An Experimental Performance Evaluation of Autoscaling Policies for Complex Workflows

Alexey Ilyushkin[1], Ahmed Ali-Eldin[2], Nikolas Herbst[3],
Alessandro V. Papadopoulos[4], Bogdan Ghiț[1], Dick Epema[1], Alexandru Iosup[1]

[1]Delft University of Technology, the Netherlands [2]Umeå University, Sweden
[3]University of Würzburg, Germany [4]Mälardalen University, Sweden

{a.s.ilyushkin, b.i.ghit, d.h.j.epema, a.iosup}@tudelft.nl, ahmeda@cs.umu.se,
nikolas.herbst@uni-wuerzburg.de, alessandro.papadopoulos@mdh.se

ABSTRACT

Simplifying the task of resource management and scheduling for customers, while still delivering complex Quality-of-Service (QoS), is key to cloud computing. Many autoscaling policies have been proposed in the past decade to decide on behalf of cloud customers when and how to provision resources to a cloud application 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. In our work, we conduct an *experimental* performance evaluation of autoscaling policies, using as application model workflows, a commonly used formalism for automating resource management for applications with well-defined yet complex structure. 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 7 policies, we conduct various forms of pairwise and group comparisons. We report both individual and aggregated metrics. Our results highlight the trade-offs between the suggested policies, and thus enable a better understanding of the current state-of-the-art.

1. INTRODUCTION

Cloud computing is a model of outsourcing IT services on demand, pay-per-use. To make this model useful for a variety of customers, cloud operators have to simplify the process of obtaining and managing a useful supply of 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 already exist, both general autoscalers for request-response applications [5, 9, 43, 16, 34] and autoscalers for more task- and structure-oriented applications such as workflows [7, 14, 10, 37, 12]. 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, as no method currently exists to systematically evaluate and compare autoscalers. To alleviate this problem, in this work we propose and use the first systematic method to evaluate and compare experimentally the performance of autoscalers for workflow-based workloads running in cloud settings.

The lack of a method for comparing autoscalers derives in our view from scientific and industry practice. For the past decade, much academic work has focused on building basic mechanisms and autoscalers for specific applications, and thus there was little related work to compare to, and the threshold for publication has been kept low to develop the community. In industry, 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 [21, 36].) However, for the past two years a collaboration started within SPEC Research's Cloud Group that highlighted the need for deeper, systematic work in the evaluation and comparison of autoscalers, raising questions such as *How to evaluate the performance of individual autoscalers?*, and *How to compare autoscalers?*

Among the many application types, our focus on workflow-based workloads is motivated by two aspects. First, we are motivated by the increasing popularity [39, 40] of workflows for science and engineering [1, 25, 27], big data [28], and business applications [41], and by the ability of workflows to express complex applications whose interconnected tasks can be managed automatically on behalf of cloud customers [24]. Second, although generic autoscalers focus mainly on QoS aspects, such as throughput, response-time and cost constraints, state-of-the-art autoscalers can also take into account application structure [31]. *How does the performance of generic and of workflow-specific autoscalers differ?*

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 generic and of workflow-specific autoscalers depend on workflow-based workload characteristics?*

ICPE'17, April 22–26, 2017, L'Aquila, Italy

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 generic and workflow-specific autoscalers, for more than 10 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 6), 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.

2. A MODEL FOR ELASTIC CLOUD PLATFORMS

The autoscaling problem is an incarnation of the dynamic provisioning problem that has been studied in the literature for over a decade [8]. While in essence trying to solve the problem of how much capacity to provision given a certain QoS, most 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; 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 the autoscaling algorithms in two major groups: general and workflow-specific. Examples of general autoscalers include algorithms for allocating virtual machines (VMs) in data-centers, or algorithms for spawning web-server instances, etc. 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 running workflow structure, job dependencies, and expected runtimes of each task [32]. In most cases, the autoscaler is integrated with the task scheduler [31].

While 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. Most autoscaling-related papers seldom go beyond meeting some predefined metrics, e.g., with respect to response time or throughput. While the performance of most autoscalers is very dependent on how they are configured, this configuration is rarely discussed. Thus, 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 study aims to fill this gap by evaluating a set of algorithms in realistic setting.

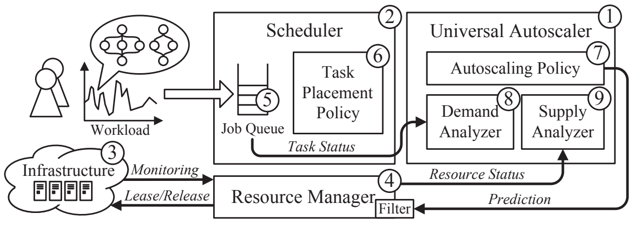


Figure 1: Elastic cloud platform.

