* to meet the time constraint (red or hexagon number 6). The number of new *job partitions* are limited by the TaScaaS configuration parameters to avoid overloading the *worker infrastructures*. The creation of new *job partitions* consists of adding the corresponding entries in the table, and create the configuration files in the *Configuration storage*. After that, the new *job partitions* could be requested to be processed by the *worker infrastructures* with the same procedure as in the *Infrastructure life cycle* zone.

To support worker failures, although the behaviour depends on the used load balancer system, RUPER-LB will check the latest communication timestamp of each *job partition* to consider the corresponding process as inactive or active. Inactive *job partitions* are not considered during the iterations distribution, and the corresponding ones will be distributed among the remaining active *job partitions*.

Finally, *workers* should send a finish request to TaScaaS to inform that this partition will not process more iterations. This request will be also handled by the *LB* function (red or hexagon number 4, finish). Although this request is intended to be sent when a *partition* has finished its assigned iterations, it could be used when the *worker* must stop the execution due to external reasons and there are remaining iterations to compute. For example, if the worker is running on a spot instance [40] it can be interrupted at any moment, depending on the used configuration. Another case is a worker running on a Lambda function, which should send a finish request when the execution time is approaching the configured timeout. When the load balancer receives this request, this *job partition* is flagged as finished in the table (red or hexagon number 5, finish), and will not be further considered to distribute the remaining iterations. Notice that if a *job partition* finishes before the assigned iterations have been reached, its remaining iterations are redistributed among the active *job partitions*.

Concerning the results handling, during, or at the end, the *job partition* processing, the *worker* could require to store results data. The procedure is represented in the purple or *Results* zone and is described following. First, the *worker* send an upload request to the REST API, which will be handled by the *Data* function (purple or diamond number 1). As response to this request, the *Data* function will send back a

presigned URL to upload the results in the file storage (purple or diamond number 2). Like the procedure to obtain the input data, this approach allows to store data in the file storage without granting permissions to the *worker infrastructure* (purple or diamond number 3).

Even though TaScaaS has been designed to be used with a decentralised serverless load balancer system, it can be used as a simple job dispatcher and results storage for applications which do not support load balancing. Specifying a negative expected execution time in the *job* configuration file will disable the load balancer for this *job*. Nevertheless, TaScaaS will keep helping the *worker infrastructures* to fit the incoming workload, as is described in the next section. Further details and examples can be found in the TaScaaS repository.[3]

B. WORKER INFRASTRUCTURE LIFE CYCLE

The *worker infrastructures* life cycle, as explained in section III-A, begins with the registration in the TaScaaS service (green or *Infrastructure life cycle* zone of the Figure 2). After that, each *worker infrastructure* must perform regular *update requests* (green or square number 3). The usefulness of this is twofold. First, the *update* informs TaScaaS that the *worker infrastructure* is still alive. If a *worker infrastructure* does not send any *update request* during the time specified in the TaScaaS configuration, it will be marked as inactive. Thus, it will not be considered to calculate the whole system computing capacity. Furthermore, if the *worker infrastructure* does not send an *update request* after a configurable amount of time, it will be removed from the table, requiring repeating the registration procedure to be able to receive *job partitions*. Secondly, the *update request* can be used to update the *worker infrastructure* information stored by TaScaaS (green or square number 4). This information includes the current number of slots and the maximum achievable slots. The schema of this procedure is included in the green or *Infrastructure life cycle* zone.

As mentioned, TaScaaS will assists the *worker infrastructures* to scale their slots according to the system workload. This is done in two steps. In the first one, during the time specified at the TaScaaS deployment configuration, the information about the number of dispatched and queued *job partitions* is stored. Then, combining this information with the total number of available slots of all the registered *worker infrastructures*, and their maximum achievable slots, TaScaaS calculates the required percentage of achievable slots to tackle the incoming workload. This information is sent to the *worker infrastructures* frontends within the *update* and *job requests* responses. After the data measure step, TaScaaS waits for the same amount of time to let the *worker infrastructures* scale according to the received information. Notice that the *worker infrastructures* should send the new number of available slots using an *update request*, as explained before. Finally, to remove a *worker infrastructure* from the TaScaaS register, their frontend must perform a disconnect request.

[3] https://github.com/grycap/TaScaaS/tree/main/examples

C. PARTITIONING PROCEDURE

In this section we will discuss how jobs are partitioned in TaScaaS and which metrics are used for that purpose. First, the iterations processed during a job or partition execution can be approximated to the mean speed in the interval $[t_0, t_1]$ as,

$$n^I_{done} = \int_{t_0}^{t_1} s(t)dt \approx \Delta t \cdot \bar{s}(t_0, t_1) \tag{1}$$

Since the job execution is split in several partitions, the mean speed depends on the individual speeds of each partition. However, as TaScaaS uses a load balance system, we can assume that the global speed is the result of summing up the processing speeds of all individual job partitions (N^p),

$$\bar{s}^t(t_0, t_1) = \sum_{k=1}^{N^p} \bar{s}_k(t_0, t_1) \tag{2}$$

In addition, we can divide the speed measures in smaller time intervals to be able to obtain the processing performance behaviour with more precision in each instant,

$$\bar{s}^t(0, t) = \sum_{l=1}^{n^t} \bar{s}^t(t_l, t_{l+1}) \setminus t_1 = 0, t_{n^t+1} = t \tag{3}$$

With the previous considerations, the condition to be satisfied to achieve a job processing time restriction (t_{max}) is given by the equation 4,

$$t_{max} \geq \frac{n^I_0}{\bar{s}^t(0, t_{max})} \tag{4}$$

where n^I_0 corresponds to the total number of initial iterations to be processed. However, if we want to achieve the execution time constrain at any instant t during the execution, the equation 4 can be rewritten as a function of the remaining time and the remaining iterations to process,

$$t_{max} - t \geq \frac{n^I(t)}{\bar{s}^t(t, t_{max})} \tag{5}$$

where $\bar{s}^t(t, t_{max})$ corresponds to the remaining time interval mean speed, which can differ significantly compared with the previous measured mean speeds due the fluctuating performance behaviour. The remaining iterations to process can be obtained from the number of processed iterations (equation 1) as,

$$n^I(t) = n^I_0 - t \cdot \bar{s}^t(0, t) \tag{6}$$

Replacing in the equation 5 we obtain the condition to be satisfied at any instant t,

$$t_{max} - t \geq \frac{n^I_0 - t \cdot \bar{s}^t(0, t)}{\bar{s}^t(t, t_{max})} \tag{7}$$

