Performance-Feedback Autoscaling with Budget Constraints for Cloud-based Workloads of Workflows

Technical Report on the IEEE Cluster 2019 homonym article

Alexey Ilyushkin[1], André Bauer[2], Alessandro V. Papadopoulos[3],
Ewa Deelman[4], and Alexandru Iosup[5,1]

[1]Delft University of Technology, the Netherlands
[2]University of Würzburg, Germany
[3]Mälardalen University, Sweden
[4]University of Southern California, USA
[5]Vrije Universiteit Amsterdam, the Netherlands

a.s.ilyushkin@tudelft.nl, andre.bauer@uni-wuerzburg.de,
alessandro.papadopoulos@mdh.se, deelman@isi.edu, a.iosup@vu.nl

Abstract

The growing popularity of workflows in the cloud domain promoted the development of sophisticated autoscaling policies that allow automatic allocation and deallocation of resources. However, many state-of-the-art autoscaling policies for workflows are mostly plan-based or designed for batches (ensembles) of workflows. This reduces their flexibility when dealing with workloads of workflows, as the workloads are often subject to unpredictable resource demand fluctuations. Moreover, autoscaling in clouds almost always imposes budget constraints that should be satisfied. The budget-aware autoscalers for workflows usually require task runtime estimates to be provided beforehand, which is not always possible when dealing with workloads due to their dynamic nature. To address these issues, we propose a novel Performance-Feedback Autoscaler (PFA) that is budget-aware and does not require task runtime estimates for its operation. Instead, it uses the performance-feedback loop that monitors the average throughput on each resource type. We implement PFA in the popular Apache Airflow workflow management system, and compare the performance of our autoscaler with other two state-of-the-art autoscalers, and with the optimal solution obtained with the Mixed Integer Programming approach. Our results show that PFA outperforms other considered online autoscalers, as it effectively minimizes the average job slowdown by up to 47% while still satisfying the budget constraints. Moreover, PFA shows by up to 76% lower average runtime than the competitors.

Keywords: autoscaling, scheduling, workflow, performance, Airflow, workload, DAG, budget

Contents

1 Introduction 3

2 Problem Statement 4
- 2.1 Autoscaling Model . . . 4
- 2.2 Performance Metrics for Autoscaling . . . 5
  - 2.2.1 User- and System-Oriented Metrics . . . 6
  - 2.2.2 Elasticity-Oriented Metrics . . . 6

3 Autoscalers 6
- 3.1 Planning-First Autoscaler . . . 7
- 3.2 Scaling-First Autoscaler . . . 8
- 3.3 Performance-Feedback Autoscaler . . . 9
  - 3.3.1 Determining the Resource Profile . . . 9
  - 3.3.2 Token-based Demand Prediction . . . 11
  - 3.3.3 Scaling Down or Inflating the Profile . . . 12
  - 3.3.4 Allocating and Deallocating Resources . . . 12
  - 3.3.5 Task Placement . . . 12

4 Experiment Setup 12
- 4.1 Apache Airflow Deployment and Configuration . . . 12
- 4.2 Billing Setup . . . 14
- 4.3 Workloads . . . 14

5 Experiment Results 16
- 5.1 Algorithm Performance . . . 16
- 5.2 Workload Performance . . . 17
- 5.3 Elasticity Performance . . . 18
- 5.4 System-Oriented Performance . . . 20
- 5.5 Autoscaling Dynamics . . . 21

6 The Optimal Solution 22
- 6.1 Mixed Integer Programming Model . . . 22
- 6.2 Heuristics vs. the Optimal Solution . . . 24

7 Threats to Validity 25

8 Related Work 25

9 Conclusions 26

1 Introduction

The variety of workflow structures observed in modern cloud workloads and the diversity of cloud resource types require sophisticated autoscaling policies for meeting Service Level Agreements (SLAs). The problem of autoscaling for workflows previously has been often seen just from the perspective of a single user who submits a batch (an ensemble) of workflows to the cloud. Usually, it is supposed that the workflows in the batch have previously known runtime characteristics, e.g., task runtime estimates, obtained through a code analysis, a simulation, or by simply running the batch of workflows on a reference system.

This approach has been successfully adopted for executing batches of scientific workflows [21, 29, 33], which are well-studied [17] and have rather fixed patterns of execution [18], but can be too rigid for workflows in non-scientific domains [32]. Moreover, task runtime estimates have not been shown to be robust for batches in cloud settings, e.g., under multi-tenancy effects [16] and performance variability [24].

A more general approach assumes that the cloud user submits a *workload* of workflows of different types as, for example, if the cloud user runs an application serving many other, diverse users of that application. Many cloud-based services, such as Airbnb (rentals), Twitter (communication), and Netflix (video streaming), use this approach [25]. In this situation, the service can be considered as a single cloud user submitting a workload of workflow jobs. Scheduling batches of workflows in the cloud normally has the goal to minimize the makespan of the whole batch and staying within the budget. However, in the workload the number of arriving jobs can vary over time, thus, it is important to minimize the workflow response time and slowdown, as both these metrics include the possible queuing delay, and look at the system from the stability perspective as stability guarantees predictable and uninterrupted service.

Normally, a cloud user has some budget which expresses the financial limitations for allocating the cloud resources. When dealing with workloads of workflows, due to their dynamic nature, it is common that the user budget is defined per certain time interval. Since cloud resources are usually heterogeneous and have different costs, the basic goal of the autoscaler in this case is to allocate a desired number of resources and find such a combination of resource types that maximizes the workload performance while staying within the budget constraint. This also means that the resources can stay constantly running once allocated even if they are idle. The deallocation of resources will only be needed if that would help to adapt to the changing resource requirements of the workload, e.g., substitute lower number of expensive resources with higher number of cheaper ones if that helps to increase the performance. However, these requirements are not sufficient as multitenancy in the cloud could lead to possible performance degradation from a single user's perspective if there will be too many allocated but idle resources in the system. Moreover, to address modern sustainability challenges and to minimize energy waste, both end users and infrastructure owners should approach the use of resources responsibly. For example, users could be interested in saving the unused budget for achieving a sustainable business structure and minimizing the resource waste. Similarly, the infrastructure owners could have the interest in providing services that rely on sustainable infrastructure. In other words, it is not sufficient anymore to simply collect payments for the resource usage without controlling how the resources are actually utilized.

Most of the existing autoscalers for workflows belong to the group of *offline* policies [2, 5, 21]. Such policies, given a batch of workflows with known task runtimes, create a full-ahead task placement and autoscaling plan which is then strictly followed by the workflow management system that coordinates the execution. The workflows could be submitted simultaneously or with some delays—the main distinguishing feature of offline policies is that the submission times of all workflows are known in advance. Moreover, in the work by Wang et al. [31] a Mixed Integer Programming (MIP) approach was proposed that even allows to find the optimal solution to the autoscaling problem.

Only few papers consider the *online* autoscaling scenario where workflows arrive over time forming a workload [7, 22, 23], and the arrival times are not known in advance. However, such group of autoscalers mostly use the *online plan-based* approach, as they create a partial plan for the next autoscaling interval and not for the whole time horizon. The main issue with various plan-based scheduling approaches is the excessive time complexity they show when applied to workloads of workflows [30]. That could negatively affect the stability of the system in case if the autoscaling decisions take too much time, and do not scale well with workload fluctuations. Shorter autoscaling intervals that are more in line with the current trend on fine-grained billing [27] further complicate the problem, as plan-based autoscalers simply do not have enough time for making their decisions. Moreover, the plan-driven task placement can possibly delay the execution of newly arrived workflows as the scheduler will need to wait until the new plan incorporating the newly arrived tasks is constructed. In this case, the plan-based computationally-intensive solutions are not beneficial and can be substituted by simpler and faster heuristic approaches.

Thus, we clearly see further possibilities for improving autoscaling for workloads of workflows by joining the concepts inherent both to the general [3] and workflow-aware autoscalers [23]. From general autoscaler we can use the performance-feedback mechanism and the ability to derive and analyze runtime statistics during the execution. For example, instead of trying to derive task runtime estimates [9], we can look at task throughput, as the system can observe the task throughput fairly easily. From workflow-aware autoscalers, instead of constructing a partial plan, we can use less computationally intensive techniques for estimating the expected level of parallelism, and, accordingly, the resource demand.

Thus, the main research questions in this paper are:

Q1. *How to minimize workflow slowdowns within the budget constraint with unknown in advance task runtime estimates when autoscaling cloud resources for workloads of workflows?*

Q2. *Does the autoscaling policy found when answering Q1 has lower time complexity than the state-of-the-art plan-based online autoscalers?*

Q3. *How far is the performance and scalability of the policy found in Q1 from the optimal solution?*

When answering the raised questions, the contributions of this paper are the following:

1. We answer Q1 by proposing a novel online dynamic Performance-Feedback Autoscaler (PFA) that uses the resource task throughput information and a token-based LoP estimator (Section 3).
2. Through real-world experiments, PFA answers Q2 favourably by outperforming two state-of-the-art plan-based online autoscalers Planning First (PLF) and Scaling First (SCF) (Section 5).
3. We answer Q3 by comparing all the considered autoscalers with the optimal solution obtained from a Mixed Integer Programming model (Section 6).

2 Problem Statement

This section presents the model for the problem of autoscaling for workloads of workflows. The section also presents a set of metrics we use to evaluate the performance of the workloads and the performance of the studied autoscalers.

2.1 Autoscaling Model

We consider a public cloud computing system which is a subject to an arriving workload of workflows. The workload consists of multiple independent sub-workloads each belonging to an independent user. Each job in the workload is a workflow, and each component of the workflow is a task. Each workflow has single entry task and single exit task. The tasks belonging to a single workflow can exchange data through a shared file system. The task is considered *eligible* when all of its precedence constraints are satisfied, e.g., when all of its required input files are available in the shared file storage. The *workflow size* is the total number of tasks in a workflow.

The cloud computing system allows every user to dynamically allocate and deallocate computing resources of various types, where each resource type has a specific cost. Each resource can be in either of the following four states: *down*, *idle*, *busy*, or *booting*. The resource is *down* when it is deallocated and it is not reserved for any user. The resource is *idle* when it is allocated, reserved for a certain user, but has no currently assigned task. The resource is *busy* when it has a task assigned. The resource is *booting* when it is in the transition state between the down and idle states. Once the resource is allocated, the user is charged and the resource is reserved for the user until the end of the resource billing period, where the *billing period* is the minimal time for which the cloud resource can be reserved for a particular user. Each user has a certain operational budget per autoscaling interval, and the total cost of all the resources reserved for the user on the autoscaling interval cannot exceed the user budget. After the allocation, the resource spends some time in the booting state, while already being reserved for the user, without being able to execute any user tasks. At the end of the billing period, the resource can be deallocated or its reservation can be prolonged for the next billing period. In our model, the duration of the billing period equals to the duration of the autoscaling period. Before transitioning into the down state for deallocation, the resource should always pass the idle state first. The resource deallocation happens instantaneously. The *system size* is the maximal resource capacity which is available for the system users.

Since the number of eligible tasks from each user varies over time, the system employs an *autoscaler* to automatically control the number of allocated resources on per-user basis. The separate *scheduler* is responsible for placing tasks onto the allocated resources. In this work, we focus on periodic autoscaling, so that the autoscaler is invoked at fixed intervals by the workflow management system and monitors the controlled cloud environment. Accordingly, the *autoscaling interval* is the time between any two invocations of the autoscaler.

