ORCA – Online Research @ Cardiff

This is an Open Access document downloaded from ORCA, Cardiff University's institutional repository:https://orca.cardiff.ac.uk/id/eprint/96043/

This is the author’s version of a work that was submitted to / accepted for publication.

Citation for final published version:

Tolosana-Calasanz, Rafael, Diaz-Montes, Javier, Rana, Omer F. and Parashar, Manish 2017. Feedback-control & queueing theory-based resource management for streaming applications. IEEE Transactions on Parallel and Distributed Systems 28 (4) , pp. 1061-1075. 10.1109/TPDS.2016.2603510

Publishers page: http://dx.doi.org/10.1109/TPDS.2016.2603510

Please note:
Changes made as a result of publishing processes such as copy-editing, formatting and page numbers may not be reflected in this version. For the definitive version of this publication, please refer to the published source. You are advised to consult the publisher’s version if you wish to cite this paper.

This version is being made available in accordance with publisher policies. See http://orca.cf.ac.uk/policies.html for usage policies. Copyright and moral rights for publications made available in ORCA are retained by the copyright holders.

Feedback-Control & Queueing Theory-based Resource Management for Streaming Applications

Rafael Tolosana-Calasanz[1], Javier Diaz-Montes[2], Omer Rana[3] and Manish Parashar[2]
[1] Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain
[2] Rutgers Discovery Informatics Institute, Rutgers University, USA
[3] School of Computer Science & Informatics, Cardiff University, UK

Abstract—Recent advances in sensor technologies and instrumentation have led to an extraordinary growth of data sources and streaming applications. A wide variety of devices, from smart phones to dedicated sensors, have the capability of collecting and streaming large amounts of data at unprecedented rates. A number of distinct streaming data models have been proposed. Typical applications for this include smart cites & built environments for instance, where sensor-based infrastructures continue to increase in scale and variety. Understanding how such streaming content can be processed within some time threshold remains a non-trivial and important research topic. We investigate how a cloud-based computational infrastructure can autonomically respond to such streaming content, offering Quality of Service guarantees. We propose an autonomic controller (based on feedback control and queueing theory) to elastically provision virtual machines to meet performance targets associated with a particular data stream. Evaluation is carried out using a federated Cloud-based infrastructure (implemented using CometCloud) – where the allocation of new resources can be based on: (i) differences between sites, i.e. types of resources supported (e.g. GPU vs. CPU only), (ii) cost of execution; (iii) failure rate and likely resilience, etc. In particular, we demonstrate how Little's Law –a widely used result in queuing theory– can be adapted to support dynamic control in the context of such resource provisioning.

Index Terms—Elastic resource provisioning, autonomic systems, feedback control.

1 Introduction

Over the last years, the proliferation of geographically distributed sensors has generated large volumes of data becoming available. This has led to a proliferation of applications that receive raw data continuously, constituting streams of data, which are transmitted over long periods of time from different data source (sensor) nodes with different complexities, ranging from smart phones to specialist instruments. Oftentimes, these applications require data to be timely processed and delivered in order to take operational actions. Data rates and generation timelines can also vary significantly across these different types of infrastructures – depending on the complexity of the sensors or instruments involved. Based on their latency requirements, streaming applications can be classified in two main types, namely low latency and medium to high latency: a) Low latency applications require response times in the order of milliseconds and typically handle hundreds or thousands of events per second, where each element involves small amount of computation – e.g., financial streams, intrusion detection, fraud detection; in contrast, b) medium to high latency applications have response times in the order of seconds, minutes, or even hours, their workloads are typically coarse-grained and more complex, hence involving more computational resources. We can find examples of these in areas such as surveillance and monitoring [1], *smart*-traffic management, data analysis of electricity meter data to support "Smart (Power) Grids" [2], energy management in smart building [3], and geo-spatial imaging processing [4].

In this paper, we focus on streaming applications with medium to high latency. Typically, these applications require data elements, arriving into the system, to be processed within a time threshold (deadline). Moreover, the processing may involve the execution of complex simulations or control algorithms that are typically computationally intensive and that are often executed as *batch* processes. For instance, a smart electric grid application for charging of electric vehicle batteries of an electric area requires to gather charging requests in order to subsequently compute an optimized scheduling algorithm that preserves a number of constraints and satisfies users' preferences. Such a scheduling has to be done within a time threshold, typically around 15 minutes.

When a provider manages a number of such applications with a shared, elastic computing infrastructure, understanding how such streaming content can be processed within some time threshold, so that the number of computational resources can be minimised remains an important challenge. More specifically, a *time slack* appears when the time required for processing is less than the overall period time. Therefore, it can be defined as the remaining difference between the deadline (established in the SLA) and the actual processing time. Data elements from different applications can be buffered by the provider for the given time slack, before starting the processing.

The cloud computing paradigm and its technologies can be exploited for processing such applications. According to

NIST, the cloud computing model [5] enables on-demand access to a shared pool of configurable computing resources, which can be elastically provisioned and released to scale rapidly with workload. In consequence, cloud computing leads to a remarkable change of paradigm: While in the past, the policy of evenly sharing the workload of applications among the distributed resources was typically dominant, in the cloud, one can automatically provision (release) the required computational resources on-demand, according to variations in the workload. On the other hand, the cloud paradigm in real practice has to face a number of challenges due to current maturity of technologies. For instance, such challenges include the fact that computational resources may not perform as expected [6]–[8], making any performance prediction difficult. Also, a rapid scale triggered by workload may also be compromised, as there is currently a significant overhead in the provisioning and in the release of virtual machines (VMs), and such overhead is often asymmetric (i.e. slightly smaller for the release). Another challenge is that cloud providers cannot offer unlimited resources and applications need to operate with multiple cloud providers, triggering new inter-cloud challenges [9].

The problem of cloud resource management has been studied for stateless Web requests [10]–[14], but in such a context, requests need to be processed on-line and as soon as the request is received. Such efforts make use of autonomic principles in conjunction with elasticity in cloud environments, enabling a system to dynamically provision computational resources depending on changes in the workload. Nonetheless, the introduction of the time slack for the data elements, before processing, produces variations to that problem, which require a different approach.

In this paper, we propose a resource management strategy for the aforementioned streaming applications that require continuous processing of data elements within a time threshold (deadline) and that are computationally intensive, – typically involving the execution of batch jobs. The strategy is achieved by means of an autonomic controller, which is based on feedback control and makes use of system properties derived from queueing theory. Its objective is to optimize the number of cloud computational resources (generally VMs) allocated to a *particular data stream*, so that data elements are processed within an established time threshold on average, while the number of VMs is minimal. For such an objective, the data elements of a *given data stream are buffered on a queue* for the maximum time (*time slack*) that, together with the processing time, does not violate the time threshold on average: There will be some jobs over the objective and some others under the objective, we assume scenarios where such postulate is acceptable.

Our controller manages the processing rate of data elements by (de-)allocating VMs to meet its objective. It is worth highlighting that such a processing rate is adaptive to changing conditions –depending on unpredictable variations of the incoming rate of data elements and unexpected performance fluctuations of computational resources. Our controller has two options to control the time slack: (i) to control the waiting time of data elements directly, or (ii) as, in real practice, information about waiting time of data elements may not always be available or may be very difficult to obtain, we decided to make use of Little's law (LL) for deriving waiting time from arrival rate and queue (buffer) size. In other words, the controller reacts by (de)-allocating resources on changes of the queue size, leading to an indirect control of the time slack. It should also be noted that although LL has been widely studied for infinite times and stationary conditions, recent studies show that it also holds under finite times and non-stationary conditions [15].

2 Background: Little's Law

A queuing system comprises [15] a number of discrete objects often called items, arriving at some rate within a system. The stream of arrivals enters the system, joins one or more queues, eventually receives a service, and exits in a departure stream. In general terms, services perform operations over items, which involve an amount of time (e.g., in computing, we often call such an action, processing, and to the amount of time it involves, processing or execution time). Typically, a queue appears as a result of differences between arrival and service rates –when arrival rates are higher than actual service rates (e.g. in terms of computing, when items arrive into the systems faster than actually being processed). The queue prevents items from being lost, as they are buffered on the queue. LL is a mathematical relationship between three variables: The average number of items in a queuing system, denoted by L, equals the average arrival rate of items, λ, multiplied by the average waiting time of an item, W. Thus, $L = \lambda * W$. At the time Little first proposed his theorem, he was focused on infinite intervals of time and stationary conditions.

LL in Practice. However, after more than fifty years of research and practice, LL has also shown that it holds under a great number of conditions, including finite intervals of time, which provides significant value for engineering design and operational problem solving [15]. The following theorems, as formulated in the 50-year LL retrospective paper [15], incorporate these conditions implicitly, which are of extremely importance for practical purposes.

Theorem 1. (from [15]) LL Over $[0, T]$. LL.1. For a queuing system observed over $[0, T]$ that is empty at 0 and at T and has $0 < T < \infty$, $L = \lambda * W$ holds.

A number of important remarks for *practical consideration* can be obtained from the previous theorem. First, not only does LL hold in finite intervals of time, but also under *non-stationary* conditions. A stationary process is one whose joint probability distribution does not change over time, which is not the case in common practice. Actually, in real practice, the probability distribution of arrival items may be non-stationary. In a computing system, for instance, the arrival of data elements to a computing system for processing can be subject to sudden data bursts or spikes. Analogously, performance of computational resources can also be subject to sudden variations. The importance of LL.1 is that it states that LL continues to be true under both stationary and *non-stationary* conditions during *finite* intervals of time. Second, LL also holds independent of the queue discipline, such as FIFO (first in, first out), LIFO (last in, first out), random, etc. Furthermore, a generalization of LL.1 is also proposed in [15] by eliminating the restriction of zero starting and ending queues in the interval $[0, T]$.