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 the selected architecture of the proposed unified elastic cloud platform properly reflects the approaches used in modern commercial solutions.

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 (2) and a virtual infrastructure service (3) which maintains a set of computing resources. A resource manager (4) monitors the infrastructure and controls the resource provisioning. Users submit their complex jobs directly to the scheduler which maintains a single job queue (5). The tasks from the queued jobs are mapped on the computing resources in accordance to the task placement policy (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*. In contrast, in event-based autoscaling the autoscaler is invoked on every change in the demand curve. However, we do not use the event-based autoscaling approach because it can be derived from the interval-based autoscaling with rather short autoscaling interval. Additionally, the event-based autoscaling would make the experimental setup more complex and would require to incorporate extra processing logic into the scheduler.

The autoscaling service implements an autoscaling policy (7) and has a demand analyzer (8) which uses information about running and queued jobs to compute the momentary demand value. The supply analyzer (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.

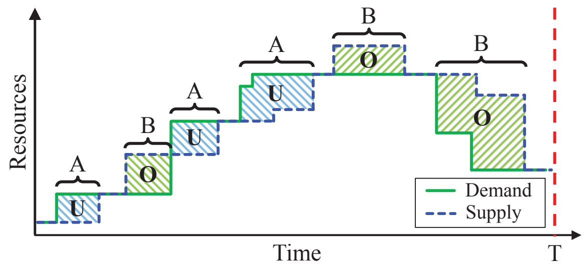


Figure 2: The supply and demand curves illustrating the under- and over-provisioning periods (A and B) quantified in the number of resources (areas U and O).

2.3 Workflows as Main Applications

For our experiments, we use complex workflows as a workload for our system. A *workflow* is a set of tasks with precedence constraints among them. Alternatively it can be represented in the form of a Directed Acyclic Graph (DAG). Each workflow task can start its execution when all of its input constraints are satisfied (e.g., when the 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 the considered autoscaling algorithms in more stringent conditions. Additionally, depending on the DAG structure, workflows can also reflect the behavior of other popular jobs 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 as the number of tasks it has. We focus on workflows which only consist of tasks requiring a single processor core.

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. Notably, some of these metrics have been endorsed by the Research Group of the Standard Performance Evaluation Corporation (SPEC) [21]. The user-oriented metrics are aimed to assess the impact of autoscaler usage on the speed of workflow execution.

3.1 Supply and Demand

All the considered system-oriented metrics are based on the analysis of discrete supply and demand curves. 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 workflows, we define the momentary demand as the number of eligible and running tasks in all the queued workflows, as in our model a resource can only process one task at a time. 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 *under-provisioning accuracy* metric a_U is defined as the average fraction by which the demand exceeds the supply:

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

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, $(x)^+ := \max(x, 0)$ is the positive part of x, and Δt is the time elapsed between two subsequent measurements. Analogously, we define the *over-provisioning accuracy* a_O as:

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

Figure 2 shows an intuition of the meaning of the provided accuracy metrics. 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 based on 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:

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

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

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. Thus, the challenge is to ensure that enough resources are provided at any point in time, but at the same time distinguish themselves from competitors by minimizing the amount of over-provisioned resources. Considering this, the defined separate accuracy metrics for over- and under-provisioning allow providers to better communicate their autoscaling capabilities and customers to select the provider or autoscaling algorithm 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. Thus we present an additional over-provisioning accuracy metric m_U which is equal to the average number of idle resources during the experiment time:

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

where u_t is the number of idle resources at time t.

3.3 Wrong-Provisioning Timeshare

The timing aspect of elasticity is characterized from the viewpoint of the *wrong-provisioning timeshare* on the one hand, and from the viewpoint of the induced *instability* accounting for superfluous or missed adaptations on the other hand [20].

The accuracy metrics allow no reasoning as to whether the average amount of under-/over-provisioned resources results from a few big deviations between demand and supply or if it is rather caused by a constant small deviation. To address this, the following two metrics are designed to 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 $\text{sign}(x)$ be the sign function of x, the overall timeshare spent in under- or over-provisioned states can be computed as:

$$t_U := \frac{1}{T}\sum_{t=1}^{T}(\text{sign}(d_t - s_t))^+ \Delta t,$$

$$t_O := \frac{1}{T}\sum_{t=1}^{T}(\text{sign}(s_t - d_t))^+ \Delta t.$$

3.4 Instability

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. The *instability* metric k captures this instability and inertia of elasticity mechanisms. 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 fraction of time the supply and demand curves move in opposite directions is defined as:

$$k := \frac{1}{T-1}\sum_{t=2}^{T}\min((\text{sign}(\Delta s_t) - \text{sign}(\Delta d_t))^+, 1)\Delta t.$$

Similarly, we define a complementary metric k' which captures the moments when the curves move towards each other:

$$k' := \frac{1}{T-1}\sum_{t=2}^{T}\min((\text{sign}(\Delta d_t) - \text{sign}(\Delta s_t))^+, 1)\Delta t.$$

If supply follows demand perfectly then both instability metrics are equal to zero.