Despite that the system has resources of different types, we assume that there is no direct dependency between the cost of a resource type and the execution speeds of tasks running on it. Our motivation is based on the assumption that while some tasks can benefit from additional CPU cores, other tasks can be sequential in their nature and, thus, can show better performance on resources with fewer cores but with higher CPU frequency. Similar assumptions can be made for different RAM or storage requirements, etc.

The autoscaler and the scheduler operate in tandem with the goal to *minimize workflow response time within the budget constraint.* This can be achieved by allocating enough resources and finding an appropriate resource profile which guarantees required performance. By *resource profile* we understand a specific combination of resource types within the set of resources currently allocated for the user. Additionally, the autoscaler can have a goal to achieve fairness among multiple users. Since the resources are reserved for each user until the end of their billing period, during that period each resource can execute only tasks from the reserving user. This implies that the scheduler is not able to control the fairness among the users as it is only allowed to place tasks belonging to a certain user to the resources that are reserved for the same user. Thus, the only way to control fairness, by which in this chapter we understand maintaining average task throughput proportional to the user budget, is by controlling the number of allocated resources within the resource profile. Thus, if the autoscaler is fairness-aware, it should consider in addition to the budget constraint also the fairness constraint.

We do not include deadlines in this study as, in contrast to the offline approach, in the dynamic workload scheduling deadlines can only be roughly estimated. Furthermore, for workloads the deadline compliance depends on the system utilization, thus, the deadlines that were derived previously at a certain utilization level can be easily invalid for other utilizations. The dynamic nature of autoscaling for workloads makes the response time minimization and the stability of the system more important goals rather than the deadline compliance. It is also reasonable to assume that the response time minimization usually increases the number of met deadlines. Additionally, in our model we allow users to assign numeric priorities to workflows so that they can indicate which workflows are more important and should be processed faster.

2.2 Performance Metrics for Autoscaling

The system is constantly monitored by its users and operators, who assess its performance for a set of metrics commonly used in autoscaling settings [14].

2.2.1 User- and System-Oriented Metrics

As a main user-oriented metric we use *slowdown* which is defined in steps as follows: The *waiting time* is the time that a workflow spends in the system before starting executing its first task. The *makespan* is the time between the start of the first task of the workflow and until the completion of its last task. The *response time* of a workflow is the sum of its waiting time and its makespan. The *slowdown* is the ratio of response time of a workflow in a busy system to the ideal workflow makespan obtained from a reference system. We also consider the *monetary cost per autoscaling interval* as a user-oriented metric. By monetary cost we understand the total cost of the allocated resources during any autoscaling interval. As a system-oriented metrics we use the *percentage of busy resources* throughout the experiment and the *percentage of allocated resources per autoscaling interval.*

2.2.2 Elasticity-Oriented Metrics

To evaluate the performance of the considered autoscaler, we take the elasticity into account. In our model, we allow each resource to run only a single workflow task at a time. Accordingly, the momentary *demand* equals to the number of currently running and eligible workflow tasks (submitted by a particular user). By the resource *demand* d_t we understand the minimal number of resources required for fulfilling a given performance-related Service Level Objective (SLO) at time $t \in [1, T]$. By the resource *supply* s_t we understand the number of currently allocated (to the user) resources which are not in the down state at time $t \in [1, T]$. The maximal number of resources that can be supplied R is limited.

The *under-provisioning accuracy* a_U is defined as the average fraction of missing resources required to meet the SLO. Similarly, the *over-provisioning accuracy* a_O is the average fraction of resources that the autoscaler supplies in excess of the current demand. Both metrics can be formulated as:

$$a_U = \frac{1}{T \cdot R} \cdot \sum_{t=1}^{T} \max(d_t - s_t, 0), \tag{1}$$

$$a_O = \frac{1}{T \cdot R} \cdot \sum_{t=1}^{T} \max(s_t - d_t, 0). \tag{2}$$

The *under-provisioning time share* t_U is the time relative to the measurement duration, in which the system has insufficient resources, whereas, the *over-provisioning time share* t_O is the time relative to the measurement duration, in which the system has more resources than required. Both metrics can be computed as:

$$t_U = \frac{1}{T} \cdot \sum_{t=1}^{T} \max(\text{sgn}(d_t - s_t), 0), \tag{3}$$

$$t_O = \frac{1}{T} \cdot \sum_{t=1}^{T} \max(\text{sgn}(s_t - d_t), 0). \tag{4}$$

3 Autoscalers

This section explains in detail two state-of-the-art budget-aware autoscalers, that require task runtime estimates for their operation, and presents our novel autoscaler, which, in contrast, operates without explicitly provided task runtime estimates. The considered state-of-the-art autoscalers were proposed by Mao and Humphrey [23] and designed specifically for workloads of workflows. The relevance of these autoscalers is supported by the recent survey [20].

3.1 Planning-First Autoscaler

The Planning First (PLF) [23] autoscaling policy uses currently eligible tasks to allocate resources within a budget constraint. Even though the name of the policy in the original paper is Scheduling First, further we refer to it as Planning First, as this policy basically creates an execution plan for the tasks within the autoscaling interval. The autoscaler consists of six steps which are executed on every policy invocation, i.e., for every autoscaling interval:

i. Distribute the user budget among the workflows based on their priority.

ii. Perform initial supply prediction by determining the number of each resource type to allocate within the budget constraint.

iii. Consolidate the budget left after the initial supply prediction.

iv. Allocate the resources according to the predicted supply.

v. Create an execution plan for the upcoming autoscaling interval.

vi. Deallocate idle resources which do not have any tasks planned and are approaching the end of their billing period.

In the first step, the policy computes the cost of already allocated resources, deducts their cost from the user budget, and distributes the remaining budget to individual workflows proportionally to their priority, so that higher priority workflows get bigger budgets.

In the second step, the policy iterates through the eligible tasks of the workflow, sorted in the descending order of their workflow priorities, and for each task, while there is enough budget, it finds the resource type allowing to finish the task in the shortest time. The tasks are not assigned to the resources, only the number of resources of each type is determined. If the budget is over, the autoscaler proceeds to the third step—the budget consolidation. In the original paper, the loop break condition depends on the cost of the cheapest resource in the system so that already after the second step the policy can overspend the budget (for each workflow) by the cost difference between the fastest resources and the cheapest one. To avoid this, we modify the policy and use the cost of the fastest resource for the currently processed workflow task instead.

In the third step, the policy performs budget consolidation, as some budget can be left by individual workflows after the initial supply prediction. There are two reasons why the initially distributed budget may not be fully spent: some workflows could have not enough eligible tasks, or some workflows could have remaining budget smaller than the cost of the fastest resource. So that these remaining per-workflow budgets can be redistributed among the workflows from the same user to include more fastest resources in the allocation plan. This allows to determine fastest resource types for the remaining higher priority eligible tasks that were not processed in the second step. After this step, the autoscaler produces the final predicted number of instances of each type which should be allocated. It also specifies for some or all eligible tasks on which resource types they should run. Some eligible tasks belonging to lower priority workflows still could be without assigned resource types, as the cost of their fastest resources did not fit within the budget constraint.

In the fourth step, the policy performs so-called resource consolidation which basically means creation of an execution plan on the already allocated (at the moment of the autoscaler invocation) and newly allocated resources (after the third step) for the upcoming autoscaling interval. For that, the policy determines actual resources (not just the resource types) for each workflow task and tries to fill the resources in the plan with tasks until the end of the autoscaling interval. This is necessary, as after the third step only (a subset of) eligible tasks get the resource type assigned—those, that were used to predict the supply. Accordingly, the number of resources in the plan equals the number of running tasks and the number of tasks that have the resource type assigned after the third step. As the original paper used simulations, many very important details, which are crucial when implementing the policy in a real system, are missing or imprecise. Further we provide our interpretation of the resource consolidation step.

In the fourth step, the policy allocates the resources according to the predicted supply.

In the fifth step, the policy performs resource consolidation, i.e., it creates a task placement plan for the upcoming autoscaling interval, while processing the workflows in the random order. The newly allocated resources are considered as booting, thus, the planner takes into account the allocation delay which is supposed to be known in advance. The execution plan is initialized with tasks that are already running at the moment of the autoscaler invocation. Then the policy adds in the plan the eligible tasks that got the resource type assigned during the second or third autoscaling steps. The eligible tasks with known resource types are first assigned to idle resources of that type. If there are no idle resources, the planner checks the booting and busy resources of the same type, which of those will become available earlier, and places the eligible tasks on the earliest one. After that, all the resources in the plan should have at least one task assigned. Finally, all the remaining eligible and not yet eligible tasks are processed while maintaining the precedence constraints, i.e., a task is added to the plan if all of its parents are already in the plan. Each task is placed on the resource which is at the moment of task placement provides the minimal earliest possible start time. The planning process continues until there are no tasks that can start their execution before the end of the autoscaling interval.

Finally, in the sixth step, the resources that did not get any tasks assigned in the previous steps and that are approaching the end of their billing period are deallocated.

3.2 Scaling-First Autoscaler

The Scaling First (SCF) [23] autoscaling policy first creates for each workflow an individual execution plan (without considering resource allocation constraints), and then scales the plan so that it fits within the user budget constraint. The policy consists of five major steps:

i. Perform initial supply prediction by creating a per-workflow execution plan without limiting the number of resources.

ii. Scale the initial prediction to fit within the budget constraint, and consolidate the remaining budget.

iii. Allocate the resources according to the predicted supply.

iv. Create an execution plan for the upcoming autoscaling interval.

v. Deallocate idle resources in the same way as in the PLF policy.

In the first step, the policy creates an independent (from other workflows) per-workflow plan neither considering the system resource allocation limits nor considering the budget constraint. Thus, the number of resources in each plan can be bigger than the actual number of maximally available resources in the system. Since the original paper does not clearly explains this step, we present our detailed interpretation of the procedure for creating the per-workflow plan which uses similar logic as the resource consolidation step. First, the policy selects all the already running tasks of the current workflow and places all of them in the plan. Their resource types are already known, as well as the expected finish times. Second, the policy selects all the eligible tasks and places them on their fastest resource types, calculating the appropriate expected finish time. Third, all the other not yet eligible tasks are placed in the plan on their fastest resources (if those required fastest resources are not yet in the plan then they are added) so that the earliest possible start time for each task is minimized at the moment of its addition to the plan. Similarly to the resource consolidation step of PLF, a task is added to the plan only if all of its parents are already in the plan. The final number of resources for each resource type that should be supplied is calculated as the rounded up sum of the runtimes of planned tasks on each resource type divided by the length of the autoscaling interval.

In the second step, for each resource type the policy proportionally scales the initially predicted supply by multiplying it by the factor calculated as the fraction of the user budget and the total cost of initially predicted resources. Since the number of resources is integer, some remaining budget can be left after scaling the initial supply. This remaining budget is used to allocate more resources, if possible. For that the policy iterates in a round robin manner through the predicted in the first step resource types until even the resource of the cheapest type cannot be allocated.

In the third step, the policy allocates the resources according to the predicted supply.

In the fourth step, the policy performs resource consolidation. We modify the resource consolidation approach described in the original paper for SCF, as it is does not mention the situation when the number of resources of a certain type after the scaling step is zero. Instead, we use the approach similar to our interpretation of resource consolidation for the PLF policy. There are two differences between SCF and PLF. First, in PLF before the resource consolidation step some (or all) eligible tasks have the resource type already assigned, while in SCF the information on the preferred resource types from the first step is completely discarded. Second, in SCF the tasks are added to the plan in the order of their workflow priorities, so that higher priority tasks are added to the plan earlier.

