An Experimental Performance Evaluation of Autoscalers for Complex Workflows

Ilyushkin, Alexey; Ali-Eldin, Ahmed; Herbst, Nikolas; Bauer, André; Papadopoulos, Alessandro; Epema, Dick; Iosup, Alexandru

DOI
10.1145/3164537

Publication date
2018

Document Version
Accepted author manuscript

Published in
ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)

An Experimental Performance Evaluation of Autoscalers for Complex Workflows

ALEXEY ILYUSHKIN, Delft University of Technology
AHMED ALI-ELDIN, Umeå University and UMass, Amherst
NIKOLAS HERBST and ANDRÉ BAUER, University of Würzburg
ALESSANDRO V. PAPADOPOULOS, Mälardalen University
DICK EPEMA, Delft University of Technology
ALEXANDRU IOSUP, Vrije Universiteit Amsterdam and Delft University of Technology

Elasticity is one of the main features of cloud computing allowing customers to scale their resources based on the workload. Many autoscalers have been proposed in the past decade to decide on behalf of cloud customers when and how to provision resources to a cloud application based on the workload utilizing cloud elasticity features. However, in prior work, when a new policy is proposed, it is seldom compared to the state-of-the-art, and is often compared only to static provisioning using a predefined QoS target. This reduces the ability of cloud customers and of cloud operators to choose and deploy an autoscaling policy as there is seldom enough analysis on the performance of the autoscalers in different operating conditions and with different applications. In our work, we conduct an *experimental* performance evaluation of autoscaling policies, using as workflows, a commonly used formalism for automating resource management for applications with well-defined yet complex structures. We present a detailed comparative study of general state-of-the-art autoscaling policies, along with two new workflow-specific policies. To understand the performance differences between the seven policies, we conduct various experiments and compare their performance in both pairwise and group comparisons. We report both individual and aggregated metrics. As many workflows have deadline requirements on the tasks, we study the effect of autoscaling on workflow deadlines. Additionally, we look into the effect of autoscaling on the accounted and hourly-based charged costs, and evaluate performance variability caused by the autoscaler selection for each group of workflow sizes. Our results highlight the trade-offs between the suggested policies, how they can impact meeting the deadlines, and how they perform in different operating conditions, thus enabling a better understanding of the current state-of-the-art.

CCS Concepts: **•General and reference →Cross-computing tools and techniques; •Networks →Cloud computing; •Computer systems organization →Self-organizing autonomic computing; •Software and its engineering →Virtual machines;**

This is a significantly extended and more comprehensive version of our ICPE 2017 article [28]. This work is supported by the Dutch projects Vidi MagnaData and KIEM KIESA, by Commit and the Commit projects IV-E and Commissioner, and by Research Group of the Standard Performance Evaluation Corporation (SPEC). Ali-Eldin is funded by the Swedish Research Council (VR) project Cloud Control, the Swedish Government's strategic research project eSSENCE, and NSF grant #1422245. N. Herbst and A. Bauer are funded by the German Research Foundation (DFG) under grant No. KO 3445/11-1.

Authors' addresses: A. Ilyushkin and D. Epema, EEMCS, Distributed Systems, P.O. Box 5031, 2600 GA, Delft, The Netherlands; emails: a.s.ilyushkin@tudelft.nl, d.h.j.epema@tudelft.nl; A. Ali-Eldin, Plan 4 MIT-huset B443, Umeå universitet 901 87, Umeå, Sweden, email: ahmeda@cs.umu.se; N. Herbst and A. Bauer, Informatik II, Universität Würzburg, Am Hubland, D-97074 Würzburg, Germany, emails: nikolas.herbst@uni-wuerzburg.de, andre.bauer@uni-wuerzburg.de; A.V. Papadopoulos, Högskoleplan 1, 721 23, Västerås, Sweden, email: alessandro.papadopoulos@mdh.se; A. Iosup, VU Amsterdam, WN Building, P4.14, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands, email: a.iosup@vu.nl.

Additional Key Words and Phrases: Autoscaling, Elasticity, Scientific Workflows, Benchmarking, Metrics

ACM Reference format:
Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Andre Bauer, Alessandro V. Papadopoulos, Dick Epema, and Alexandru Iosup. 2017. An Experimental Performance Evaluation of Autoscalers for Complex Workflows. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 30 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Cloud computing emerged as a computing model where computing services and resources are outsourced on an on-demand pay-per-use basis. To make this model useful for a variety of customers, cloud operators try to simplify the process of obtaining and managing the provided services. To this end, cloud operators make available to their customers various autoscaling policies (*autoscalers*, *AS*), which are essentially parametrized cloud-scheduling algorithms that dynamically regulate the amount of resources allocated to a cloud application based on the load demand and the Quality-of-Service (QoS) requirements typically set by the customer. Many autoscalers have been proposed in the literature, both general autoscalers for request-response applications [5, 11, 22, 43, 53] and autoscalers for more task- and structure-oriented applications such as workflows [3, 9, 13, 16, 19, 46]. The selection of an appropriate autoscaling policy is crucial, as a good choice can lead to significant performance and financial benefits for cloud customers, and to improved flexibility and ability to meet QoS requirements for cloud operators. Selecting among the proposed autoscalers is not easy, which raises the problem of finding a systematic, method-based approach to comprehensively evaluate and compare autoscalers. The lack of such an approach derives in our view from ongoing scientific and industry practice. For the past decade, much academic work has focused on building basic mechanisms and autoscalers for specific applications, with very limited work spent in comparisons with the state-of-the-art. In industry settings, much attention has been put on building cloud infrastructures that enable autoscaling as a mechanism, and relatively less on providing good libraries of autoscalers for customers to choose from. (The authors' own prior work reflects this situation [26, 45].) To alleviate this problem, in this work we propose and use the first systematic experimental method to evaluate and compare the performance of autoscalers using workflow-based workloads running in cloud settings as a use-case application.

Modern workflows have different structures, sizes, task types, run-time properties, and performance requirements, and thus raise specific and important challenges in assessing the performance of autoscalers: *How does the performance of general and of workflow-specific autoscalers depend on workflow-based workload characteristics?* Among the many application types, our focus on workflow-based workloads is motivated by three aspects. First, there is an increasing popularity [49, 50] of workflows for science and engineering [1, 33, 35], big data [36], and business applications [51], and by the ability of workflows to express complex applications whose interconnected tasks can be managed automatically on behalf of cloud customers [32]. Second, although general autoscalers focus mainly on QoS aspects, such as throughput, response-time and cost constraints, some autoscalers take into account application structure [39]. One of the main questions we want to answer with this work is: *How does the performance of general and of workflow-specific autoscalers differ?*

Another interesting aspect that arise with workflow scaling is the effect of the autoscalers on workflow deadlines. Many of the workloads have deadlines, basically a bound on the tolerated time by the user between the workflow submission and when the computation results are available. Having enough capacity in the system for the workflows to finish their processing before the deadline can be severely affected by the presence of autoscalers. For that we enforce per-workflow and per-workload deadlines which allow us to see: *How does each autoscaler affect deadline violations?*

One of the core properties of an autoscaler is the ability to minimize operational costs while keeping the required performance level. We calculate the incurred costs for each considered autoscaler which allow us to answer the questions: *How do the autoscalers affect charged and accounted costs?*, and *How do the autoscalers find a balance between performance and cost?*

Towards addressing the aforementioned questions, our contribution is three-fold:

(1) We design a comprehensive method for evaluating and comparing autoscalers (Sections 2–4). Our method includes a model for elastic cloud platforms (Section 2), identifying a set of relevant metrics for assessing autoscaler performance (Section 3), and a taxonomy and survey of exemplary general and workflow-specific autoscalers (Section 4).

(2) Using the method, we comprehensively and experimentally quantify the performance of 7 general and workflow-specific autoscalers, for more than 15 performance metrics (Section 5). We show the differences between various policy types, analyze parametrization effects, evaluate the influence of workload characteristics on individual performance metrics, and explain the reasons for the performance variability we observe in practice.

(3) We also compare the autoscalers systematically (Section 8), through 3 main approaches: a pair-wise comparison specific to round-robin tournaments, a comparison of fractional differences between each system and an ideal system derived from the experimental results, and a head-to-head comparison of several aggregated metrics.

This article extends of our previous work on evaluating the performance of autoscalers for workflows [28]. In this article, we provide a deeper analysis of our previous experimental results for the same set of autoscalers. We include additional user-oriented metrics, explore cost models that distinguish accounted from charged CPU-hours, introduce enforced deadline-based SLAs with two types of soft deadlines, add the analysis of performance variability of user-oriented metrics, and extend the competitions for grading the autoscalers.

2 A MODEL FOR ELASTIC CLOUD PLATFORMS

Autoscaling is an incarnation of the dynamic provisioning problem that has been studied in the literature for over a decade [10]: many autoscalers in essence try to solve the problem of how much capacity to provision given a certain QoS, state-of-the-art algorithms published make different assumptions on the underlying environment, mode of operation, or workload used. It is thus important to identify the key requirements of all algorithms, and establish a fair cloud system for comparison.

2.1 Requirements

In order to improve the QoS and decrease costs of a running application, an ideal autoscaler proactively predicts and provisions resources such that: a) there is always enough capacity to handle the workload with no under-provisioning affecting the QoS requirements; b) the cost is kept minimal by reducing the number of resources not used at any given time, thus reducing over-provisioning; and c) the autoscaler does not cause consistency and/or stability issues in the running applications.

Since there are no perfect predictors, no ideal autoscaler exists. There is thus a need to have better understanding of the capabilities of the various available autoscalers in comparison to each other. For our work, we classify autoscaling algorithms in two major groups: general and workflow-specific. Examples of general autoscalers include algorithms for allocating virtual machines (VMs) in datacenters. They are general because they mostly take their decisions using only external properties of the controlled system, e.g., workload arrival rates, or the output from the system, e.g., response time. In contrast, workflow-specific autoscalers base their decisions on detailed knowledge about the

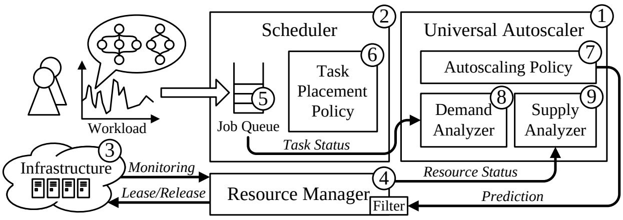


Fig. 1. Elastic cloud platform.

running workflow structure, job dependencies, and expected runtimes of each task [41]. Often, the autoscaler is integrated with the task scheduler [39].

Although many autoscaling algorithms targeting different use case scenarios have been proposed in the literature, they are rarely compared to previously published work. In addition, they are usually tested on a limited set of relatively short traces. Many autoscaling-related articles seldom go beyond meeting some predefined QoS bound, e.g., with respect to response time or throughput, that is often set (artificially) by the authors. Although the performance of many autoscalers is very dependent on how they are configured, this configuration is rarely discussed. To the best of our knowledge, there are no major comparative studies that analyse the performance of various autoscalers in realistic environments with complex applications. Our work aims to fill this gap.

2.2 Architecture Overview

Keeping the diversity of used cloud applications and underlying computing architectures in mind, we setup an elastic cloud platform architecture (Figure 1), which allows for comparable experiments by providing relatively equal conditions for different autoscaling algorithms and different workloads. The equal size of the virtual computing environment, which is agnostic to the used application type, is the major common property of the system in our model. We believe that our architecture represents modern elastic cloud platforms properly and reflects approaches used in many commercial solutions.

While our experiments should be valid for any cloud platform, we decided not to run any experiments on public clouds for two main reasons. First, scientific workflows have been shown to be cost-inefficient on public clouds [30, 55]. Secondly, as this project aims to fairly benchmark the performance of autoscalers, public clouds will introduce variability due to the platforms as public cloud VM performance can vary considerably [30].

The core of our system is the autoscaling service (Component 1 in Figure 1) that runs independently as a REST service. The experimental testbed consists of a scheduler (Component 2) and a virtual infrastructure service (Component 3) which maintains a set of computing resources. A resource manager (Component 4) monitors the infrastructure and controls the resource provisioning. Users submit their complex jobs directly to the scheduler which maintains a single job queue (Component 5). The tasks from the queued jobs are mapped on the computing resources in accordance to the task placement policy (Component 6). The scheduler periodically calls the autoscaling service providing it with monitoring data from the last time period. We refer to this period as the *autoscaling interval*.

The autoscaling service implements an autoscaling policy (Component 7) and has a demand analyzer (Component 8) which uses information about running and queued jobs to compute the momentary demand value. The supply analyzer (Component 9) computes the momentary supply value by analyzing the status of computing resources. The autoscaling service responds to the scheduler with the predicted number of resources which should be allocated or deallocated. Before applying the prediction, the resource manager filters it trimming the obtained value by the maximal number of available resources. To avoid error accumulation, the autoscaling interval is usually chosen so that the provisioning actions made during the autoscaling interval has already taken effect. Thus it is guaranteed that the provisioning time is always shorter than the autoscaling interval. In case when the provisioning time is longer than the autoscaling interval, the resource manager should apply the prediction only partially considering the number of "straggling" resources. In practice, it means that the resource manager should consider booting VMs as fully provisioned resources.