As we are assuming an unpredictable behaviour of the performance, we can't predict the value of $\bar{s}^t(t, t_{max})$. Instead, if at the t instant the number of speed measures is n^t, TaScaaS

will approximate $\bar{s}^t(t, t_{max})$ to the latest available measure (n^t), i.e.,

$$\bar{s}^t(t, t_{max}) \approx \bar{s}^t(t_{n^t}, t_{n^t+1}) \tag{8}$$

as $\bar{s}^t(t_{n^t}, t_{n^t+1})$ provides the most accurate measure of the actual performance of the system. Then, if the condition of the equation 9 is not satisfied,

$$t_{max} - t \geq \frac{n_0^I - t \cdot \bar{s}^t(0, t)}{\bar{s}^t(t_{n^t}, t_{n^t+1})} \tag{9}$$

TaScaaS will split the job in more partitions to satisfy the condition. To calculate the number of required new partitions, TaScaaS assumes that the new partitions will be processed by workers with an equal mean speed determined by the equation 10,

$$\bar{s} = \frac{\bar{s}^t(t_{n^t}, t_{n^t+1})}{N^p} \tag{10}$$

where N^p is the number of actual partitions. This assumption is necessary because TaScaaS does not have information about where the new partitions will be executed. Thus, the new speed is expected to increase, in mean value, as,

$$\bar{s}^t(t_{n^t}, t_{n^t+1}) \rightarrow \bar{s}^t(t_{n^t}, t_{n^t+1}) \frac{N_f^p}{N_0^p} \tag{11}$$

where N_f^p and N_0^p corresponds to the total number of partitions after and before the split respectively. Then, replacing equation 11 in 9, we obtain the number of required partitions to achieve the time constraint at each instant t,

$$N_f^p = \left(\frac{N_0^p}{t_{max} - t}\right)\left(\frac{n_0^I - t \cdot \bar{s}^t(0, t)}{\bar{s}^t(t_{n^t}, t_{n^t+1})}\right) \tag{12}$$

Finally, the number of new partitions (N_{new}^p) will be assigned according to the maximum number of partitions constrain (N_{max}^p),

$$N_{new}^p = \begin{cases} N_f^p - N_0^p & N_f^p \leq N_{max}^p \\ N_{max}^p - N_0^p & N_f^p > N_{max}^p \end{cases} \tag{13}$$

Notice that the relation of the equation 8 will be, usually, false. Therefore, this procedure is repeated during each job execution to adapt the number of partitions according to the new measured speeds. Also, although the number of partitions can be reduced according to the equation 12, TaScaaS will not decrease that number to avoid continuous partition deletions and creations on fluctuating environments. Thus, depending on the behaviour of the performance, in some cases TaScaaS could produce and maintain an overpartitioning, causing the job to finish earlier than the specified time constraint. This effect can be caused in the specific case where the performance maintains a constant increase trend after a significant degradation measured in the previous intervals. Nevertheless, this approach ensures that the time constraint will be achieved, and ensures the synchronisation of partition processing times due the load balance system.

To compare the TaScaaS dynamic partitioning approach with a static one, consider now that no load balance system is used to balance the job partitions. Instead, the number of partitions, and their assigned iterations, is defined initially and can't be changed. In this scenario, the execution time of each partition is determined by the equation 14,

$$t_j = \frac{n_j^I}{\bar{s}_j(0, t_j)} \tag{14}$$

where n_j^I is the number of iterations assigned to the partition number j. Therefore, the whole job execution time will be determined by the maximum value of t_j. In the better case, the approach used to perform the iteration partitioning will produce exactly the same execution time $\bar{t}$ for all partitions,

$$t_j = \bar{t} \quad \forall j \tag{15}$$

and the corresponding process speed will be,

$$s^t(0, \bar{t}) = \frac{n^I}{\bar{t}} = \sum_{j=1}^{N^p} \frac{n_j^I}{\bar{t}} = \sum_{j=1}^{N^p} \bar{s}_j(0, \bar{t}) \tag{16}$$

which is equivalent to the equation 2, meaning that, in the best possible case, we can reproduce a global execution speed equivalent to the balanced approach. Also, the number of partitions cannot be adapted to compensate performance drops. Thus, normally, a static approach will achieve worst results, as we will discuss in the section V applied to the results obtained in the section IV.

IV. RESULTS

We tested the TaScaaS behaviour using three different *worker infrastructure* types simultaneously. The first *worker infrastructure* has been deployed on an on-premises cloud and consists on 32 slots running on Intel Xeon (Skylake, IBRS) processors. The second worker infrastructure has been deployed on the EGI Federated Cloud (IFCA-LCG2 site) and consists on 16 slots running on Intel(R) Xeon(R) CPU E5-26700 @ 2.60GHz processors. Finally, the third consists on a single computer with an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz processor, which has been assigned a maximum of 8 slots. These three *worker infrastructures* will be used to process concurrently all the incoming *jobs* via the TaScaaS service.

To perform the tests, due the importance of the radiation transport simulations in clinical applications and the long execution times involved, we have executed Monte Carlo simulations using PenRed [29], which is a framework for radiation transport simulations using Monte Carlo techniques. Its executions are iterative-based where each iteration corresponds to the simulation of a primary particle and all the secondary particles produced by interactions with matter. Being a probabilistic process, the required computation time of each iteration usually differs. However, as the number of simulated iterations must be big enough to reduce the statistical uncertainties, the mean speed measured in iterations per second converges to a stable value. Thus, to be able to

control the expected execution time and force the usage of different number of partitions for each job, we have defined a set of simulations. Each one has a different execution time constraint to be achieved by TaScaaS, but exactly the same configuration and number of iterations to simulate: 10^6 primary particles.