Theorem 2. (from [15]) LL over $[0, T]$. LL.2. For a queuing system observed over $[0, T]$ that has $0 < T < \infty$, $L = \lambda * W$ holds.

In addition to the remarks derived from LL.1, LL.2 also ensures that LL holds when the queuing system is not empty at the beginning or at the end of the interval of time. Nevertheless, the conservation of items still needs to be guaranteed for LL to be held – there are no lost items. In a computing system, for instance, at the beginning or the end of the interval of time, data elements to be processed can be present on the queue, but no data element disappears during the computation –e.g. due to a failure.

Using LL, it is possible for average waiting times to be estimated by dividing the average number of elements on the queue by the average arrival rate. Nevertheless, as it is pointed out in [16], this simple indirect estimator tends to be significantly biased, when arrival rates are time varying and service processing times are relatively long. Surrogate estimators are also proposed in [16] for such cases.

Traffic Intensity and Queuing Time. An important concept in Queueing Theory is that of *traffic intensity*, which is the ratio of the incoming and outcoming rates of a queueing system.

Definition 1. Given a multi-server queueing system, its *traffic intensity*, denoted as ϱ, is: $\varrho = \frac{\lambda}{c\mu}$, where λ is the average arrival rate of items into the system, c is the number of server instances, and μ is the average throughput per server. In order for a system with a finite number of servers to be stable, its traffic intensity must be $0 \leq \varrho < 1$. Nevertheless, this is not the case when considering an infinite number of servers. In that case, the system always has enough servers, and we are then more interested in the number of busy servers and their expectation [17].

Our approach assumes that we always have enough resources, which is analogous to having infinite number of resources. We are just interested in the number of resources active and in operation and in switching them on and off depending on the circumstances and the SLA. We make use of the traffic intensity for deriving the number of computational resources (server instances in the definition) required during the execution. More details can be found in Section 4.2. Additionally, we also characterize the average time that each data element of a data stream spends on the queue.

Definition 2. The average time, T, items spend within a queueing system can be characterized by $T = W + S$, where W is the average time a data element spends waiting for a resource in the queue, and S the average time required to process a data element.

3 Data Stream Model of Computation

A data stream D of length k is a sequence $D = \{d_1, d_2, \ldots d_k\}$ with $k \in \mathbb{N}$ of digitally encoded data coming from a finite universe O that depends on the application –it can be letters, numbers, XML tags or any other finite set of elements. In general terms, specific characteristics differentiate these data streams from traditional databases: (i) input data elements may only be available during an interval of time, after which random access may be difficult or impossible; (ii) data elements arrive in real time and continuously; and (iii) the arrival rate of data elements can be bursty in nature and unpredictable, and data elements may arrive in the system out-of-order.

3.1 Driving Use Cases

As discussed above, with the proliferation of geographically distributed sensors, a variety of applications have emerged in different areas such as surveillance and monitoring, *smart-* traffic management, etc. For instance, a smart electric grid application for charging of electric vehicle batteries of an electric area requires to gather charging requests every control period (e.g. every 15 minutes) and subsequently to compute an optimized scheduling algorithm (i.e. one schedule per area), so that a number of constraints are preserved and users' preferences met. Such a scheduling has to be done before the next requests arrive in the next control period (consider, for example [18], where a breadth-first search algorithm is executed for each geographic area to prioritize on who should be selected for charging, given that demand exceeds supply). Another similar application scenario is that of the smart building management [19], which collects measurements of temperature, humidity and people density within a building every established period of time (e.g. every 5 minutes), and automates and optimizes a number of controlled parameters by means of the EnergyPlus model.

On the other hand, the study of marine ecosystems is vital for understanding environmental effects and though undersea video data is available, it is tedious to analyse. The EU-funded Fish4Knowledge project [20] developed algorithms and a distributed infrastructure in order to support automated video analysis of undersea video data. The idea is that videos are recorded and transmitted continuously and periodically for subsequently processing. The processing of each video is computationally intensive analysis and independent of the others.

Hence, we can conclude that there are a number of applications arising with the proliferation of distributed sensors. Many of them share common characteristics: they are typically computationally intensive, but they do not require an immediate response, as data elements arriving into the system need to be processed within a time threshold (deadline). Such a deadline is in the order of seconds, minutes, hours, or even days, rather than in the order of milliseconds, and this deadline is one of the key metrics in the SLA. Moreover, the processing may involve the execution of complex simulations or control algorithms that are often executed as *batch* processes, independent of the rest (stateless).

3.2 Functional and Non-Functional Requirements

For the specification of the operations to be applied to each data stream, we consider the workflow streaming model of computation [21]. This model of computation consists of a sequence of one or more tasks applied sequentially to a vector of input data elements as they are received from sensors. We assume that a data element can pass through a workflow pipeline task as it is produced by its predecessor (avoiding blocking semantics). In consequence,

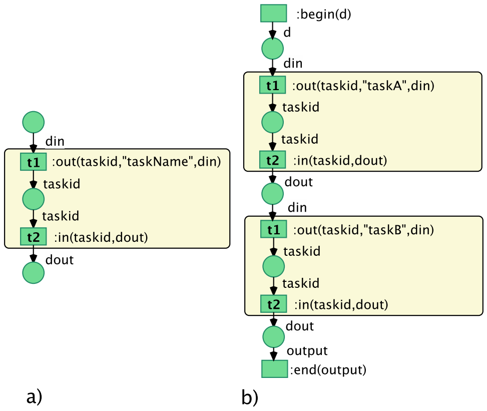


Fig. 1: a) Linda-based Streaming Workflow Task Pattern; b) Streaming Workflow Example

unlike other pipeline models of computation, multiple data elements could be executing the same task at a time, or even data elements can finalize their execution in an out-of-order manner.

In our proposal, a streaming workflow is specified as depicted by Fig. 1, in terms of Petri nets (i.e. actually in terms of a high-level Petri net class called Reference nets). Each of our workflow tasks are specified in an abstract way – without binding to any computational resource. A task (see Fig. 1 a)) is expressed in terms of two Linda operations: (i) an *out* operation that writes the operation name and its arguments into a tuple space, and (ii) an *in* operation retrieving the results from a tuple space upon completion. The Linda communication & coordination paradigm is based on communication orthogonality in the interactions, whereby interacting peers do not have any prior knowledge about each other. Hence, our workflow tasks are uncoupled in time and space from the execution environment that will execute it, and this provides greater degrees of flexibility in the execution. More detailed information regarding this usage of Linda, workflow tasks and Reference nets can be found in [22]. As the Reference nets are actually interpreted, these specifications are also utilized for the actual execution of the coordination mechanism. Another important aspect is that, depending on the application semantics and functional requirements, a task will generate one or multiple computational jobs for the processing. In this paper, for simplicity, we are considering that a task generates one single job.

Fig. 1 b) shows a streaming workflow example. Data elements are introduced at the initial transition (Channel *begin*), and once a data element is fully processed, it is retrieved from Channel *end*. That workflow is a sequential composition of two tasks, named "task A" and "task B", respectively. In particular, tasks are expected to be connected by means of a data dependency; the output *dout* of a tasks becomes the input *din* of the following task in the sequence. It is important to note that the model is following streaming semantics, whereby multiple data elements can be executing a task simultaneously, and even the processing for a data element can finalise out of order for previously arrived data elements. More details of the specification and the execution semantics can be found in [22].

As for the non-functional requirements, SLA consists of a threshold deadline (δ). The primary evaluation metric considered in this work is the number of data elements computed within the threshold deadline (δ). The default number of data elements that are required to meet the threshold deadline is 50% (i.e.: at least 50% of the data elements must have a response time (T) lesser or equals δ. The remaining jobs are not guaranteed to be completed within δ as they are computed in a best effort manner, but they cannot be discarded. Moreover, higher success percentages (e.g. 75% or 90%) can also be achieved as discussed later in Section 5.2.2.

Moreover, resource allocation to support this streaming model is undertaken on a per-second basis (adopted from similar concept in Amazon Lambda, where resource allocation and pricing is based on a 100ms period [1]). Longer resource allocation periods (of 1 hour) are inefficient and unsuited for fine grain streaming processing approaches. This work focuses on a much higher frequency allocation strategy (as observed in Amazon Lambda) to account for fluctuations in the associated data stream.

3.3 Time Series Model

There are a number of data stream models of computation reported in literature and they can be analyzed and better understood by formalizing the semantics and relationship between the data elements forming a stream. An attempt to formalize these semantics was accomplished in [23]. In this paper, we make use of the *time series model*, in which the result of the processing of any data element of the stream is independent of the rest and computations are completely independent. It is suitable for monitoring and for observations in time that do not need to establish relationships with past measurements.

Along with the time series model, we also considered *one-pass processing* [23], thereby each data element arriving into the system is processed only once and it cannot be retrieved later again. In accordance with the characteristics of the applications we are based on (see Section 3.1), data elements arrive into the system and have to be processed within a period of time, after which access to arbitrary past items for processing is not useful.

3.4 Computational Capacity Requirements

Given all the previous characteristics of the data stream model of computation, we have identified the following key requirements for an autonomic elastic resource management:

Support for Data Intensive Workloads: The type of processing required for such type of data streams is computationally intensive and it can require a large amount of computational resources, involving parallel or distributed complex simulations, optimization algorithms, or the execution of forecasting models.

Enforce Quality of Service (QoS): The computation of each data element is a critical process in time that must be computed within established time slots between controlled requests. Typically, the processing time (S) is less than the overall due time for the control period (*deadline*), allowing data elements to be buffered prior to their processing. Moreover, in some scenarios, exceeding the overall amount of

[1] https://aws.amazon.com/lambda/pricing/

time for performing the computation may be allowed by the SLA of the application. The resource management policy should provide mechanisms for enforcing QoS.