The fifth step of the SCF policy is identical to the resource deallocation step of PLF.

3.3 Performance-Feedback Autoscaler

In this section, we present our novel Performance-Feedback Autoscaler (PFA), which we developed considering the limitations of the state-of-the-art workflow-specific autoscalers, and based on observations on the performance of general and workflow-specific autoscalers from the literature [14, 21].

We expect PFA to achieve better elasticity performance, as it constantly monitors the historical resource throughputs to derive faster resource types, and relies on a low complexity workload approximator to predict the future demand. Moreover, the dynamic task placement, used together with PFA, is expected to further reduce task waiting times and increase the resource utilization.

The PFA autoscaler consists of the following steps:

S1. Determine the resource profile using the historical throughputs.

S2. Determine the number of resources (the supply) that can be allocated within the resource profile with the user budget.

S3. Estimate future workload resource demand using the token propagation approach and historical throughput information.

S4. Scale down the profile-based supply if it is higher than the predicted demand to avoid wasting resources.

S5. Allocate the predicted number of resources.

S6. Deallocate idle resources which are staying idle the longest and approaching the end of their billing period.

3.3.1 Determining the Resource Profile

The first two steps of the autoscaler use throughput information to derive the initial resource profile. PFA relies on two alternative smoothing mechanisms for the historical throughput: Moving Average (MA) and Exponentially Weighted Moving Average (EWMA) for smoothing out possible throughput fluctuations.

In the first step, on the autoscaling interval t for each resource type i and each user j the average resource throughput $\tau_{i,j}(t)$ is defined as:

$$\tau_{i,j}(t) = \begin{cases} \frac{c_{i,j}(t)}{n_{i,j}(t)}, & \text{if } n_{i,j}(t) > 0, \\ 0, & \text{otherwise}, \end{cases} \tag{5}$$

where $c_{i,j}(t)$ is the number of completed tasks on the interval, and $n_{i,j}(t)$ is the number of allocated resources on the interval. This allows to compute the instant throughput-based resource type ratios:

$$\hat{\rho}_{i,j}(t) = \begin{cases} \dfrac{\tau_{i,j}(t)}{\sum\limits_{r \in \mathcal{R}} \tau_{r,j}(t)}, & \text{if } \sum\limits_{r \in \mathcal{R}} \tau_{r,j}(t) > 0, \\ 0, & \text{otherwise}, \end{cases} \tag{6}$$

Table 1: Symbols used for the PFA autoscaler.

Inputs	
t	The autoscaling interval, $t \in \mathbb{Z}_{\geq 0}$, where $t = 0$ corresponds to the earliest autoscaling interval
m	The lookup depth for MA and TBA, $m \in [0, t]$
α	The EWMA smoothing factor, $\alpha \in [0, 1)$
$\mathcal{R}$	The set of resource types, $i \in \mathcal{R}$
$\mathcal{U}$	The set of users, $j \in \mathcal{U}$
q_i	The resource cost on any single autoscaling interval
b_j	The user budget for a single autoscaling interval
System Measurables	
$\tau_{i,j}(t)$	The average throughput
$c_{i,j}(t)$	The number of completed tasks
$n_{i,j}(t)$	The number of allocated resources
Derived Values	
$\hat{\rho}_{i,j}(t)$	The instant resource type ratio
$\rho_{i,j}(t)$	The smoothed resource type ratio
$\nu_{i,j}(t)$	The budget fraction available for the resource type
$\zeta_j(t)$	The lookup depth for the token-based approximator
$\theta_j(t)$	The number of tasks in the visited future eligible sets
$\lambda_j(t)$	The token-approximated LoP
$\sigma_j(t)$	The token-approximated demand for all resource types
$\hat{\mu}_{i,j}(t)$	The throughput-based number of resources to allocate
$\tilde{\mu}_j(t)$	The throughput-based number of resources to allocate
$\mu_{i,j}(t)$	The final corrected number of resources to allocate
$P_{i,j}(t)$	The history of non-zero total resource ratios for MA
$T_j(t)$	The history of non-zero total throughputs for MA used with TBA

where $\mathcal{R}$ is the set of all the resource types in the system. MA uses resource type ratios that are not zero for all the resource types:

$$P_{i,j}(t) = \left\{ \hat{\rho}_{i,j}(t-k) : \sum_{r \in \mathcal{R}} \hat{\rho}_{r,j}(t-k) > 0, \forall k \in [0, m] \right\}. \tag{7}$$

The MA-smoothed resource ratios over m previous observations are computed as:

$$\rho_{i,j}(t) = \begin{cases} \frac{1}{|P_{i,j}(t)|} \cdot \sum\limits_{k \in P_{i,j}(t)} k, & \text{if } \sum\limits_{k \in P_{i,j}(t)} k > 0, \\ \frac{1}{|\mathcal{R}|}, & \text{otherwise,} \end{cases} \tag{8}$$

where $|X|$ denotes the cardinality of a set X. If any resource has zero historical throughput all the resource types, instead, get an equal share. This allows the system to collect the throughput history for all the resource types. For the EWMA smoothing method, the smoothed resource ratios are computed as:

$$\rho_{i,j}(t) = \begin{cases} \alpha \cdot \rho_{i,j}(t-1) + (1-\alpha) \cdot \hat{\rho}_{i,j}(t), & \text{if } \hat{\rho}_{r,j}(t) > 0, \forall r \in \mathcal{R}, \\ \frac{1}{|\mathcal{R}|}, & \text{otherwise,} \end{cases} \tag{9}$$

with $\alpha \in [0, 1)$ being the smoothing factor. The parameter α represents the degree of weighting decrease of the past values of $\rho_{i,j}$. A small value of α (close to 0) corresponds to the non-averaged value of $\rho_{i,j}$, i.e., $\hat{\rho}_{i,j}$, while a high value (close to 1), corresponds to a smoother signal over time.

In the second step, based on the resource ratio produced in the first step, we calculate the number of resources of each type that can be allocated with the user budget. For that, we define the fraction $\nu_{i,j}(t)$ of the user budget that we can spend on each resource type according to the resource ratio, knowing the cost q_i of each resource type i:

$$\nu_{i,j}(t) = \frac{q_i \cdot \rho_{i,j}(t)}{\sum\limits_{r \in \mathcal{R}} \big(q_r \cdot \rho_{r,j}(t)\big)}. \tag{10}$$

Accordingly, for each user j the number of resources of type i that can be allocated with the user budget b_j is calculated as:

$$\hat{\mu}_{i,j}(t) = \left\lfloor \frac{b_j \cdot \nu_{i,j}(t)}{q_i} \right\rfloor, \tag{11}$$

supposing that the budget is large enough to allocate at least one instance of each resource type, where $\lfloor x \rfloor$ denotes the floor function of a real number x. Summing up the $\hat{\mu}_{i,j}(t)$ values for all the resource types we calculate the total resource supply that can be achieved with the obtained resource profile:

$$\tilde{\mu}_j(t) = \sum_{r \in \mathcal{R}} \hat{\mu}_{r,j}(t). \tag{12}$$

3.3.2 Token-based Demand Prediction

In the third step, the resource demand $\sigma_j(t)$ is predicted using the Token-based Approximator (TBA), similar to the one described in [14]. For that, TBA considers all the submitted and not yet finished workflows of the user as a single workflow, excluding finished tasks, and places tokens in all the tasks that either have no parents or whose parents has already finished. Then in successive steps TBA moves these tokens to all the tasks all of whose parents already hold a token or were earlier tokenized. TBA records the total number of token movements and, after each step, the number of tokenized nodes.

The intuition is to evaluate the number of "waves" of tasks (future eligible sets) that will finish during the autoscaling interval. When the lookup depth $\zeta_j(t)$ or the final task of the joint workflow is reached, the largest recorded number of tokenized nodes is the approximated LoP $\lambda_j(t)$, and the total number of token movements $\theta_j(t)$ is the number of tasks in the visited future eligible sets. To limit the TBA lookup depth $\zeta_j(t)$, we use the average historical task throughput among all the resource types smoothed either with MA over m previous autoscaling intervals, or with EWMA. For MA, the set of historical average throughputs for all the resource types with skipped intervals with zero total throughput is defined as:

$$T_j(t) = \left\{ \tau_{i,j}(t-k) : \sum_{r \in \mathcal{R}} \tau_{r,j}(t-k) > 0, \forall k \in [0, m], \forall i \in \mathcal{R} \right\}, \tag{13}$$

With MA, the TBA lookup depth is calculated as:

$$\zeta_j(t) = \begin{cases} \left\lceil \frac{1}{|T_j(t)|} \cdot \sum\limits_{k \in T_j(t)} k \right\rceil, & \text{if } \sum\limits_{k \in T_j(t)} k > 0, \\ \infty, & \text{otherwise,} \end{cases} \tag{14}$$

where $\lceil x \rceil$ denotes the ceiling function of a real number x. Accordingly, the resource demand $\sigma_j(t)$ with MA is computed as:

$$\sigma_j(t) = \begin{cases} \left\lceil \theta_j(t) \cdot |T_j(t)| \cdot \left(\sum_{k \in T_j(t)} k \right)^{-1} \right\rceil, & \text{if } \sum\limits_{k \in T_j(t)} k > 0, \\ \lambda_j(t), & \text{otherwise.} \end{cases} \tag{15}$$

With EWMA, the TBA lookup depth is calculated as:

$$\zeta_j(t) = \begin{cases} \left\lceil \alpha \cdot \zeta_j(t-1) + (1-\alpha) \cdot \frac{\sum\limits_{r \in \mathcal{R}} \tau_{r,j}(t)}{|\mathcal{R}|} \right\rceil, & \text{if } \sum\limits_{r \in \mathcal{R}} \tau_{r,j}(t) > 0, \\ \infty, & \text{otherwise.} \end{cases} \tag{16}$$

And the resource demand $\sigma_j(t)$ with EWMA is computed as:

$$\sigma_j(t) = \begin{cases} \left\lceil \theta_j(t) \cdot |\mathcal{R}| \cdot \Big(\sum\limits_{r \in \mathcal{R}} \tau_{r,j}(t) \Big)^{-1} \right\rceil, & \text{if } \sum\limits_{r \in \mathcal{R}} \tau_{r,j}(t) > 0, \\ \lambda_j(t), & \text{otherwise.} \end{cases} \tag{17}$$

3.3.3 Scaling Down or Inflating the Profile

In the fourth step, we scale down or inflate, if necessary, the resource supply calculated using the resource profile to match the predicted resource demand $\sigma_j(t)$. Scaling down prevents allocation of potentially idle resources, and gives space to other users to utilize the resources. Inflating the profile, despite creating possible imbalance in the throughput-based resource ratio, helps to cope with sudden demand surges by increasing the total throughput. If $\tilde{\mu}_j(t)$ exceeds the predicted demand $\sigma_j(t)$, we proportionally scale down the $\hat{\mu}_{i,j}(t)$ values:

$$\mu_{i,j}(t) = \left\lceil \frac{\sigma_j(t)}{\tilde{\mu}_j(t)} \cdot \hat{\mu}_{i,j}(t) \right\rceil . \tag{18}$$

If $\tilde{\mu}_j(t)$ is lower than the predicted demand $\sigma_j(t)$, we inflate the resource as follows: (i) Sort the resources in the ascending order of their resource type cost. (ii) For each resource type i, except the most expensive one, try to add to the original resource profile $\hat{\mu}_{i,j}(t)$ as many resources of that type as possible, until there is no budget available or until $\tilde{\mu}_j(t)$ reaches $\sigma_j(t)$. This produces the inflated $\mu_{i,j}(t)$ values. (iii) If $\sigma_j(t)$ is not yet reached, starting from the second cheapest resource k, try to remove one instance of it from $\mu_{k,j}(t)$ and, instead, add a number of instances to the previous cheapest resource type $\mu_{k-1,j}(t)$. This does not change the total cost of the resource profile, but increases $\tilde{\mu}_j(t)$. Continue, until the total number of resources in the profile reaches $\sigma_j(t)$ or no more such exchanges are possible.