The executed simulation corresponds to one of the examples provided in the PenRed package, specifically the $1-disc-vr$ example, which details and description can be found in their documentation and github repository.[4] This one consists on a point source of monoenergetic electrons with a energy of $40KeV$. The beam aims to a cylindrical copper phantom. However, as TaScaaS is agnostic to the running application, the results are valid for other simulations and applications that meet the already discussed characteristics. All the required configuration, data base and geometry files are included in the example provided by PenRed, thus can be executed directly once the code has been compiled following the instructions of the corresponding documentation.

Since the number of mean operations for each simulation converges to the same value, the differences on the required computation power is determined by the execution time constraint. Furthermore, we have checked that the measured speed, in an isolated environment, requires a few minutes to stabilise, being very stable after $3-5$ minutes. Thus, to create jobs with different computational power requirements, a set of 4 jobs have been created, whose characteristics are summarised in the Table 3. The simulation time constraints for each job type have been set to 3600, 2700, 2100, 1500 and 900 seconds, expecting to require more partitions for lower time constraints. Taking as a reference the computing power required by job executions of the first type, the following types require a 133%, 171%, 240% and a 400%, respectively, of the computational power required by the type 1 simulations. This selection provides a wide range of jobs with different requirements.

TABLE 3. Job types with the corresponding time constraint, in seconds, and the required computational power relative to type 1.

Type	Time Constraint (s)	Relative cost (%)
1	3600	100
2	2700	133
3	2100	171
4	1500	240
5	900	400

During the experimentation, we will measure the execution time in seconds of each submitted job and the number of created partitions to fit the time constrain. In addition, all the reports done by each job partition of each job have been registered to evaluate the performance behaviour of the system. This data includes the speed, measured in iterations per second, of each time interval, the timestamp of each report, and the evolution of the assigned iterations to each partition.

[4]https://github.com/PenRed/PenRed

For the following analysis, we have considered that the time spent by TaScaaS to balance the execution is minimal. This is because the used balancer system has been designed to introduce a negligible overhead on the whole execution. Moreover, we have extracted the execution time information of the TaScaaS Lambda functions to ensure that assumption. The minimum execution time for all the Lambda invocations is approximately 300 ms, the maximum 1000 ms and the mean value 400 ms. Considering that TaScaaS has been configured to perform a mean value of 20 reports during the execution of each *job partition*, and considering also synchronous blocking communications, in the worst case, the overhead introduced by the TaScaaS processing time is 20 seconds. Since the mean execution time of the lambda functions is lower than 1 second, this overhead is very overestimated. Furthermore, all Lambda functions have been configured with a capacity of 512 MB. Therefore, this overhead could be reduced increasing the computing power of the TaScaaS functions. Nevertheless, considering that the execution time of each simulation partition is on the order of hundreds or thousands of seconds, this overhead is effectively negligible. Regarding communication time, the message data required by the load balancer is lesser than 1 KB, thus, the communication time can be neglected too. Furthermore, the communication can be done asynchronously to continue the execution while waiting for the TaScaaS response.

The tests have been performed during almost three hours launching jobs with different time constraints in irregular intervals and frequencies. The results are shown in Figure 3, where the purple line represents the number of concurrent active partitions for each timestamp and the green line represents the number of slots required by TaScaaS to process all the workload and finish the executions in the user-defined time limit. The figure shows how TaScaaS fits the incoming workload and maintains a pool of free slots to be able to process a possible peak of incoming jobs or new partitions. Then, when TaScaaS neither receives new jobs nor requires to launch more partitions, it decreases the number of required slots efficiently.

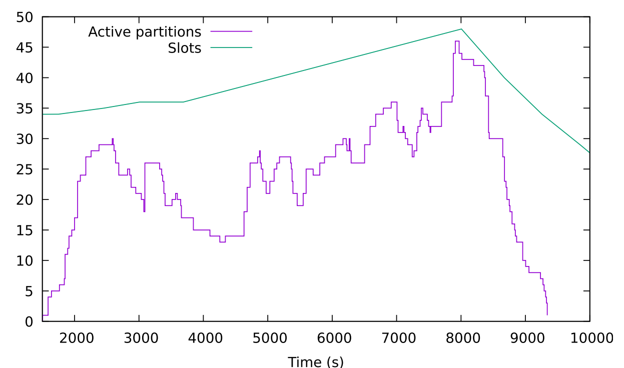


FIGURE 3. Number of active partitions running on the worker infrastructures (dark) and number of required slots by TaScaaS (light) at each timestamp.

Notice that TaScaaS not only must be prepared to allocate new incoming jobs, but also new partitions of the currently

running jobs. Since TaScaaS does not have any knowledge about the running applications nor the heterogeneity behaviour in the environment where the applications run, it only relies on the reported speeds during the execution to allocate new partitions to meet the configured execution time. Thus, a job that initially runs a single partition could require many more partitions to achieve the time constraint. Furthermore, our tests have been done in the worst scenario. First, we have launched all the jobs with only a single initial *partition*, relying on TaScaaS to scale the number of *partitions*. Secondly, we have fixed the scaling step time to 300 seconds, which is relatively low compared to the application execution times and forces TaScaaS to perform predictions with less statistics.

In a real application the user could estimate a mean value of the required partitions and set it during the job configuration. For example, using TaScaaS itself, the user could create sufficient jobs to fill the worker infrastructures and get an upfront distribution of the required partitions per job. As the worker infrastructures are working at full capacity, our test scenario can be considered as the slowest, providing a mean value of the number of required partitions in the worst case. This approach avoids TaScaaS to launch several new partitions during the job execution, providing a more predictable workload input.

