Scheduling Workloads of Workflows in Clusters and Clouds

Ilyushkin, Alexey

DOI
10.4233/uuid:18b7ff8a-6e70-4c9b-8b8e-c5c595fd40ea

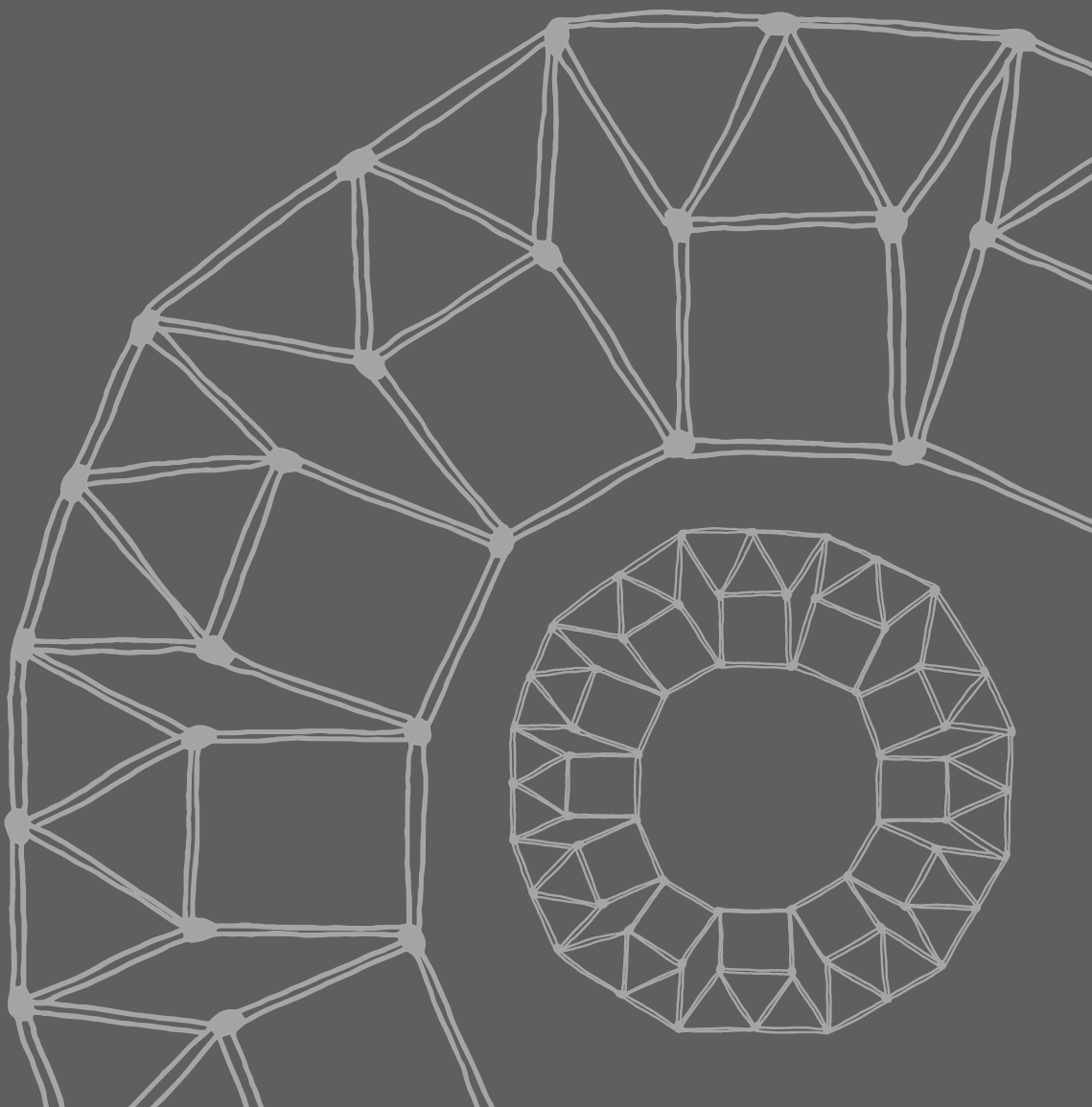
Publication date
2019

Document Version
Final published version

Citation (APA)
Ilyushkin, A. (2019). *Scheduling Workloads of Workflows in Clusters and Clouds*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:18b7ff8a-6e70-4c9b-8b8e-c5c595fd40ea