3.3.4 Allocating and Deallocating Resources

In the fifth step, the predicted number of resources is allocated according to the $\mu_{i,j}(t)$ values while taking into account the already allocated resources and the physical system constraints. In the sixth step, PFA de-allocates at maximum the number of idle resources that exceeds the predicted supply. While de-allocating the idle resources, PFA gives priority to those that approach the end of their billing interval.

3.3.5 Task Placement

The PFA autoscaler can operate with various independent task placement policies. As both of the state-of-the-art autoscalers considered in this work, PLF and SCF, employ user-defined workflow priorities and both have embedded task-placement policies which construct an execution plan, for comparability purposes together with PFA we use dynamic task placement policy which also considers user-defined workflow priorities. Our task placement policy assigns eligible tasks according to the priority of their workflows to the first available idle resource of any type.

4 Experiment Setup

This section describes the setup we used to conduct the experiments and the synthetic workloads of workflows.

4.1 Apache Airflow Deployment and Configuration

Our setup is based on the Apache Airflow WMS [4] (v1.9.0) which we extended by adding an autoscaling component with a resource manager. We choose Airflow since it is open source, it is written in Python, and it uses Python-based workflow descriptors, making it rather easy to integrate our code using the existing Airflow codebase. Airflow has reasonable performance for running workloads of workflows and for the autoscaler evaluation purposes. Moreover, Google provides Airflow as its Cloud Composer service [10]. The architecture of our system is presented in Figure 1. All the components of the system are deployed on a cluster with the following characteristics. Head node: Intel Xeon X5650 @ 2.67GHz

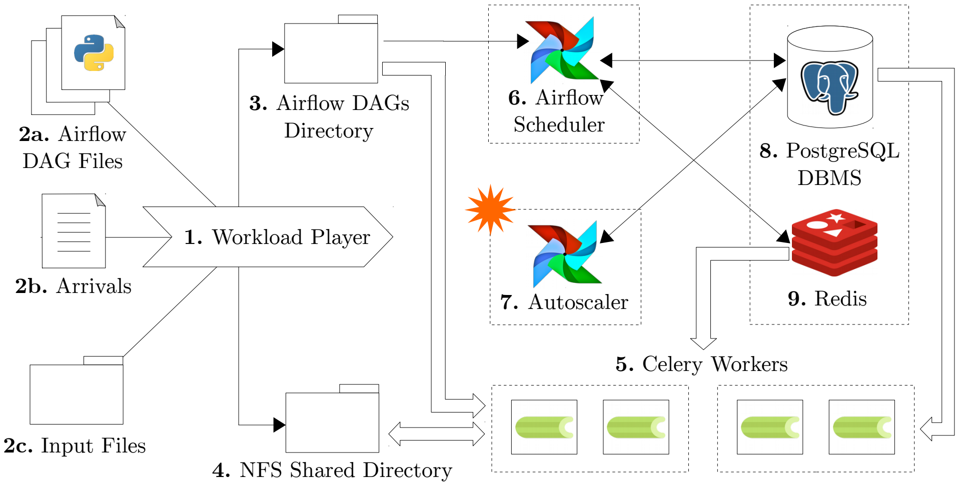


Figure 1: The architecture of the system.

CPU, 49GB RAM, 18TB HDD. 32 compute nodes: Intel Xeon E5620 @ 2.40GHz CPU, 24GB RAM, 2TB HDD. The cluster employs the QDR InfiniBand interconnect and 1 Gbit/s Ethernet at the compute nodes and 10 Gbit/s Ethernet on the head node. All the nodes are running CentOS (v7.4.1708). The average measured Network File System (NFS) access speed is 550 MB/s.

The Workload Player (Component 1 in Figure 1) emulates the Poisson workflow arrivals by sequentially copying workflow descriptors (Component 2a) to the Airflow DAGs directory (Component 3) according to the interarrival times which are read from the Arrivals file (Component 2b). The interarrival times are pre-generated knowing the average total workflow execution time in the workload and the size of the system, so that the imposed average system utilization is kept around 20%. We choose this relatively low imposed utilization to better evaluate the considered autoscalers as it minimizes the amount of time when the demand significantly exceeds the maximal achievable supply. When the descriptor appears in the Airflow DAG directory, the Workload Player issues the '`trigger_dag`' Airflow command to start the workflow execution. In each workflow descriptor we define an identifier of the user who owns the workflow. Together with the workflow descriptor, the Workload Player copies the required input files (Component 2c) to a shared directory in the Network File System (NFS) (Component 4) which is accessible to all the cluster nodes. The Airflow system does not provide specific interface for accessing workflow files, thus, the workflow code is responsible for file access operations. Each task can start its execution when all of its input files are read. Similarly, each task is considered as finished when all its output files are written. The minimal delay between any two dependent tasks is equal to the sum of these two values.

All the Airflow components communicate through the central Airflow database which is in our setup deployed in the PostgreSQL database management system (Component 8). Our setup uses the Celery [8] distributed task queue (version 4.1.1) with Redis [26] (Component 9) in-memory database (version 4.0.10) for sending tasks to the worker nodes. Each worker node runs 8 Celery workers (Component 5)—one per CPU core. In total we deploy 64 Celery workers on 8 worker nodes.

The Airflow Scheduler (Component 6) is responsible for placing eligible tasks for execution to the resources (Celery workers). The default Airflow scheduler is an online dynamic scheduler as it simply sends the eligible tasks in the order of their priority to the single Celery queue which is monitored by the worker processes. Even though, Airflow supports pools of workers, it does not have functionality to monitor the status of each individual worker, and does not support assigning workers to users. To implement this functionality we introduce individual Celery queues for each worker (resource) and guarantee that no new task is placed in the resource's queue if the queue is not empty, so that the queue can hold one task at maximum. We add a table in the central Airflow database which, for each queue (i.e., resource), stores the information on its current status and the identifier of the user who reserved the resource. Such an approach is required since PLF and SCF autoscalers are not only responsible for the resource allocation but also partially take the work from the scheduler by constructing the tasks placement plan for the whole autoscaling interval. Thus, for PLF and SCF autoscalers our Airflow Scheduler simply places tasks to the idle resources just according to the plan. However, since our PFA autoscaler does not create any plan, the modified Airflow Scheduler, when working in tandem with PFA, makes its own task placement decisions by sending eligible tasks according to their workflow priority and task priority to the first idle resource. In all the cases, the modified Airflow Scheduler only places tasks that belong to a specific user to the resources that are reserved for the same user.

The Autoscaler (Component 7) is a novel independent component which is implemented from scratch but heavily relies on the existing Airflow codebase. The Autoscaler implements all the three considered autoscaling policies which can be configured through the main Airflow configuration file. The Autoscaler monitors the status of resources and changes their status through the central Airflow database.

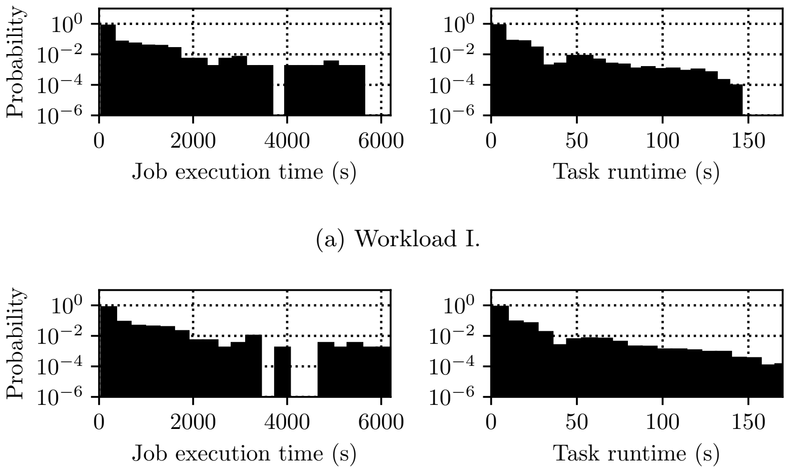
The Workload Player, Airflow Scheduler, and Autoscaler are running on individual worker nodes. The PostgreSQL database management system is co-located with the Redis in-memory database on the the head node.

4.2 Billing Setup

We configure the system with two Small and Large resource types with different costs. Further, we use the generic currency sign ¤ when referring to monetary costs. An instance of Small resource type costs 1¤ per billing interval, while a Large instance costs 5¤ per billing interval. The system is configured to allocate at maximum 64 resources of which 32 of type Small and 32 of type Large. Accordingly, the maximal budget that all the users can spend per autoscaling interval is 192¤. If both users have joint budget which allows to purchase more resources than the system can provide the users will be competing between each other. We shuffle the users before executing the autoscalers for each of them. In the simple case when the system would have only a single resource type any of the considered autoscalers would not make sense as each user would simply get the number of instances which its budget allows to allocate at maximum. We use autoscaling interval of one minute to be in line with the current trend on fine-grained billing [27]. Since we report the imposed system utilization, we believe that the same behavior should be observed for shorter or longer autoscaling intervals, if the utilization will be accordingly adjusted.

4.3 Workloads

We use two workloads Workload I and Workload II, each consisting of 600 workflows divided in three sets with 200 workflows each. That allows us to perform three repetitions of each experiment. Both workloads use the same 600 workflow structures, but differ in the task runtime characteristics. 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 [15] 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 [19] is a bioinformatics workflow used to discover bacterial regulatory RNAs. We take the workflow structures, the task runtime distributions and file sizes from the Bharathi generator [6, 18]. We scale down the original task runtimes and file sizes to reduce the total execution time of the workloads by dividing the original values from the generator by 30 and rounding them to the nearest integer. We guarantee that the minimal task runtime is 1 second and the minimal files size is 1KB. Since we have two resource types in our model, for each task we take its scaled down runtime and generate one extra task runtime using the uniform distribution. For Workload I the maximal deviation for the second task runtime from the original task runtime is 50%, and the original and new task runtimes are randomly assigned to the resource types. For Workload II the maximal deviation from the original task runtime is 100%, and the new generated task runtime is always assigned to the second resource type. In both workloads each workflow has a randomly assigned priority in the range from 0 to 9. Figure 2 presents task runtime and job runtime distributions of the workload. The details of each workload are summarized in Table 2.



(a) Workload I.

(b) Workload II.

Figure 2: Statistical characteristics of the workloads. The vertical axes have a log scale.

Table 2: Characteristics of Workloads I and II.