Furthermore, the scale step time must be selected as a compromise between response speed and resources allocation accuracy. A faster scale step will provide a faster response time for peaks and troughs of new *partitions*. However, this produces larger fluctuations on the workload measures due to the lower statistic, providing a less accurate allocation of resources, like the case shown in Figure 3. Notice that a slower scale step will provide a more accurate, and probably stable, value of the mean workload. Therefore, the allocated resources will be less underused. However, the system will be less responsive to new *partition* peaks, which may cause some jobs to take longer to start in this case. The election of these configuration parameters depends strongly on the executing applications, the execution time ranges, the worker infrastructures, the rate of incoming jobs and its variability, etc. Nonetheless, the user can obtain an occupation diagram like the shown in the Figure 3 to be able to adjust these parameters. Nevertheless, the results show that TaScaaS can handle adverse configurations efficiently.

Turning to the ability to meet the execution time constraints, we have grouped the simulations with the same time limits. Figure 4 represents the execution time with points, the number of used partitions is represented by the blue histogram, and the time constraint by the red line. This one corresponds to the jobs of 1500 seconds. As we can see, all the jobs satisfy the time limit. Although job number 6 slightly exceeds the execution time goal, the excess time is negligible compared to the total execution time. Therefore, TaScaaS does not request a new worker in order to save resources.

Notice that the execution times present huge differences among jobs which perform the same simulation. This is

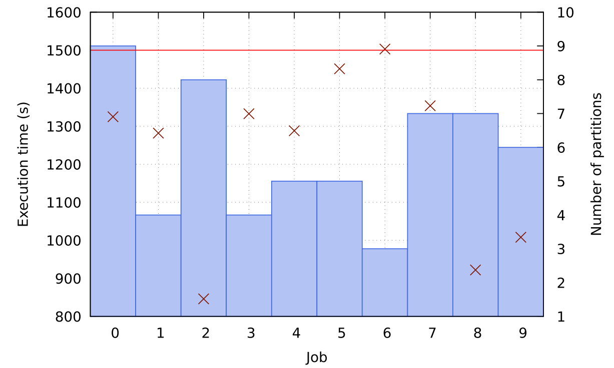


FIGURE 4. Execution times (points) and number of partitions (blue histogram) required by each job with a time constraint of 1500 seconds (red line).

because each *worker infrastructure* have different capabilities and TaScaaS does not know where a job partition will be executed. This effect could be exacerbated if the *worker infrastructures* present heterogeneity on their *workers* capabilities, either because differences with the underlying hardware or because the resources sharing among both, own *partition* executions and tenants. Also, this causes important differences on the number of required partitions, even between jobs with similar execution times.

The simulations with the other execution time limits presents the same behaviour, as shown in Figures 5 and 6 for 2100 and 2700 seconds respectively.

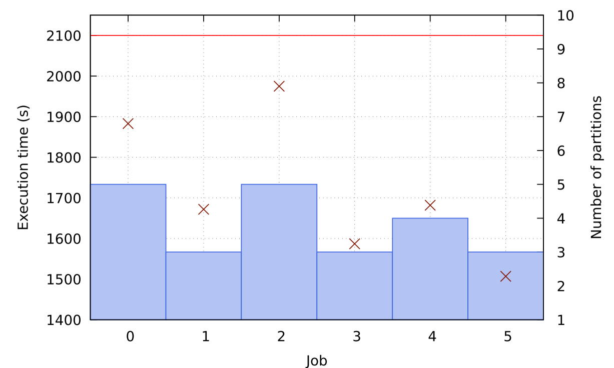


FIGURE 5. Execution times (points) and number of partitions (blue histogram) required by each job with a time constraint of 2100 seconds (red line).

An explanation for the differences in the number of used partitions for similar simulations could be the heterogeneity of the worker infrastructure processors. However, the *noisy neighbour* has also a non-negligible effect. To exemplify this fact, Figures 7 and 8 represents the performance behaviour evolution of the partitions of two specific jobs i.e. the time evolution of the simulation speed for each partition. In these figures, the boxes on the X axis mark the partition start, the vertical lines indicate the first speed report performed by the partition, and the remaining line shows the mean speed evolution. Notice that at the partition start TaScaaS does not have a speed measure, and this is why it is shown as 0 in the figure.

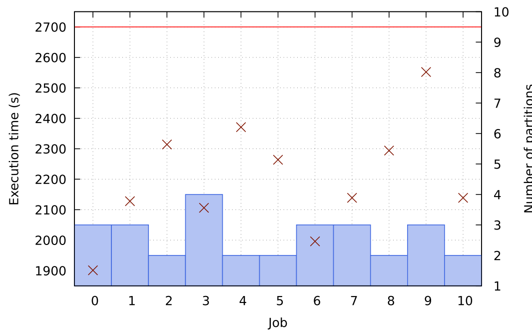


FIGURE 6. Execution times (points) and number of partitions (blue histogram) required by each job with a time constraint of 2700 seconds (red line).

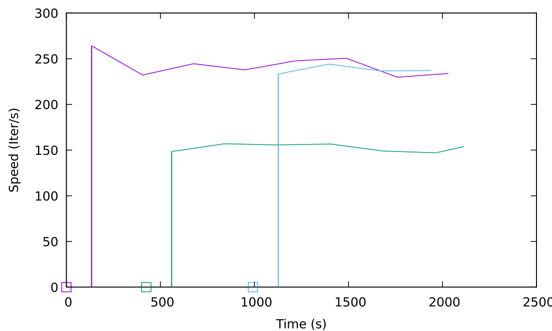


FIGURE 7. Execution speed at each timestamp for the job number 1 from type 2 jobs set.

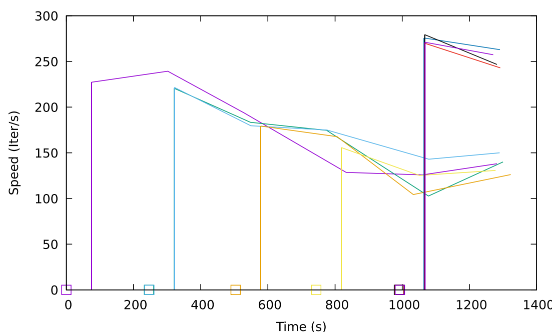


FIGURE 8. Execution speed at each timestamp for the job 1 from the simulations with a time constraint of 1500 seconds.