2.3 Workflows as Main Applications

For our experiments, we use complex workflows as our system workload. A *workflow* is a set of tasks with precedence constraints among them. Alternatively it can be represented as a Directed Acyclic Graph (DAG). Each workflow task can start execution when all of its input constraints are satisfied (e.g., when input files are ready). Each task can have multiple inputs and multiple outputs. The precedence constraints make workflow scheduling non-work-conserving as there may be idle processors in the system while there are no waiting tasks with all their dependencies satisfied. This property is one of the reasons we selected workflows for our experiments, since it puts an autoscaling algorithm in more stringent conditions. Additionally, depending on the DAG structure, workflows can also reflect the behavior of other popular job types such as web-requests, bags-of-tasks, or parallel applications. One whole workflow in our setup is considered as a job. The *size* of a workflow is defined by its number of tasks. We assume that every workflow has a single entry node and a single exit node. The *critical path* of a workflow is the longest path from its entry node to its exit node. The *critical path length* is the sum of all task runtimes and inter-task communication times along the critical path. We focus on workflows where tasks require a single processor core only.

2.4 Deadlines for Workflows

Scheduling of workflows is often time critical and involves meeting deadlines for the processing times. For example, workflows for processing satellite data should handle the received information while the satellite makes a turn around the planet [18, 47]. Such workflows should finish before a new batch of information is received. Another example is the case of modern cloud services where the user pays per time slot and the deadlines are bounded to the lengths of these time slots [40]. We assume that deadlines for workflows is set per-workflow and per-workload basis. Per-workflow deadlines are unique for every workflow and are normally assigned based on user (statistical) estimates of the possible workflow runtimes. Per-workload deadlines are common when processing batches of workflows. In this case, a per-workload deadline applies to all the workflows in the workload. We consider soft deadlines which can be violated without affecting the execution of a workflow. Soft deadlines is a measure that can additionally reflect induced infrastructure costs for users.

3 PERFORMANCE METRICS FOR ASSESSING AUTOSCALERS

We use both system- and user-oriented evaluation metrics to assess the performance of the autoscalers. The system-oriented metrics quantify over-provisioning, under-provisioning, and stability of the provisioning, all of which are commonly used in the literature [5, 6, 26, 27]. All the considered system-oriented metrics are based on the analysis of discrete supply and demand curves. The user-oriented metrics assess the impact of autoscaler usage on the workflow execution speed.

3.1 Supply and Demand

The resource *demand* induced by a load is understood as the minimal amount of resources required for fulfilling a given performance-related service level objective (SLO). In the context of our workflow model, a resource can only process one task at a time. We thus define the momentary demand as the

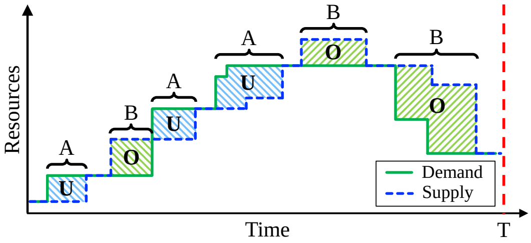


Fig. 2. The supply and demand curves illustrating the under- and over-provisioning periods (A and B) quantified in the number of resources (areas U and O).

number of eligible and running tasks in all the queued workflows. Extending the model to include resource sharing is trivial by using the average number of tasks processed by a resource instead.

Accordingly, the *supply* is the monitored number of provisioned resources that are either idle, booting or processing tasks. Figure 2 shows an example of the two curves. If demand exceeds supply, there is a shortage of available resources (under-provisioning) denoted by intervals A and areas U in the figure. In contrast, over-provisioning is denoted by intervals B and areas O.

3.2 Accuracy

Let the resource demand at a given time t be d_t, and the resource supply s_t. The average *under-provisioning accuracy* metric a_U is defined as the average fraction by which the demand exceeds the supply. Similarly, *over-provisioning accuracy* a_O is defined as the average fraction by which the supply exceeds the demand. Both metrics can be computed as:

$$a_U := \frac{1}{T \cdot R} \sum_{t=1}^{T} (d_t - s_t)^+, \qquad a_O := \frac{1}{T \cdot R} \sum_{t=1}^{T} (s_t - d_t)^+,$$

where T is the time horizon of the experiment expressed in time steps, R is the total number of resources available in the current experimental setup, and $(d_t - s_t)^+ := \max(d_t - s_t, 0)$, i.e., only the positive values of $d_t - s_t$. The intuition behind the two accuracy metrics is shown in Figure 2. Under-provisioning accuracy a_U is equivalent to summing the areas U where the resource demand exceeds the supply normalized by the duration of the measurement period T. Similarly, the over-provisioning accuracy metric a_O is the sum of areas O where the resource supply exceeds the demand.

It is also possible to normalize the metrics by the actual resource demand, obtaining therefore a normalized, and more fair indicator. In particular, the two metrics can be modified as:

$$\bar{a}_U := \frac{1}{T} \sum_{t=1}^{T} \frac{(d_t - s_t)^+}{\max(d_t, \varepsilon)}, \qquad \bar{a}_O := \frac{1}{T} \sum_{t=1}^{T} \frac{(s_t - d_t)^+}{\max(d_t, \varepsilon)},$$

with $\varepsilon > 0$; in our setting we selected $\varepsilon = 1$. The normalized metrics are particularly useful when the resource demand has a large variance over time, and it can assume both large and small values. In fact, under-provisioning of 1 resource unit when 2 resource units are requested is much more harmful than under-provisioning 1 resource unit when 1000 resource units are requested. Therefore, this type of normalization allows a more fair evaluation of the obtainable performance.

Since under-provisioning results in violating SLOs, a customer might want to use a platform that minimizes under-provisioning ensuring that enough resources are provided at any point in time, but at the same time minimizing the amount of over-provisioned resources. The defined separate accuracy metrics for over- and under-provisioning allow providers to better communicate their autoscaling capabilities and customers to select an autoscaler that best matches their needs. In the context of workflows, over-provisioning accuracy can also be represented in the number of idle resources (i.e. the resources which were excessively provisioned and currently are not utilized).

In ideal situation when an autoscaler perfectly follows the demand curve, there should be no idle resources as the system will always have enough eligible tasks to run. Although, intuitively it seems that over-provisioned resources should always be idle, in situations when the actual demand exceeds the estimated demand (from the autoscaler's perspective), the over-provisioned resources may not necessarily be idle. Since a_O and $\overline{a}_O$ metrics do not particularly distinguish the amount of idle resources in the system, we present an additional over-provisioning accuracy metric m_U which measures the average number of idle resources during the experiment time. If u_t is the number of idle resources at time t, m_U can be defined as:

$$m_U := \frac{1}{T \cdot R} \sum_{t=1}^{T} u_t,$$

3.3 Wrong-Provisioning Timeshare

The accuracy metrics do not distinguish cases when the average amount of under-/over-provisioned resources results from a few large deviations between demand and supply or rather by a constant small deviation. To address this, the following two metrics provide insights about the fraction of time in which under- or over-provisioning occurs. As visualized in Figure 2, the following metrics t_U and t_O are computed by summing the total amount of time spent in an under- A or over-provisioned B state normalized by the duration of the measurement period. Letting $\operatorname{sgn}(x)$ be the sign function of x, the overall timeshare spent in under- or over-provisioned states can be computed as:

$$t_U := \frac{1}{T} \sum_{t=1}^{T} (\operatorname{sgn}(d_t - s_t))^+, \qquad t_O := \frac{1}{T} \sum_{t=1}^{T} (\operatorname{sgn}(s_t - d_t))^+.$$

3.4 Instability of Elasticity

Although the accuracy and timeshare metrics characterize important aspects of elasticity, platforms can still behave differently while producing the same metric values for accuracy and wrong-provisioning timeshare. We define two *instability* metrics k and k' which capture this instability and inertia of the autoscalers. A low stability increases adaptation overheads and costs (e.g., in case of instance-hour-based pricing), whereas a high level of inertia results in a decreased SLO compliance.

Letting $\Delta d_t := d_t - d_{t-1}$, and $\Delta s_t := s_t - s_{t-1}$, the *instability* metric k which shows the average fraction of time of over-provisioning trends in the system is defined as:

$$k := \frac{1}{T-1} \sum_{t=2}^{T} \mathbb{1}_{\operatorname{sgn}(\Delta s_t) > \operatorname{sgn}(\Delta d_t)},$$

where by over-provisioning trends we mean situations when supply increases while demand is stable or when supply increases while demand decreases or when supply is stable while demand decreases. Similarly, we define k' which shows the average fraction of time of under-provisioning trends:

$$k' := \frac{1}{T-1} \sum_{t=2}^{T} \mathbb{1}_{\operatorname{sgn}(\Delta s_t) < \operatorname{sgn}(\Delta d_t)},$$

where under-provisioning trends are situations when demand increases while supply is stable or when demand increases while supply decreases or when demand is stable while supply decreases.

Both metrics k and k' do not capture neutral situations when both supply and demand move in the same direction (have the same sign) or both stay stable. Thus, if supply follows demand perfectly then both instability metrics are equal to zero.

3.5 User-oriented Metrics

To assess the performance of autoscaling policies from the time perspective, we employ the (average) elastic slowdown as a main user metric together with a set of traditional metrics. The (average) *elastic slowdown* is defined in steps as follows.

- The *wait time* T_w of a workflow is the time between its arrival and the start of its first task.
- The *makespan* T_m of a workflow is the time between the start of its first task until the completion of its last task.
- The *response time* T_r of a workflow is the sum of its wait time and its makespan: $T_r := T_w + T_m$.
- The *elastic slowdown* S_e of a workflow is its response time in a system which uses an autoscaler (where the workflow runs simultaneously with other workflows) normalized by its response time T_r' in a system of the same size without an autoscaler (where the workflow runs simultaneously with the same set of other workflows and where a certain amount of resources is constantly allocated): $S_e := T_r \;/\; T_r'$.

In ideal situation, where jobs do not experience slowdown due to the use of an autoscaler, the optimal value for S_e is 1. When S_e is less than 1, then the workflow is accelerated by the autoscaler. The *schedule length ratio* (SLR) [7] normalizes the response time of a workflow in a busy system with an autoscaler (as in the elastic slowdown) by the minimal possible critical path length of a workflow. The minimal possible critical path length is calculated using known a priori workflow task runtime estimates and the time required to transfer files between every two neighboring workflow tasks (also known as link weight). SLR, especially for the case without an autoscaler, shows how the system and the used scheduling policy slow down a workload. We also calculate the *average task throughput* $\overline{T}$ which is defined as the number of tasks processed per time unit.

For each workflow, we define its *deadline proximity ratio* D_p as $D_p := T_r/D$, where T_r is the completion time of a workflow and D is the deadline. D_p is calculated for each workflow after the completion of its last task.

3.6 Cost-oriented Metrics

We define the *average number of resources* $\overline{V}$ as

$$\overline{V} := \frac{1}{T}\sum_{t=1}^{T} s_t,$$

which is the number of resources allocated during the experiment to compute the *gain* of using an autoscaler. Though $\overline{V}$ shows th average e resource consumption of the autoscaler, it does not reflect cost in relation to common cloud pricing models.

We thus introduce the used CPU hours metric. To calculate the CPU hours, we distinguish between two different pricing models: *accounted CPU hours* and *charged CPU hours*. Figure 3 shows both pricing models where the orange blocks represent the charged CPU hours and the green blocks are the accounted CPU hours. The *accounted CPU hours* h_j for a VM j can be defined as

$$h_j := \sum_{t=1}^{T} s_{t,j},$$

where $s_{t,j}$ is the number of supplied resources at time t for the VM j. Hence, h_j represents the effective used CPU hours. Accordingly, we can compute the *average accounted CPU hours per VM* that is defined as $\overline{h} := \frac{1}{|VM|}\sum_{j=1}^{|VM|} h_j$, where $|VM|$ is the number of VMs. Therefore, we define the *runtime speedup* R as $R := \overline{h}_n/\overline{h}_a$, where $\overline{h}_a$ is the consumed CPU hours when using the autoscaler and $\overline{h}_n$ the CPU hours when all resources are running throughout the experiment. If R is greater

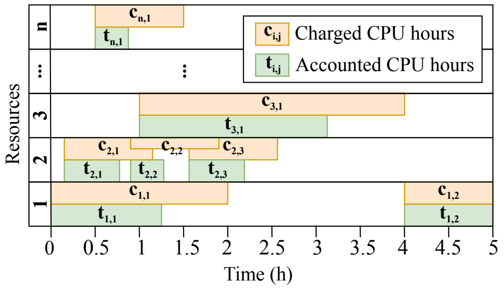


Fig. 3. This example shows which CPU hours are accounted and which CPU hours are charged.

Source of Information	**Timeliness of Information**	
	Long-term	**Current/Recent**
Server (General)	Hist, Reg, ConPaaS	React, Adapt
Job (WF-specific)	Plan	Token

Table 1. The two-dimensional taxonomy of the considered autoscalers.

than 1 the autoscaler saves CPU hours, otherwise it uses equal or more CPU hours than the no autoscaling scenario.