Elastic/On-demand Provisioning: The computational capacity must be adjusted to the overall requests. Therefore, a resource management strategy needs to allocate the appropriated number of computational resources to process unpredictable and variable workloads while satisfying QoS constraints, as described above. The underlying infrastructure must be able to support admission control and enable a variable processing rate per stream. The adopted mechanisms should be based on autonomic principles, so that they are resilient and self-adaptive to variations in the historical traces without requiring human intervention.

All the above requirements are used to design the system architecture and resource management policy described and subsequently evaluated in the following sections.

4 Queuing-Theory-based Approach for Data Stream Processing in Clouds

We propose a data stream processing strategy that is modeled using feedback control principles for the self-management of computational resources and that makes use of queuing theory. Next, we describe the conceptual architectural of our system and the autonomic loop operations.

4.1 Generic Architecture

Our proposed architecture can process a number of independent data streams simultaneously as depicted in Fig. 2. Each data stream is assigned to a queue (buffer) in the system, so that when a data element arrives, it is redirected to its corresponding queue and waits until there are computational resources available, and after the processing stage, the results are transmitted to the destination. In this activity, a data stream follows three main stages: (i) data admission and control, (ii) processing, and (iii) data transfer to the destination. The explicit goal of an admission control policy is to prevent the system from accepting workloads in violation of high-level system policies: for example, the admission control policy may only accept a maximum income rate (λ) of X data elements per stream. The admission control component can decide on buffering the arriving elements or even discarding them, in case the processing supports such a policy (i.e. by losing some precision). In our case, we assume that, we have enough resources to satisfy the computational needs of the applications and that our buffer is always sufficient so that all data elements are accepted. We also assume that a data stream has its own functional requirements and previously negotiated Service Level Agreements (SLAs) that are translated into QoS requirements.

In order to achieve this, there is a controller per data stream, whose main objective is to reduce the number of computational resources (VMs) whilst still enforcing QoS targets. Two main scaling mechanisms can be found in literature: *vertical scaling* and *horizontal scaling*. Vertical scaling keeps the number of VMs of an application constant, but increases the amount of resources allocated to each one of them. This can be done by either migrating the VMs to more powerful servers or by keeping the VMs on the same servers but increasing their share of the CPU time. The first alternative involves additional overhead; the VM is stopped, a snapshot of it is taken, the file is transported to a more powerful server, and, finally, the VM is restated at the new site. Horizontal scaling is the most common mode of scaling on a cloud; it is achieved by increasing the number of VMs as the load increases and reducing the number of VMs when the load decreases. This strategy typically takes advantage of a set of physical machines and limits the number of VMs that can concurrently live within a single physical machine.

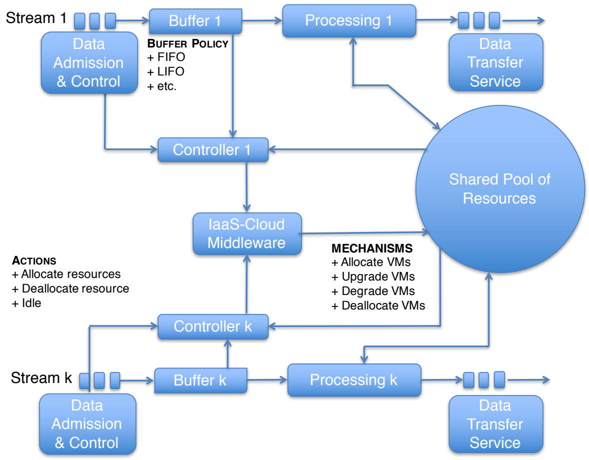


Fig. 2: Generic Queueing-Theory-based System Architecture for Cloud Resource Provisioning for Streaming Applications

Currently, our controller considers horizontal scaling and it ideally targets fine grain billing models, i.e. in a per second basis, inspired by Amazon Lambda, as mentioned before. However, we also wanted to ensure efficient usage of resources for cases where cloud providers offer business models billing in a per hour basis. In a per hour billing model, your usage is rounded to the next hour even if you release the resource back to the provider. For this reason, in our model we decided to use a shared pool of resources where resources are made available to satisfy the computational demands of our streams. We assume that the medium- and long-term policies for resource provisioning are capable of estimating the number of resources during the peak workload periods. Nevertheless, only when this pool is exhausted, we require a new on-demand resource from other cloud providers. Since the overheads of provisioning and de-provisioning can severely affect the performance of our controller when those overheads are in a similar order of magnitude to processing times, we propose the following strategy: Resources that are in the shared pool and reach the end of their allocation/usage period are released back to the resource provider. Thus, when a data stream de-allocates a VM, instead of switching it off, it remains operative in the shared pool for a period of time, aiming at satisfying the provisioning needs for any other data stream. Resources that are part of the shared pool do not introduce additional monetary costs, and the overheads are alleviated. In the following subsections, the operations carried out by the controller are explained further.

4.2 Autonomic Elasticity for Streaming Applications

We consider that the QoS objective of our streams is to guarantee certain response time or deadline. As a resource

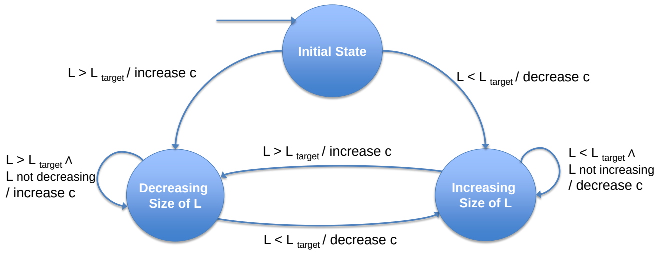


Fig. 3: Mealy State Machine Modeling the Controller's Behaviour

provider, we need to guarantee such a QoS objective, while minimizing the amount of resources used. For such a purpose, we either directly monitor the maximum time that a job (i.e. data element) can stay on the queue waiting to be processed, on average, without violating its QoS guarantee; or alternately, we take advantage of LL, in order to calculate the waiting time from the queue size. Thus, the objective of the proposed controller is that the average waiting time W on the queue for a data element equals the maximum allowed, defined as *time slack*. As depicted in Fig. 3, a Mealy state machine that consists of 3 states can model the controller: (i) initial state, (ii) decreasing size of L state, and (iii) increasing size of L state. It can take 3 different actions: (i) to allocate VMs, if L > L*; (ii) to de-allocate VMs, if L < L* and (iii) not to take an action, in any other case. In our approach, the controller follows a MAPE (monitoring, analysis, planning and execution) loop. We assume that the arrival rate of each stream and their processing times cannot be estimated in advance. Presently, we do not have control over the size of the shared pool of resources and it is used in a best effort manner, i.e. resources with time left in their current allocation period are made available to others. If a resource reaches the end of that period, it is released for use by other operations.

Monitoring. During this stage, the following system variables are monitored: (i) queue size L, which is the control output; (ii) the arrival rate for a data stream λ; and (iii) the processing time of each data element, which allows us to calculate S. These variables are recorded and computed every time a job enters or leaves the system. On the other hand, we have variables that are calculated periodically at intervals, which allow our controller to take operational decisions. This includes the process of applying LL, which requires average values in order to hold, and allows the controller to act on L (average number of jobs on the queue) rather than on W (average wait time of a job in the queue). Average values may increase the response time of our controller to changes in the environment. In order to mitigate the effects of long running averages and allow the system to rapidly react upon changes (i.e. bursty conditions of λ and unexpected performance of VMs), we could limit the amount of data used to take operational decisions. Specifically, we propose two alternatives:

- *Average Reset:* The first approach consists of discarding previously monitored data, when a change in the input pattern is detected. This resets the average values and helps to mitigate the effects of long running averages, which allows the system to quickly react upon changes in the demand. Therefore, the overall execution time is divided in sub-periods of execution: a sub-period starts when a reset in the average values is accomplished. Firstly, according to LL.2 (see Theorem 2), LL holds for finite periods of time even if the system is not empty at the beginning or at the end of any sub-period (otherwise, W could not be derived by L). The challenge of such an approach, however, is to efficiently and effectively detect variations in monitoring that require a change of sub-period. From the two variables considered, arrival rate λ and average execution time S, there are a number of studies proposing different data burst detectors [24]. In the case of average execution time, we can also consider thresholds to identify variations in VM performance.
- *Moving Average:* The second strategy is to compute moving averages. In a moving average only a subsequence, a window, of a time series is considered for computing the average: older values of the series are being discarded with the arrival of new values. Thereby, the effect of long running averages is limited. Nevertheless, it may be challenging to determine the window size: if the window is too small, the derived values for W may tend to be biased, especially in cases where the processing time is too long (see Section 2). In contrast, if the window is too big, then the same problem of arithmetic averages appears.

It is also worth highlighting that the performance variation of computational resources is considered implicitly in our model by assuming the application can take an arbitrary time to process a data element.

Analysis. In this phase, there are two main steps: The setpoint (that is, the objective queue size $L*$) is computed and then the monitored average value of L for the period of time T is compared to $L*$. According to the difference (the error) found, an action is decided: (i) to remain idle, (ii) to increase computational power, or (iii) to reduce computational power.