The first one, corresponds to the job number 1 from type 2 jobs set, which has a time constraint of 2700 seconds. According to the data shown in the figure 6, TaScaaS has created three partitions to fit the time constrain, whose individual speeds evolution are represented by each colour line. The figure, shows a simple behaviour where the execution of each *partition* presents an approximately constant average speed. So, in this case, the environment presents insignificant time dependent fluctuations and is only affected by a constant heterogeneity, probably caused by hardware differences. Once the first partition starts to process (Purple box), at the second box (Green) TaScaaS calculates that the *job* require one more *partition* to meet the execution time limit. However, this second *partition* has been received by a slower *worker*, and is not sufficient to meet the time. To solve it, TaScaaS requests a new *partition*, which, this time, is processed by a faster *worker*. Since TaScaaS works with the mean speed of all partitions, after the second job begins execution, it estimates that the *job* requires a third *partition* with a speed of, approximately, 200 iterations per second. Since the third *partition* is faster, the execution time will be significantly lower than the configured limit, which can explain some of the discrepancies seen between the execution times of the same kind of simulations.

Secondly, Figure 8 corresponds to the job number 0 from type 4 jobs set. This one has a time constraint of 1500 seconds and, according to the data in Figure 4, TaScaaS has created 9 partitions to achieve it. The figure shows a constantly decrease of the speed of each partition. So, as opposed to the previous case, the system capabilities fluctuate during the execution. More specifically, the system decreases its performance and, therefore, this could not only be caused due to the hardware heterogeneity. To mitigate this issue, TaScaaS creates new partitions several times to compensate the continuous decreasing performance, and meet the time constraint. Notice that the execution time of this application is not expected to decrease as the execution advances, as we have explained before and proved in several executions, like the one shown in Figure 7.

As the average speed of this application is expected to be constant if the environment conditions do not change, this behaviour is caused by the competition of several processes for the same resources. It could be caused by both the interference of our own partitions running on the same workers and by the effect of tenants (noisy neighbours). Nevertheless, TaScaaS has handled correctly both cases, creating new *partitions* only when needed. Notice that the *partitions* of the same *job* can be executed by any *worker* of any *worker infrastructure*. Thus, increasing the number of *partitions* does not necessarily overload the same *worker*.

This results demonstrate that TaScaaS can correctly handle the capability variability on heterogeneous environments correcting its effect on the execution time. Notice that TaScaaS has no information about the cause of the speed fluctuation, so it can handle also applications whose speed is not constant throughout the execution.

V. DISCUSSION

To quantify the efficiency improvement achieved by TaScaaS, following, we will compare the results obtained in section IV with a static partitioning approach with no balance, as many of the works discussed in section II. So, for that analysis, we will assume that each job is partitioned according to the system performance information, which can be measured from previous job executions. However, the iterations cannot be reassigned once the partitions have been sent to be computed. As the possible combination of the characteristics involving a single job executions are too big, considering job types, number of partitions, resource where the execution is computed, possible performance fluctuations etc. the study will be carried out using the measured data from the previous experiment, which will allow to compare both results.

First, in Figure 9, the mean speed of each job partition execution is represented by points. It can be seen that these points are split in two groups according to the measured speeds, which corresponds to the two different types of points in Figure 9. In addition, the line of each group represents the fitted evolution of the mean speed according to the measured data. The area surrounding the line represents one standard deviation of the fitted model. Although we could classify the measures according to the worker hardware where the partition has been executed, in many providers services the information about the underlying hardware where the execution is carried out is not, or only partially, accessible. Therefore, we will classify the resource types according to the observable measures. However, notice that the fits standard deviations of each group are about 15% and 2% for the slower and faster group respectively, which are significantly lower than the performance fluctuations measured in different cloud services by the works discussed in the section I.

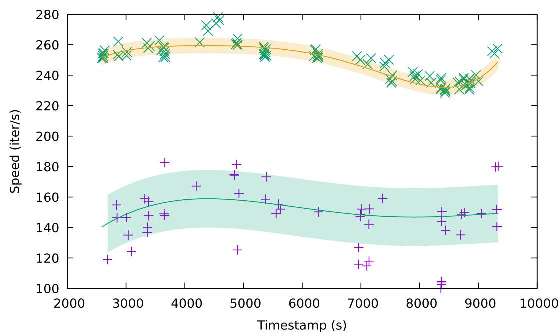


FIGURE 9. Mean execution speed for each processed partition (points). The measures are divided according to two speed levels. The lines represents the fitted time dependency of the mean speed for each set of points. The area surrounding the line represents one standard deviation of the fitted model.

Moreover, the measured mean speeds of a single partition, whose execution is performed completely in the same worker, presents standard deviations up to 50% and 10% for the slower and faster group. Thus, considering that the speed in this kind of applications is determined by the slowest process, because the processes are not balanced, the expected real results will be worst than the predictions of our fitted models. This fact can be seen in the residuals of both fits, represented in Figures 10 and 11 for the slower and faster group respectively.

Then, as we discussed in the section III-C, the job execution can be estimated using the equation 14 applied to the slowest partition, whose mean speed can be predicted by the fitted models. As we cannot know the specific speed of each partition before the execution starts, we will assume that all partitions computed in the same performance group have an equal iteration assignation. Also, to minimise the differences in execution time of partitions executed in different groups, the iteration assignation for the partition of each group must satisfy the relation,

$$\frac{n_f^I}{\bar{s}_f(t)} = \frac{n_s^I}{\bar{s}_s(t)} \tag{17}$$

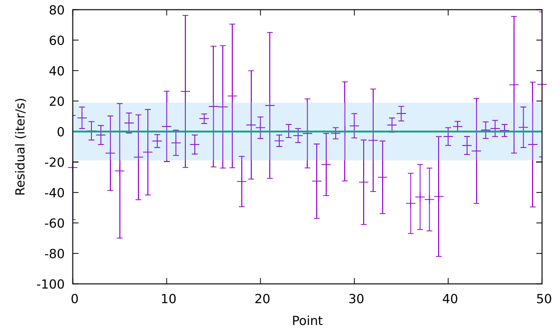


FIGURE 10. Residuals of the fit from the slower group. Error bars represent one standard deviation of measured points. The area surrounding the line represents one standard deviation of the fitted model.