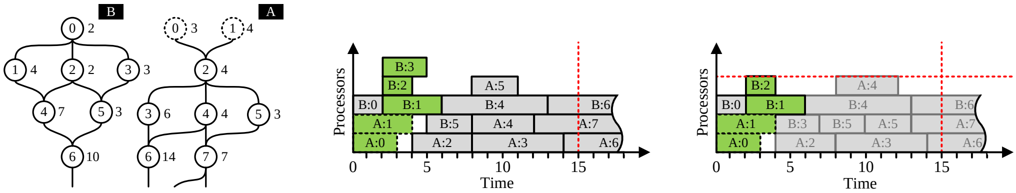
In contrast to the *accounted CPU hours*, the *charged cost* C_j represents the opened CPU hours that have to be paid for the virtual machine j. The charged cost C_j for a VM j, is defined as:

$$C_j := \sum_{i=1}^{N} \left\lceil \frac{\Delta_{i,j}}{P_{i,j}} \right\rceil K_{i,j},$$

where N is the number of scaling events, $\Delta_{i,j}$ is the time elapsed between two supply events i and $i-1$ for the VM j, $P_{i,j}$ is the charge period for VM j, and $K_{i,j}$ is the charge cost for VM j. Notice that $P_{i,j}$ and $K_{i,j}$ can vary over time, but here we consider them as constants for the sake of simplicity. In the following, we selected $P_{i,j} = 60$ minutes, and $K_{i,j} = 1$. The *average charged CPU hours per VM* $\overline{C} := \frac{1}{|VM|} \sum_{j=1}^{|VM|} C_j$, with $|VM|$ being the number of VMs. Similarly to the *accounted CPU hours*, we can compute the *charged speedup* $\tilde{C}$ as $\tilde{C} := \overline{C}_n / \overline{C}_a$, where $\overline{C}_a$ is the average charge cost of the autoscaler and $\overline{C}_n$ is the average charge cost when all the resources are running throughout the experiment. If $\tilde{C}$ is greater than 1 the autoscaler saves costs, otherwise it spends equal or more costs than the no autoscaling scenario.

4 AUTOSCALING POLICIES

For evaluation, we select five representative general autoscalers and propose two workflow-specific autoscalers. We classify them using a taxonomy along two dimensions and summarize the survey of common autoscaling policies across these dimensions in Table 1. The taxonomy allows us to ensure the proper coverage of the design space. We identify four groups of autoscalers, which differ in the way they treat the workload information. The first group consists of general autoscalers Hist, Reg, and ConPaaS which require server-specific information and use historical data to make their predictions. The second group consists of React and Adapt autoscalers which also require server-specific information for their operation but they do not use history to make autoscaling decisions. The last two groups use job-specific information (e.g., structure of a workflow) and also differ in a way they deal with the historical data: Plan needs detailed per-task information while Token needs far less historical data and only requires a runtime estimate for the whole job. Further, we present all



(a) Job queue. (b) Unlimited partial plan. (c) Limited partial plan.

Fig. 4. The Plan autoscaling algorithm.

the autoscalers in more detail. When introducing each autoscaler we additionally indicate in the title of the section to which dimensions of the taxonomy it belongs.

4.1 General Autoscaling Policies

As different autoscalers exhibit varying performance, five existing general autoscalers have been selected. By a general autoscaler, we refer to autoscalers that have been published for more general workloads including multi-tier applications, but that are not designed particularly for workflow applications. The five autoscalers can be used on a wide range of scenarios with no human tuning. We implement four state-of-the-art autoscalers that fall in this criteria. In addition, we acquire the source codes of one open-source state-of-the-art autoscaler. The selected methods have been published in the following years 2008 [53] (with an earlier version published in 2005 [52]), 2009 [11], 2011 [31], 2012 [3, 5], and 2014 [22]. The selected autoscalers are well-cited representatives of the autoscaler groups identified in the extensive survey [37].

4.1.1 General Autoscalers for Workflows. All of the chosen general autoscalers have been designed to control performance metrics that are still less commonly used for workflow applications, namely, request response time, and throughput. The reason is that historically, workflow applications were rather big or were submitted in batches [50]. However, emerging workflow types require quick system reaction such as the usage of workflows in areas where they were less popular, e.g., for complex web requests, making the use of general autoscalers more promising.

The autoscalers aimed to control the response time are designed such that they try to infer a relationship between the response time, request arrival rates, and the average number of requests that can be served per VM per unit time. Then, based on the number of request arrivals, infer a suitable amount of resources. This technique is widely used in the literature [24, 37] due to the non-linearity in the relationship between the response time and allocated resources.

A similarity does exist though between workflows and other cloud workloads. A task in a workflow job can be considered as a long running request. The number of tasks becoming eligible can be considered as the request arrival rate for workflows. Therefore, we have adapted the general autoscalers to perform the scaling based on the number of task arrivals per unit time.

4.1.2 The React Policy (Server, Current). Chieu et al. [11] present a dynamic scaling algorithm for automated provisioning of VM resources based on the number of concurrent users, the number of active connections, the number of requests per second, and the average response time per request. The algorithm first determines the current web application instances with active sessions above or below a given utilization. If the number of overloaded instances is greater than a predefined threshold, new web application instances are provisioned, started, and then added to the front-end load-balancer. If two instances are underutilized with at least one instance having no active session, the idle instance is removed from the load-balancer and shutdown from the system. In each case the technique *React*s to the workload change. For the remainder of this article, we refer to this technique as *React*. The main reason we are including this algorithm in the analysis is that this algorithm is the baseline algorithm in our opinion since it is one of the simplest possible workload predictors. We have implemented this autoscaler for our experiments.

4.1.3 The Adapt Policy (Server, Recent).

Ali-Eldin et al. [3, 5] propose an autonomous elasticity controller that changes the number of VMs allocated to a service based on both monitored load changes and predictions of future load. We refer to this technique as *Adapt*. The predictions are based on the rate of change of the request arrival rate, i.e., the slope of the workload, and aims at detecting the envelope of the workload. The designed controller *Adapt*s to sudden load changes and prevents premature release of resources, reducing oscillations in the resource provisioning. *Adapt* tries to improve the performance in terms of number of delayed requests, and the average number of queued requests, at the cost of some resource over-provisioning.

4.1.4 The Hist Policy (Server, Long-term).

Urgaonkar et al. [53] propose a provisioning technique for multi-tier Internet applications. The proposed methodology adopts a queuing model to determine how many resources to allocate in each tier of the application. A predictive technique based on building *Hist*ograms of historical request arrival rates is used to determine the amount of resources to provision at an hourly time scale. Reactive provisioning is used to correct errors in the long-term predictions or to react to unanticipated flash crowds. The authors also propose a novel datacenter architecture that uses VM monitors to reduce provisioning overheads. The technique is shown to be able to improve responsiveness of the system, also in the case of a flash crowd. We refer to this technique as *Hist*. We have implemented this autoscaler for our experiments.

4.1.5 The Reg Policy (Server, Long-term).

Iqbal et al. propose a regression-based autoscaler (hereafter called *Reg*) [31]. The autoscaler has a reactive component for scale-up decisions and a predictive component for scale-down decisions. When the capacity is less than the load, a scale-up decision is taken and new VMs are added to the service in a way similar to *React*. For scale-down, the predictive component uses a second order regression to predict future load. The regression model is recomputed using the complete history of the workload when a new measurement is available. If current load is less than the provisioned capacity, a scale-down decision is taken using the regression model. This autoscaler was performing badly in our experiments due to two factors; first, building a regression model for the full history of measurements for every new monitoring data point is a time consuming task. Second, distant past history becomes less relevant as time proceeds. After contacting the authors, we have modified the algorithm such that the regression model is evaluated for only the past 60 monitoring data points.

4.1.6 The ConPaaS Policy (Server, Long-term).

ConPaaS, proposed by Fernandez et al. [22]. The algorithm scales a web application in response to changes in throughput at fixed intervals of 10 minutes. The predictor forecasts the future service demand using standard time series analysis techniques, e.g., Linear Regression, Auto Regressive Moving Average (ARMA), etc. The code for this autoscaler is open source. We downloaded the authors' implementation.

4.2 Workflow-Specific Autoscaling Policies

In this section, we present two workflow-specific autoscalers designed by us. Their designs are inspired by previous work in this field and adapted to our situation. The presented autoscalers differ in a way they use workflow structural information and task runtime estimates.

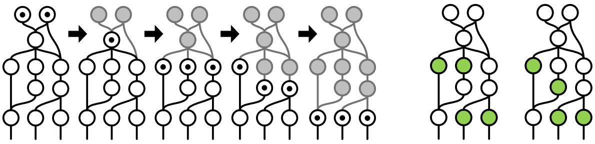
4.2.1 The Plan Policy (Job, Long-term).

This autoscaler makes predictions by constructing and analyzing a partial execution *Plan* of a workflow. Thus it uses the workflow structure and workflow task runtime estimates. The idea is partially based on static workflow schedulers [2]. On each call, the policy constructs a partial execution plan considering both workflows with running tasks and workflows waiting in the queue. The maximal number of processors which are used by this plan is returned as a prediction. The time duration of the plan is limited by the autoscaling interval. The plan is two-dimensional, where one dimension is time and another is processors (VMs).

The policy employs the same task placement strategy as the scheduler. In our case, the jobs from the main job queue are processed in first-come, first-served (FCFS) order and the tasks are prioritized in ascending order of their identifier (each task of a workflow is supposed to be assigned with a unique numeric identifier). For already running tasks, the runtimes are calculated as a remaining time to their completion. The algorithm operates as follows. On each call it initializes an empty plan with start time 0. Then it sequentially tries to add tasks in the plan in such as their starting times are minimal. The algorithm adds a task to the plan only if it is eligible or its parents are already in the plan. The plan construction lasts until there are no tasks which can be added in the plan or until the minimal possible task start time equals or exceeds the planning threshold (which is equal to the autoscaling interval), or until the processor limit is reached. If the processor limit is reached then this is returned as the prediction. Otherwise, the prediction is calculated as the maximal number of processors ever used by the plan within the planning interval.

Figure 4 shows an example of the operation of the algorithm. In Figure 4a, we show the job queue at the moment when the autoscaler is called. The queue contains two workflows A and B, where A is at the head of the queue. Each workflow task is represented by a circle with an identifier within it and runtime in time units on the right. Tasks A:0 and A:1 are already running, finished tasks are not shown. The autoscaling interval (a threshold) is equal to 15 time units and is represented by a vertical red dashed line. Figure 4b shows an example of an unlimited plan where the processor limit is not reached. In this case the maximal number of processors used within the 15 time units interval is 5 which equals to the number of green rectangles in the figure (A:0, A:1, B:1, B:2, B:3). Figure 4c shows a plan where the number of available processors is limited by 4 (the horizontal red dashed line). In this case, the algorithm stops constructing the plan after placing task B:2 and returns the prediction, which simply equals to the maximal number of available processors (i.e., 4).

4.2.2 The Token Policy (Job, Recent). The *Token* policy uses structural information of a DAG and does not directly consider task runtimes to make predictions and instead requires an estimated execution time of the whole workflow. It uses tokens to estimate the *Level of Parallelism* (LoP) of a workflow [29] by simulating an execution "wave" through a DAG. The operation of the algorithm is illustrated in Figure 5. The algorithm processes the workflows in the queue in the FCFS order. In the beginning, the algorithm picks a workflow from the queue and places tokens in all of its entry tasks. Then in successive steps it moves these tokens to all the nodes all of whose parents already hold a token or were earlier tokenized. After each step, the number of tokenized nodes is recorded. For each workflow, the number of propagation steps is limited by a certain depth δ, which is defined as $\delta = (\Delta t \cdot N)/L$, where Δt is the autoscaling interval, N is the number of tasks on the critical path of



(a) Token propagation steps. (b) Two real LoPs.

Fig. 5. The token-based LoP approximation.

the workflow, and L is the total duration of the tasks on the critical path of the workflow. Thus, the intuition is to evaluate the number of "waves" of tasks (future eligible sets) that will finish during the autoscaling interval. When δ or the final task of a workflow is reached, the largest recorded number of tokenized nodes is the approximated LoP value. The algorithm stops when the prediction value exceeds the maximal total number of available processors or when the end of the queue is reached. The final prediction is the sum of all of the separate approximated LoPs of the considered workflows.

The token-based algorithm does not guarantee the correct estimation of the LoP. The quality of the estimation depends on the DAG structure. In Figure 5a the estimated LoP of 3 is lower than the maximal possible LoP of 4 in Figure 5b. However, in our previous work [29], we showed that this method provides meaningful results for popular workflow structures.

5 EXPERIMENTAL EVALUATION

In this section, we present the workloads and the configuration of the cloud infrastructure we use for the experimental evaluation of the unified cloud system introduced in Section 2. To design our workloads, we use a set of representative scientific workflows. We take an experimental approach to evaluate chosen autoscaling algorithms with an extensive set of experiments in a virtualized environment deployed on our multi-cluster system.

5.1 Setup of Workflow-based Workloads

We choose three popular scientific workflows from different fields, namely Montage, LIGO, and SIPHT. The main reason for our choice is the existence of validated models for these workflow types. Montage [33] is used to build a mosaic image of the sky on the basis of smaller images obtained from different telescopes. LIGO [1] is used by the Laser Interferometer Gravitational-Wave Observatory (LIGO) to detect gravitational waves. SIPHT [35] is a bioinformatics workflow used to discover

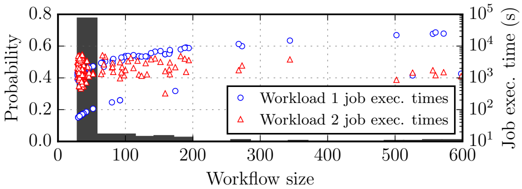


Fig. 6. The distribution of job sizes in the workloads (histogram, left vertical axis) and the dependency between the job size and its execution time (glyphs, right vertical axis). The right vertical axis is in log scale.