4.2.1 Establishing the Reference Setpoint ($L*$)

In accordance with Definition 2, the average time for data elements of a data stream in the system is given by: $T = W + S$, where W is the average queuing time, and S is the average processing time. We assume that the QoS for a data stream involves the processing of data elements within a deadline, δ, on average. Therefore, in order to enforce QoS, the following must be fulfilled:

$$T = W + S \leq \delta \tag{1}$$

As we are utilizing an elastic pay-as-you-go infrastructure, as a load balancing policy, we want to meet QoS while minimizing the number of computational resources. This policy is fully satisfied when $T_{max} = W_{max} + S = \delta$. Therefore, when $W_{max} = \delta - S$, we obtain the maximum time slack, i.e. the maximum time a data element can spend on the queue without violating the QoS, which minimises the number of computational resources. Alternatively, with W_{max} and by applying LL, we can change the variable and obtain the reference setpoint, the maximum average number of data elements on the queue, $L*$:

$$L* = \lambda W_{max} = \lambda(\delta - S) \tag{2}$$

It can be seen that $L*$ depends on λ, the arrival rate, and S, the average processing time in the system. Under variable and unpredictable workload and performance times, $L*$ is likely to vary over time unpredictably. Thus, the challenge for the controller is to make use of IaaS elastic actions, in order to maintain $L*$ in its objective value.

4.2.2 Determining an Action

After the reference point is set up, the controller establishes upper and lower thresholds around it, incorporating hysteresis in the reference setpoint to avoid oscillations. Then, the current queue size (which directly affects the waiting time of data elements) is compared to the thresholds: if it is less that the lower one, the computational power is decreased (as there are more resources than required), and the computational power is increased, when the queue size is higher than the upper threshold (it should be noticed that being above the objective queue size means that the waiting time will be violated). At this point, the controller decides an action: (i) to remain idle, (ii) to increase computational power, or (iii) to decrease computational power.

Capacity Planning. Currently, we are only considering horizontal scaling for our controller's actions. Therefore, the controller can provision and de-provision a discrete number of computational resources (VMs). We assume that the infrastructure has limits in terms of the amount of VMs that can be provisioned as a whole – this number is set by the infrastructure provider and it considers both the number of physical machines and the number of VMs per physical machine. However, as discussed above, in this work, we assume that we always have enough resources. We also assume homogeneity in the computational resources and leave considering multiple types of resources to achieve finer grain actions for future work. In our approach, we adaptively regulate the number of provisioned VMs, c, such that the QoS of the data stream is enforced. In order to find the number of active VMs (c), we make use of the traffic intensity, ϱ, (which defines the relationship between input and output, as defined in Section 2). Thus, by equating Eq. 1 to 1, and solving for c, we obtain the number of computational resources that are able to output elements at the same rate as the are entering the system: $\lceil c \rceil = \frac{\lambda}{\mu}$, with $c \in \mathbb{N}$. It should be noticed that as we have a discrete number of resources, we round up to the nearest integer.

Therefore, the actions for decreasing and increasing the waiting time corresponds to $c + \Delta c$ and $c - \Delta c$, respectively, where Δc represents the increment of computational elements. The lower Δc is, the slower the reaction of the controller. Ideally, the quickest reaction is desired, but if the control action is too large, it may lead to controller oscillations, instability provoked by switching rapidly and violently between different configurations (namely in this case, over-provisioning and under-provisioning states). Typically, this Δc number of resources is experimentally determined.

Execution. The execution stage provides mechanisms to perform the necessary changes to the system. The proposed plan has to be implemented by performing a series of actions that will elastically provisioning or de-provisioning appropriated resources. Hence, in order for the controller to implement the plan, it has to interact with the middleware to execute those actions and make sure the resulting status of the systems is as planned. Launching a VM may often lead to a considerable delay, which in turn may provoke instability in the controller. In this paper, the architecture described above incorporates a shared pool of VMs that are shared across all the data streams and which removes such a launch delay. As commented before, we assume no shortage of resources for testing our controller. Once an action is taken, the amount of time t required for the controller to achieve the desired state can be characterised as follows: $t = t_d + \frac{L*-L_0}{\mu-\lambda}$, where t_d is the associated time delay for the action, $L*-L_0$ is the variation of the queue size that the action aims to achieve (i.e., $L*$ is the target queue size), and $\mu - \lambda$ is the variation of data elements in the queue. Since the action modifies the computational power (as discussed in the capacity planning, by making use of Eq. 1), it affects the output rate μ. Therefore, there is a trade-off between the number of computational resources involved for the action and the time to reach the target queue size. Moreover, as discussed previously, the quickest reaction is desired, but without generating instabilities in the controller.

4.3 Implementation of the Approach in CometCloud

We validated our controller within the CometCloud system [9], [25]. In order to specify the batch operations to be applied to the data streams and enable this functionality within CometCloud, we make use of streaming workflow technologies, as specified in [22]. We, therefore, designed and implemented a workflow interpreter that can process multiple data streams simultaneously coming from different distributed sources for processing. The workflow interpreter coordinates the processing of the data elements (based on dependency constraints identified in the workflow specifications). The controller from Section 4.2 is integrated with the interpreter so that the system also dynamically scales the required resources on demand, while enforcing the particular QoS requirements for each data stream. For both purposes, the interpreter interacts with CometCloud in two different ways: (i) To dispatch processing data elements to distributed computational resources in CometCloud's federation (by inserting tuples into the CometCloud's tuplespace and by retrieving the results back), and (ii) to monitor and control the computational resources around the tuple space for dynamically scaling them on demand. The role of the queues (buffers) described in Fig. 2 is provided by CometCloud's tuplespace, which is a peer-to-peer overlay that can scale across machines [25]. In this section, we briefly describe these components. However, more details and a Petri net-based specification of the workflow interpreter can be found in our previous work in [22].

Streaming Workflow Interpreter. The streaming workflow interpreter works as follows. It is responsible for creating and enacting the workflow instances and for receiving the data elements of the streams. When a data element arrives in the interpreter, it is injected into the corresponding streaming workflow pipeline instance for processing. Then, as a data element advances through its corresponding workflow pipeline, tuples are written to and retrieved from the CometCloud's tuplespace. Initially, the interpreter allocates a number of worker nodes to each data stream, based on historical executions. These nodes will withdraw workflow tuples, perform the required computations over the data element, and finally they will write the result tuple back into the tuplespace. The result tuple will be taken by the streaming interpreter, which will re-direct it to the corresponding workflow instance.

Hence, the CometCloud's tuplespace acts as a queue for a data stream, where data elements are waiting to be retrieved and processed by worker nodes. CometCloud's middleware components can be configured so that workflow jobs from the same data stream are stored within the same location.

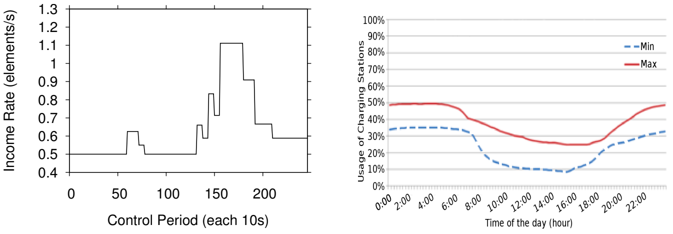
5 Experimental Validation

5.1 Experiment Methodology

We have performed a set of experiments to validate our theoretical formulation. We have considered two scenarios, one using synthetic data and other using real data from electrical vehicle (EV) charging stations [26]. Using this streaming workflow, we want to observe the behaviour of our autonomic strategy under significant changes in the input rate. We consider that the SLA of an streaming workflow is to guarantee the deadline of its data elements on average, while minimizing the number of resources allocated to such streaming workflow. As such, we consider two metrics: (i) The number of data elements that meet the deadline and (ii) the amount of resources used. In order to validate our metrics, we have computed a baseline case where the number of VMs required to maintain the queue size (L) to zero are provisioned. In such a baseline, all of the jobs meet the deadline (except at the initial time as the system is empty), but the number of VM hours is over-provisioned. We compared such a baseline to the performance of our proposal, and we also measured the number of jobs in our proposal that are not meeting the deadline. Further details for our evaluation success criteria can be found in Section 5.4.

Synthetic scenario: In these experiments, we consider a dynamic streaming workflow that changes its incoming data rate (λ) over time, as shown in Fig. 4a. The incoming data rate is based on the number of tweets per second during the 2014 WorldCup Semi-final between Brazil-Germany, obtained from Twitter Data [2]. The streaming workflow starts with a rate of 0.5 jobs (data elements) per second (i.e. one job each 2 seconds), then it increases its incoming rate up to 0.625 jobs per second (i.e. one job each 1.6 seconds) in period 60–72, and keeps varying over time as shown in Fig. 4a. The idea of this set of experiments is twofold: (i) observe the behaviour of our controller when receiving large number of data elements per minute for processing, and (ii) observe how quickly our controller is able to adapt to abrupt changes in the input rate, while minimizing the number of resources used. In these experiments, the execution time for the jobs are generated randomly between 4 and 10 seconds. The duration of the jobs was adjusted considering the limitations of our computational resources and the maximum data incoming rate.

Real scenario: In these experiments, we consider a real workload from EV charging stations. Fig. 4b shows aggregated data of the usage of the charging stations over the day. We used this information to model our workload, which is scaled down to reduce the time needed to execute the experiments. In this scenario, the input rate of the workload is fixed to one data element every 10 seconds, and the deadline for each job is always 300 seconds. The execution time of the incoming jobs varies with the usage of the charging stations. The higher the usage is, the higher the execution time. Thus, the execution time of the data elements received in the rage 0 to 3000 seconds of the experiment is 100 seconds (from 0:00 to 7:00 hours in Fig. 4b); from 3000 to 6000 seconds (from 7:00 to 15:00 hours in Fig. 4b), the execution time is 60 seconds; from 6000 to 6700 is 40 seconds (from 15:00 to 16:30 hours in Fig. 4b); and from 6700 to 9000 seconds is 100 seconds (from 16:30 to 23:00 hours in Fig. 4b).