A.S. Ilyushkin

Scheduling Workloads of Workflows in Clusters and Clouds



Scheduling Workloads of Workflows in Clusters and Clouds

Alexey Sergeyevich Ilyushkin

Scheduling Workloads of Workflows in Clusters and Clouds

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus
Prof. dr. ir. T.H.J.J. van der Hagen,
Chair of the Board for Doctorates
to be defended publicly on
Thursday 19 December 2019 at 15:00 o'clock

by

Alexey Sergeyevich Ilyushkin

Master of Science in
Software for Computing Machinery and Automated Systems,
Penza State University, Russia,
born in Kuznetsk, Penza oblast, USSR.

This dissertation has been approved by the promotors:

Prof. dr. ir. D.H.J. Epema
Prof. dr. ir. A. Iosup

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof. dr. ir. D.H.J. Epema	Delft University of Technology, promotor
Prof. dr. ir. A. Iosup	VU University Amsterdam and Delft University of Technology, promotor

Independent members:

Prof. dr. K.G. Langendoen	Delft University of Technology
Prof. dr. ir. B.R.H.M. Haverkort	Tilburg University
Prof. dr. E. Deelman	University of Southern California, USA
Prof. dr. R. Prodan	University of Klagenfurt, Austria
Dr. P. Grosso	University of Amsterdam
Prof. dr. E. Visser	Delft University of Technology, reserve member

The work described in this dissertation has been carried out in the ASCI graduate school. ASCI dissertation number 409. This work was supported by the Dutch national program COMMIT within the IV-e (e-Infrastructure Virtualization for e-Science Applications) project. Part of this work has been done in collaboration with the Standard Performance Evaluation Corporation (SPEC) within the Cloud Research Group.

Keywords: scheduling, workflow, directed acyclic graph, workload, queuing theory, slowdown, fairness, autoscaling, resource provisioning, allocation, cluster, datacenter, cloud computing, distributed computing

Email: alexey.ilyushkin@yandex.ru

Printed by: Gildeprint B.V., Enschede, The Netherlands

Cover by: A.A. Andreev. The cover shows an artistic interpretation of a regular matchstick graph.

ISBN 978-94-6366-228-4

Typeset by the author with the LaTeX document preparation system.

An electronic version of this dissertation is available at http://repository.tudelft.nl.

В память о маме — Марине Юрьевне Илюшкиной.
In memory of my mother Marina Yuryevna Ilyushkina.

Contents

1

Introduction

Many activities can be split up into multiple smaller, interconnected tasks, which all together are often known by the general term *workflow*. Workflows are widely used in various spheres ranging from managing manufacturing activities to orchestrating computing jobs. The history of workflows started in the pre-electronic-computing era in the early 20$^{\text{th}}$ century. Even though the term "workflow" was not yet in use at that time, the principles that underlie the modern workflow concept were already present. Originally, workflows were mostly adopted to express the *flow* of materials, for example, machine parts, between multiple *workers* to define supply chains and perform planning at the factory level. Taylor, Adamiecki, and Gantt were among the first who proposed to use scientific methods to improve industrial efficiency [151, 121, 44]. While Taylor proposed new management approaches, Adamiecki and Gantt are mostly known for their charts for visualizing a project schedule with dependencies between individual tasks. One of the most prominent early examples of industrial optimization is the introduction of mass production technologies for car manufacturing by the Ford Motor Company. Such manufacturing-related workflows nowadays are classified as *production workflows* [106]. These early developments helped to create planning techniques for complex manufacturing processes with the goal to achieve better work balancing, increase overall worker performance, and make the manufacturing process continuous.