Property	Workload 1	Workload 2
Mean task runtime	33.52 s	33.29 s
Median task runtime	2.15 s	2.65 s
σ of task runtime	65.40 s	87.19 s
Mean job execution time	2,325 s	2,309 s
Median job execution time	1,357 s	1,939 s
σ of job execution time	3,859 s	1,219 s
Total task runtime	465,095 s	461,921 s
Mean workflow size	69 tasks	
Median workflow size	35 tasks	
σ of workflow size	98 tasks	
Total number of tasks	13,876 tasks	

Table 2. Statistical characteristics of the workloads, σ stands for standard deviation.

We generate synthetic workflows using the workflow generator by Bharathi et al. [8, 34]. Each workflow is represented by a set of task executables and a set of input files. We use two workloads: a primary Workload 1 and a secondary Workload 2 each consisting of 200 workflows of different sizes in range from 30 to 600. Each workload contains an equal mixture of all of the three considered workflow types. As with many other workloads in computer systems, in practice, workflows are usually small, but very large ones may exist too [44]. Therefore, in our experiments we distinguish small, medium, and large workflows, which constitute fractions of 75%, 20%, and 5% of the workload. The size of the small, the medium, and the large workflows is uniformly distributed on the intervals [30, 39], [40, 199], and [200, 600], respectively. The distribution of the job sizes in the

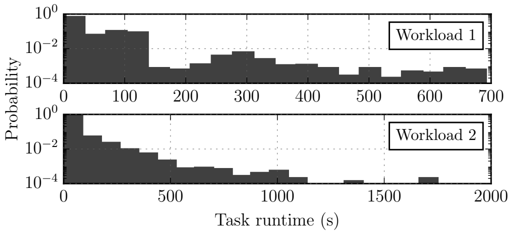


Fig. 7. The distribution of task runtimes in the workloads (the horizontal axes have different scales, and the vertical axes are in log scale).

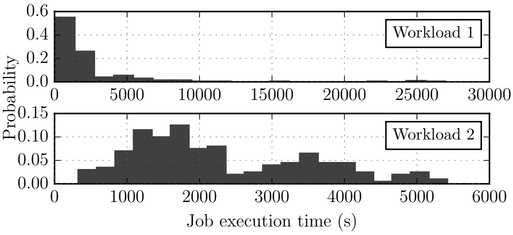


Fig. 8. The distribution of job execution times in the workloads (all the axes have different scales).

workloads is presented in Figure 6. Figure 7 shows the distribution of task runtimes. Figure 8 shows the distribution of job execution times T_e in the workloads. Note, that the histogram of job execution times shows the *total execution time* of a job which is the sum of all the job's task runtimes. The job will be running this amount of time if and only if all of its tasks are executed sequentially. However, normally workflows have both parallel and sequential parts. Thus the job execution times reported in Figure 8 should not be confused with the actual observed makespans T_m of workflows running in a parallel system.

For Workload 1, we use the original job execution time distribution from the Bharathi generator. For Workload 2, we keep the same job structures as in Workload 1, but change the job execution times using a two-stage hyper-Gamma distribution derived from the model presented in [38]. The shape and scale parameters (α, β) for each Gamma distribution are set to (5.0, 323.73) and (45.0, 88.291), respectively. Their proportions in the overall distribution are 0.7 and 0.3. Table 2 summarizes the properties of both workloads.

5.2 Setup of the Private Cloud Deployment

To schedule and execute workflows, we use the experimental setup in Figure 1 (Section 2). The KOALA scheduler is used for scheduling workflow tasks [20] on the DAS-4[1] cluster deployed at TU Delft. Our cluster consists of 32 nodes interconnected through QDR InfiniBand with 8-core 2.4GHz CPU and 24GB of RAM each. As a cloud middleware, OpenNebula is used to manage VM deployment, and configuration. The implementation and the VM images are available upon request. The REST interface between the scheduler and general autoscalers in our architecture makes it extendable and allows to use other autoscaling policies. The workflow-specific autoscalers are implemented within KOALA, though other custom policies can also be added.

The execution environment for a single workflow consists of a single head VM and multiple worker VMs. The head VM uses a single CPU core and 4GB of RAM, while each worker VM uses a single core and 1GB of RAM. Tasks are then scheduled on the VMs. The workload generator with the workflow runners run on a dedicated node. The workflow runner coordinates the workflow execution by following the task placement commands from the scheduler. The runner is also responsible for copying files (task executables, input and output files) to and from the VMs in the virtual cluster. For data storage and transfer, we use a Network File System (NFS). This implies that if the head VM and worker VM are located on the same physical node, the data transfer time between them is negligible. In other cases, data transfer delays occur. The measured mean NFS write speed for 10 tests of transferring 1 GB is 280 MB/s. We use this value to calculate critical path lengths for the SLR metric.

[1] http://www.cs.vu.nl/das4

Compared to job execution times, file transfer delays and the scheduling overheads are negligible. All tasks write their intermediate results directly to the shared storage reducing data transfer delays for all workflows. A task can run as soon as all of its dependencies are satisfied. The runner copies all input files for a workflow to the virtual cluster before starting the execution. Thus, the impact from file transfer delays between tasks on performance is negligible. Tasks are scheduled using *greedy backfilling* as it has been shown to perform well when task execution times are unknown a priori [29]. During the experiment, only the autoscaler has access to the information about job execution times and task runtimes. Note, that for all considered autoscalers, we do not perform any task preemptions. If an autoscaler requires to stop a certain number of VMs, the scheduler only releases those VMs which are idle. The scheduler first releases the VMs which are idle the longest.

5.3 Experiment Configuration

To configure the general autoscalers we use the average number of tasks a single resource (VM) is able to process per autoscaling interval (hereafter called *service rate*). The autoscaling interval, or the time between two autoscaling actions, is set to 30 seconds for all of our experiments.

We test with three different configurations in our experiments, where we change the value of the service rate parameter or the VM provisioning latency. The service rate in a request-response system is usually the average number of requests that can be served per VM. This parameter is either estimated online, e.g., using an analytical model to relate response time, as the one used in Hist [52], or offline [17, 24]. For a task-based workload, there are multiple options including using the mean task service time, the median task service time, or something in between.

In the first configuration, we assume that a VM serves on average 1 task per autoscaling interval, i.e., 2 tasks per minute. We derive this value by rounding the service rate calculated based on the mean task runtimes to the nearest integer (Table 2). This service rate allows us to perform additional comparison between general and workflow-specific autoscalers as the demand curves have the same dimension. In the second configuration, we use the median task runtime of Workload 1 which gives a service rate equal to 14 (also rounding to the nearest integer) tasks per autoscaling interval, i.e., 28 tasks per minute. The general autoscalers using the second configuration are marked with a star ($\star$) symbol. While in the first two configurations we guarantee that all the provisioned VMs are booted at the moment when the autoscaler is invoked, in the third configuration the VM booting time of 45 s exceeds the autoscaling interval of 30 s. This configuration is also used to test workflow-specific autoscalers. The autoscalers using the third configuration are marked with a diamond ($\diamond$).

For all the configurations and for both workloads the workload player periodically submits workflows to KOALA to impose the average load on the system about 40%. The workflows submitted to the system arrive according to a Poisson process. The mean inter-arrival interval is 117.77 s which results into arrival rate of 30.57 jobs per hour. Thus, the minimal duration of each experiment is approximately 6.5 h. If the autoscaler tends to under-provision resources or the provisioning time in the system is rather large then the experiment can take longer. We choose this relatively low utilization level on purpose to decrease the number of situations when the demand exceeds the maximum possible supply ceiling. Additionally, as workflow scheduling is non-work-conserving the system can saturate even at low utilizations. Thus, low utilization allows us to see better the dynamic behavior of the autoscalers by minimizing the number of extreme cases.

We are aware that in computing clouds the job arrivals are often affected by the time of the day and various external influences resulting in burstiness [21]. Even for a datacenter which serves requests from all around the globe the intensity of job arrivals could vary as the distribution of world population is not even across different time zones, etc. We leave the experiments with other job arrival patterns to future work. However, the arrival of workflow tasks is non-Poissonian and depends

Type	AS	a_U	a_O	$\bar{a}_U$	$\bar{a}_O$	t_U	t_O	k	k'	m_U
		%	%	%	%	%	%	%	%	%
General	React	2	6	5	50	15	84	20	32	7
	React$^\diamond$	6	5	13	40	32	64	21	32	6
	Adapt	4	4	8	27	23	51	21	34	5
	Hist	1	60	1	737	2	97	**17**	43	60
	Reg	3	8	6	51	17	51	20	**31**	8
	ConPaaS	2	33	5	273	11	76	20	40	34
	React*	**0**	19	**0**	179	2	98	19	64	**0**
	Adapt*	**0**	16	1	150	4	96	19	63	**0**
	Hist*	**0**	25	1	463	5	95	20	60	1
	Reg*	**0**	12	1	78	5	94	20	62	**0**
	ConPaaS*	**0**	44	1	1092	**1**	98	21	45	7
WF-specific	Plan	3	4	7	24	20	43	20	32	5
	Plan$^\diamond$	7	**3**	16	**17**	35	**33**	20	34	4
	Token	3	6	7	35	16	53	20	33	7
None	No AS	0	73	0	869	0	100	17	43	73

Table 3. Calculated *autoscaling metrics* for the main set of experiments with *Workload 1*. The diamond symbol ($\diamond$) marks the experiments where the VM booting time is longer than the autoscaling interval and service rate parameter is set to 1.0. The star symbol ($\star$) marks general autoscalers configured with service rate 14.0. All the other general autoscalers are configured with service rate 1.0. Best values in each column are highlighted in bold, except the No AS case.

on the workflow structure and on the distribution of task runtimes within a workflow. Thus, the diversity of the workflow structures, workflow sizes, and task runtime distributions which we use, allows us to suppose that our setup is representative. Furthermore, for the considered duration of the experiment of approximately 6.5 h, we can simply claim that our emulated system serves requests from multiple independent users.

5.4 Experimental Results

The main findings from our experiments are the following:

(1) Workflow-specific autoscalers perform slightly better than general-autoscalers but require more detailed job information.
(2) General autoscalers show comparable performance but their parametrization is crucial.
(3) Autoscalers reduce the average number of used resources, but slow down the jobs.
(4) Although autoscalers tend to reduce the accounted resource runtime, the charged time can easily be higher compared to not using autoscaling.
(5) Long VM booting times negatively affect the autoscalers' performance, and mostly affect small and medium job sizes.
(6) Over-provisioning could partially contribute for better fairness between different job sizes.
(7) Autoscalers with better autoscaling metrics show higher variability of deadline violations.
(8) No autoscaler outperforms all other autoscalers with all configurations and/or metrics.

5.4.1 Analysis of Elasticity. To show the trade-offs between the autoscalers, we use the metrics described in Section 3. While calculating the system-oriented metrics, we exclude periods where the demand exceeds 50 VMs, the total number of available VMs. Since system-oriented metrics are normalized by time, this approach does not bias the results.

The aggregated metrics for all experiments are presented in Table 3, Table 4, and Table 5. Considering the cases where VMs are booting faster than the autoscaling interval, Table 3 shows that the autoscalers under-provision between 1% (using Hist) and 8% (using Adapt) less resources from the demand needs. Hist's superior under-provisioning with respect to others comes at the cost of on average provisioning 7 times the actual demand, compared to 24% over-provisioning for Plan.

Type	AS	S_e	S_e (S)	S_e (M)	S_e (L)	SLR	$\overline{T}$	$\overline{V}$	$\overline{h}$	$\overline{C}$	R	$\tilde{C}$
		frac.	frac.	frac.	frac.	frac.	tasks/h	VMs	CPU · h	CPU · h	frac.	frac.
General	React	1.23	1.24	1.20	1.21	3.71	2,071	23.89	3.30	36.68	2.02	0.19
	React$^\diamond$	1.57	1.60	1.52	1.33	4.59	2,066	24.01	3.56	33.38	1.87	0.21
	Adapt	1.28	1.32	1.20	1.15	3.82	2,071	22.86	3.22	48.14	2.00	0.15
	Hist	**1.05**	**1.05**	**1.04**	**1.02**	**3.17**	**2,076**	44.81	6.21	9.90	1.08	0.71
	Reg	1.29	1.32	1.20	1.11	3.89	2,071	24.42	3.36	38.08	1.98	0.18
	ConPaaS	1.18	1.22	1.07	1.06	3.5	2,071	34.50	4.72	41.28	1.42	0.17
	React*	17.32	20.69	8.76	4.06	45.66	2,011	20.13	**2.91**	**5.06**	**2.30**	**1.38**
	Adapt*	20.06	23.25	12.26	6.08	52.74	2,026	20.49	3.14	7.54	2.13	0.92
	Hist*	12.93	15.30	7.00	3.24	34.88	2,016	20.87	3.15	7.14	2.12	0.92
	Reg*	25.57	30.04	14.38	7.22	69.84	1,997	**20.11**	2.93	5.76	2.29	1.21
	ConPaaS*	2.11	2.12	2.26	1.25	5.90	2,061	25.15	3.70	41.20	1.81	0.17
WF-specific	Plan	1.27	1.29	1.23	1.11	3.77	2,071	23.34	3.51	44.26	1.90	0.16
	Plan$^\diamond$	1.48	1.54	1.35	1.15	4.3	2,066	22.13	3.38	38.04	1.98	0.18
	Token	1.25	1.28	1.20	1.20	3.71	2,071	23.88	3.31	46.34	2.02	0.15
None	No AS	1.00	1.00	1.00	1.00	3.07	2,076	50.00	6.68	7.00	1.00	1.00

Table 4. Calculated *user-oriented and cost-oriented metrics* for the main set of experiments with *Workload 1*. The diamond symbol ($\diamond$) marks the experiments where the VM booting time is longer than the autoscaling interval and service rate parameter is set to 1.0. The star symbol ($\star$) marks general autoscalers configured with service rate 14.0. All the other general autoscalers are configured with service rate 1.0. The metric S_e as well presented for small (S), medium (M), and large (L) job sizes. Best values in each column are highlighted in bold, except the No AS case.