Property	WL I	WL II
Total workflows in all three sets	600	
Total tasks in all three sets	44,340	
Mean number of tasks in a workflow	74	
Median number of tasks in a workflow	38	
Standard deviation of number of tasks in a workflow	95	
Mean job execution time [s]	467	508
Median job execution time [s]	276	303
Standard deviation of job execution times [s]	692	761
Mean task runtime (averaged for both resource types) [s]	6.3	6.9
Median task runtime (averaged for both resource types) [s]	1.5	1.5
Standard deviation of task runtimes (averaged for both resource types) [s]	13.7	15.4
Mean task runtime on the Small resource [s]	6.3	8.2
Mean task runtime on the Large resource [s]	6.3	5.5
Total task runtime (averaged for both resource types) [ks]	280	305
Mean task input data size [MB]	578	
Median task input data size [MB]	138	
Standard deviation task input data size [MB]	1,364	
Mean task output data size [MB]	213	
Median task output data size [MB]	9	
Standard deviation task output data size [MB]	2,224	
Total task input data size (including read duplicates*) [TB]	25,6	
Total task output data size [TB]	9,4	

* When different tasks read the same file.

Table 3: Experiment configurations.

Sec.	Budget Configuration	PFA Configuration	WL
§5.2	eq. 60¤, 80¤, 100¤, 120¤; diff. 120¤/80¤	$m = 10, 20, 30$; $\alpha = 0.7, 0.8, 0.9$	I
§5.3	eq. 60¤, 100¤, 120¤; diff. 120¤/80¤	$m = 10$; $\alpha = 0.7$	I
§5.4	eq. 60¤, 100¤, 120¤; diff. 120¤/80¤	$m = 10$; $\alpha = 0.7$	I
§5.5	eq. 120¤	$m = 10$	I
§5.2	eq. 100¤	$m = 10$; $\alpha = 0.7$	II

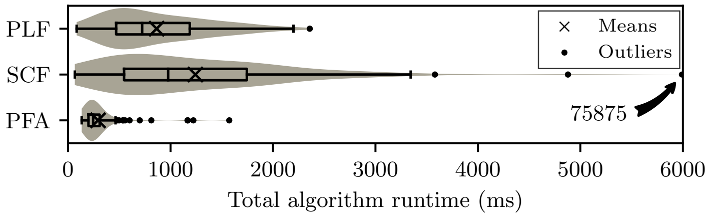


Figure 3: Variability of total runtimes for all the considered autoscalers.

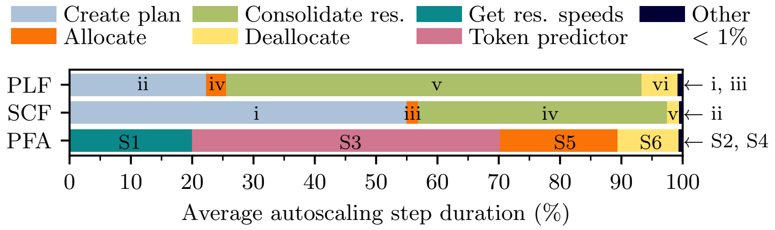


Figure 4: Average duration of each autoscaling step within the total algorithm runtime for each considered autoscaler.

5 Experiment Results

In this section, we present our experimental results. We first analyze the runtimes of the considered autoscalers obtained during the experiments. Then we investigate how varying the budget affects the workload performance and how it differs between the users. Finally, we analyze the system-oriented, and elasticity metrics. We report two experiment configurations, where we assign either equal budgets (eq.) to both users or different budgets (diff.) for each user. The sets of experiment configurations with regard to the experiment results sections are summarized in Table 3. Our results show that our PFA autoscaler shows up to 76% lower average algorithm runtime when given the same workload as PLF and SCF, while reducing by up to 47% the average job slowdowns

5.1 Algorithm Performance

Figure 3 shows the variability of total algorithm runtimes executed at every autoscaler invocation. The runtime of the algorithm varies depending on the number of workflows that are currently in the system and depending on their characteristics. We can also see that both plan-based autoscalers PLF and SCF have 3–4 times longer average runtimes and show higher runtime variability than our PFA autoscaler. Moreover, SCF autoscaler has one large outlier when it was running for 76 seconds, thus, exceeding the length of the autoscaling interval and delaying the workload! Such behavior is very unfavourable as it can negatively affect the stability of a WMS during sudden demand surges.

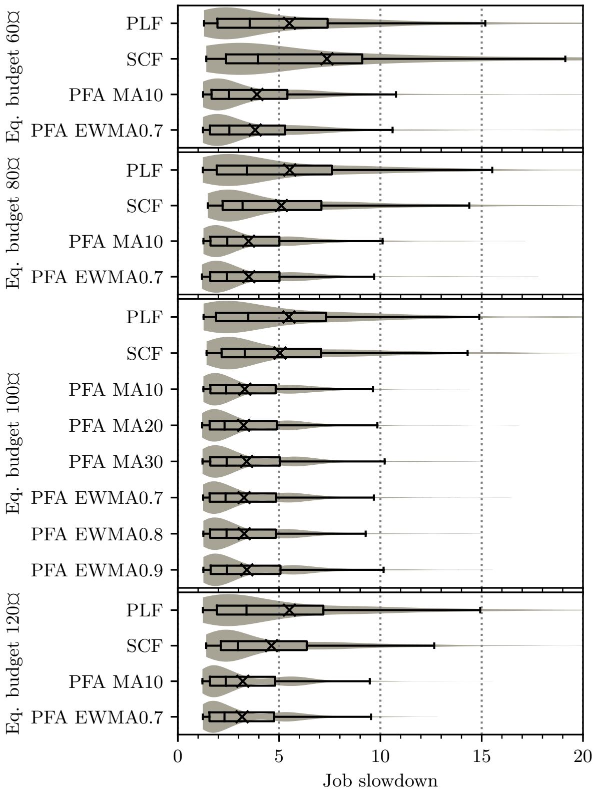


Figure 5: Variability of job slowdowns for all the studied autoscalers with equal budgets of 60¤, 80¤, 100¤, and 120¤ when running WL I. PFA autoscaler was executed with different smoothing methods. Means are marked with ×.

Figure 4 shows the average duration of each autoscaling stage as a percentage of the average total algorithm runtime. For PLF and SCF autoscalers the planning and the resource consolidation steps take up 95% of their total execution time. Resource consolidation is basically responsible for making the task placement plan. For our PFA autoscaler the token-based demand prediction takes on average 50% of the total execution time.

5.2 Workload Performance

Further, we analyze the job slowdowns to investigate how the considered autoscalers affect the workload performance from the end-user perspective. Figure 5 shows the variability of job slowdowns in two configurations where both users are assigned with equal budgets of 60¤, 80¤, 100¤, or 120¤, accordingly. In this figure we can see a clear trend where higher budgets decrease the average job slowdown as well as decrease the slowdown variability. We use the configuration with equal budget 100¤ as a baseline, as then each user can allocate at max. 52% of the system resources. With 100¤, the PFA policy is executed with various MA history depths m of 10, 20, and 30, and with EWMA α values of 0.7, 0.8, and 0.9. We can observe that PFA in any of the considered configurations shows lower average job slowdowns, as well as lower slowdown variability than PLF and SCF. Different PFA smoothing methods do not significantly affect the PFA performance.

Figure 6 shows job slowdown variability for the configuration where User 1 has higher budget 120¤ than User 2 with budget 80¤. We can conclude, that all the considered autoscalers guarantee that the user with the higher budget gets better performance, since User 1 has lower average job slowdowns and lower slowdown variability. PFA autoscaler for both users shows better workload performance than PLF and SCF.

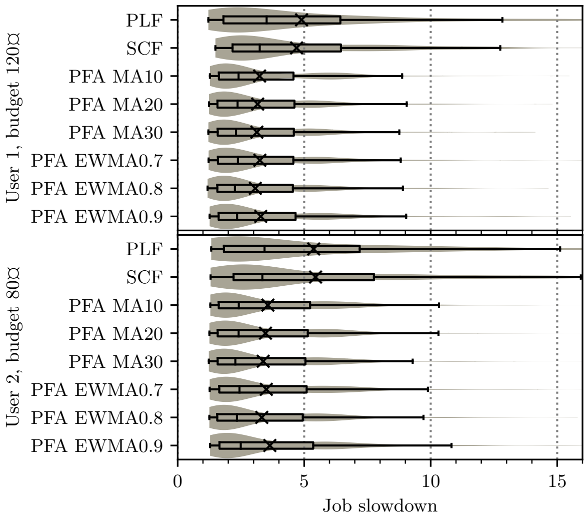


Figure 6: Variability of job slowdowns for all the studied autoscalers with different budgets for each user when running WL I. PFA autoscaler was executed with different smoothing methods. Means are marked with ×.

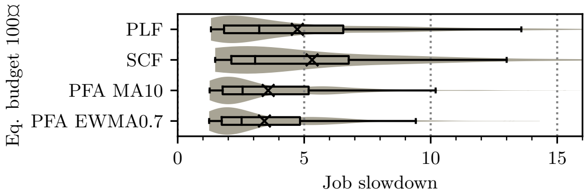


Figure 7: Variability of job slowdowns for all the studied autoscalers with equal budgets for both users when running WL II. PFA autoscaler was executed with MA depth 10 and EWMA with pole value 0.7. Means are marked with ×.

Figure 7 presents job slowdowns for configuration with equal budget 100¤ for Workload II. The observed trend is the same as in Figures 5 and 6. The tasks in WL II on average run faster on the Large resource type than on the Small resource type (see Table 2). Thus, we can conclude, that all the considered autoscalers can successfully operate with workloads where tasks “prefer” a specific resource type.

From Figure 8 and Figure 9 we can see that no autoscalers exceed the budget constraint for the configurations with equal and different budgets. SCF on average spends more budget than PLF and PFA. Our PFA autoscaler shows comparable mean costs to PLF, but lower median costs at higher budgets. Moreover, for PFA, when the budget is large enough, the distribution of allocated costs skews towards lower values. For all the autoscalers most of the cost comes from the Large resource type, as it is more expensive. Further, when presenting the results, we do not plot some experiment configurations if these configurations show no significant difference. E.g., from Figure 8 we omit the results with the equal budget 80¤, and from Figure 9 we omit the results for PFA with the configurations MA $m = 20$, 30, and $\alpha = 0.8$, 0.9.

5.3 Elasticity Performance

Figure 10 shows the considered elasticity metrics for the configurations with equal budget of 60¤, 100¤, and 120¤ for User 1. We do not report values for User 2, as we do not observe significant difference between the users. Figure 11 shows elasticity metrics for both user for the configuration with different budgets of 120¤ and 80¤ for User 1 and User 2, accordingly. When calculating the elasticity metrics, we skip the periods where demand exceeds the maximal resource number of 64. The resource demand can vary significantly even at relatively low utilization of 20% as it depends on the structure and LoP of the workflows.