3.5 User-Oriented Metrics

To assess the autoscaling policies from the user perspective, we employ the (average) elastic slowdown, which is defined in steps in the following way.

The *wait time* T_w of a workflow is the time between its arrival and the start of its first task. The *execution time* T_e of a workflow is the sum of the runtimes of all its tasks. 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 *slowdown* S of a workflow is its response time (in a busy system, when the workflow runs simultaneously with other workflows) normalized by its makespan T'_m in an empty system of the same size (when the workflow has exclusive access to all the resources): $S = T_r / T'_m$. 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 a workflow accelerates from the use of an autoscaler. We define the *average number of resources* $\overline{V}$ the system utilized during the experiment to compute the *gain* of using an autoscaler. We also calculate the *average task throughput* $\overline{T}$ which is defined as the number of tasks processed per time unit (e.g., second, minute or hour).

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.

4. AUTOSCALING POLICIES

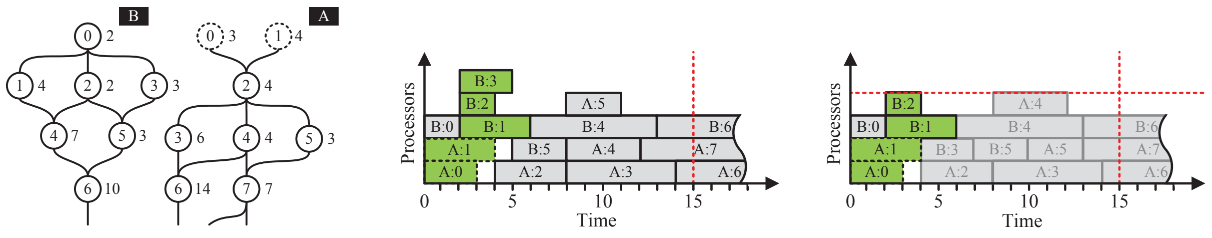
For evaluation, we select five representative general autoscalers and propose two workflow-specific autoscalers. We classify them using a taxonomy along two dimensions and summarize the survey of common autoscaling policies across these dimensions in Table 1. The taxonomy allows us to ensure the proper 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 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 [43] (with an earlier version published in 2005 [42]), 2009 [9], 2011 [23], 2012 [5, 3], and 2014 [16]. This is a representative set of the development of cloud autoscalers design across the past 10 years.

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 [40]. However, emerging workflow types which require quick system reaction and the usage of workflows in the areas in which they were less popular, e.g., for



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

Figure 3: The Plan autoscaling algorithm.

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 [18, 29] 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. [9] 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 rest of the paper, we refer to this technique as *React*. The authors introduce the scaling algorithm but provide no experiments to show the performance of the proposed autoscaler. 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. [5, 3] 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. The algorithm was tested using a simulated environment using a non-scaled version of the FIFA 1998 worldcup server traces, traces from a Google cluster and traces from Wikipedia.

4.1.4 The Hist Policy (Server, Long-term)

Urgaonkar et al. [43] 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 Virtual Machine (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*. The authors test their approach using two open source applications, RUBIS which is an implementation of the core functionality of an auctioning site, and Rubbos a bulletin-board application modeled after an online news forum. The testing is performed using 13 VMs running on Xen. Traces from the FIFA 1998 worldcup servers are scaled in time and intensity and used in the experiments. 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*) [23]. 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. [16]. 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 dimension 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 3 shows an example of the operation of the algorithm. In Figure 3a 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 3b 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 3c 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 [22] by simulating an execution "wave" through a DAG. The operation of the algorithm is illustrated in Figure 4. 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

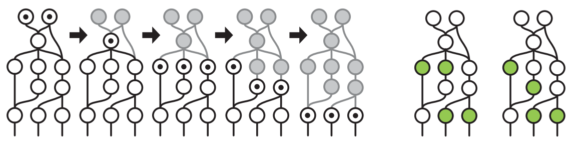

(a) Token propagation steps. (b) Two real LoPs.

Figure 4: The token-based LoP approximation.

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 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 4a the estimated LoP of 3 is lower than the maximal possible LoP of 4 in Figure 4b. However, in our previous work [22], 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 [25] 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. And SIPHT [27] is a bioinformatics workflow used to discover bacterial regulatory RNAs.