Type	AS	a_U	a_O	$\bar{a}_U$	$\bar{a}_O$	t_U	t_O	k	k'	m_U	S_e	S_e (S)	S_e (M)	S_e (L)	$\overline{T}$	$\overline{V}$
		%	%	%	%	%	%	%	%	%	frac.	frac.	frac.	frac.	tasks/h	VMs
General	React	2	7	4	36	17	81	21	**32**	7	1.11	1.08	1.19	1.21	**1,905**	22.83
	Hist	**1**	46	**1**	338	**5**	94	**19**	41	46	**1.05**	**1.03**	**1.10**	**1.16**	**1,905**	40.82
WF-specific	Plan	3	**4**	7	**13**	22	**39**	21	**32**	**4**	1.12	1.10	1.18	1.13	**1,905**	**21.32**
None	No AS	0	66	0	563	0	100	19	41	66	1.00	1.00	1.00	1.00	1,910	50.00

Table 5. Calculated *autoscaling and user-oriented metrics* for the additional set of experiments with *Workload 2*. The metric S_e as well presented for small (S), medium (M), and large (L) job sizes. Best values in each column are highlighted in bold, except the No AS case.

The React$^\diamond$ and Plan$^\diamond$ policies with longer booting VMs in Table 3 show slightly different results compared to the runs with faster booting VMs. We picked only these two policies to have one from each group of autoscalers. Both React$^\diamond$ and Plan$^\diamond$ tend to under-provision more when the VM provisioning time is longer. The job slowdowns in Table 4 are also higher. From Figure 10 we can clearly see that for ($\diamond$) autoscalers the supply curve is always lagging behind the demand. Thus, we can conclude that longer provisioning times decrease the number of available resources for the workload. We can also notice that the average number of idle VMs decreases for React$^\diamond$ and for Plan$^\diamond$ as the tasks more fully utilize provisioned VMs.

For the general policies configured with service rate 1.0 and for workflow-specific policies in Table 4 and Table 5 job elastic slowdowns show low variability. We can conclude that the resources either significantly over-provisioned (Hist and ConPaaS) or already provisioned resources are underutilized (React, Adapt, Reg, Plan, and Token). The non-zero values of m_U metric in these cases confirm our assumption.

5.4.2 The Influence of Different Workloads. The difference is also visible between the two used workloads. While Workload 1 has the majority of short jobs, Workload 2 has a more equal distribution of job execution times and is thus less bursty. Elastic job slowdowns in both tables confirm this tendency. For Workload 2 they slightly increase (the Plan policy in Table 5 is an exception) while going from small to large job sizes. We do not run Workload 2 with service rate different from 1.0 as we expect that the trend will be the same as for Workload 1.

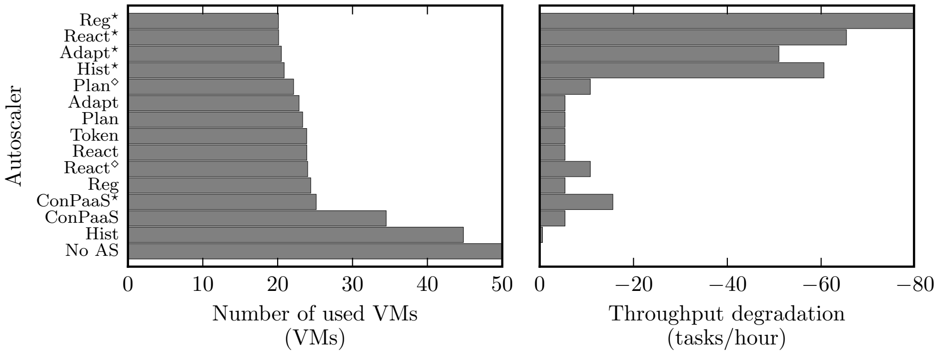


Fig. 9. The average number of used VMs during the experiment and the average throughput degradation (compared with the no autoscaler case). All results are given for Workload 1.

The system-oriented metrics do not vary much between the workloads. For example, compare React in Table 3 and Table 4 with React in Table 5. Only Hist over-provisions less while running with Workload 2 as can be explained by lower burstiness of the workload.

5.4.3 The Dynamics of Autoscaling. Figure 10 shows the system dynamics for each autoscaling policy while executing Workload 1. Some of the autoscalers have a tendency to over-provision resources (Hist and ConPaaS). The other policies appear to be following the demand curve more or less closely. Note, that the demand curve has different shape for each autoscaler as the autoscaling properties affect the order in which workflow tasks become eligible.

The workflow-specific Plan policy follows the demand curve quite well and shows results similar to general autoscalers React, Adapt, and Reg running with service rate of 1.0. However, if a policy follows the demand too close that increases job slowdowns as seen in Table 4. This trade-off is intuitive. The best policy when it comes to reducing slowdown is to always have more capacity than needed as this will allow any task to run as soon as it becomes eligible.

5.4.4 The Influence of Service Rate Parameter on the Autoscaling Dynamics. The most noticeable differences in the results are between general autoscalers running with service rate 1.0 and with service rate 14.0, the median service rate based on the mean task runtime. Figure 11 shows general autoscalers running with the same Workload 1 as in Figure 10. The demand curves in these two figures look very different, except for ConPaaS and ConPaaS⋆. In addition, the supply curves do not follow the demand curves closely anymore. Although the service rate chosen is the median, it does not reflect the temporal properties of the workload when it comes to the length of running tasks. If longer jobs occur in parallel, a queue of tasks will build up resulting in enormous system slowdowns. This is clear from Table 4 where the slowdown between the two service rates is 10 to 15 times larger when using a larger service rate. The k' metric also increases for service rate 14.0 as the autoscalers need to estimate more while computing the next predicted supply value and thus the curves are not so well synchronized. On the one hand, using a larger service rate significantly reduces the average charged CPU hours, potentially reducing the costs of operations for a user to almost one sixths of running with lower service rate.

5.4.5 The Trade-off Between Resource Usage and Performance. Here, we study the influence of the number of used VMs on the throughput. We evaluate only two user-oriented metrics: the throughput degradation in tasks per hour compared with the no autoscaler case and the number of used resources (VMs). The values of these metrics are plotted in Figure 9. For example, for React the throughput degradation of –24 tasks per hour contributes only to 1.16% of the hourly throughput.

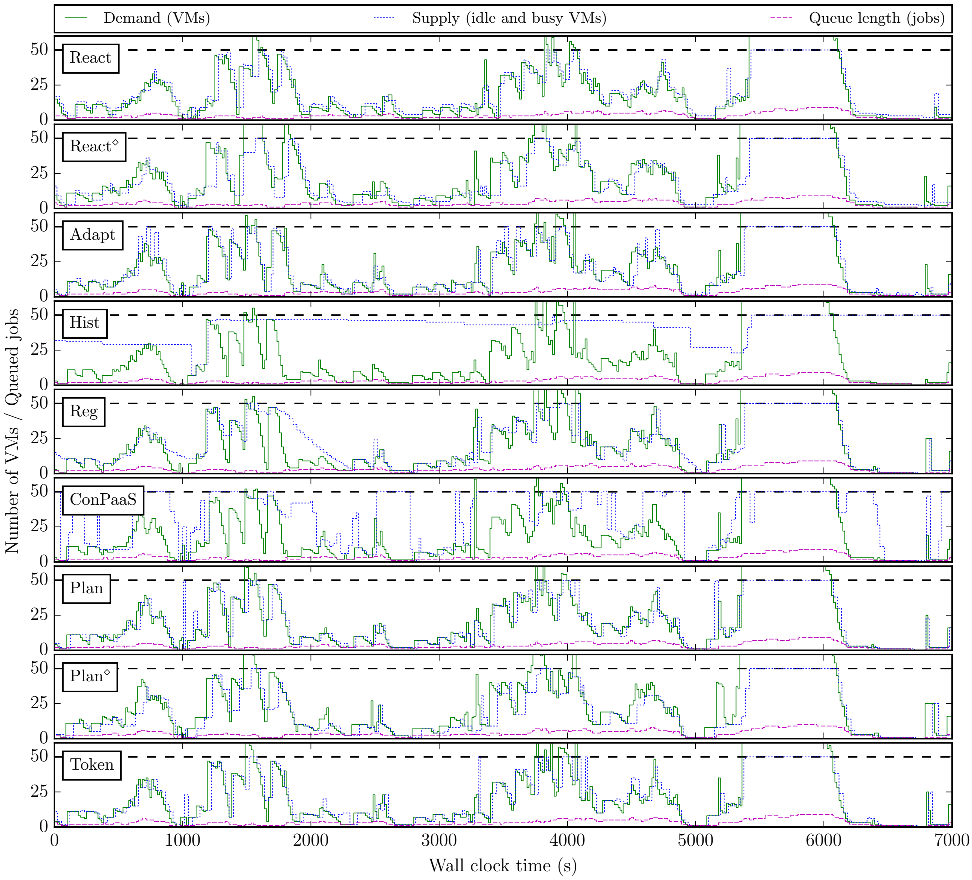


Fig. 10. The experimental dynamics of five general autoscaling policies (Workload 1, service rate 1.0) and two workflow-specific policies during the cropped period of 7,000 s. The horizontal dashed line indicates the resource limit of 50 VMs. The diamond symbol (⋄) marks the experiments where the VM booting time is longer than the autoscaling interval.

In Figure 9, we can see that $\overline{T}$ is definitely affected by the $\overline{V}$. The variation of $\overline{T}$ depends on the properties of the workload such as task durations, the total number of tasks in the workload, and the number of tasks per job.

From these results we can conclude the following. Hist over-provisions quite a lot and achieves low throughput degradation. ConPaaS also over-provisions but the throughput is not much affected because its supply curve is very volatile. ConPaaS* over-provisions less than ConPaaS as it “supposes” that the system needs less active VMs to process the same workload. Accordingly, the throughput degradation for ConPaaS* is also bigger. Reg, React, and Adapt configured with service rate 1.0 as well as Token and Plan show almost similar results. Plan and Token policies show good balance between the number of used VMs and the throughput. Parametrization with the service rate of 14.0 (based on the median task runtime) decreases the performance by allocating less VMs. We can also see that longer booting VMs (React$^\diamond$ and Plan$^\diamond$) negatively affect the throughput.

5.5 Performance of Enforced Deadline-based SLAs

In this section, we analyze how enforced deadline-based SLAs perform for the considered autoscalers. We use Workload 1 and configure the general autoscalers with service rate 1.0. We look at two cases.

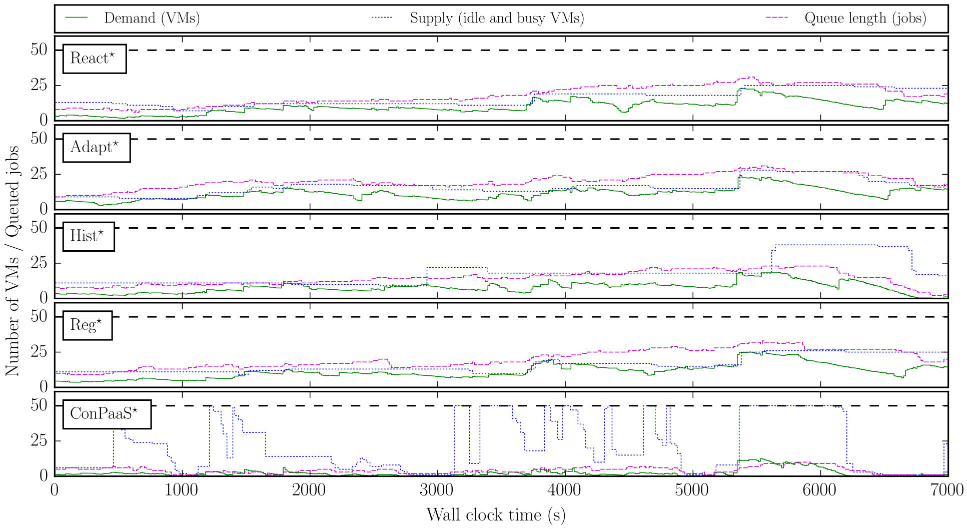
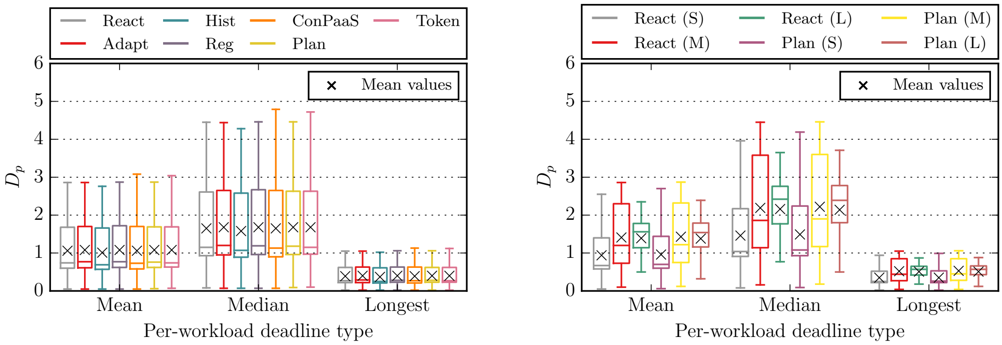


Fig. 11. The experimental dynamics of all the five considered general autoscaling policies (Workload 1, service rate 14.0) during the cropped interval of 7,000 s. The horizontal dashed line indicates the resource limit of 50 VMs.