In the subsequent years, the demand for rationalization only increased, which led to the emergence of new mathematical optimization techniques for improving workflow performance. For example, in 1939, Kantorovich established the principles of linear programming [93, 94] when helping to optimize plywood production. The productivity of veneer peeling machines depended on the material being processed, which in turn affected the final product output for this group of machines. The new method exploited this circumstance to optimally distribute the materials among the machines to maximize the product output. In 1947, Danzig published the Simplex method [49] for solving linear programming problems of higher dimensionality, which gained success due to the appearance of electronic computers. In the late

1

1950s, Walker and Kelley developed the Critical Path method [95] minimizing the project cost and completion time. This method optimized project plans by considering the tasks lying on the longest path within a plan—the *critical path*. The tasks on the critical path, which determine the lower bound of the project duration, were given priority.

Later on, the planning and rationalization principles started to find application in other non-purely industrial spheres, e.g., for formalizing and controlling the flow of documents and finances within organizations [37, 69]. Such administrative (organizational) workflow applications are currently classified as *business workflows* [36]. The progress in computing machinery and in software enriched the tools for describing workflows in a completely digital form and further facilitated automated planning.

Finally, workflows started to get used for automating computations, especially in distributed systems. Such workflows are currently classified as *computing workflows*. A workflow (WF), or, alternatively, a Directed Acyclic Graph (DAG), is a convenient concept for representing complex computing jobs, as a typical distributed computing application consists of tasks which communicate with each other. The tasks can perform different roles, which are dictated by the objectives of the application. While one task produces data, the dependent tasks consume them. With workflows, it is possible to define the relationships between the tasks and to control the execution flow.

Computing workflows represent a wide scope of application structures: parallel applications, e.g., Message Passing Interface (MPI) applications, data processing frameworks such as MapReduce [51], and conveniently parallel batch applications such as bags-of-tasks [86]. The most prominent examples of modern computing workflow applications are used in scientific research, especially in support of large-scale physics experiments. Examples are the Large Hadron Collider (LHC), which produced over 50 petabytes of physics data in 2016 [125] and already helped to discover the Higgs boson [20], and the Laser Interferometer Gravitational Observatory (LIGO), which generates one petabyte of data per year [96] and helped to successfully detect gravitational waves [21]. Of course, there are many workflow applications in other scientific fields, such as computational chemistry and biology [129, 143]. Moreover, (often smaller) workflows are used for non-scientific applications, e.g., for steering computations in clouds [57], processing sensor data [113], and orchestrating Internet-of-things devices [165].

The most popular representation of workflows is by means of DAGs. In this dissertation by *job* we understand the whole workflow DAG, by *task* we understand a node of a workflow DAG, and by a *precedence constraint* we understand an edge of the DAG connecting two tasks. The *size* of a workflow is defined as the number of its tasks. A workflow task is *eligible* for execution when all of its ancestors, according to the workflow structure, have completed and all the required data have arrived over the communication links.

Figure 1.1 shows an example of a very simple workflow DAG which consists of five nodes which represent individual computational tasks $t_1, t_2, ..., t_5$ and five edges which represent data and control dependencies (precedence constraints) $w_1, w_2, ..., w_5$. Task t_1 is an *entry task* and task t_5 is an *exit task*. The tasks with

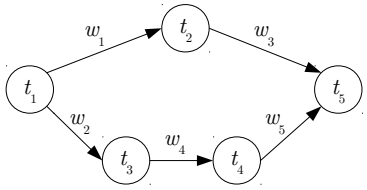


Figure 1.1: An example of a workflow.

outgoing edges are called *predecessors* or *parent tasks* with regard to the tasks in which these edges enter. The tasks with incoming edges are called *successors* or *child tasks* with regard to the tasks from where these edges originate. For example, t_1 is a predecessor of t_2 and t_3, and t_2 and t_3 are successors of t_1. In every workflow DAG it is possible to distinguish basic inter-task interaction patterns which are often observed in practice:

1. The *pipeline* consists of a chain of tasks each of which (except for the first task in the chain) has only one precedence constraint in the DAG from its predecessor in the chain. In Figure 1.1, the tasks t_3 and t_4 constitute a pipeline. The data produced by task t_3 are transferred via link w_4 and consumed by task t_4.
2. The *data distribution* or *map* operation models a situation when multiple tasks have precedence constraints only to a single ancestor task. In Figure 1.1, a map operation consists of tasks t_1, t_2, t_3 and the data transfer links between them, namely, w_1 and w_2. The data produced by task t_1 are distributed (or mapped) to tasks t_2 and t_3 via communication links w_1 and w_2.
3. The *data aggregation* or *reduce* operation models a situation when a single task has precedence constraints to multiple ancestor tasks. In Figure 1.1, a reduce operation consists of tasks t_2, t_4, t_5 and the data transfer links between them, w_3 and w_5. The data produced independently by tasks t_2 and t_4 are transferred via links w_3 and w_5 to task t_5 for aggregation.
4. The *parallel processing* model or *bag-of-tasks* represents the parallel independent execution of multiple tasks which do not have precedence constraints among them. In Figure 1.1 parallel processing is represented by tasks t_2 and t_3, and by t_2 and t_4. The tasks in either of these pairs can be executed in parallel.

1.1. Workflow Scheduling Approaches

The growing applicability of computing workflows has led to the development of appropriate algorithms designed for scheduling them efficiently. Workflow scheduling can be seen from the task placement and resource allocation perspectives, where by *task placement* we understand the mapping of tasks to computing resources, and by *resource allocation* we understand the allocation and deallocation of computing resources. The allocation and deallocation of resources, e.g., virtual

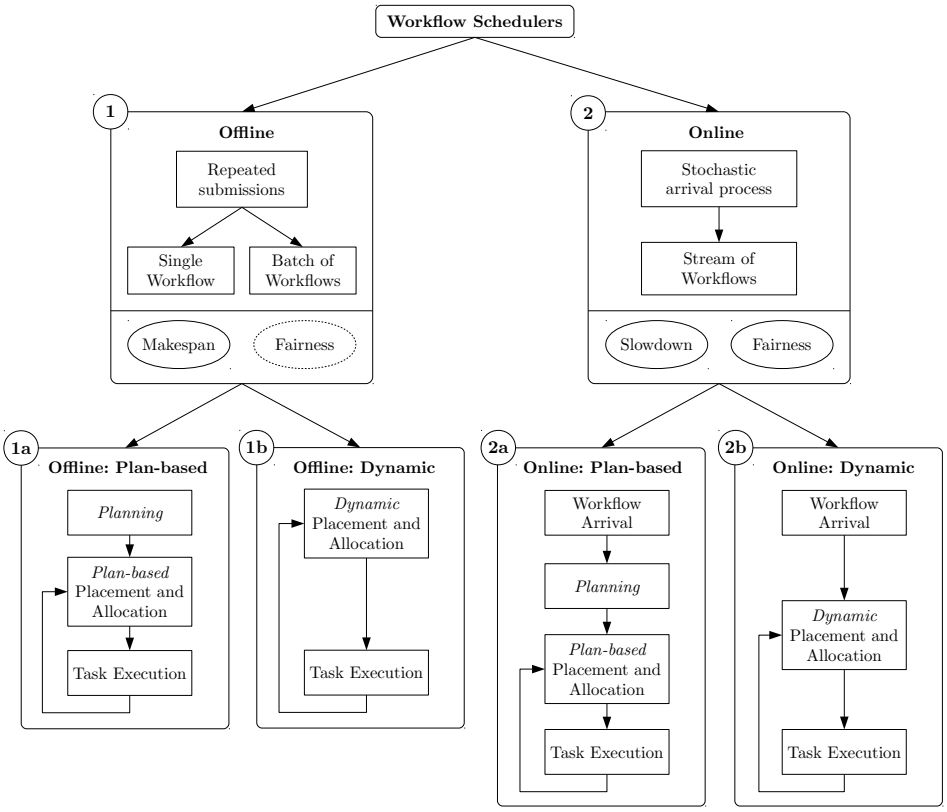


Figure 1.2: A hierarchy of classes of workflow scheduling approaches for task placement and resource allocation.

machines in a public cloud, is usually controlled automatically using *autoscaling* [72]. Figure 1.2 presents the hierarchy of classes of workflow scheduling approaches for task placement and resource allocation used in this dissertation.