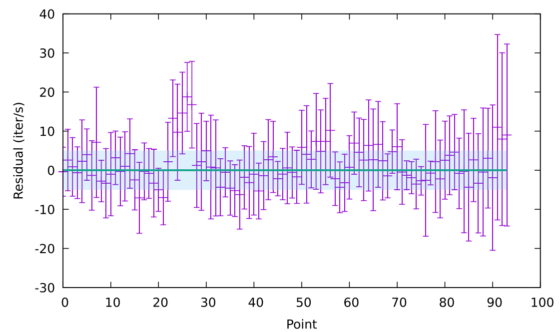


FIGURE 11. Residuals of the fit from the slower group. Error bars represent one standard deviation. The area surrounding the line represents one standard deviation of the fitted model.

where n_f^I and n_s^I are the number of iterations assigned to each partition executed in a fast and slow worker respectively, and $\bar{s}_f(t)$ and $\bar{s}_s(t)$ the corresponding mean speeds. Then the number of assigned iterations to partitions of the slower group can be obtained as follows,

$$\begin{aligned} n^I &= n_f^I n_f + n_s^I n_s \\ n^I &= n_s^I \frac{\bar{s}_f(t)}{\bar{s}_s(t)} n_f + n_s^I n_s \\ n^I &= n_s^I \left(\frac{\bar{s}_f(t)}{\bar{s}_s(t)} n_f + n_s \right) \\ n_s^I &= \frac{n^I}{\frac{\bar{s}_f(t)}{\bar{s}_s(t)} n_f + n_s} \end{aligned} \tag{18}$$

and n_f^I can be calculated using the equation 17. Also, to fit the time constraint, the condition of the equation 19 must be satisfied,

$$t_{max} < \frac{n_i^I}{\bar{s}_i(t)} \quad \forall i \in s, f \tag{19}$$

where i is the partition index and n_i^I takes the n_f^I or n_s^I values depending on the group to which it belongs, the fast or the slow respectively. However, if we suppose normally distributed mean speeds around our fitted model, the partitions execution time will have a non negligible probability of not satisfying the equation 19. As a single slow partition will delay the whole run, the probability to produce a delay is

equivalent to the probability to have a single slow partition. To quantify this effect, the Table 4 shows the delay intervals corresponding to 1σ, 2σ, 3σ and greater than 3σ for both groups, the fast and the slow. For each interval, the dependency of the probability with the number of partitions of each group is shown, and the specific probability for 5 partitions has been calculated, which is, approximately, the mean number of partitions used in our experimentation. In the worst case, the delay produced in the whole job execution time will be equal to the top limit of each delay interval, which possibly causes the condition of the equation 19 to not be met.

TABLE 4. Expected delay probabilities caused by performance fluctuations for both groups, fast and slow. The variable *n*, represents the number of partitions belonging to the specific group.

Delay(fast)	Delay(slow)	Probability	Probability $n = 5$
$(0, 2]\%$	$(0, 15]\%$	$1 - (0.659)^n$	87.57%
$(2, 4]\%$	$(15, 30]\%$	$1 - (0.864)^n$	51.85%
$(4, 6]\%$	$(30, 45]\%$	$1 - (0.979)^n$	10.07%
$> 6\%$	$> 45\%$	$1 - (0.999)^n$	0.5%

Notice also that, due the symmetry of the distribution, the same probabilities can be applied to partitions which mean speed is faster than the model prediction. Thus, these analysis can be also used to calculate the probability to have underused resources. Although the mean speed of our model can be artificially decreased to increase the probability to satisfy the time constraint, for example multiplying the speed by a "security" factor $\rho \in (0, 1) \setminus \overline{s}'(t) = \rho\overline{s}(t)$, this will cause an increment of the required resources to perform the calculus, producing an unnecessary resources over-provisioning. Moreover, this method will not handle the differences in partitions execution speeds, remaining faster resources potentially unused when their partial execution finishes. Furthermore, this method will produce an excess of partitions, which will increase the post-processing cost of the partial results. Depending on the application and the quantity of generated data, an excessive partitioning could produce post-processing times comparable to the execution time.

However, notice that if the fluctuations of our infrastructures are sufficiently low, like the faster performance group, the delays could be assumed and the use of a static approach may be good enough. Thus, to evaluate the suitability of using TaScaaS, could be useful to perform a benchmark following the procedure discussed in this section.

VI. CONCLUSION

In this work, we presented TaScaaS an open source serverless job scheduler and load balancer service to distribute and balance jobs among multiple heterogeneous infrastructures deployed or accessed by the user. As it is deployed on AWS Lambda, it benefits from the AWS free tier, minimising the cost of its execution. Also, TaScaaS is created as a serverless application, so it produces a cost only when it is used. Moreover, AWS Lambda provides a highly scalable environment, thus TaScaaS is capable to handle a large number of simultaneous workers.

We have demonstrated how TaScaaS overcomes static partitioning approaches depending on the performance fluctuations of the available infrastructures, which affect not only public cloud providers, but also on-premises and federated cloud infrastructures. In addition, TaScaaS has proved its capabilities to handle efficiently both kinds of heterogeneity, on hardware and due to sharing resources across multiple tenants. Furthermore, TaScaaS correctly handles time constraints in the execution time in this kind of environments.