(a) All autoscalers and all workflows.

(b) React and Plan autoscalers, (S) small, (M) medium, and (L) large workflow sizes.

Fig. 12. The deadline proximity ratio for the three types of enforced per-workload deadlines (Workload 1).

In the first case, the deadlines for each workflow are set on per-workload basis, in the second case the deadlines for each workflow are calculated individually, thus set on per-workflow basis. In both cases, we use response times of workflows in a system without an autoscaler as the reference.

In the first case, for all the workflows we assign the same deadline which is based on the statistical characteristics of the whole workload. This approach imitates a situation where only high level statistics of the workload profile are available for calculating the deadlines. Although, in our case we use the whole workload, in other situations the size of the observed period (or the number of considered workflows) could be different and limited by the practicability, e.g., when the execution statistics are not available a priori and should be collected automatically on-the-fly [14]. We use three scenarios to assign per-workload based deadlines:

(1) The longest response time of a workflow in the system without an autoscaler (1,247 s).

(2) The mean response time of workflows in the system without an autoscaler (459 s).
(3) The median response time of workflows in the system without an autoscaler (295 s).

Figure 12a shows deadline proximity ratios for all the three scenarios of enforced per-workload deadlines. Figure 12b shows the variability of deadline proximity ratios for small, medium, and large workflows for two selected autoscalers: React and Plan.

In the second case, we use per-workflow deadlines. Each workflow is assigned with a unique deadline which is equal to its response time in a system without an autoscaler. We vary the per-workflow deadlines by multiplying them by a certain error factor which we select from the following list of values [0.8, 0.9, 0.95, 1.0, 1.05, 1.1, 1.2, 1.3]. Note, for the error factor 1.0 the deadline proximity ratio coincides with the elastic slowdown. By altering the deadlines, we can observe how the incorrectly set deadlines, due to inaccuracies in estimation methods [12, 21], affect deadline violations.

Figure 13 shows the variability of deadline proximity ratios for all the workflows in Workload 1. Figure 14 shows the variability of deadline proximity ratios between three considered groups of workflow sizes for React and Plan policies.

Varying per-workload deadlines using the similar approach as for per-workflow deadlines does not show significant difference in the results. Interestingly, Figure 12a shows low variability between different autoscalers within the same deadline type. The main reason is that the majority of jobs in Workload 1 are rather short. Thus, when the per-workload deadline is applied to each separate workflow, it leaves enough free space to allow the workflow to meet its deadline (despite the possible negative effects of elastic slowdown). Comparing different scenarios of per-workload deadlines, we can definitely see that the per-workload deadline which is based on the longest response time, allows almost all workflows to finish before the deadline. The mean-based per-workload deadline also shows good results since mean deadline violation ratio is very close to 1. The median-based per-workload deadline is not the best solution as it increases deadline violations, shifts median deadline proximity ratio up to 1 and leads to higher variability in the deadline violations.

Comparing the deadline violations between React and Plan autoscalers in Figure 12b we can see that for the same job sizes, e.g., React (S) and Plan (S) deadline proximity ratios are almost identical. We can conclude that for per-workload deadlines the influence of autoscalers is less pronounced because each workflow has enough time to meet the deadline even despite the elastic slowdown. The general trend between different scenarios in Figure 12b is similar to Figure 12a, i.e., longest deadline scenario shows the best results and Median the worst. Table 6 additionally shows numerical variability characteristics for the selected per-workload and per-workflow scenarios, also for (⋄) and (⋆) autoscaler configurations.

Intuitively, the further the deadline, the lower the deadline violation ratio. Figure 13 confirms this proposition and indicates how varying per-workflow deadlines affects the deadline proximity ratio. Notably, the considered autoscalers show different variability in deadline proximity ratios. Comparing error factors 0.8 and 1.3, the variability increases when more deadlines are violated. However, the difference in variability of deadline proximity ratios between the autoscalers stays stable. For example, we can see that for all the error factors Adapt stably shows slightly higher variability in deadline violations and shows comparable to Reg mean deadline proximity ratios. Hist and ConPaaS which over-provision more exhibit better deadline proximity ratios.

Comparing Figure 14 with Figure 12b we can notice that with per-workload deadlines more medium and large jobs do not meet deadlines while with per-workflow deadlines more smaller jobs do not meet their deadlines. This is due to the method for calculating per-workflow deadlines. Workload 1 has the majority of small workflows. Smaller workflows get smaller "safety margin" after multiplying their response time in a system without an autoscaler by the error factor.

Type	AS	S_e			D_p per-workload, mean scenario				D_p per-workflow, err. factor 1.0			
		σ	γ_1	γ_2	mean	σ	γ_1	γ_2	mean	σ	γ_1	γ_2
General	React	0.42	2.93	9.13	1.06	**0.77**	0.63	–0.79	1.23	0.42	2.93	9.13
	React$^\diamond$	1.12	3.65	14.85	1.17	**0.77**	0.56	–0.8	1.57	1.12	3.65	14.85
	Adapt	0.55	3.65	16.14	1.08	0.78	0.63	–0.77	1.28	0.55	3.65	16.14
	Hist	**0.14**	4.2	20.6	**1.01**	**0.77**	0.62	–0.82	**1.05**	**0.14**	4.2	20.6
	Reg	0.59	3.3	11.04	1.08	**0.77**	0.61	–0.81	1.29	0.59	3.3	11.04
	ConPaaS	0.49	5.19	31.85	1.06	0.8	0.66	–0.75	1.18	0.49	5.19	31.85
	React*	34.75	2.81	6.76	4	1.56	**0.08**	–0.77	17.32	34.75	2.81	6.76
	Adapt*	25.12	2.64	5.75	4.65	1.48	–0.6	0.18	20.06	38.67	2.51	4.92
	Hist*	38.67	2.51	4.92	3.23	1.18	–0.45	**–0.15**	12.93	25.12	2.64	5.75
	Reg*	45.91	**2.13**	**2.99**	5.46	1.32	0.17	–0.43	25.57	45.91	**2.13**	**2.99**
	ConPaaS*	3.43	7.47	62.58	1.27	0.79	0.44	–0.89	2.11	3.43	7.47	62.58
WF-specific	Plan	0.47	2.84	7.82	1.08	0.78	0.63	–0.79	1.27	0.47	2.84	7.82
	Plan$^\diamond$	0.96	5.05	36.26	1.15	0.78	0.58	–0.77	1.48	0.96	5.05	36.26
	Token	0.44	3.04	11.28	1.08	0.79	0.67	–0.69	1.25	0.44	3.04	11.28
None	No AS	0	0	–3	1	0.77	0.64	–0.81	1	0	0	–3

Table 6. Additional *variability characteristics* of S_e and D_p metrics for per-workload and per-workflow cases for *Workload 1*. Best values in each column are highlighted in bold, except the No AS case. σ is standard deviation, γ_1 is skewness and γ_2 is kurtosis.

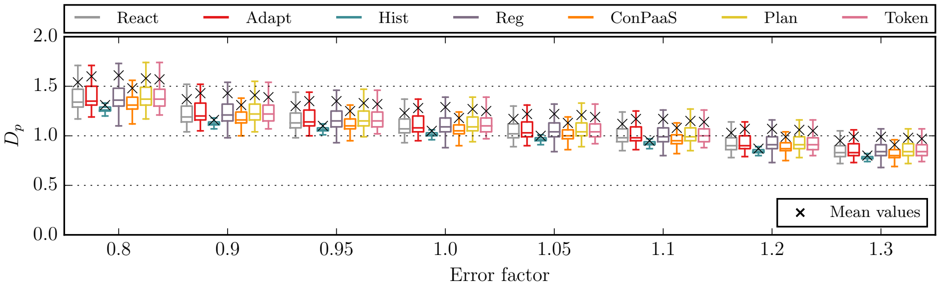


Fig. 13. The effect of changing enforced per-workflow deadlines on the deadline proximity ratio (for all workflows in Workload 1).

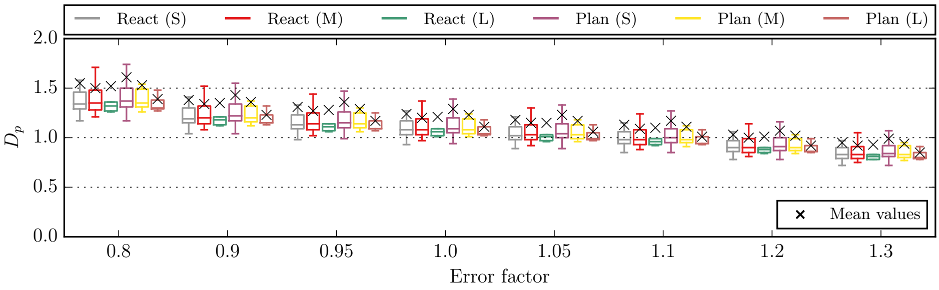


Fig. 14. The effect of changing enforced per-workflow deadlines on the deadline proximity ratio for React and Plan autoscalers plotted separately for (S) small, (M) medium, and (L) large workflow sizes in Workload 1.

6 ANALYSIS OF PERFORMANCE VARIABILITY

We analyze in more detail the performance variability of user metrics and focus only on Workload 1 since Workload 2 does not show significant differences in the results.

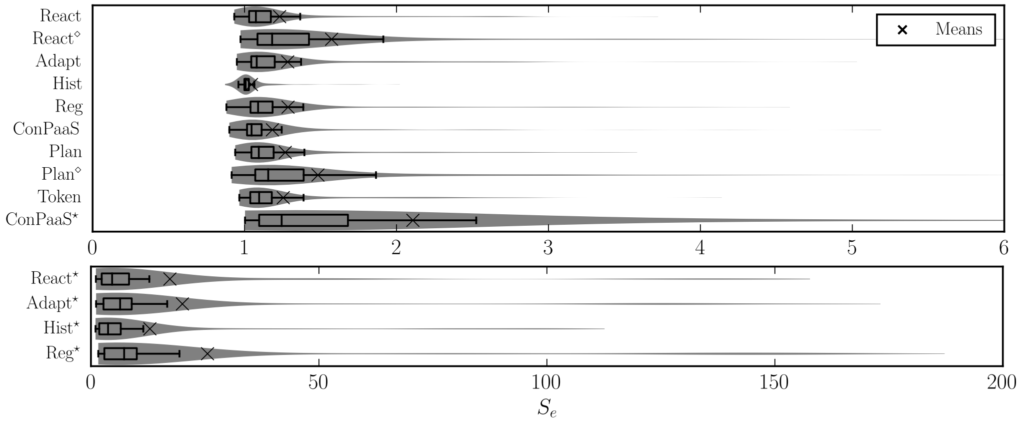


Fig. 15. Variability of elastic slowdowns for all the workflows in Workload 1. Values in wider parts of shaded areas are more probable than those in narrower parts.

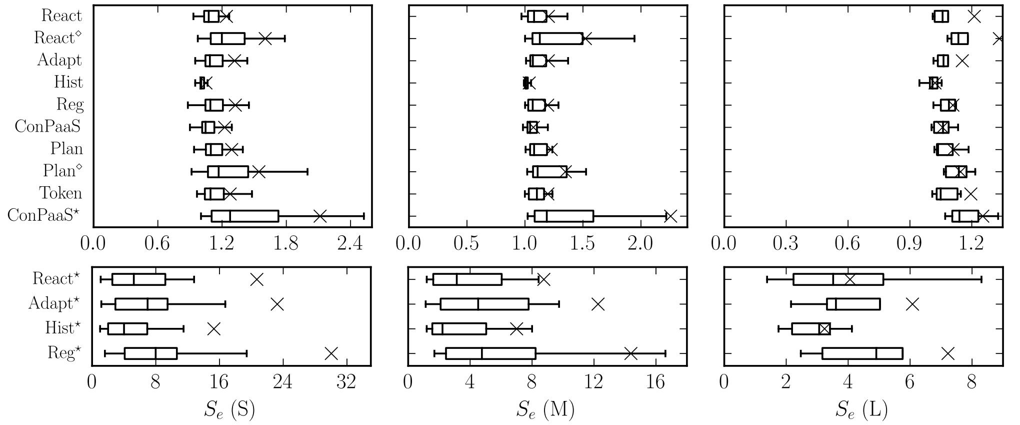


Fig. 16. Variability of elastic slowdowns for (S) small, (M) medium, and (L) large workflow sizes in Workload 1.