We generate synthetic workflows using a workflow generator by Bharathi et al. [26, 6]. 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 consisted 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 [35]. 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

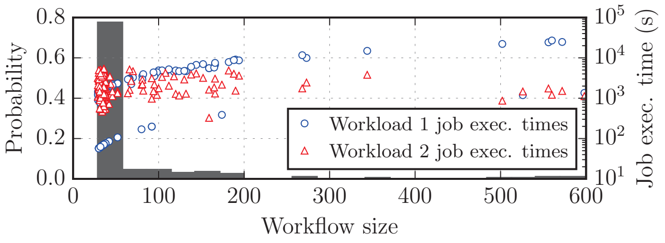


Figure 5: The distribution of job sizes in the workloads (histogram, left vertical axis) and the dependency between the job size and its execution time (glyphs, right vertical axis). The right vertical axis is in log scale.

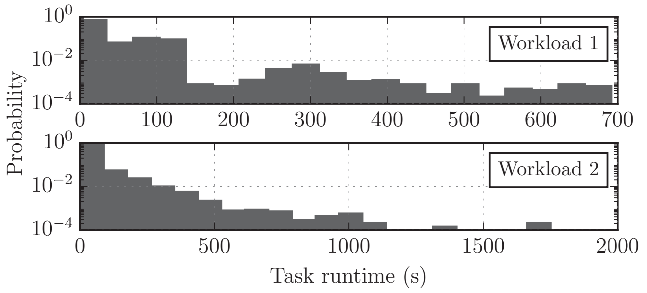


Figure 6: The distribution of task runtimes in the workloads (the horizontal axes have different scales, and the vertical axes are in log scale).

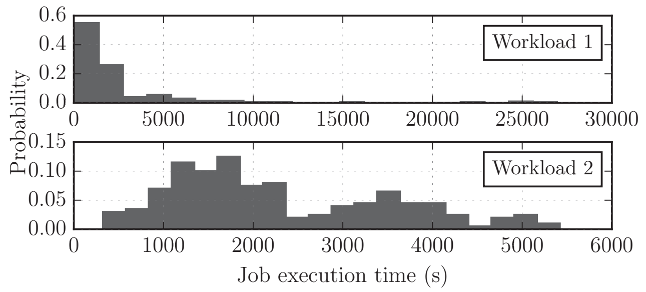


Figure 7: The distribution of job execution times in the workloads (all the axes have different scales).

job sizes in the workloads is presented in Figure 5. Figure 6 shows the distribution of task runtimes. Figure 7 shows the distribution of job execution times T_e in the workloads. 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 [30]. 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 [15] 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.

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

Property	Workload 1	Workload 2
Mean task runtime	33.52 s	33.29 s
Median task runtime	2.15 s	2.65 s
SD 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
SD 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	
SD of workflow size	98 tasks	
Total task number	13,876 tasks	

Table 2: Statistical characteristics of the workloads. SD stands for standard deviation.

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.

Compared to the job execution time, file transfer delays and the scheduling overhead are negligible. All tasks write their intermediate results directly to the shared storage to reduce data transfer delays for all workflows. A task can then run as soon as all of its dependencies are satisfied. Additionally, the runner copies all input files for a workflow to the virtual cluster before starting the execution. Thus, the impact from the file transfer delay between tasks on the system 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 [22]. During the experiment only the autoscaler has the access to the information about job execution times and task runtimes.

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 [42], or offline [18, 13]. 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 to the nearest integer the service rate calculated based on the mean task runtime in the workloads (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.

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 operational costs but slow down the jobs.
4. Long VM booting times negatively affect the performance.
5. No autoscaler outperforms all other autoscalers with all configurations and/or metrics.

5.4.1 Analysis of Elasticity Metrics

To better show the trade-offs between the autoscalers, we use the metrics described in Section 3. While calculating the system-oriented metrics, we excluded the periods where the demand curve exceeds the maximal available resource limit of 50 VMs. Since all the system-oriented metrics are normalized by the time of the experiment this approach does not bias the results.

The aggregated metrics for all the experimental configurations are presented in Table 3 and Table 4. 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 27% over-provisioning for Adapt.

The policies with longer booting VMs in Table 3 show slightly different results compared to the runs with faster booting VMs. Both React$^\diamond$ and Plan$^\diamond$ tend to under-provision more when the VM provisioning time is longer. The job slowdowns are also higher. We picked only these two policies to have one from each group of autoscalers. 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$^\diamond$ as the tasks more fully utilize provisioned VMs. m_U does not change for Plan$^\diamond$ as it over-provisions less.