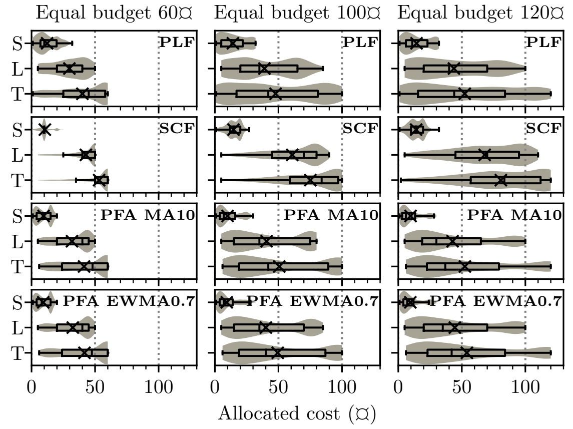


Figure 8: Variability of monetary cost for allocated resources per billing interval for User 1 for the studied autoscalers with equal budgets of 60¤, 100¤, and 120¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

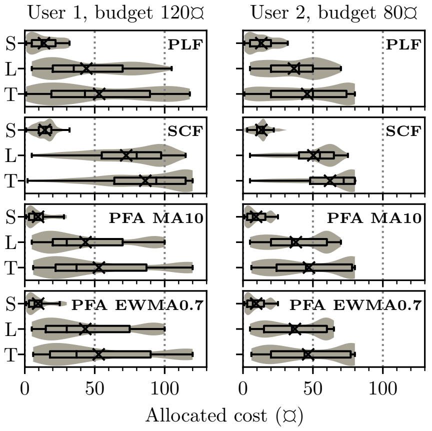


Figure 9: Variability of monetary cost for allocated resources per billing interval for each user for the studied autoscalers with different budgets when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

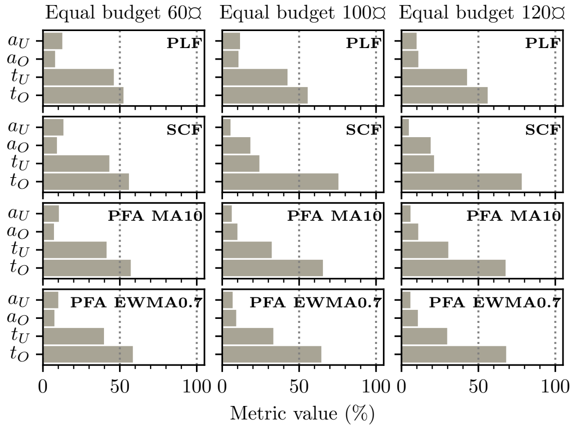


Figure 10: Elasticity metrics for User 1 for the studied autoscalers with equal budgets of 60¤, 100¤, 120¤ when running WL I. For all the metrics lower values are better.

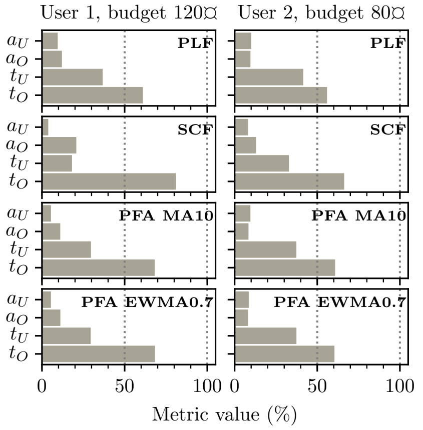


Figure 11: Elasticity metrics for each user for the studied autoscalers with different budgets 120¤ and 80¤ when running WL I. For all the metrics lower values are better.

In Figure 10 we can see that for budgets 100¤ and 120¤, SCF shows in all the plots the worst values for a_O and t_O, it also has the best values for a_U and t_U. In other words, SCF tends to over-provision. For example, in the configuration with budget 100¤, SCF over-provisions for almost 75% of the time with on average 18% too many resources and has in 24% of the time on average 5% too few resources. At lower budget 60¤, SCF spends less time over-provisioning, but still shows on average the worst over-provisioning accuracy of 8.5%.

In contrast, PLF with budget 100¤ has in 42% of the time on average 11% too few resources. Thus, PLF tends to under-provision the system and has the worst values for a_u (except for budget 60¤, where SCF is the worst), t_U and only the best values for t_O.

Our PFA autoscaler shows the best values for a_O. Further, PFA has the second best values for a_U with budgets 100¤ and 120¤. For budget 60¤ PFA shows the best value for a_U, t_U, but the words value for t_O, which is, however, compensated by low a_O. In general, PFA is more accurate than the other two autoscalers, as it has the lowest summed up a_U and a_O accuracy values. Moreover, spending more time under- or over-provisioning with higher accuracy is more favourable than spending less time under-

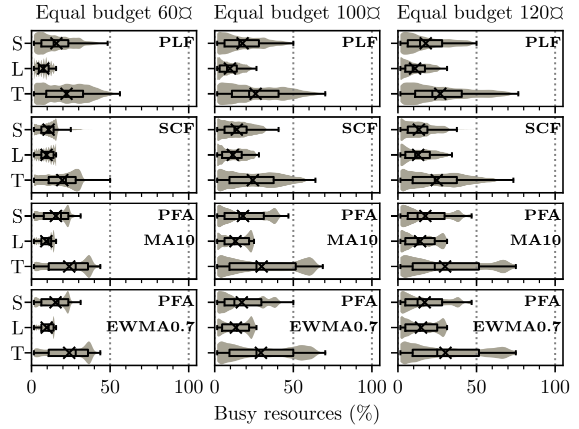


Figure 12: Variability of busy resources for User 1 for the studied autoscalers with equal budgets of 60¤, 100¤, 120¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

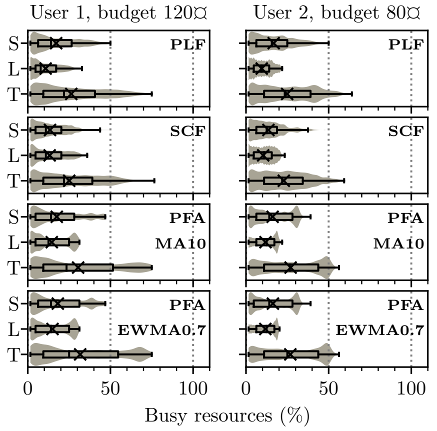


Figure 13: Variability of busy resources for each user with different budgets 120¤ and 80¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

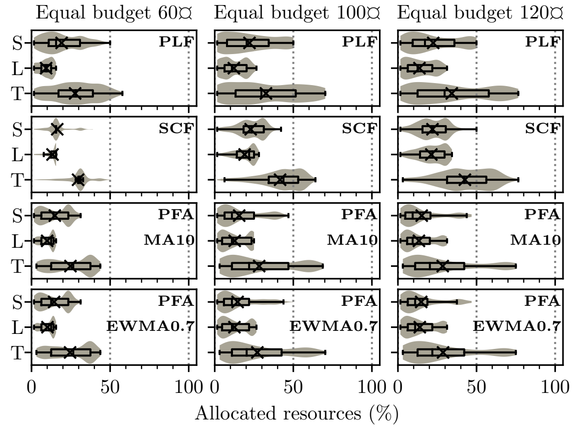


Figure 14: Variability of allocated resources for User 1 for the studied autoscalers with equal budgets of 60¤, 100¤, 120¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

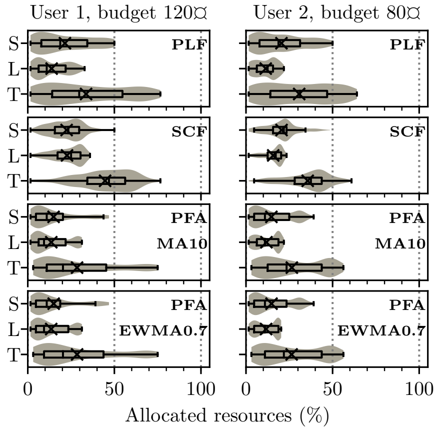


Figure 15: Variability of allocated resources for each user with different budgets 120¤ and 80¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

or over-provisioning with lower accuracy. The same trends can be observed for the configuration with different budgets in Figure 11. From this we conclude that our approach is more likely to satisfy the user SLOs, which is also confirmed by the workload performance results. Although, PFA does not use known in advance task runtime estimates, it is more accurate when applied to workload of workflows than the plan-based autoscalers.

5.4 System-Oriented Performance

We look at the percentage of busy and allocated resources throughout the experiment to evaluate the system-oriented performance, as these metrics shows how effectively the resources are utilized, and how many resources are actually allocated. Figure 12 presents the percentage of busy resources for the configurations with equal budget of 60¤, 100¤, and 120¤ for User 1. Figure 14 shows the variability of allocated resources for the same configuration and the same user. We do not report values for User 2, as we do not observe significant difference between the users.

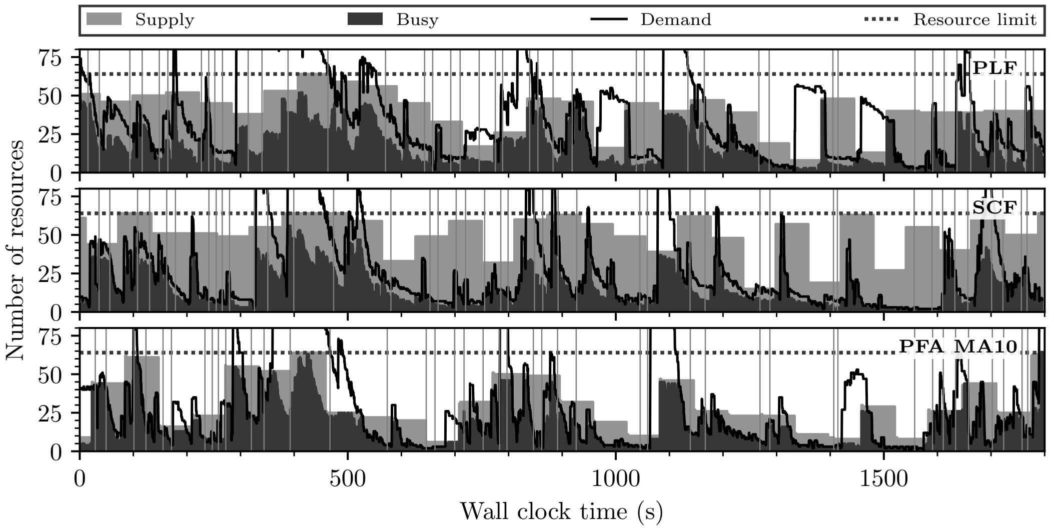


Figure 16: The dynamics of autoscaling on a cropped interval of 1,800 seconds for both users with equal budget 120¤. Vertical lines indicate workflow arrivals.

Figure 13 shows the percentage of busy resources for both user for the configuration with different budgets of 120¤ and 80¤ for User 1 and User 2, accordingly. Figure 15 shows the percentage of allocated resources for different budgets also for both users.

SCF shows lowest average number of busy resources, which correlates with the elasticity results, as SCF tends to over-provision more. PFA shows higher and also more balanced use of the resources. We can also see that the variability of busy resources increases together with the budget. Looking at the variability of allocated resources, we observe that PLF and SCF on average allocate more resources than PFA, this correlates with the results on monetary costs from Section 5.2. For higher budgets PFA tends to spend more time allocating less resources than the other two autoscalers. For lower budgets the difference between the autoscalers decreases. Thus, we can conclude, that, in contrast to PLF and SCF, PFA allocates and uses the resources more efficiently, while given the same budget.

5.5 Autoscaling Dynamics

We further study the dynamics of the obtained Airflow traces to better understand the performance differences between the autoscalers. Figure 16 shows the snapshots of autoscaling dynamics on a cropped interval of 1,800 seconds for both users with equal budget of 120¤. We rely on the configuration with 120¤ as it shows higher supply variability. We can see that PLF and SCF autoscalers have higher demand values—the number of waiting eligible tasks. Both PLF and SCF show lower resource utilization (the number of busy resources) as in between the autoscaler invocations the tasks are waiting for being included in the plan. Moreover, we can see how PLF makes wrong predictions, e.g., at the time around 1350 seconds, as PLF makes its predictions using the tasks that are eligible at the moment it is invoked. Similar-looking spikes can be observed for PFA, e.g., at the time around 1425 seconds, however, it is caused by a double workflow arrival within the autoscaling interval. We can also observe different shapes of demand curves in each plot, as the number of eligible tasks depends on the throughput, that, in turn, depends on the number of allocated resources and the efficiency of the task placement policy utilized by the scheduler. Interestingly, for PFA on the interval 350–450 seconds the allocated resources are not fully utilized, even though the demand exceeds the resource ceiling. The reason for this is the latency caused by the Airflow system.