In future versions of TaScaaS we will implement improvements such as an adaptive system to select the scale step time and change it according to the incoming workload, and support to deploy the TaScaaS service on other cloud providers. As it is accessed via HTTPS requests, the *worker infrastructures* are agnostic about where the TaScaaS back-end is running, and changing the provider require no changes on the infrastructure side. We will also investigate its behaviour in scenarios of computing continuum where resources from the edge are used in coordination with resources from on-premises and public Clouds. This will allow to achieve load balancing across highly heterogeneous computing platforms across a variety of infrastructures.

ACKNOWLEDGMENT

The authors would like to thank the EGI Applications on Demand service to provide part of the resources used for this work.

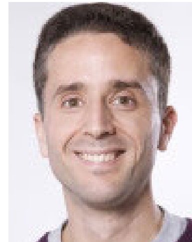
REFERENCES

[1] C. Valdes-Cortez, F. Ballester, J. Vijande, V. Gimenez, V. Gimenez-Alventosa, J. Perez-Calatayud, Y. Niatsetski, and P. Andreo, "Depth-dose measurement corrections for the surface electronic brachytherapy beams of an Esteya unit: A Monte Carlo study," *Phys. Med. Biol.*, vol. 65, no. 24, Dec. 2020, Art. no. 245026.
[2] V. Giménez-Alventosa, V. Giménez, F. Ballester, J. Vijande, and P. Andreo, "Monte Carlo calculation of beam quality correction factors for PTW cylindrical ionization chambers in photon beams," *Phys. Med. Biol.*, vol. 65, no. 20, Oct. 2020, Art. no. 205005.
[3] J. Ericson, M. Mohammadian, and F. Santana, "Analysis of performance variability in public cloud computing," in *Proc. IEEE Int. Conf. Inf. Reuse Integr. (IRI)*, Aug. 2017, pp. 308–314.
[4] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2011, pp. 104–113.
[5] P. Leitner and J. Cito, "Patterns in the chaos—A study of performance variation and predictability in public IaaS clouds," *ACM Trans. Internet Technol.*, vol. 16, no. 3, pp. 1–23, Aug. 2016.
[6] *AWS*. Accessed: Feb. 18, 2021. [Online]. Available: https://aws.amazon.com/
[7] J. Schad, J. Dittrich, and J. A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 460–471, Sep. 2010.
[8] S. Shankar, J. M. Acken, and N. K. Sehgal, "Measuring performance variability in the clouds," *IETE Tech. Rev.*, vol. 35, no. 6, pp. 656–660, Nov. 2018.
[9] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless MapReduce on AWS Lambda," *Future Gener. Comput. Syst.*, vol. 97, pp. 259–274, Aug. 2019.
[10] A. Abedi and T. Brecht, "Conducting repeatable experiments in highly variable cloud computing environments," in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng.*, New York, NY, USA, Apr. 2017, pp. 287–292.
[11] V. G. Alventosa, G. M. Martínez, and J. D. S. Quilis, "RUPER-LB: Load balancing embarrasingly parallel applications in unpredictable cloud environments," 2020, *arXiv:2005.06361*. [Online]. Available: https://arxiv.org/abs/2005.06361