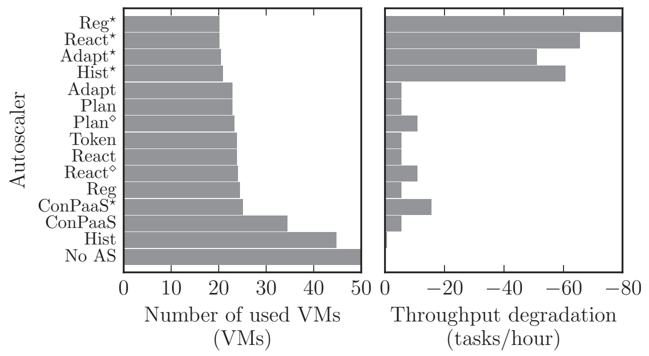


Figure 8: The average number of used VMs during the experiment and the average throughput degradation (compared with the no autoscaler case). All results are given for Workload 1.

For the general policies configured with service rate 1.0 and for workflow-specific policies in Table 3 and Table 4 job elastic slowdowns show low variability. We can conclude that the resources either significantly over-provisioned (Hist and ConPaaS) or already provisioned resources are running at low utilization (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 workloads. While Workload 1 has the majority of short jobs, Workload 2 has a more equal distribution of job execution times and thus less bursty. Elastic job slowdowns in both tables confirm this tendency. For Workload 2 they slightly increase (the Plan policy in Table 4 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.

The system-oriented metrics do not vary much between the workloads. For example, compare React in Table 3 with React in Table 4. 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 9 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 good and shows results similar to general autoscalers React, Adapt, and Reg running with service rate of 1.0. However, if policy follows the demand too close that increases job slowdowns. The Token policy, due to its specifics, needs to guess more while predicting the future demand and thus tends to over-provision a bit more.

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. Figure 10 shows the selection of general autoscalers running with the same workload as in Figure 9. The demand curves in these two figures look very different, as well as the supply curve does not follow the demand curve so close anymore. We do not show ConPaaS in Figure 10 as it has similar supply pattern as for service

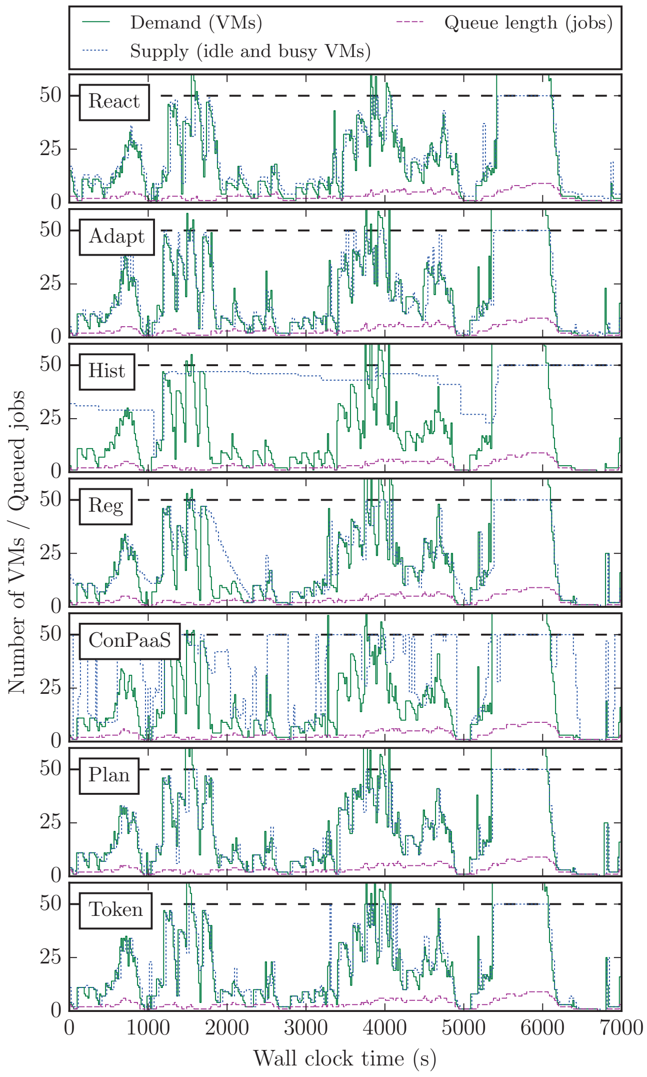


Figure 9: The experimental dynamics of five general autoscaling policies (Workload 1, service rate 1.0) and two workflow-specific policies during the cropped period of 7,000 s. The horizontal dashed line indicates the resource limit of 50 VMs.

rate 1.0, and Reg looks quite similar to React and Adapt. 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.

5.4.5 *The Trade-off Between Operational Cost 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 8. For example, for React the throughput degradation of –24 tasks per hour contributes only to 1.16% of the hourly throughput. In Figure 8, 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.