The phases of the autoscaling intervals are slightly drifting between the plots, as we did not set a goal to deliberately synchronize the phases of different autoscalers. The drifting is caused by possible internal delays in the Airflow system during the demand surges, and by occasional delays in the autoscalers, e.g., when SCF runs for too long, as shown in Figure 3. That is why, to minimize the possible effect of such drifting, we run three independent subsets of workflows within the workload.

Table 4: Symbols used for the MIP model.

$\mathcal{T}$	The set of time slots $\mathcal{T} = \{1, 2, \dots, T\}$
$\mathcal{W}$	The set of workflows $\mathcal{W} = \{1, 2, \dots, W\}$
$\mathcal{S}$	The set of workflows tasks $\mathcal{S} = \{1, 2, \dots, S\}$
$\mathcal{M}$	The set of billing intervals $\mathcal{M} = \{1, 2, \dots, M\}$
$\mathcal{V}$	The set of resources $\mathcal{V} = \{1, 2, \dots, V\}$
L	Number of time slots per billing interval
B	Budget per billing interval
A_w	Arrival time of workflow w
D_w	Earliest possible completion time for workflow w
$R_{j,k}$	Runtime of task j on resource k
P_k	Cost of running resource k on a billing interval
$l_{i,j}$	Equals one if task i depends on task j
$h_w(t)$	The value of workflow w finishing at time t
$x_{j,k}^t$	Binary variable, equals one if task j starts at time t on resource k
y_k^m	Binary variable, equals one if resource k is active on billing interval m
z_k^m	Integer variable, determines the number of active time slots for resource k on billing interval m
u_w^t	Binary variable, equals one if workflow w finishes at time t

6 The Optimal Solution

In this section, to validate the performance of the considered policies, we compare the results obtained from the Airflow system with the optimal solution obtained from solving the optimization problem represented as a Mixed Integer Programming (MIP) model. For that we modify the MIP model proposed by Wang et al. [31] to incorporate budget constraints, while following the similar notation, and implement the model in the Gurobi [11] solver (v. 8.0.1). The goal of the solver is to find the optimal plan which, under the budget and resource constraints, finds the task placement plan and determines the number of resources of each type that should be allocated on each autoscaling interval, so that the response time of each workflow is minimized.

6.1 Mixed Integer Programming Model

The MIP model presents time as a set $\mathcal{T} = \{1, 2, \dots, T\}$ of discrete time slots of equal duration, where T is the furthest time horizon. The times slots are grouped into M billing intervals. Each billing interval consists of L time slots. The set of billing intervals is denoted by $\mathcal{M} = \{1, 2, \dots, M\}$, and T is divisible by L, so that $M = T/L$. The budget B is given per billing interval and should not be exceeded. The input of the problem is a set of workflows $\mathcal{W} = \{1, 2, \dots, W\}$, where each workflows contain tasks. All the tasks in all the workflows are represented by the set $\mathcal{S} = \{1, 2, \dots, S\}$, where each task can belong to a single workflow only. The task precedence constraints are represented by a binary matrix $(l_{i,j}), \forall i, j \in S$, where $l_{i,j} = 1$ if task i depends on task j, i.e., i can start only after j has finished, and $l_{i,j} = 0$ otherwise. By convention, each $l_{i,i} = 0$. Each workflow w has an arrival time A_w, known in advance length B_w of its critical path, and earliest possible completion time D_w, so that $D_w = A_w + B_w - 1$. The model also defines a set of computing resources $\mathcal{V} = \{1, 2, \dots, V\}$.

If a task is scheduled on a resource, it runs on it exclusively until completion. To represent the task assignment, we use binary decision variables $x_{j,k}^t$, where $x_{j,k}^t = 1$ if task i is scheduled to run on resource k starting at time slot t, and $x_{j,k}^t = 0$ otherwise. Each task should start only once, which we specify as follows:

$$\sum_{k \in \mathcal{V}} \sum_{t \in \mathcal{T}} x_{j,k}^t = 1, \forall j \in \mathcal{S}. \tag{19}$$

Let the integer variable $0 \le z_k^m \le L$ denote the number of active time slots on each resource k on billing interval m. This requires the following constraints:

$$\sum_{t=(m-1)\cdot L+1}^{m\cdot L} \sum_{j\in S} \sum_{r=\max(1,t-R_{j,k}+1)}^{t} x_{j,k}^{r} = z_k^m, \forall k \in \mathcal{V}, \forall m \in \mathcal{M}. \tag{20}$$

Let the binary variable y_k^m denote the active/idle state of each resource k on billing interval m, with $y_k^m = 1$ if some tasks are assigned on the resource, and $y_k^m = 0$ otherwise. If the resource has no tasks scheduled, it is considered deallocated, however if even a single task is assigned to the resource, it is considered active. Accordingly, we define the following constraints:

$$y_k^m = \min(1, z_k^m), \forall k \in \mathcal{V}, \forall m \in \mathcal{M}. \tag{21}$$

The tasks are not allowed to overlap, i.e., for each time slot and each resource at most one task is allowed to occupy the time slot on that resource. Let $R_{j,k}$ denote the running time of task j on resource k which is known in advance. The non-overlapping constraints are specified as follows:

$$\sum_{j\in\mathcal{S}} \sum_{r=\max(1,t-R_{i,k}+1)}^{t} x_{j,k}^{r} \le 1, \forall k \in \mathcal{V}, \forall t \in \mathcal{T}. \tag{22}$$

The precedence constraints are formulated as follows:

$$\Big(\sum_{k\in\mathcal{V}}\sum_{t\in\mathcal{T}} t\cdot x_{i,k}^t - \sum_{k\in\mathcal{V}}\sum_{t\in\mathcal{T}} (t+R_{j,k})\cdot x_{j,k}^t\Big)\cdot l_{i,j} \ge 0, \qquad \forall i,j \in \mathcal{S}. \tag{23}$$

Further we formulate the constraints that no tasks of any workflow can be scheduled to start before its arrival time:

$$\sum_{k\in\mathcal{V}}\sum_{t\in\mathcal{T}} t\cdot x_{j,k}^t \ge A_w, \forall w \in \mathcal{W}, \forall j \in w. \tag{24}$$

Since the optimization goal is to minimize the workflow response time within the given budget, we represent it as a profit maximization problem where higher profit corresponds to a shorter response time. For that, let $h_w : \{1, 2, \ldots\} \to \mathbb{R}$ be a non-increasing value function, where $h_w(t)$ represents the value gained depending on the time slot t where the workflow w is finished:

$$h_w(t) = \begin{cases} 1, & \text{if } t \le D_w, \\ D_w - t, & \text{otherwise.} \end{cases} \tag{25}$$

For each workflow w and each time t we define a binary variable u_w^t, where $u_w^t = 1$ if workflow w is completed at time t. Since each workflow can finish only once, we formulate the following constraints:

$$\sum_{t\in\mathcal{T}} u_w^t = 1, \forall w \in \mathcal{W}. \tag{26}$$

The completion time of the workflow can be written as $\sum_{t\in\mathcal{T}} t\cdot u_w^t$. Accordingly, the constraint that all the tasks of a workflow w are completed by the workflow completion time can be formulated as follows:

$$\sum_{k\in\mathcal{V}}\sum_{t\in\mathcal{T}} (t+R_{j,k}-1)\cdot x_{j,k}^t \le \sum_{t\in\mathcal{T}} t\cdot u_w^t, \forall w \in \mathcal{W}, \forall j \in w. \tag{27}$$

Let P_k be the cost of resource k, then the budget constraints are defined as follows:

$$\sum_{k\in\mathcal{V}} P_k \cdot y_k^m \le B, \forall m \in \mathcal{M}. \tag{28}$$

Finally, we can formulate the profit maximization objective:

$$\begin{aligned} &\max \sum_{w\in\mathcal{W}}\sum_{t\in\mathcal{T}} h_w(t)\cdot u_w^t \\ &\text{s.t. } (19)(20)(21)(22)(23)(24)(26)(27)\ (28). \end{aligned} \tag{29}$$

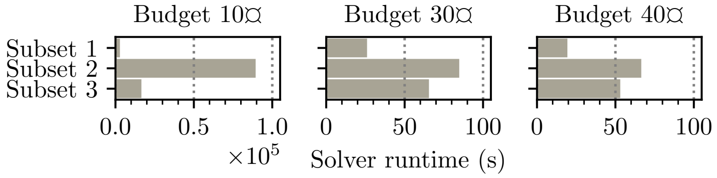


Figure 17: Solver runtimes for all three subsets of workflows with different budget constraints. In the left plot the x axis has much larger scale than in the other two plots.

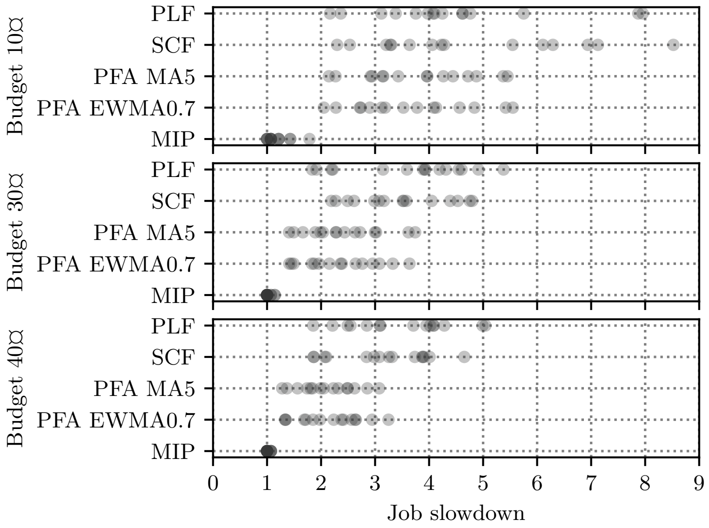


Figure 18: The variability of job slowdown for different budgets for Airflow-based and MIP results.

6.2 Heuristics vs. the Optimal Solution

We use three subsets of five workflows each from Workload I, submitted with a fixed interval of 30 seconds in a system with 16 resources (vs. 64 resources in other experiments) of two types with 8 resources in each. For this reason, the maximal budget required to allocate all the system resources is 48¤. The first group of five workflows consists of one Montage, one SIPHT, and three LIGO workflows, with 183 tasks in total. The second group contains one Montage, two SIPHT and two LIGO workflows, with 241 task in total. The third group contains two Montage, one SIPHT, and two LIGO workflows with 199 tasks in total. Further, we refer to these 15 workflows as the MIP workload. We limit the number of considered workflows and limit the number of resources due to much higher expected computational effort for finding the optimal solution versus the considered heuristic approaches. To make the workflows compatible with the MIP model, we round their task runtimes to 5 seconds, which is the duration of the time slot in the MIP model, and set the sizes of all the exchanged files to zero to make the comparison more fair. We configure the MIP model with 16, 18, or 19 billing intervals (depending on the workload subset) and set the length of each billing interval to 3 time slots. Accordingly, we set the Airflow autoscaling interval to 15 seconds. We use a single user only, and find the optimal solutions with three different budgets of 10¤, 30¤, and 40¤.