(a) Arrival Rate of Data Elements

(b) Aggregated Charging Availability in Residential Areas

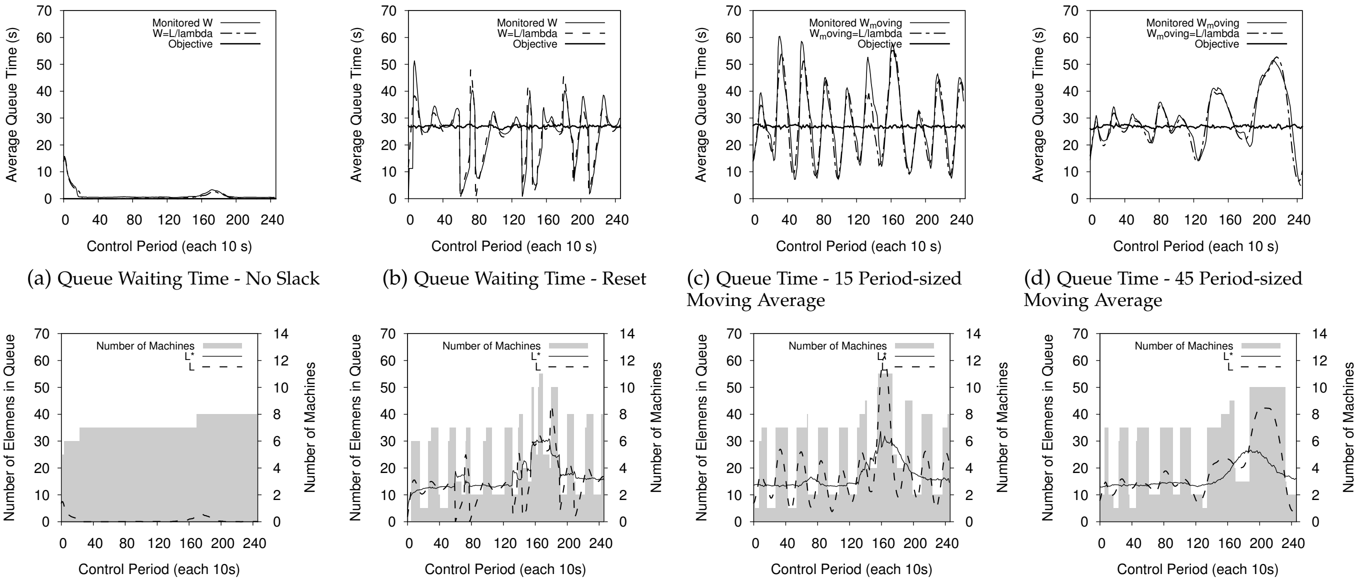
Fig. 4: Scenario Workload

We have deployed an actual testbed between the Universidad de Zaragoza, Spain and Rutgers University, USA. In this testbed, we have the autonomic control layer at Zaragoza and the resource management layer (CometCloud) at Rutgers. Jobs are generated at Zaragoza and sent to Rutgers for computation, which, in turn, sends results and monitoring data back to Zaragoza. Thus, while the actual infrastructure is deployed and operationally ready for real applications, in our experiments, the execution of individual jobs are emulated using the sleep function once they reach their corresponding processing machine. This ensures the reproducibility of the experiments. The computational resources located at Rutgers are part of a cluster-based infrastructure with 32 dedicated cluster machines. Each node has 8 cores, 6 GB memory, 146 GB storage, and a Gigabit Ethernet connection. The measured latency on the network is 0.227ms on average. Two configurations of these infrastructures are used: (i) For the *Synthetic scenario*, we consider an HPC-Cloud infrastructure, where the resources of our cluster are offered using a cloud abstraction and therefore, we can provision and de-provision them elastically and on-demand. Since we are using an HPC-Cloud, where resources are allocated by simply starting workers (e.g., Docker container) on different machines, the overhead of provisioning machines in this infrastructure is not significant and therefore not considered at the moment. (ii) For the *Real scenario*, we consider a Cloud infrastructure, where the resources of our cluster are offered using a cloud abstraction and they incur in provisioning overheads when allocating resources, as in a virtualized cloud. Specifically, we use a provisioning overhead between 60 and 80 seconds, which corresponds to Amazon Web Services cloud [27]. The interconnection network overhead between Zaragoza and Rutgers is 130 ms on average.

5.2 Synthetic Scenario Experiments

In addition to validating the performance of our controller, we also want to validate LL.1 and LL.2 theorems in practice.

[2] https://twitter.com/TwitterData



(a) Queue Waiting Time - No Slack

(b) Queue Waiting Time - Reset

(c) Queue Time - 15 Period-sized Moving Average

(d) Queue Time - 45 Period-sized Moving Average

(e) Queue Length - No Slack

(f) Queue Length - Reset

(g) Queue Length - 15 Period-sized Moving Average

(h) Queue Length - 45 Period-sized Moving Average

Fig. 5: Summary of experiment - Variable λ and S for the synthetic scenario. Action is ± 2 machines over the optimum. Bar plots representing the number of provisioned machines use the right Y axis, while all line plots use the left Y axis.

We have performed a set of experiments using variable execution time for each data element. Since we are working with average results, we recognize that this may hinder the responsiveness of the system under significant changes in the steaming input rate (λ). For this reason, we study two different strategies: (i) Reset: perform a reset of the average values when a burst is identified; and (ii) Moving average: use sliding windows to calculate moving averages. Additionally, we use a baseline strategy that, unlike previous strategies, does not make use of the slack. We call this strategy *no slack*.

5.2.1 Variable Lambda and Variable Execution Time

We have performed a set of experiments using a variable execution time of the data elements (S). According to LL, the autonomic controller should be able to adjust to these changes by looking at S. As we mentioned in the previous section, we consider that processing a job takes between 4 and 10 seconds (i.e. $1/\mu \in [4, 10]$ seconds). Nevertheless, the controller does not know this information a priori. The controller records the observed execution time to calculate the average value to use in the planning. The deadline established by the user is 33 seconds, which means that the maximum time that a job can stay waiting in the queue is between 23 and 29 seconds (this is our slack, W), depending on its execution time. Since measuring the queue time for each data element in our system is not trivial, we use the number of elements in the queue and the incoming rate to estimate the queue waiting time W (see Equation 2 in Section 4.2.1). Using this information, we determine the number of machines we need to allocate or release, see Section 4.2.2. In these experiments, we allocate or release two machines over or under the suboptimal value (Δc) to increase or reduce the queue size. This offset value was obtained empirically for the conditions of our experiments and we leave for future work deducting from monitored data such value. Moreover, we consider that the system is empty at the beginning and has only one machine allocated for this workflow. Fig. 5 collects the experimental results, when using a fixed execution time.

Figs. 5a and 5e present the results of our baseline experiment. In this experiment, our controller tries to satisfy the computational demands by provisioning or deprovisioning resources without considering the use of the slack. We can observe that after the system is initialized, the number of elements in the queue tends to zero – given enough resources in the system. As a consequence, the amount of provisioned resources is maximized.

Figs. 5b and 5f present the results of the experiments using the reset strategy. Fig. 5b depicts how the queue waiting time of the system evolves over time. We can observe how the "W=L/λ" (W calculated using LL) and the "Monitored W" (the actual measured W) oscillate around the objective waiting time, as expected. This shows that LL holds valid even with a variable execution time and that the system is able to adapt itself by autonomously finding the right configuration parameters. At the beginning of the experiment, the system shows a spike in the waiting time due to the low number of resources, but this is quickly acknowledged by our autonomic controller and new resources are provisioned to reduce the waiting time of the data elements. This can be clearly observed in Fig. 5f, where we show the actual size of the queue L and the objective size of the queue $L*$, as well as the number of allocated machines.

After this initial spike, the system allocates new resources, which reduce the size of the queue. When the queue size is under the target, the waiting time is reduced as well, hence complying with the QoS. In this current status (around period 10), we continue reducing the queue size and processing elements faster than required. Therefore, the resource provider is using more resources than needed. This situation is corrected by releasing some resources, which in turn increases the size of the queue and the waiting time. For this reason, the system is continuously oscillating around the objective, but, as we can observe in Fig. 5b, these oscillations are close to the objective. On the control period number 60, the streaming's input rate increases (see Fig. 4a). We assume that our system is able to detect this situation and performs a reset of the average values, recorded until now, to allow for faster adaptability (in Section 6, we discuss about techniques to do this). Similarly, more resets were performed when sudden changes of the input rate were detected. This is represented in Figs. 5b and 5f.

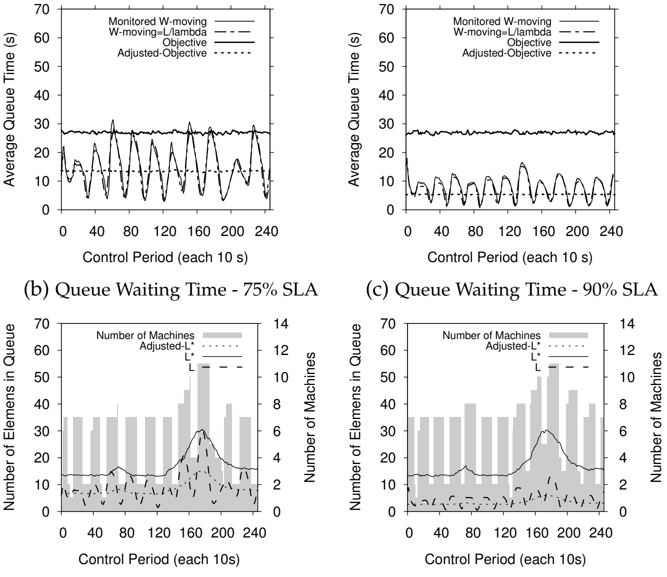
Additionally, we have performed experiments using the moving average strategy. First, we have used a 15 period-sized moving average. Results are shown in Figs. 5c and 5g. Using a small period-sized, moving average has the advantage of quickly realizing of changes in the input rate (λ), but at the same time this ability can be counter-productive if the system becomes unstable due to "short memory". In this case, we can observe that the system regulates itself and it is always oscillating around the optimum. However, as opposed to the reset strategy, we can see that in this case the oscillations are considerably larger, even in stationary conditions (e.g., before period 60). For example, Fig. 5c shows waiting times that goes from around 60 seconds to as low as 10 seconds. Moreover, we can observe that, even with a small window, it takes several periods to realize that a change in the input rate happened, which translates into large number of elements in the queue (L) and hence large waiting times (W). The main reason is that the moving average "softens" the changes in the input rate, as it is clearly shown by the objective queue size ($L*$) in Fig. 5g.