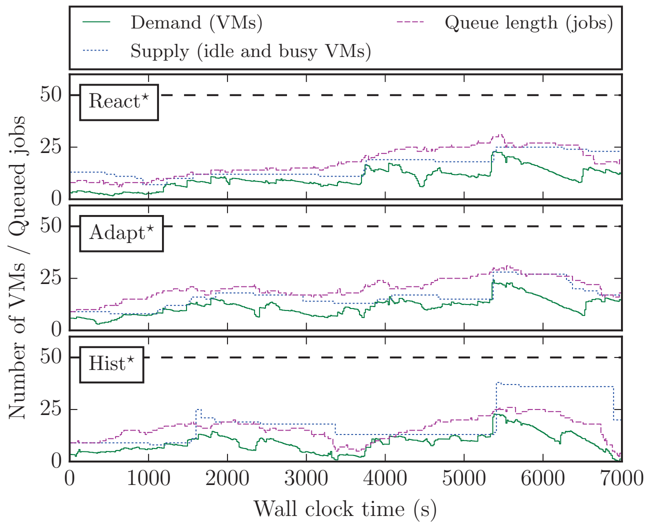


Figure 10: The experimental dynamics of three selected autoscaling policies (Workload 1, service rate 14.0) during the cropped interval of 7,000 s. The horizontal dashed line indicates the resource limit of 50 VMs.

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, Token, Plan, and Adapt show almost similar results for service rate 1.0. 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.

6. WHICH POLICY IS THE BEST?

At first glance, considering all the computed metrics and all the autoscalers it is hard to distinguish the winners. Comparing only $\overline{V}$ and $\overline{T}$ 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. Thus, it is necessary to establish a procedure 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 as the most comprehensive. To include all the computed metrics into consideration we utilize two raking methods based on pairwise and fractional difference comparisons. Additionally, we aggregate elasticity and user metrics using an approach from our previous work.

6.1 Pairwise Comparison

In this section, we rank the autoscalers using the pairwise comparison method [11]. 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. We consider system metrics (a_U, a_O), (t_U, t_O), (k, k'), and user metrics S_e, $\overline{V}$, and $\overline{T}$. We do not consider $\overline{a}_U$ and $\overline{a}_O$, as well as m_U due to redundancy with the selected accuracy metrics. For all the metrics except $\overline{T}$, smaller value is better. In case when smaller value is better, for a pair of two autoscalers A and B, autoscaler A accumulates one point if the value of its certain metric is lower than the value of the same metric of autoscaler B. In case when bigger is better, autoscaler B gets the

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
	React	2	6	5	50	15	84	20	32	7	1.23	1.24	1.20	1.21	2,071	23.89
	React$^\diamond$	6	5	13	40	32	64	21	32	6	1.57	1.60	1.52	1.33	2,066	24.01
	Adapt	4	4	8	27	23	51	21	34	5	1.28	1.32	1.20	1.15	2,071	22.86
	Hist	1	60	1	737	2	97	**17**	43	60	**1.05**	**1.05**	**1.04**	**1.02**	**2,076**	44.81
	Reg	3	8	6	51	17	51	20	**31**	8	1.29	1.32	1.20	1.11	2,071	24.42
General	ConPaaS	2	33	5	273	11	76	20	40	34	1.18	1.22	1.07	1.06	2,071	34.50
	React*	**0**	19	**0**	179	2	98	19	64	**0**	17.32	20.69	8.76	4.06	2,011	20.13
	Adapt*	**0**	16	1	150	4	96	19	63	**0**	20.06	23.25	12.26	6.08	2,026	20.49
	Hist*	**0**	25	1	463	5	95	20	60	1	12.93	15.30	7.00	3.24	2,016	20.87
	Reg*	**0**	12	1	78	5	94	20	62	**0**	25.57	30.04	14.38	7.22	1,997	**20.11**
	ConPaaS*	**0**	44	1	1092	**1**	98	21	45	7	2.11	2.12	2.26	1.25	2,061	25.15
	Plan	3	4	7	19	20	41	20	**31**	4	1.30	1.32	1.28	1.18	2,071	22.93
WF-specific	Plan$^\diamond$	8	**3**	17	**16**	38	**35**	21	32	4	1.64	1.72	1.44	1.38	2,066	23.31
	Token	3	6	7	35	16	53	20	33	7	1.25	1.28	1.20	1.20	2,071	23.88
None	No AS	0	73	0	869	0	100	17	43	73	1.00	1.00	1.00	1.00	2,076	50.00

Table 3: Calculated metrics for the main set of experiments with all the considered autoscalers and 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**	6	**11**	22	**39**	21	**32**	**4**	1.16	1.13	1.24	1.28	**1,905**	**21.89**
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 4: Calculated metrics for the additional set of experiments with selected autoscalers and 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.

point. If both values are equal then both autoscalers get half point each. The results of the comparison are given in Table 5. The bigger the number of points the better.

6.2 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 from Table 3 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 5.