6.1 Overall

Figure 15 shows variability of elastic slowdowns for all the workflows in Workload 1. From the shape of the violin plots we can conclude that all the distributions are long-tailed. For autoscalers configured with service rate 1.0 the distributions are unimodal, and for autoscalers React⋆, Adapt⋆, Hist⋆, and Reg⋆ the distributions have second small peak on the right side of the S_e axis. Interestingly, heavy over-provisioning by Hist decreases elastic slowdown variability. Policies configured with service rate 14.0 show significantly higher variability and for this reason are plotted separately. ConPaaS, which over-provisions 2–3 times less than Hist, but still more than all the other autoscalers within the same configuration, shows variability comparable to e.g., React and Token. Increased VM booting time not only increases the mean value of the elastic slowdown but also doubles the variability. Adapt and Reg, while showing almost identical mean slowdowns, differ in the values which are below 1. Reg speeds up more workflows than Adapt does. The reason is probably related to the smoother downturns in the supply curve after demand decreases (Figure 10).

Additionally, in Table 6 we report numerical variability characteristics for all the policies with Workload 1: standard deviation (σ), skewness (γ_1), and kurtosis (γ_2) for S_e and for D_p for both deadline cases. For per-workload D_p we report only values for the scenario when the per-workload deadline is set based on the mean response time in a system without an autoscaler (Mean scenario, Section 5.5). For per-workflow D_p we present the situation when the error factor is 1.0. For both D_p cases we additionally report their mean values for all the policies with Workload 1. Note, S_e means are reported in Table 4. Skewness allows us to see how symmetric the distribution is. The sign of skewness shows in which direction the distribution is tilted. Comparing slowdowns in Table 6 with Figure 15, we can observe that positive skewness basically means that the tail of the distribution is located on the right from its mean. Higher kurtosis shows that the distribution has infrequent extreme outliers. Negative kurtosis means that the distribution is more "flat" and has "thinner tails", e.g., a uniform distribution.

6.2 Performance Variability per Workflow Size

Figure 16 depicts variability of elastic slowdowns per size group for (S) small, (M) medium, and (L) large workflows in Workload 1. We do not show the distributions in Figure 16 as it is long-tailed and similar to Figure 15. To better show the interquartile ranges we do not plot the outliers.

For all the autoscalers configured with service rate 1.0 large workflows show the lowest elastic slowdown variability, while small workflows suffer more. This trend is similar to the differences in deadline violation ratios between the same groups of job sizes in Figure 12b and Figure 14.

For longer booting VMs ($\diamond$) the variability of elastic slowdowns for large jobs is comparable to similar configurations with shorter VM booting times. Small- and medium-sized jobs are more affected by the longer booting VMs. General autoscalers with service rate 14.0 ($\star$) show much worse results and have many outliers which shift the mean values to the right. This confirms the importance of look into the variability of elastic slowdowns, especially when comparing the autoscalers.

7 AUTOSCALER CONFIGURATION AND CHARGING MODEL

Configuring autoscalers in public clouds is key to cost savings. Understanding the pricing model and how an autoscaler can affect the total costs is important. To give an example, we compare accounted costs, where the actual resource usage is paid for, and hourly-charged costs having a constant hourly, fixed-price scheme, e.g., like the one used by Amazon AWS for on-demand instances. Figure 17 shows accounted cost for each autoscaler and Figure 18 shows charged cost for each autoscaler.

Besides average accounted and charged cost per VM, we analyze the *charged speedup* $\tilde{C}$ and *runtime speedup* R compared to the scenario without autoscaling (see Table 4). While comparing *average accounted CPU hours per VM* $\overline{h}$ of the different autoscaler policies from Figure 17, it can be concluded that each autoscaling policy reduces the *average accounted CPU hours per VM*, i.e., the autoscalers e.g., React, Adapt, Reg, Plan, and Token have values between 3 and 4 CPU·h, whereas the no autoscaling scenario has 6.68 CPU·h. This result is more clear when we compute R. Here, the autoscaler Token and React use 2.02 times less VMs compared to the scenario where all VMs are running throughout the whole experiment. As the ($\star$) autoscalers adapt the system based on another resource demand, these autoscalers use less VMs than the others. Indeed, the achieved user metrics are worse compared to the other autoscalers.

Following the hourly pricing scheme, where the autoscalers start and stop VMs with no regard to the per-hour billing model, all autoscalers use more *average charged CPU hours per VM* $\overline{C}$ than the no autoscaling scenario (7 CPU·h), see Figure 18 or Table 4. This shows the danger of misconfiguring an autoscaler on a public cloud. One can end up paying between 50% to 600% more than when not using an autoscaler at all.

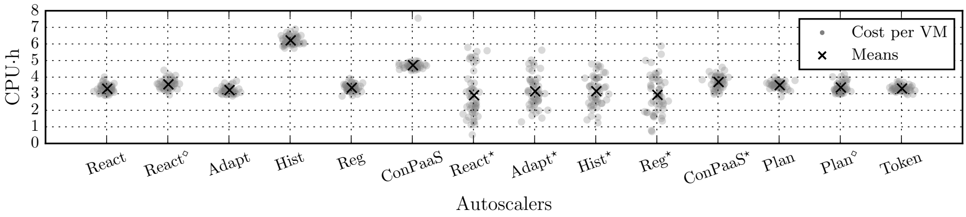


Fig. 17. Accounted cost in CPU-hours for all the considered autoscalers running Workload 1. For each policy, per VM costs are displayed with a slight random horizontal shift to avoid excessive superimposing.

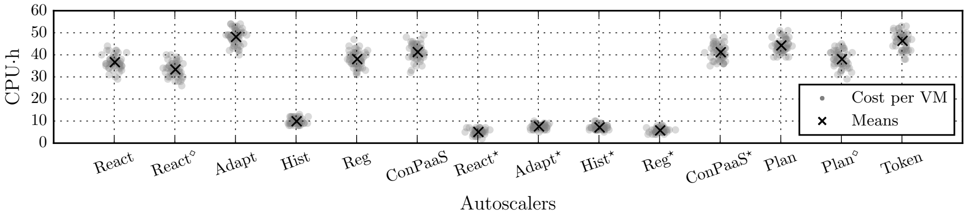


Fig. 18. Charged cost in CPU-hours for the autoscalers running Workload 1. For each policy, per VM costs are displayed with a slight random horizontal shift to avoid excessive superimposing.

8 WHICH POLICY IS THE BEST?

Considering all the computed metrics and all the autoscalers, there are many trade-offs when picking an autoscaler. Comparing for example only the average number of virtual machines $\overline{V}$ and average throughput $\overline{T}$ metrics could be insufficient. Definitely, there is no single best and the final choice of a policy depends on many factors: application choice, optimization goals, etc. In this section, we try to show some possible procedures to allow the comparison of autoscalers in such a multilateral evaluation. For all the assessments presented in this section we use the set of experiments with Workload 1. To include all the computed metrics into consideration, we utilize two ranking methods based on pairwise and fractional difference comparisons. Additionally, we aggregate elasticity and user metrics to scores normalized to a reference as done in benchmarking contexts.

8.1 Pairwise Comparison

In this section, we rank the autoscalers using the pairwise comparison method [15]. In this method, for each algorithm we pairwise compare the value of each metric with the value of the same metric of all the other autoscalers. When a metric's smaller value is better, for a pair of autoscalers A and B, autoscaler A accumulates one point if the value of this metric is lower than the value for autoscaler B. In case when bigger is better, autoscaler B gets the point. If both values are equal then both autoscalers get half point each. This method provides some ranking of the autoscalers, but its results do not fully capture the trade-offs as an autoscaler can get a point for being marginally better than another autoscaler in one metric, while being considerably worse than all of them in another metric.

We consider system metrics (a_U, a_O), (t_U, t_O), (k, k'), and user metrics S_e, the standard deviation of S_e, and $\overline{h}$. $\overline{T}$ and D_p are excluded since they correlate with S_e. We do not consider $\overline{a}_U$ and $\overline{a}_O$, as well as m_U due to redundancy with the selected accuracy metrics. The usage of $\overline{h}$ allows us to exclude $\overline{V}$ from the consideration. For all the metrics smaller value is better. The results of the comparison are given in Table 7. The bigger the number of points the better. The maximal number of points which each autoscaler can ever obtain is limited by the product of the number of considered metrics and the number of compared autoscalers.

AS	Pairwise	Fractional	Elasticity	User	Overall
	points	frac.	s_i	u_i	o_i
React	**79.5**	**2.46**	2.20	**1.28**	1.68
React$^{\diamond}$	49	5.30	1.99	1.09	1.47
Adapt	63.5	3.5	2.38	1.27	1.74
Hist	69.5	2.83	1.07	1.01	1.04
Reg	69	2.89	2.26	1.24	1.67
ConPaaS	61	3.12	1.34	1.10	1.21
React*	59	30.45	1.41	0.36	0.72
Adapt*	60	23.22	1.49	0.33	0.7
Hist*	53.5	33.62	1.3	0.4	0.73
Reg*	54.5	40.28	1.65	0.3	0.7
ConPaaS*	44	4.69	1.15	0.93	1.03
Plan	75	2.96	**2.67**	1.22	**1.81**
Plan$^{\diamond}$	61	5.42	2.28	1.16	1.62
Token	74	2.6	2.37	1.27	1.74
No AS	72.5	2.89	1	1	1

Table 7. The pairwise and fractional comparison, the aggregated elasticity and user metrics. The winners in each category (except No AS) are highlighted in bold.

8.2 Fractional Difference Comparison

In this section, we rank the autoscalers using the fractional difference method comparing all the autoscalers with an ideal case. For ideal case, we construct an empirical ideal system that achieves the best performance for all the metrics we consider. Note, this system does not exist in practice. Thus, the ideal system is a system which compiles all the optimal values for the considered metrics (here we use the same metrics as for Pairwise Comparison in Section 8.1), including the No Autoscaler case. For each metric m_i, we compute its best value b_i which is either minimum or maximum value from the set of metric's values, depending on the metric (e.g., among our metrics only for $\overline{T}$ the biggest value is the best). For each autoscaler the score p for the metric j is computed as following:

$$p_j := \sum_{i=1}^{M} \frac{|m_i - b_i|}{\max(b_i, \varepsilon)},$$

where M is the total number of considered metrics, and $\varepsilon > 0$, which is here set to $\varepsilon = 1$. The final score of an autoscaler is the average of all the individual p_j scores. The final score shows the fraction by which the autoscaler differs from the empirically established ideal system. Thus, the smaller the final score the better. The results of the comparison are given in Table 7.

8.3 Elasticity and User Metrics Scores

In this section, we aggregate elasticity and user metrics to scores as proposed by Fleming et al. [23]. As commonly done in the benchmarking domain, we select a baseline as reference to compute speedup ratios and in a second step compute the averages of the speedups using an unweighted geometric mean. We choose the metric results with no active autoscaler as baseline. The unitless scores allow for a consisted ranking of autoscalers with 1 as reference value. The larger the score, the better the rating. The scoring could be extended by user-defined weights.

For each autoscaler i we group the set of elasticity metrics based on the covered aspects into three groups: (I) accuracy as $a_i = a_{U,i} + a_{O,i}$, (II) wrong provisioning timeshare $t_i = t_{U,i} + t_{O,i}$, and (III) instability $\kappa_i = k_i + k'_i$. For the elasticity scores, we do not consider $\overline{a}_U$ and $\overline{a}_O$, and m_U metrics to avoid redundancy in elasticity aspects. The user score comprises the average runtime of VM instances $\overline{h}_i$ and the elastic slowdown $S_{e,i}$ as speedup ratios for each autoscaler i. The average throughput $\overline{T}$ is not considered as it is inversely dependent on the elastic slowdown S_e.

The elasticity scores s_i and user scores u_i are computed with respect to the baseline No Autoscaler scenario b for each autoscaler i. The overall score o_i is the geometric mean of elasticity scores s_c and user scores u_c:

$$s_i = \left(\frac{a_b}{a_i} \cdot \frac{t_b}{t_i} \cdot \frac{\kappa_b}{\kappa_i}\right)^{1/3}, \qquad u_i = \sqrt{\frac{\overline{h}_b}{\overline{h}_i} \cdot \frac{S_{e,b}}{S_{e,i}}}, \qquad o_i = \sqrt{s_i \cdot u_i}.$$

The resulting ranking is presented in Table 7. Using the described metric aggregation approach and concerning the elasticity s_i and overall o_i scores, Plan outperforms the general autoscalers. The React policy shows the best results from a user perspective in u_i, while our Token policy and the general Adapt policy follow React with a small score difference of 0.01. Hist and ConPaaS perform slightly better than a system without an autoscaler in this context. Strong impact on the autoscalers has the service rate parameter (★), a smaller impact can be observed for the experiments with longer provisioning time (◇).

9 THREATS TO VALIDITY

The limitations of the study are mainly expressed in the constrained number of considered job types and autoscalers. Improvements can be achieved by adding extra workloads with different characteristics to ideally consider wider spectrum of major job types that benefit from autoscaling. For example, data analytics workflows, streaming workflow applications, and workflows requiring quick reaction time [54]. Additionally, it is possible to report the job slowdown per workflow type. To make the study more applicable to cloud environments, one can extend the set of workflow-related autoscalers with algorithms which consider job deadlines and costs [13, 41].

One interesting aspect is the possible interpretations of the metric values. While our metrics are application-agnostic, their interpretation is not. They can be viewed as raw metrics which, in a proper service-level agreement, can be assigned with certain thresholds and interpretation.