Figs. 5d and 5h show results using a 45 period-sized moving average. In this case, we can observe that the behaviour of the system in stationary conditions is very good and it keeps the size of the queue (L) and therefore the waiting time (W) oscillating closely to the objective. However, since it keeps a large window of values, we can see that after the period 60, the system does not behave well and all values are above the objectives for a long time until the window leaves behind those small values and realizes that has to correct the situation by allocating more resources. In Fig. 5h, we can see that the target queue size ($L*$) changes very slowly due to the large window size and therefore a large number of elements remain over $L*$.

5.2.2 Adjusting QoS

As we mentioned in Section 3.1, the default behaviour of our controller is to achieve that at least a 50 % of the jobs that enter the system are completed within a given deadline. The reason is that LL holds for average values and we use LL to calculate the maximum waiting time (slack) that jobs can be on the queue. Additionally, we can consider that using no slack allows the system to meet the deadline for all jobs. Therefore, we could approximate the amount of slack we want to use to achieve different QoS. For example, one way of doing this is by linear regression: Assuming a linear relationship between the amount of slack and the percentage of jobs that are going to meet the deadline. In this way, using the equation of the line we can set the percentage of jobs that we want to meet the deadline and obtain the adjusted maximum waiting time (or adjusted slack). The equation of the line is typically defined as $y = mx + b$, where m is the slope and b is the y-intercept. Considering our two known points to be $(100, 0)$ and $(50, 1)$ for the cases with 0 slack and 100 % jobs completed within the deadline, and 100 % of the slack and 50 % jobs completed within the deadline. This results in $m = -0.02$ and $b = 2$.

Next, we performed two experiments where the QoS is guaranteeing that at least a 75 % and 90 % of the jobs are completed within the deadline, respectively. For this experiments, we used a 15 Period-sized moving average and scenario described in Fig. 4a. By applying the linear regression discussed, we obtain that to achieve a 75 % of jobs completed, we need to use a 50 % of the slack calculated using LL, while to achieve a 90 % of jobs, we need to limit the use of the slack to just a 20 %. In our experiments this translates into adjusting our objective waiting time (W) and queue size ($L*$). Figs. 6 collects the results. We can observe in Fig. 6b that for the 75 % case, the average waiting time fluctuates around the adjusted objective and only around a 13 % ob jobs missed the deadline. On the other hand, Fig. 6c shows that for the 90 % case, all averages are within the deadline (only one job finished beyond the deadline). Figs. 6d and 6e show that when reducing the amount of slack, the amount of resources allocated increases and progressively approaches our baseline (No Slack). Although the linear assumption resulted to be overly conservative for our tested use case, we conclude that adjusting the slack allows us to adjust the offered QoS.



(b) Queue Waiting Time - 75% SLA

(c) Queue Waiting Time - 90% SLA

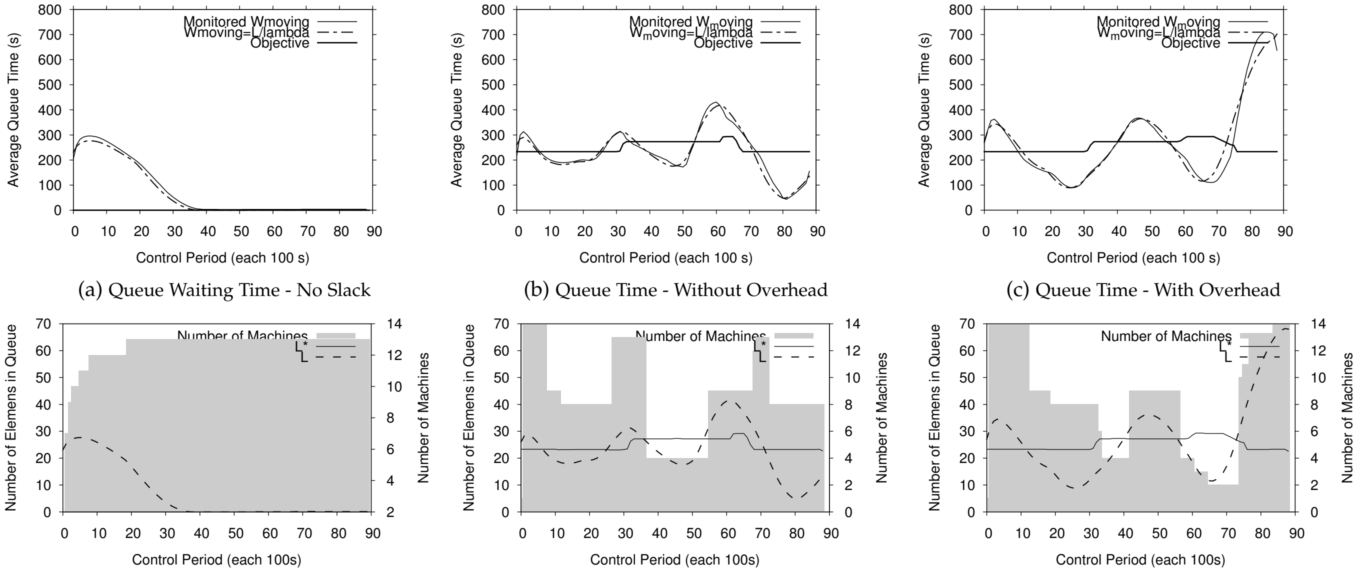
(d) Queue Length - 75% SLA

(e) Queue Length - 90% SLA

Fig. 6: Summary of experiment - Adjusting QoS to 75% and 90% of the elements meeting the deadline. Heterogeneous Execution Time and 15 Period-sized Moving Average. Action is ± 2 machines over the optimum. Bar plots representing the number of provisioned machines use the right Y axis, while all line plots use the left Y axis.

5.3 Real Scenario Experiments

In these experiments, we want to observe the behaviour of our controller in both infrastructures: An HPC-cloud (no provisioning overhead) and a virtualized cloud (with



(a) Queue Waiting Time - No Slack

(b) Queue Time - Without Overhead

(c) Queue Time - With Overhead

(d) Queue Length - No Slack

(e) Queue Length - Without Overhead

(f) Queue Length - With Overhead

Fig. 7: Summary of experiment - Electrical Vehicle Charging Stations. Bar plots representing the number of provisioned machines use the right Y axis, while all line plots use the left Y axis.

provisioning overhead). In these experiments, the execution time of the jobs varies between 40 and 100 seconds, as we described in Section 5.1. We assume that the controller does not know this information a priori and it reacts to these changes using the observed data. For these experiments, we use a 200 period-size moving average. Fig. 7 collects the results of the experiments.

First, Figs. 7d and 7d show the baseline experiment. Again, this experiment shows the maximum number of resources required to process data elements as soon as possible (i.e. minimum time of completion).

Figs. 7c and 7e show the results when using the HPC-cloud infrastructure (no provisioning overhead). As in previous cases, we observe that LL also holds valid during the duration of the experiment. Both figures show how the controller recognizes the changes in the execution time of the workload and changes the objectives, i.e., objective in Fig. 7c and L* in Fig. 7e. This translates into increasing or decreasing the number of provisioned resources to adjust the size of the queue.

Similarly, in Figs. 7c and 7f, we can observe the results when using a virtualized cloud infrastructure (with provisioning overhead). As in the previous case, we see that LL holds valid. However, we observe certain delay caused by the overheads when provisioning resources. At the beginning of the experiment, a larger number of jobs get accumulated in the queue and it takes longer time to correct the situation, which is caused by the provisioning overheads. We also observe how around control period 70, one of the worst-case scenarios for a reactive resource manager. We have very few allocated resources when all of a sudden the execution time of the jobs increases from 40 seconds to 100 seconds. This causes a spike in the number of elements accumulated in the queue. Once again, a large number of resources are provisioned but the overheads delay the effectiveness of our control actions.

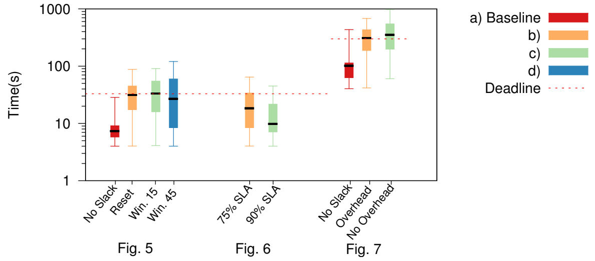


Fig. 8: Data Element Completion Time (waiting time plus execution time). Y axis is in logarithmic scale. Labels a),b),c),and d) identify the experiment by looking at the first row of plots contained in Figs. 5, 6, and 7.

5.4 Completion Time Analysis

In this Section, we analyze the completion time of the data elements processed in each one of the experiments executed before. In this paper, we proposed an approach to leverage the slack of jobs (i.e. remaining of deadline minus execution time and overheads), aiming at minimizing the amount of resources provisioned to satisfy the workload given a specific SLA. In our case, we chose that our SLA was to meet the data elements' deadline on average. Next, we evaluate two specific metrics: (i) The amount of resources used, and (ii) the SLA assurance. Fig. 8 collects these results.