6.3 Aggregated Elasticity and User Metrics

In this section, we aggregate both elasticity and user metrics as proposed by Fleming et al. [17]. As commonly done in the benchmarking domain, we select a baseline as reference to compute speedup ratios and then average the speedups using an unweighted geometric mean. We choose as baseline the metric results with no active autoscaler. We group the elasticity metrics based on the covered aspects into three groups: accuracy (a_U, a_O), wrong provisioning timeshare (t_U, t_O), and instability (k, k'). We do not consider $\bar{a}_U$ and $\bar{a}_O$, and m_U metrics. In addition, we compute a ratio based on the user metrics $\overline{V}$ (average number of VMs), the elastic slowdown S_e and the average throughput $\overline{T}$ to represent a balance between user-experienced performance and resource consumption. An overall ratio combines the user and elasticity ratios with the geometric mean. The resulting ranking is presented in Table 5. Using the described metric aggregation approach, the workflow-specific autoscaler Plan outperforms the generic ones. The Token policy is ranked on the same level as the generic Adapt and React policies. Hist and ConPaaS perform slightly better than 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 ($\diamond$).

7. 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 [44]. Additionally, it is possible to report the job slowdown per workflow type. To make the study more appli-

AS	Pairwise	Fractional	Elasticity	User	Overall
	points	frac.	frac.	frac.	frac.
React	73.5	**2.23**	2.20	**1.19**	1.62
React$^\diamond$	51.0	4.50	1.99	1.10	1.48
Adapt	66.0	3.17	2.38	**1.19**	1.69
Hist	70.0	2.83	1.07	1.02	1.04
Reg	69.5	2.53	2.26	1.17	1.62
ConPaaS	62.0	2.84	1.34	1.07	1.20
React*	57.0	2.96	1.41	0.52	0.85
Adapt*	60.0	3.37	1.49	0.49	0.86
Hist*	55.0	3.02	1.30	0.56	0.86
Reg*	56.0	3.94	1.65	0.45	0.87
ConPaaS*	44.5	2.06	1.15	0.98	1.06
Plan	**78.5**	2.68	**2.72**	**1.19**	**1.80**
Plan$^\diamond$	59.0	5.23	2.18	1.09	1.54
Token	71.0	2.36	2.37	**1.19**	1.68
No AS	72.0	3.01	1.00	1.00	1.00

Table 5: The pairwise and fractional comparison, the aggregated elasticity and user metrics. The winners in each category (except No AS) are highlighted in bold.

cable to cloud environments, one can extend the set of workflow-related autoscalers with algorithms which consider job deadlines and costs [32, 10].

One of the interesting aspects is related to possible meanings of metric values. In fact our metrics are application-agnostic but their interpretation is not. In this sense, they can be seen as a raw metrics which, however, in a proper service-level agreement, can be assigned with certain thresholds and interpretation.

The experimental setup used in this paper 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 [38]. 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.

8. 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 [29, 36]. One exception is a tool that tries to utilise 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. [31] discuss the scheduling problem of ensembles of scientific workflows in clouds while considering cost- and deadline-constraints. Mao et al. [33] optimize the performance of cloud workflows within budget constraints. They propose two algorithms, namely, *scheduling-first* and *scaling-first*. Cushing et al. [10] deal with prediction-based autoscaling of scientific data-centric workflows. Buyn et al. [7] 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. [14] 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. [19] propose a design and evaluate the performance of a workflow execution engine with self-tuning capabilities. The engine is purely reactive and does not employ workload prediction. It has an autonomic controller to automatically reconfigure itself to adjust to the changes in the resource demand.

9. CONCLUSION AND ONGOING WORK

The ability to select autoscalers is beneficial for customers and operators of cloud computing, because it enables simple control over cloud elasticity and thus facilitates an on-demand, pay-per-view delivery of IT services. In this work, we have proposed 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 10 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 have evaluated 7 generic 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 wins 4 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, and to conduct through the SPEC Research Group vendor-driven experiments.

10. REFERENCES

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

[2] S. Abrishami, M. Naghibzadeh, and D. H. J. Epema. Cost-Driven Scheduling of Grid Workflows using Partial Critical Paths. *IEEE TPDS*, 23:1400–1414, 2012.

[3] A. Ali-Eldin et al. Efficient Provisioning of Bursty Scientific Workloads on the Cloud using Adaptive Elasticity Control. In *ScienceCloud Workshop*, 2012.

[4] A. Ali-Eldin et al. Workload Classification for Efficient Auto-Scaling of Cloud Resources. Technical report, Umeå University, Lund University, 2013.

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

[6] S. Bharathi et al. Characterization of Scientific Workflows. In *WORKS Workshop*, 2008.

[7] E.-K. Byun et al. Cost Optimized Provisioning of Elastic Resources for Application Workflows. *FGCS*, 27:1011–1026, 2011.