[12] *Serverless Framework*. Accessed: Jan. 25, 2021. [Online]. Available: https://www.serverless.com/
[13] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proc. 11th ACM Symp. Cloud Comput. (SoCC)*, New York, NY, USA, Oct. 2020, pp. 1–15.
[14] M. Pawlik, P. Banach, and M. Malawski, "Adaptation of workflow application scheduling algorithm to serverless infrastructure," in *Euro-Par 2019: Parallel Processing Workshops*, U. Schwardmann, C. Boehme, D. B. Heras, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R. R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, S. L. Scott, Eds. Cham, Switzerland: Springer, 2020, pp. 345–356.
[15] *AWS Lambda Limits*. Accessed: Feb. 18, 2021. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html
[16] *Google Cloud Functions Limits*. Accessed: Feb. 18, 2021. [Online]. Available: https://cloud.google.com/functions/quotas
[17] K. Mahajan, D. Figueiredo, V. Misra, and D. Rubenstein, "Optimal pricing for serverless computing," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2019, pp. 1–6.
[18] W. Li, Y. Xia, M. Zhou, X. Sun, and Q. Zhu, "Fluctuation-aware and predictive workflow scheduling in cost-effective infrastructure-as-a-service clouds," *IEEE Access*, vol. 6, pp. 61488–61502, 2018.
[19] L. F. Bittencourt and E. R. M. Madeira, "HCOC: A cost optimization algorithm for workflow scheduling in hybrid clouds," *J. Internet Services Appl.*, vol. 2, no. 3, pp. 207–227, Dec. 2011.
[20] M. Taufer and A. L. Rosenberg, "Scheduling DAG-based workflows on single cloud instances: High-performance and cost effectiveness with a static scheduler," *Int. J. High Perform. Comput. Appl.*, vol. 31, no. 1, pp. 19–31, Jan. 2017.
[21] H. Cao, H. Jin, X. Wu, S. Wu, and X. Shi, "DAGMap: Efficient scheduling for DAG grid workflow job," in *Proc. 9th IEEE/ACM Int. Conf. Grid Comput.*, Sep. 2008, pp. 17–24.
[22] *HTCondor*. Accessed: Feb. 17, 2021. [Online]. Available: https://research.cs.wisc.edu/htcondor/index.html
[23] F. Salvat, "PENELOPE-2006: A code system for Monte Carlo simulation of electron and photon transport," in *Proc. OECD/NEA Data Bank*, Issy-Les-Moulineaux, France, 2019, p. 7. [Online]. Available: http://www.nea.fr/lists/penelope.html
[24] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. J. N. I. Barrand, and F. Behner, "GEANT4—A simulation toolkit," *Nucl. Instrum. Methods Phys. Res. A, Accel. Spectrom. Detect. Assoc. Equip.*, vol. 506, no. 3, pp. 250–303, 2003.
[25] A. Ferrari, P. R. Sala, A. Fassò, and J. Ranft, *FLUKA: A Multi-Particle Transport Code (Program Version 2005)* (CERN Yellow Reports: Monographs). Geneva, Switzerland: CERN, 2005. [Online]. Available: http://cds.cern.ch/record/898301, doi: 10.5170/CERN-2005-010.
[26] I. Kawrakow, E. Mainegra-Hing, D. W. O. Rogers, F. Tessier, and B. R. B. Walters, "The EGSnrc code system: Monte Carlo simulation of electron and photon transport," Nat. Res. Council Canada, Ottawa, ON, Canada, Tech. Rep. NRCC Rep. PIRS-701, 2019.
[27] R. A. Forster and T. N. K. Godfrey, "MCNP—A general Monte Carlo code for neutron and photon transport," in *Monte-Carlo Methods and Applications in Neutronics, Photonics and Statistical Physics*, R. Alcouffe, R. Dautray, A. Forster, G. Ledanois, and B. Mercier, Eds. Berlin, Germany: Springer, 1985, pp. 33–55, doi: 10.1007/BFb0049033.
[28] J. Sempau, A. Badal, and L. Brualla, "A PENELOPE-based system for the automated Monte Carlo simulation of clinacs and voxelized geometries—Application to far-from-axis fields," *Med. Phys.*, vol. 38, no. 11, pp. 5887–5895, Oct. 2011.
[29] V. Giménez-Alventosa, V. G. Gómez, and S. Oliver, "PenRed: An extensible and parallel Monte-Carlo framework for radiation transport based on PENELOPE," *Comput. Phys. Commun.*, vol. 267, Oct. 2021, Art. no. 108065.
[30] S. Jan, D. Benoit, E. Becheva, T. Carlier, F. Cassol, P. Descourt, T. Frisson, L. Grevillot, L. Guigues, L. Maigne, C. Morel, Y. Perrot, N. Rehfeld, D. Sarrut, D. R. Schaart, S. Stute, U. Pietrzyk, D. Visvikis, N. Zahra, and I. Buvat, "GATE V6: A major enhancement of the GATE simulation platform enabling modelling of CT and radiotherapy," *Phys. Med. Biol.*, vol. 56, no. 4, pp. 881–901, Jan. 2011.
[31] L. Beaulieu, Å. C. Tedgren, J.-F. Carrier, S. D. Davis, F. Mourtada, M. J. Rivard, R. M. Thomson, F. Verhaegen, T. A. Wareing, and J. F. Williamson, "Report of the task group 186 on model-based dose calculation methods in brachytherapy beyond the TG-43 formalism: Current status and recommendations for clinical implementation," *Med. Phys.*, vol. 39, no. 10, pp. 6208–6236, Sep. 2012.
[32] M. Caballer, I. Blanquer, G. Moltó, and C. de Alfonso, "Dynamic management of virtual infrastructures," *J. Grid Comput.*, vol. 13, no. 1, pp. 53–70, Mar. 2015.
[33] *Terraform*. Accessed: Feb. 18, 2021. [Online]. Available: https://www.terraform.io/
[34] *Amazon Simple Storage Service (Amazon S3)*. Accessed: Feb. 18, 2021. [Online]. Available: https://aws.amazon.com/s3/
[35] *Amazon DynamoDB*. Accessed: Feb. 18, 2021. [Online]. Available: https://aws.amazon.com/dynamodb/
[36] *AWS Lambda*. Accessed: Feb. 18, 2021. [Online]. Available: https://aws.amazon.com/lambda/
[37] *AWS API-Gateway*. Accessed: Feb. 18, 2021. [Online]. Available: https://aws.amazon.com/api-gateway/
[38] *Azure*. Accessed: Feb. 6, 2021. [Online]. Available: https://azure.microsoft.com/es-es/
[39] *Google Cloud*. Accessed: Feb. 6, 2021. [Online]. Available: https://cloud.google.com/
[40] *AWS Spot Instance*. Accessed: Mar. 15, 2021. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html

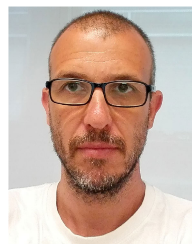
VICENT GIMÉNEZ-ALVENTOSA received the joint B.Sc. and M.Sc. degrees in physics and the master's degree in advanced physics from the Universitat de València (UV), Spain, in 2014 and 2015, respectively, and the master's degree in parallel and distributed computing from the Universitat Politècnica de València (UPV), Spain, in 2017. He is currently pursuing the Ph.D. degree with the Institute of Instrumentation for Molecular Imaging (I3M).

Since 2018, he has been a member of the Grid and High Performance Computing Research Group (GRyCAP), I3M.



GERMÁN MOLTÓ received the B.Sc. and Ph.D. degrees in computer science from the Universitat Politècnica de València (UPV), Spain, in 2002 and 2007, respectively.

Since 2002, he has been a member of the Grid and High Performance Computing Research Group (GRyCAP), Institute of Instrumentation for Molecular Imaging (I3M). He is currently an Associate Professor with the Department of Computer Systems and Computation (DSIC), UPV. He has participated in several European projects, such as INDIGO-DataCloud, EOSC-HUB, DEEP Hybrid-DataCloud, and AI-SPRINT. He has led national research projects in the area of cloud computing. His broad research interests include cloud computing and scientific computing.



J. DAMIAN SEGRELLES received the B.Sc. and Ph.D. degrees in computer science from the Universitat Poliècnica de València (UPV), in 2000 and 2008, respectively.

He is a member of the Grid and High Performance Computing Research Group (GRyCAP), Institute of Instrumentation for Molecular Imaging (I3M), since 2001. He is currently an Associate Professor with the Department of Computer Systems and Computation (DSIC), UPV. He has been involved in grid and cloud technologies and medical image processing, since eight years ago. He has participated more than 20 regional, national, and European research projects. He has coauthored more than 40 papers in international conferences, national conferences, and workshops, as well as more than 15 articles in high-impact journals referenced in the Journal Citation Reports (JCR)/Science Citation Index (SCI). He has also coauthored more than 15 papers in international conferences, national conferences, and workshops related to education. He belongs to the ICAPA Education Research Team.

• • •