In previous experiments, we observed oscillations around the target value. We can observe now how the median completion time for the baseline approaches is typically very far from the deadline. However, in the rest of the cases – b), c), and d) – all our strategies show how the median completion time is very close to the deadline. This means that when the slack is not used, we are wasting a significant amount of resources. Specifically, in our experiments, in comparison with the baseline, the proposed approach saved between a 32% and a 54 % machine-hours in the experiments of Fig. 5, between 35% and 17% in the experiments of Fig. 6, and between a 47% and a 51% in the experiments of Fig 7. Additionally, the median of the data elements' completion time also tells us that the proposed SLA is satisfied. Although, the dispersion of the completion times is large, at times, our approach managed to complete around 50% of the jobs within the deadline. Furthermore, it is also worth highlighting that there is a trade-off between the interquartile range and the oscillation frequency introduced in the controller: The smaller the interquartile range, the smaller the hysteresis required, and then the higher the oscillation frequency.

6 Related Work

The two biggest fields in which LL has been applied are operations management (OM), and computer science and engineering (CSI) [15]. On the other hand, with the extraordinary development of computer technologies, there is a significant amount of case studies ranging from computer architecture to computer networks and distributed systems. The most significant case studies from OM and CSI are described in [15].

There is a significant amount of work in deadline based scheduling algorithms for multimedia applications in communication networks. In general terms, the function of these scheduling algorithms is to select the session whose head-of-line (HOL) packet is to be transmitted next through the network. This process is based on the QoS requirements. A survey that provides an overview can be found in [28], [29]. In other words, these proposals aim to evenly share the workload (packets) onto a shared resource (the network). In contrast, in our approach, based on the cloud paradigm principles, we adapt the computational power (resources) to the workload: We adaptively provision the number of computational resources to a data stream in accordance to workload variations and resource performance.

A number of studies developed autonomic policies and mechanisms for elasticity in clouds. In [10], the AGILE system provides medium-term resource demand predictions for achieving enough time to scale up the computational resources in advance, minimizing VM launching overheads. In comparison to our approach, AGILE is application agnostic and does not consider the characteristics of streaming applications. For the NoSQL cluster scenario, TIRAMOLA was presented in [11]. It can self-resize a NoSQL cluster according to user-defined policies. Decisions on (de-)allocating VMs from a cluster are modelled as a Markov Decision Process and taken in real-time. Autoflex [12] is a service agnostic system for autonomic scaling of VMs that combines both reactive and proactive approaches. A purely reactive resource provisioning approach was proposed in [13] under the YinzCam system, which provides cloud-hosted service for real-time Web scores, news, etc. The workload considered exhibits significant spikes. Hence, the controller is designed so that the scaling up action is done much faster than the scaling down. Again, this approach is designed to enable service operations to be on-line and responding without any time-slack. More recently, vertical scalability was also studied in [14] by means of different performance models that enable to map performance to capacity. Autonomic computing was studied to provide opportunistic in-transit processing by taking advantage of the estimated "slack" that is available at different stages of a data-intensive workflow [30].

The increasing deployment of sensor network infrastructures has led to large volumes of data becoming available, leading to new challenges in storing, processing and transmitting such data [31]. For that reason, stream processing frameworks such as Yahoo's S4 [32], or IBM InfoSphere Streams [33] provide streaming programming abstractions to build and deploy jobs as distributed applications at scale for commodity clusters and clouds. Nevertheless, even that these systems support high input data rates; they do not consider variable input rates, which is our focus in this paper. In some other approaches, the parallelism is extracted from the data stream query operators they provide, Aurora [34], Borealis [35] and Stream Cloud [36], which differs that in our case, we explicitly exploit the parallelism by having multiple data elements in multiple workflow pipelines. In this area, Spark has popularized the idea of discretized streams to process streams as a sequence of discrete micro-batches, which improves fault recovery [37]. These micro-batches are dynamically allocated across workers based on data locality and availability. While Spark assumes a ready-to-use cluster of workers, our autonomic approach intends to elastically provision and de-provision machines to minimise the operational costs of processing the streams.

Our work is closely related to three approaches. In [38], the goal is to allocate resources dynamically from a cloud, so that the processing rate can match the rate of data arrival. They also consider variable transient input rates. Our approach is more general; as such a case corresponds in our approach to a scenario where the time slack is zero. Moreover, we make use of a federation of heterogeneous resources and we propose autonomic based mechanisms and policies for the selection of resources. In [39], the authors propose a workflow specification where each job consists of one or more alternate implementations with different non-functional properties, so that the system can choose any of them dynamically at runtime. In this paper, we have not considered dynamism at workflow-level, but our dynamic provisioning of resources is accomplished in a federation of heterogeneous resources. Finally, the work in [40]–[42] consists of a sequence of nodes, where each node has multiple data buffers and computational resources – whose numbers can be adjusted in an elastic way. They utilize the token bucket model for regulating data injection rates into such nodes. As before, they do not consider time slacks. Another important difference to our approach is that instead of utilizing multiple nodes, we assume CometCloud system as a coordination mechanism that can outsource the computation when required.

Finally, the problem of detecting spikes in cloud workloads can be beneficial not only for the purpose in this paper of resetting monitoring average values and starting a new monitored period, but also for making proactive resource management decisions. Indeed, unanticipated changes in workload characteristics can potentially lead to service slowdown and might end in service-failure due to insufficient resource allocation. In [24], the authors investigate methods for detecting spikes in cloud workloads. In particular, they developed methods that make use of signal processing techniques. Previous efforts have also been made on modeling and characterizing workloads and spikes. The work in [43] presents a detailed workload characterization study of the 1998 World Cup Web site. In [44], the authors analyse a number of real workload and data spikes and from the results they propose and validate a model of stateful spikes that allow them to synthesize volume and data spikes, and that can be used for cloud providers.

7 Discussion

The results presented in this paper show the feasibility of using LL for capacity planning of cloud-like resources for finite periods of time under non-stationary conditions. We observed how our controller is able to react upon changes in the data element input rate of a stream. By making use of the *time slack* of each data element (the time that a data element can spend on a queue before processing), we are able to guarantee the QoS of a stream, while minimizing the amount of resources used and hence the operational costs.

We have observed that our controller also has to deal with certain "inertia" that delays the effect of our control actions. Such an "inertia" is influenced by two factors: (i) the effect that average values can have upon variabilities in the environment (i.e. bursty conditions of λ and unexpected performance of VMs), and (ii) the actual processing time that, once an action is taken, it delays the effect of the action on the queue size. In order to mitigate the effects of long running averages and allow the system to rapidly react upon changes, we propose to limit the amount of data used to take operational decisions. Two alternative approaches can be taken, i.e. either to reset average values upon detecting variability, hence, the overall execution time is divided in sub-periods of execution (a sub-period starts when a reset is the average values is accomplished), or to compute moving averages. The effect caused by the processing time cannot be avoided, but if the control period of the controller is not long enough, it could even lead to an unstable system that is constantly oscillating, by drastically allocating and deallocating resources.

From our experiments, we have also observed that the discretization of our control action (i.e., translating an optimal number of resources to provision or de-provision into the closest natural numbers space) can lead to suboptimal solutions. This issue can be addressed by considering heterogeneous resources that can be categorized – similar to the flavours or type of instances we find in cloud infrastructure, to allow finer grained decisions that lead to a more efficient use of our resources. Finally, we introduced overheads when provisioning resources, like the ones typically found in virtualized clouds. As expected, these overheads caused a delay between a control action and its observable effect in the workload of the system. Naturally, the larger the overheads the larger the amount of time needed for our control actions to take effect. This is a well-known issue in resource management and can only be solved by using predictive techniques that allow us to provision resources ahead of time.

8 Conclusions and Future Work

The proliferation of geographically distributed sensors has led to large volumes of data becoming available and to a great number of applications that need to process such data volumes with a number of purposes such as surveillance and monitoring, *smart-* traffic management, cities, and energy management in buildings. Typically, these applications require that when data elements arrive, they are processed within a time threshold (deadline specified in their Service Level Agreement, SLA). Moreover, the processing may involve the execution of complex simulations or control algorithms that are typically computationally intensive and that are often executed as *batch* processes.

In this paper, we propose an autonomic queueing theory-based controller for provisioning computational resources (generally VMs from a cloud IaaS) to a number of such data stream applications. The controller elastically provisions VMs to meet performance targets associated with a particular data stream. In particular, when data elements arrive, they are buffered until there are VMs available. The controller controls the time data elements are buffered by (de-)allocating VMs to a data stream. With such actions, it achieves that data elements spend the maximum time on the queue, so that, together with the processing time, the time threshold (deadline) is not violated and the VMs are minimised. Besides, rather than monitoring waiting times, as information about waiting time of data elements may not always be available or may be very difficult to obtain, the controller makes use of LL to derive waiting times (W) from queue size (L) and arrival rate (λ). The waiting time on the queue, however, can vary depending on data bursts or performance variations during processing. The controller monitors the input rates and the execution times, periodically computes the target waiting times (queue sizes), and regulates allocated VMs accordingly to meet SLAs. We implemented our controller on top of the CometCloud system in a real federated cloud scenario and validated it through real executions.

As future work, we expect to consider different types of resources (e.g., VMs with different performance capabilities), so that the efficiency of the controller is improved. We are also planning to use some of the strategies mentioned in the related work to detect changes in the workload.