[8] J. S. Chase et al. Managing Energy and Server Resources in Hosting Centers. In *ACM SIGOPS*, 2001.

[9] T. Chieu et al. Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment. In *IEEE ICEBE*, 2009.

[10] R. Cushing et al. Prediction-Based Auto-Scaling of Scientific Workflows. In *MGC Workshop*, 2011.

[11] H. A. David. Ranking from Unbalanced Paired-Comparison Data. *Biometrika*, 74:432–436, 1987.

[12] E. De Coninck et al. Dynamic Auto-Scaling and Scheduling of Deadline Constrained Service Workloads on IaaS Clouds. *JSS*, 118:101–114, 2016.

[13] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *ACM SIGPLAN Notices*, 49:127–144, 2014.

[14] T. Dornemann, E. Juhnke, and B. Freisleben. On-Demand Resource Provisioning for BPEL Workflows using Amazon's Elastic Compute Cloud. In *9th IEEE/ACM CCGrid*, 2009.

[15] L. Fei et al. KOALA-C: A Task Allocator for Integrated Multicluster and Multicloud Environments. In *IEEE Cluster*, 2014.

[16] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling Web Applications in Heterogeneous Cloud Infrastructures. In *IEEE IC2E*, 2014.

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

[18] A. Gandhi et al. Autoscale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM TOCS*, 30, 2012.

[19] T. Heinis et al. Design and Evaluation of an Autonomic Workflow Engine. In *IEEE ICAC*, 2005.

[20] N. Herbst et al. BUNGEE: An Elasticity Benchmark for Self-adaptive IaaS Cloud Environments. In *SEAMS*, 2015.

[21] N. Herbst et al. Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics. Technical report, SPEC Research Group, Cloud Working Group, 2016.

[22] A. Ilyushkin, B. Ghit, and D. Epema. Scheduling Workloads of Workflows with Unknown Task Runtimes. In *IEEE/ACM CCGrid*, 2015.

[23] W. Iqbal et al. Adaptive Resource Provisioning for Read Intensive Multi-Tier Applications in the Cloud. *FGCS*, 27:871–879, 2011.

[24] M. Islam et al. Oozie: Towards a Scalable Workflow Management System for Hadoop. In *ACM SIGMOD Workshop SWEET*, 2012.

[25] J. C. Jacob et al. Montage: An Astronomical Image Mosaicking Toolkit. *Astrophysics Source Code Library*, 1:10036, 2010.

[26] G. Juve et al. Synthetic Workflow Generators. https://github.com/pegasus-isi/WorkflowGenerator.

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

[28] D. Logothetis et al. Stateful Bulk Processing for Incremental Analytics. In *SoCC*, 2010.

[29] T. Lorido-Botran et al. A Review of Auto-Scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, 12:559–592, 2014.

[30] U. Lublin and D. G. Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *JPDC*, 63:1105–1122, 2003.

[31] M. Malawski et al. Cost-and Deadline-Constrained Provisioning for Scientific Workflow Ensembles in IaaS Clouds. In *IEEE SC*, 2012.

[32] M. Mao and M. Humphrey. Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *SC*, 2011.

[33] M. Mao and M. Humphrey. Scaling and Scheduling to Maximize Application Performance within Budget Constraints in Cloud Workflows. In *IEEE IPDPS*, 2013.

[34] A. Naskos et al. Dependable Horizontal Scaling Based on Probabilistic Model Checking. In *IEEE/ACM CCGrid*, 2015.

[35] S. Ostermann et al. On the Characteristics of Grid Workflows. In *CoreGRID Integration Workshop*, 2008.

[36] A. V. Papadopoulos et al. 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.

[37] M. Pundir et al. Supporting On-demand Elasticity in Distributed Graph Processing. In *IEEE IC2E*, 2016.

[38] S. Spinner et al. Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation*, 92:51 – 71, 2015.

[39] D. Talia. Toward Cloud-based Big-data Analytics. *IEEE Computer Science*, pages 98–101, 2013.

[40] I. J. Taylor et al. *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2014.

[41] S. Tilloo. Running Arbitrary DAG-based Workflows in the Cloud. http://www.ebaytechblog.com/2016/04/05/running-arbitrary-dag-based-workflows-in-the-cloud.

[42] B. Urgaonkar et al. An Analytical Model for Multi-Tier Internet Services and its Applications. In *ACM SIGMETRICS*, 2005.

[43] B. Urgaonkar et al. Agile Dynamic Provisioning of Multi-Tier Internet Applications. *ACM TAAS*, 3:1:1–1:39, 2008.

[44] N. Vydyanathan et al. A Duplication Based Algorithm for Optimizing Latency under Throughput Constraints for Streaming Workflows. In *ICPP*, 2008.