The experimental setup used in this article could also be improved. Despite the fact that our private OpenNebula environment is rather representative, the number of concurrent users in Amazon EC2 and Microsoft Azure is much higher than in our case. Thus, it would be beneficial to consider public clouds to capture possible performance effects which could arise there. In addition, avoiding interval-based autoscaling in real setups could improve the quality of predictions by reacting to changes in the demand more quickly. We parametrize general autoscalers (computed service rate parameter) using the statistical properties of the whole workload as we have an access to this information. However, in the case when the workload properties are unknown different demand estimation methods can be used [48]. We do not analyze CPU utilization and RAM usage as for the considered workloads CPU and RAM information has low value as we primarily assign one task per VM and focus on performance characteristics from the perspective of job execution times.

The fractional difference comparison does not have upper bound for its scores. The comparison method can be improved by, for example, normalizing the set of values for the single metric by the maximum value from that set, but this could make the results incomparable with results obtained in other environments in the future. Another issue is related to the selection of metrics for tournaments to have a proper balance between autoscaling-, user-, and cost-oriented metrics within a single competition. The same stands for using the weights for prioritizing the considered metrics. The choice of metrics and weights solely depends on the goals which should be achieved. For this reason, since we are performing general comparison of divers autoscalers and we are not inclined towards certain metrics, we use multiple types of tournaments. It depends on the reader's preferences how the metrics and tournament results will be interpreted.

10 RELATED WORK

Our work provides the first comprehensive comparative experimental study of autoscaling for workflows. We are unaware of any similar study in terms of the methodology taken, the number of policies compared, the number of performance metrics, and the size of experiments run. The importance of comparing different autoscaling algorithms has been recently discussed in the literature but mostly from a theoretical point of view [37, 45]. One exception is a tool that tries to utilize the differences between different autoscaling policies to achieve better QoS for customers by selecting a policy based on the workload [4]. That work, nevertheless, does not include any experimental comparison or deep analysis between the performance of the autoscalers as we do in our work.

The problem of scaling workflows has been studied in the literature with a focus on designing new autoscaling policies. Malawski et al. [39] discuss the scheduling problem of ensembles of scientific workflows in clouds while considering cost- and deadline-constraints. Mao et al. [42] optimize the performance of cloud workflows within budget constraints. They propose two algorithms, namely, *scheduling-first* and *scaling-first*. Cushing et al. [13] deal with prediction-based autoscaling of scientific data-centric workflows. Buyn et al. [9] try to achieve cost-optimized provisioning of elastic resources for workflow applications. They use the *Balanced Time Scheduling (BTS)* algorithm to calculate the minimal required number of resources which will allow to execute the workflow within a given deadline. Dörnemann et al. [19] consider scheduling of Business Process Execution Language (BPEL) workflows in Amazon's elastic computing cloud. Their main findings include the methods to automatically schedule workflow tasks to underutilized hosts and to provide additional hosts in peak situations. The proposed load balancer uses the overall system load to take scaling decisions in contrast to other systems where the throughput is more important. Heinis et al. [25] propose a design and evaluate the performance of a workflow execution engine with self-tuning capabilities.

11 CONCLUSION AND ONGOING WORK

In this work, we propose a comprehensive method for comparing autoscalers when running workflow-based workloads in cloud environments. Our method includes a model for elastic cloud platforms, a set of over 15 relevant metrics for evaluating autoscalers, a taxonomy and survey of exemplary general and workflow-specific autoscalers, and experimental and analysis steps to conduct the comparison. Using our method, we evaluate 7 general and workflow-specific autoscalers, and several autoscaler variants, when used to control the capacity for a workflow-based workload running in a realistic cloud environment. Our results across the diverse metrics highlight the trade-offs of using the different autoscalers. At the best of our knowledge, the efficiency of general autoscalers was previously unknown for workflows. We show that although workflow-specific autoscalers have the privilege of knowing the workflow structure in advance, it is possible for properly configured general autoscalers to achieve similar performance. Our results demonstrate that a correct parametrization of general autoscalers is very important. In our case, the service rate parameter is not the only one to affect the performance of general autoscalers. In particular, VM booting times and the choice of the autoscaling interval are also crucial, as many general autoscalers are designed to stably operate when VM booting times do not exceed a certain threshold. Finding optimal values for parameters could be even impossible (as they could be implementation-related) and will probably require more experiments. Remarkably, our workflow-specific Plan autoscaler shows comparable results to the general React autoscaler and wins 2 out of 5 competitions while providing a good balance between operational costs and performance. The correct choice of an autoscaler is important but significantly depends on the application type. Thus, no single universal solution exists. In such a situation, the multilateral ranking methods which we use gain more importance. For the future, we plan to extend this work to consider other application models, such as request-response services and media streaming workloads, use other workflow arrival patterns, introduce deadline- and cost-awareness to workflow-specific autoscalers, and to conduct through the SPEC Research Group vendor-driven experiments.

12 ACKNOWLEDGMENTS

We thank Laurens Versluis and Mihai Neacșu from Vrije Universiteit Amsterdam for their thorough validation of our autoscalers, and anonymous reviewers for their helpful comments.

REFERENCES

[1] B. Abbott and others. 2008. Search for Gravitational Waves from Binary Inspirals in S3 and S4 LIGO Data. *Physical Review D* 77 (2008), 062002.

[2] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick H. J. Epema. 2012. Cost-Driven Scheduling of Grid Workflows using Partial Critical Paths. *IEEE TPDS* 23 (2012), 1400–1414.

[3] Ahmed Ali-Eldin and others. 2012. Efficient Provisioning of Bursty Scientific Workloads on the Cloud using Adaptive Elasticity Control. In *ScienceCloud Workshop*.

[4] Ahmed Ali-Eldin and others. 2013. *Workload Classification for Efficient Auto-Scaling of Cloud Resources*. Technical Report. Umeå University, Lund University.

[5] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. 2012. An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures. In *IEEE NOMS*.

[6] Guillermo A. Alvarez and others. 2001. Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM TOCS* 19 (2001), 483–518.

[7] Hamid Arabnejad and Jorge Barbosa. 2012. Fairness Resource Sharing for Dynamic Workflow Scheduling on Heterogeneous Systems. In *IEEE ISPA*.

[8] Shishir Bharathi and others. 2008. Characterization of Scientific Workflows. In *WORKS Workshop*.

[9] Eun-Kyu Byun and others. 2011. Cost Optimized Provisioning of Elastic Resources for Application Workflows. *FGCS* 27 (2011), 1011–1026.

[10] Jeffrey S Chase and others. 2001. Managing Energy and Server Resources in Hosting Centers. In *ACM SIGOPS*.

[11] T.C. Chieu and others. 2009. Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment. In *IEEE ICEBE*.

[12] Artem M. Chirkin and others. 2017. Execution Time Estimation for Workflow Scheduling. *FGCS* (2017).

[13] Reginald Cushing and others. 2011. Prediction-Based Auto-Scaling of Scientific Workflows. In *MGC Workshop*.

[14] Rafael Ferreira Da Silva and others. 2015. Online Task Resource Consumption Prediction for Scientific Workflows. *Parallel Processing Letters* 25 (2015).

[15] Herbert A. David. 1987. Ranking from Unbalanced Paired-Comparison Data. *Biometrika* 74 (1987), 432–436.

[16] Elias De Coninck and others. 2016. Dynamic Auto-Scaling and Scheduling of Deadline Constrained Service Workloads on IaaS Clouds. *JSS* 118 (2016), 101–114.

[17] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *ACM SIGPLAN Notices* 49 (2014), 127–144.

[18] Qiu Dishan and others. 2013. A Dynamic Scheduling Method of Earth-observing Satellites by Employing Rolling Horizon Strategy. *The Scientific World Journal* (2013).

[19] Tim Dornemann, Ernst Juhnke, and Bernd Freisleben. 2009. On-Demand Resource Provisioning for BPEL Workflows using Amazon's Elastic Compute Cloud. In *IEEE/ACM CCGrid*.

[20] Lipu Fei and others. 2014. KOALA-C: A Task Allocator for Integrated Multicluster and Multicloud Environments. In *IEEE Cluster*.

[21] Dror G. Feitelson. 2015. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press.

[22] Hector Fernandez, Guillaume Pierre, and Thilo Kielmann. 2014. Autoscaling Web Applications in Heterogeneous Cloud Infrastructures. In *IEEE IC2E*.

[23] Philip J. Fleming and John J. Wallace. 1986. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *ACM Communications* 29 (1986), 218–221.

[24] Anshul Gandhi and others. 2012. Autoscale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM TOCS* 30 (2012).

[25] Thomas Heinis and others. 2005. Design and Evaluation of an Autonomic Workflow Engine. In *IEEE ICAC*.

[26] Nikolas Herbst and others. 2016. *Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics*. Technical Report. SPEC Research Group, Cloud Working Group.

[27] Nikolas Herbst, Samuel Kounev, and Ralf Reussner. 2013. Elasticity in Cloud Computing: What it is, and What it is Not. In *ICAC*.

[28] Alexey Ilyushkin and others. 2017. An Experimental Performance Evaluation of Autoscaling Policies for Complex Workflows. In *ACM/SPEC ICPE*.

[29] Alexey Ilyushkin, Bogdan Ghit, and Dick Epema. 2015. Scheduling Workloads of Workflows with Unknown Task Runtimes. In *IEEE/ACM CCGrid*.

[30] Alexandru Iosup and others. 2011. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE TPDPS* (2011).

[31] Waheed Iqbal and others. 2011. Adaptive Resource Provisioning for Read Intensive Multi-Tier Applications in the Cloud. *FGCS* 27 (2011), 871–879.

[32] Mohammad Islam and others. 2012. Oozie: Towards a Scalable Workflow Management System for Hadoop. In *ACM SIGMOD Workshop SWEET*.

[33] Joseph C. Jacob and others. 2010. Montage: An Astronomical Image Mosaicking Toolkit. *Astrophysics Source Code Library* 1 (2010), 10036.

[34] Gideon Juve and others. 2017. Synthetic Workflow Generators. https://github.com/pegasus-isi/WorkflowGenerator. (2017).

[35] Jonathan Livny. 2012. Bioinformatic Discovery of Bacterial Regulatory RNAs Using SIPHT. In *Bacterial Regulatory RNA*.

[36] Dionysios Logothetis and others. 2010. Stateful Bulk Processing for Incremental Analytics. In *SoCC*.

[37] Tania Lorido-Botran and others. 2014. A Review of Auto-Scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing* 12 (2014), 559–592.

[38] Uri Lublin and Dror G Feitelson. 2003. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *JPDC* 63 (2003), 1105–1122.

[39] Maciej Malawski and others. 2012. Cost-and Deadline-Constrained Provisioning for Scientific Workflow Ensembles in IaaS Clouds. In *ACM/IEEE Supercomputing*.

[40] Maciej Malawski and others. 2015. Scheduling Multilevel Deadline-constrained Scientific Workflows on Clouds based on Cost Optimization. *Scientific Programming* (2015).

[41] Ming Mao and Marty Humphrey. 2011. Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *ACM/IEEE Supercomputing*.

[42] Ming Mao and Marty Humphrey. 2013. Scaling and Scheduling to Maximize Application Performance within Budget Constraints in Cloud Workflows. In *IEEE IPDPS*.

[43] Athanasios Naskos and others. 2015. Dependable Horizontal Scaling Based on Probabilistic Model Checking. In *IEEE/ACM CCGrid*.

[44] Simon Ostermann and others. 2008. On the Characteristics of Grid Workflows. In *CoreGRID Integration Workshop*.

[45] A. V. Papadopoulos and others. 2016. PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications. Tail Response Time Modeling and Control for Interactive Cloud Services. *ACM TOMPECS* (2016).

[46] Mayank Pundir and others. 2016. Supporting On-demand Elasticity in Distributed Graph Processing. In *IEEE IC2E*.

[47] Mustafizur Rahman, Xiaorong Li, and Henry Palit. 2011. Hybrid Heuristic for Scheduling Data Analytics Workflow Applications in Hybrid Cloud Environment. In *IEEE IPDPSW*.

[48] Simon Spinner and others. 2015. Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation* 92 (2015), 51 – 71.

[49] Domenico Talia. 2013. Toward Cloud-based Big-data Analytics. *IEEE Computer Science* (2013), 98–101.

[50] Ian J. Taylor and others. 2014. *Workflows for e-Science: Scientific Workflows for Grids*. Springer.

[51] Sachin Tilloo. 2017. Running Arbitrary DAG-based Workflows in the Cloud. http://www.ebaytechblog.com/2016/04/05/running-arbitrary-dag-based-workflows-in-the-cloud. (2017).

[52] Bhuvan Urgaonkar and others. 2005. An Analytical Model for Multi-Tier Internet Services and its Applications. In *ACM SIGMETRICS*.

[53] Bhuvan Urgaonkar and others. 2008. Agile Dynamic Provisioning of Multi-Tier Internet Applications. *ACM TAAS* 3 (2008), 1:1–1:39.

[54] Naga Vydyanathan and others. 2008. A Duplication Based Algorithm for Optimizing Latency under Throughput Constraints for Streaming Workflows. In *ICPP*.

[55] Katherine Yelick and others. 2011. The Magellan report on cloud computing for science. *US Department of Energy, Washington DC, USA, Tech. Rep* (2011).

Received June 2017