Acknowledgment: This work was supported in part by: The Industry and Innovation department of the Aragonese Government and European Social Funds (COSMOS research group, ref. T93), the Spanish Ministry of Education, (Framework Program CEI Iberus – Universidad de Zaragoza, for faculty staff, mobility call 2014), the Spanish Ministry of Economy (program "Programa de I+D+i Estatal de Investigación, Desarrollo e innovación Orientada a los Retos de la Sociedad" – TIN2013-40809-R), NSF via grants numbers ACI 1339036, ACI 1441376. The research at Rutgers was conducted as part of the Rutgers Discovery Informatics Institute (RDI2).

References

[1] A. Anjum, T. Abdullah, M. Tariq, Y. Baltaci, and N. Antonopoulos, "Video stream analysis in clouds: An object detection and classification framework for high performance video analytics," *IEEE Transaction on Cloud Computing*, 2016 - to appear.

[2] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna, "Adaptive rate stream processing for smart grid applications on clouds," in *Intl. workshop on Scientific cloud computing*, 2011, pp. 33–38.

[3] I. Petri, O. Rana, Y. Rezgui, H. Li, T. Beach, M. Zou, J. Diaz-Montes, and M. Parashar, "Cloud supported building data analytics," in *CCGrid*, 2014, pp. 641–650.

[4] C. Herath and B. Plale, "Streamflow programming model for data streaming in scientific workflows," in *CCGrid*, 2010, pp. 302–311.

[5] P. Mell and T. Grance, "The nist definition of cloud computing," *NIST*, vol. 53, no. 6, p. 50, 2009.

[6] J. O'Loughlin and L. Gillam, "Performance evaluation for cost-efficient public infrastructure cloud use," in *Intl. Conf. on Economics of Grids, Clouds, Systems, and Services GECON, Cardiff, UK*, 2014.

[7] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 460–471, Sep. 2010.

[8] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, "Modeling virtual machine performance: Challenges and approaches," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 3, pp. 55–60, Jan. 2010.

[9] J. Diaz-Montes, M. Diaz-Granados, M. Zou, S. Tao, and M. Parashar, "Supporting data-intensive workflows in software-defined federated multi-clouds," *IEEE Transactions on Cloud Computing*, 2015-to appear.

[10] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: elastic distributed resource scaling for infrastructure-as-a-service," in *ICAC'13, San Jose, CA, USA*, 2013, pp. 69–82.

[11] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, elastic resource provisioning for nosql clusters using TIRAMOLA," in *CCGrid, Delft, Netherlands*, 2013.

[12] F. J. A. Morais, F. V. Brasileiro, R. V. Lopes, R. A. Santos, W. Satterfield, and L. Rosa, "Autoflex: Service agnostic auto-scaling framework for iaas deployment models," in *CCGrid, Delft, Netherlands*, 2013.

[13] N. D. Mickulicz, P. Narasimhan, and R. Gandhi, "To auto scale or not to auto scale," in *ICAC, San Jose, CA*, 2013, pp. 145–151.

[14] E. B. Lakew, K. Cristian, H.-R. Francisco, and E. Erik, "Towards faster response time models for vertical elasticity," in *IEEE/ACM UCC, London, UK*, 2014, pp. 560–565.

[15] J. D. C. Little, "OR FORUM - little's law as viewed on its 50th anniversary," *Operations Research*, vol. 59, no. 3, pp. 536–549, 2011.

[16] S.-H. Kim and W. Whitt, "Statistical analysis with little's law," *Operations Research*, vol. 61, no. 4, pp. 1030–1045, 2013.

[17] L. Kleinrock, *Theory, volume 1, Queueing systems*. Wiley-interscience, 1975.

[18] P. Papadopoulos, N. Jenkins, L. Cipcigan, I. Grau, and E. Zabala, "Coordination of the charging of electric vehicles using a multi-agent system," *IEEE Transactions on Smart Grid*, vol. 4, no. 4, pp. 1802–1809, 2013.

[19] I. Petri, M. Zou, A. Zamani, J. Diaz-Montes, O. Rana, and M. Parashar, "Integrating software defined networks within a cloud federation," in *IEEE/ACM CCGrid*, 2015, pp. 179–188.

[20] "Fish4Knowledge Project. http://fish4knowledge.eu/."

[21] C. Pautasso and G. Alonso, "Parallel computing patterns for Grid workflows," in *Workshop on Workflows in Support of Large-Scale Science, Paris, France*, 2006.

[22] R. Tolosana-Calasanz, J. Diaz-Montes, O. F. Rana, and M. Parashar, "Extending cometcloud to process dynamic data streams on heterogeneous infrastructures," in *Intl. Conf. on Cloud and Autonomic Computing (ICCAC)*, 2014.

[23] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Surfing wavelets on streams: One-pass summaries for approximate aggregate queries," in *Intl. Conf. on Very Large Data Bases*, 2001.

[24] A. Mehta, J. Durango, J. Tordsson, and E. Elmroth, "Online spike detection in cloud workloads," in *IEEE Intl. Conf. on Cloud Engineering, IC2E, Tempe, AZ, USA*, 2015, pp. 446–451.

[25] J. Diaz-Montes, M. AbdelBaky, M. Zou, and M. Parashar, "Cometcloud: Enabling software-defined federations for end-to-end application workflows," *IEEE Internet Computing*, vol. 19, no. 1, 2015.

[26] "The EV Project. https://avt.inl.gov/project-type/ev-project. last time accessed aug. 2016."

[27] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, in *1st Intl. Conf. on Cloud Computing (CloudComp), Munich, Germany*, 2010.

[28] H. Fattah and C. Leung, "An overview of scheduling algorithms in wireless multimedia networks," *Wireless Communications, IEEE*, vol. 9, no. 5, pp. 76–83, 2002.

[29] R. Guérin and V. Peris, "Quality-of-service in packet networks: basic mechanisms and directions," *Computer Networks*, vol. 31, no. 3, pp. 169–189, 1999.

[30] V. Bhat, "Autonomic management of data streaming and in-transit processing for data intensive scientific workflows," Ph.D. dissertation, Rutgers University, 2008.

[31] L. Golab and M. T. Özsu, "Issues in data stream management," *SIGMOD Rec.*, vol. 32, no. 2, pp. 5–14, 2003.

[32] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *IEEE Intl. Conf. on Data Mining Workshops (ICDMW)*, 2010, pp. 170–177.

[33] A. Biem, E. Bouillet, H. Feng *et al.*, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in *ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1093–1104.

[34] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," in *1st Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, CA, 2003.

[35] D. J. Abadi, Y. Ahmad, M. Balazinska *et al.*, "The Design of the Borealis Stream Processing Engine," in *2nd Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, CA, 2005.

[36] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez, "Streamcloud: A large scale data streaming system," in *IEEE Intl. Conf. on Distributed Computing Systems*, 2010, pp. 126 –137.

[37] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM Symp. on Operating Systems Principles*, 2013, pp. 423–438.

[38] S. Vijayakumar, Q. Zhu, and G. Agrawal, "Dynamic resource provisioning for data streaming applications in a cloud environment," in *IEEE CloudCom*, 2010, pp. 441–448.

[39] A. G. Kumbhare, Y. Simmhan, and V. K. Prasanna, "Exploiting application dynamism and cloud elasticity for continuous dataflows," in *SC'13, Denver, CO, USA*, 2013.

[40] R. Tolosana-Calasanz, J. A. Bañares, and O. F. Rana, "Autonomic streaming pipeline for scientific workflows," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 16, pp. 1868–1892, 2011.

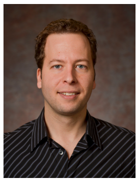
[41] J. Á. Bañares, O. F. Rana, R. Tolosana-Calasanz, and C. Pham, "Revenue creation for rate adaptive stream management in multi-tenancy environments," in *GECON*, 2013, pp. 122–137.

[42] R. Tolosana-Calasanz, J. Á. Bañares, C. Pham, and O. F. Rana, "Enforcing qos in scientific workflow systems enacted over cloud infrastructures," *Journal of Computer and System Sciences*, vol. 78, no. 5, pp. 1300–1315, 2012.

[43] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *IEEE Network*, vol. 14, no. 3, pp. 30–37, 2000.

[44] P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *1st ACM Symposium on Cloud Computing (SoCC), Indianapolis, Indiana, USA*, 2010, pp. 241–252.

Rafael Tolosana-Calasanz received his PhD in 2010 from the University of Zaragoza. He is currently an Associate Professor at the Computer Science and Systems Engineering Department at the University of Zaragoza. His research interests lie in the intersection of Distributed and Parallel Systems, and Problem Solving Environments.



Javier Diaz-Montes is a Assistant Research Professor at Rutgers University and a member of the Rutgers Discovery Informatics Institute (RDI2). He received his PhD degree in Computer Science from the Universidad de Castilla-La Mancha, Spain ("Doctor Europeus", 2010). Before joining Rutgers, he was Postdoctoral Fellow at Indiana University. His research interests are in the area of parallel and distributed computing, including autonomic and computing, virtualization, and scheduling.

Omer F. Rana is a Professor of Performance Eng. in School of Computer Science & Informatics at Cardiff University and leads the "Complex Systems" research group. He holds a Ph.D. in ”Neural Computing and Parallel Architectures" from Imperial College. Prior to joining Cardiff University, he worked as a software developer at Marshall BioTechnology Limited. His research interests extend to three main areas within computer science: problem solving environments, high performance agent systems and novel algorithms for data analysis and management.

Manish Parashar is Distinguished Professor of Computer Science at Rutgers University. He is also the founding Director of the Rutgers Discovery Informatics Institute (RDI2). His research interests are in the broad areas of Parallel and Distributed Computing and Computational and Data-Enabled Science and Engineering. Manish serves on the editorial boards and organizing committees of a large number of journals and international conferences and workshops, and has deployed several software systems