Figure 17 shows solver runtimes for all three groups of workflows with different budget constraints. Note, that the lower the budget constraint the more time the solution takes. For budget 10¤ finding the solution takes up to 88,592 seconds (24.5 hours)! This confirms that the solution time does not scale linearly as it highly depends on parameterization of the model, for example, on the chosen number of slots. Moreover, the Gurobi solver has more than 40 MIP-related internal parameters [12] that can significantly affect the performance of the solver. To somehow automate this process, Gurobi even provides a parameter tuning tool [13]. Since in our MIP model we add the budget constraints, they increase the total runtime for finding the optimal solution, compared to the runtimes reported in the paper by Wang et al. [31]. Even in the paper by Wang et al. the number of considered workflows used with the MIP solver was much higher (500), those workflow structures were very simple, and the model did not have budget constraints. From this we can conclude that the MIP approach is not suitable for autoscaling workloads of workflows.

In Figure 18 we compare the slowdowns of the workflows from the optimal plan with the slowdown obtained from running the same workflows in our Airflow setup with all the three considered autoscalers. We configure PFA with $m = 5$ for MA due to the low number of autoscaling intervals in the MIP model, for EWMA we use $\alpha = 0.7$. We do not use violin plots in Figure 18 for the distributions, as for each run there we have only 15 samples—the total number of executed workflows in all the three MIP workload subsets. We can clearly see that the slowdowns obtained from the Airflow system are up to 8 times higher than the slowdowns from the MIP solution. Note, that the slowdowns from the Airflow experiments also include the slowdown caused by the WMS itself. However, the general trend is similar to Figures 5, and 6 where our PFA autoscaler shows better workload performance than the plan-based autoscalers.

7 Threats to Validity

The limitations of this study are mostly expressed in the number of considered resource types and users. Having two resource types is the minimum requirement for comparing the performance of the considered dynamic and plan-based autoscalers. A higher number of resource types would lead to higher planning complexity of the plan-based autoscalers and would require to increase the total number of resources. Adding more resources would also increase the WMS overhead, thus, lengthening the experiment duration and bringing unnecessary complexity, while not being relevant to the conceptual part of the paper.

The main reason for having multiple concurrent users serving independent workloads is to model a background load present in real production clouds, while conducting experiments in an isolated cluster environment. The isolated cluster environment allows to achieve higher control over the setup and the factors affecting the results. Having only two users has the same reason as having two resource types—it is sufficient for the initial performance comparison of the autoscalers. We do not consider resource allocation and deallocation delays as adding them would relatively equally affect all the considered autoscalers [14], thus, making the effect of such delays not relevant to the autoscaler comparison purposes.

The choice of throughput as reliable metric can also be questioned. However, in this work we are dealing with a workload that consists of many tasks with different durations, and the scheduler that is used with PFA assigns tasks to the resources randomly, so that each resource type processes tasks with different durations. This allows for rather precise estimation of resource type speeds. In the worst case, when there are no throughput information available, PFA falls back to an equal share of resource types. Using, for example, historical task runtimes we would need to suppose that resources with shorter observed average task runtimes are faster. In contrast with task runtimes, throughput allows to easier estimate the resource speed on an autoscaling interval.

The goal of the considered autoscalers is not only to fit within the budget but also to spend it effectively. An infinite budget makes our problem equivalent to the cost minimization problem. However, considering only cost minimization without budget constraint is not realistic as cloud providers usually employ usage quotas and allow to set limits for monetary costs.

We have tried multiple parameters for MA and EWMA to find which of them reduce the experienced workflow slowdowns. The considered parameter values can be used directly in practice, however, other workload types may require additional parameter tuning.

8 Related Work

In this section, we overview specialized autoscaling policies for workflows that focus on the resource-allocation problem.

State-of-the-art autoscaling policies: The Dynamic Scaling Consolidation Scheduling (DSCS) [22], Partitioned Balanced Time Scheduling (PBTS) [7], IaaS Cloud Partial Critical Paths (IC-PCP) [2], Deadline Constrained Critical Path (DCCP) [5], Dyna [34], and Partition Problem-based Dynamic Provisioning and Scheduling (PPDPS) [28] autoscalers combine scheduling and allocation approaches, and, in contrast to the approach used in this paper, have the goal to minimize the operational cost under unlimited budget and meet (soft) workflow deadlines. DSCS, PBTS, and Dyna are online plan-based autoscalers, while IC-PCP, DCCP, and PPDPS are offline autoscalers. The Plan autoscaler [14] is an online plan-based autoscaler which does not support budget constraints and requires task runtime estimates for the workflow tasks. The Token autoscaler [14] is an online dynamic autoscaler that uses a token-based approach to estimate the demand and requires runtime estimate for the whole workflow. The Dynamic Provisioning Dynamic Scheduling (DPDS) [21] is an offline dynamic autoscaler for ensembles of scientific workflows that supports a single resource type only. The autoscaler is threshold-based, the cost- and deadline-constraints should be provided for the whole ensemble. The Static Provisioning Static Scheduling (SPSS) [21] is an offline autoscaler that creates a plan for each workflow in the ensemble, and rejects workflows that exceed the deadline or budget. BAGS [27] is a plan-based offline autoscaler that partitions workflows into bags-of-tasks and then applies a MIP-based approach to make the allocation plan. The majority of the considered works performs simulations when evaluating the proposed algorithms.

Comprehensive comparisons and benchmarks: Versluis et al. [30] and Ilyushkin et al. [14] perform comprehensive analysis of different autoscalers for workloads of workflows. Overall, these studies emphasize the need for autoscalers that can cope with workloads of workflows, but neither propose the autoscalers that support cost constraints and multiple resource types, nor assess the time taken by autoscalers to make decisions or evaluate the scalability. The recent survey [20] on cost and makespan-aware workflow scheduling in cloud provides a good overview of the current scheduling and autoscaling trends for workflows.

9 Conclusions

We presented the novel Performance-Feedback Autoscaler (PFA) for workloads of workflows. To make autoscaling decisions, PFA analyzes historical task throughput and uses current workflow structural information, instead of relying on task runtime estimates. This makes PFA easier to use, as observing task throughput normally requires less effort than obtaining task runtime estimates.

Overall, PFA has lower time-complexity and effectively minimizes workflow slowdowns, compared to two state-of-the-art online plan-based autoscalers. Our real-world experiments with the Apache Airflow workflow management system show that PFA, compared to other two autoscalers, has better applicability potential due to its good scalability when dealing with possible demand surges, and good end-user and system-oriented characteristics.

For future work, we plan to investigate the performance footprint of PFA for other resource types, e.g., memory. To further evaluate the scalability of the proposed autoscaler we will consider setups with higher number of resource types and concurrent users. To make PFA even more autonomous, we will look how to automatically configure the signal smoothing and try different feedback mechanisms.

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] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick H.J. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems (FGCS)*, 29:158–169, 2013.

[3] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An Adaptive Hybrid Elasticity Controller

for Cloud Infrastructures. In *Proceedings of the IEEE Network Operations and Management Symposium*, 2012.

[4] Apache Airflow. A platform to programmatically author, schedule and monitor workflows, February 2019. URL `https://airflow.apache.org`.

[5] Vahid Arabnejad, Kris Bubendorfer, and Bryan Ng. Scheduling deadline constrained scientific workflows on dynamically provisioned cloud resources. *Future Generation Computer Systems (FGCS)*, 75:348–364, 2017.

[6] Shishir Bharathi et al. Characterization of scientific workflows. In *Third Workshop on Workflows in Support of Large-Scale Science*, 2008.

[7] Eun-Kyu Byun et al. Cost Optimized Provisioning of Elastic Resources for Application Workflows. *Future Generation Computer Systems (FGCS)*, 27:1011–1026, 2011.

[8] Celery. Distributed task queue, February 2019. URL `http://docs.celeryproject.org/`.

[9] Artem M. Chirkin et al. Execution time estimation for workflow scheduling. *Future Generation Computer Systems (FGCS)*, 2017.

[10] Google Cloud Composer. A fully managed workflow orchestration service built on Apache Airflow, February 2019. URL `https://cloud.google.com/composer/`.

[11] Gurobi. The state-of-the-art mathematical programming solver, February 2019. URL `http://www.gurobi.com`.

[12] Gurobi Reference Manual. Parameters, February 2019. URL `https://www.gurobi.com/documentation/8.1/refman/parameters.html`.

[13] Gurobi Reference Manual. Parameter Tuning Tool, February 2019. URL `http://www.gurobi.com/documentation/8.1/refman/parameter_tuning_tool.html`.

[14] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, André Bauer, Alessandro V. Papadopoulos, Dick Epema, and Alexandru Iosup. An experimental performance evaluation of autoscalers for complex workflows. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):8, 2018.

[15] Joseph C. Jacob et al. Montage: An astronomical image mosaicking toolkit. *Astrophysics Source Code Library*, 1:10036, 2010.

[16] Vimalkumar Jeyakumar et al. EyeQ: Practical network performance isolation at the edge. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[17] Gideon Juve et al. Characterizing and profiling scientific workflows. *Future Generation Computer Systems (FGCS)*, 29:682–692, 2013.

[18] Gideon Juve et al. Synthetic workflow generators, February 2019. URL `https://github.com/pegasus-isi/WorkflowGenerator`.

[19] Jonathan Livny. Bioinformatic discovery of bacterial regulatory RNAs using SIPHT. *Bacterial Regulatory RNA: Methods and Protocols*, 905:3–14, 2012.

[20] Pingping Lu et al. A review of cost and makespan-aware workflow scheduling in clouds. *Journal of Circuits, Systems and Computers*, page 1930006, 2018.

[21] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Generation Computer Systems (FGCS)*, 48:1–18, 2015.

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

[23] Ming Mao and Marty Humphrey. Scaling and Scheduling to Maximize Application Performance within Budget Constraints in Cloud Workflows. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (ISPDP)*, 2013.

[24] Aleksander Maricq et al. Taming performance variability. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[25] Marcus Oppitz and Peter Tomsu. Cloud computing. In *Inventing the Cloud Century*, pages 267–318. Springer, 2018.

[26] Redis. In-memory data structure store, February 2019. URL `https://redis.io/`.

[27] Maria A Rodriguez and Rajkumar Buyya. Budget-driven scheduling of scientific workflows in iaas clouds with fine-grained billing periods. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 12(2):5, 2017.

[28] Vishakha Singh, Indrajeet Gupta, and Prasanta K Jana. A novel cost-efficient approach for deadline-constrained workflow scheduling by dynamic provisioning of resources. *Future Generation Computer Systems (FGCS)*, 79:95–110, 2018.

[29] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 13:260–274, 2002.

[30] Laurens Versluis, Mihai Neacsu, and Alexandru Iosup. A trace-based performance study of autoscaling workloads of workflows in datacenters. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 223–232. IEEE, 2018.

[31] Yi Wang, Ye Xia, and Shigang Chen. Using integer programming for workflow scheduling in the cloud. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, 2017.

[32] Zhenyu Wen et al. Fog orchestration for internet of things services. *IEEE Internet Computing*, 21 (2):16–24, 2017.

[33] Henan Zhao and Rizos Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *Processing of the International Parallel and Distributed Symposium*, 2006.

[34] Amelie Chi Zhou, Bingsheng He, and Cheng Liu. Monetary cost optimizations for hosting workflow-as-a-service in iaas clouds. *IEEE Transactions on Cloud Computing*, 4(1):34–48, 2016.