Originally, computing workflows were mostly executed on standalone computers or on computing clusters and grids [100, 138, 23]. This execution model is still popular, for example, in research institutions, and it usually implies a limited number of known users running a limited number of pre-defined workflows, possibly submitted and executed in *batches*. We classify scheduling policies designed specifically for static sets (i.e., initially present at the moment when the scheduling decisions are made) of workflows with fully known inter-task dependencies as *offline* policies (Block 1 in Figure 1.2). As the same workflows or batches of workflows are often executed repeatedly, either the user or the scheduling system may be able to derive reasonable estimates of the task runtimes [124, 43]. In such a situation, using not only the fully known inter-task dependencies but also the task runtime estimates, and, sometimes, the communication overheads between tasks, it is possible to create a task placement and resource allocation plan beforehand [155, 170, 23, 33]. Such policies constitute the majority of the available state-of-the-art offline policies,

and we refer to them as *offline plan-based* policies (Block 1a). Another group of offline policies, while still scheduling an initially present set of workflows, does not construct any plan, but, instead, makes the scheduling decisions on-the-fly, e.g., when a task becomes eligible or a resource becomes idle [116]. We refer to such policies as *offline dynamic* policies (Block 1b).

For offline policies, the most common performance metric from the end-user perspective is the minimization of the *makespan* of a workflow or a batch of workflows. For a single workflow, the makespan is defined as the time between the start of its first task until the completion of its last task. Accordingly, for a batch of workflows the makespan is defined as the time between the start of the first task of any workflow in the batch until the completion of the last task of any workflow in the batch. It is often supposed that a workflow or a batch is executed on resources reserved beforehand exclusively for the submitting user, so that the policy is fully aware of the number and the type of the available resources [142, 24, 164]. This means that during the execution of workflows belonging to a certain user, no direct interference on their performance can be observed from other users. Therefore, the access to computing resources among multiple users is usually outside the scope of offline policies. When scheduling a batch of workflows, the user is normally interested in the minimization of the makespan of the whole batch rather than in the minimization of the makespans of specific workflows within the batch.

Nowadays, computing workflows are used in a much wider scope compared to their earlier applications in clusters and grids. They have become a popular automation tool for controlling computations in various types of online systems, for example, within website back-ends, in data analytics services, in computing clouds, etc. [162, 158]. In such environments, multiple users can constantly submit their workflow jobs, thus generating a stochastic arrival process, leading to *workloads* (streams) of workflows.

In such workloads of workflow, every arrival can contain a unique workflow structure, so that deriving statistical information on task runtimes and inter-task communication overhead is difficult [43]. Various workflows can also have different priorities. For example, while some workflows are responsible for running business-critical infrastructure and are very sensitive to delays, e.g., processing of radar data for a weather forecasting website, other workflows process auxiliary data, e.g., user preferences in social networks which are not time-critical. Additionally, workflows can be assigned *deadlines* that define the time by which each workflow should be completed [164]. In cloud computing, the resources are paid for based on usage, so that the user normally has a certain *budget* to spend on workflow execution [119, 164]. Cloud computing services provide various types of resources with different performance characteristics and costs. The same workflow can be executed on different combinations of resource types, which leads to a variety of performance and cost options. All these factors make scheduling of workloads of workflows challenging and requires special policies [169, 77, 29]. The scheduling of workloads of workflows can be classified as an *online* problem (Block 2). In online scheduling, similarly to offline scheduling, we distinguish *online plan-based* scheduling (Block 2a) and *online dynamic* scheduling (Block 2b). The *plan-based* scheduling approach constructs a (partial) plan on every workflow arrival and

strictly follows this plan to perform task placements and resource allocations between workflow arrivals. In the *dynamic* approach, the scheduling decisions are made just-in-time, e.g., when a workflow arrives, a task becomes eligible, or a processor becomes idle.

For online policies, the response time and slowdown of workflows are more appropriate performance metrics rather than the makespan. While the *response time* captures the queuing time and the makespan, the *slowdown* is defined as the ratio between the response time and the ideal makespan, e.g., when a workflow runs in an empty system. Additionally, due to multi-tenancy, when multiple users share the same computing resources, the scheduler is responsible for fair access of workflows to the resources, where fairness can be defined in multiple ways, depending on the application. For example, fairness can be interpreted as a guarantee that a workflow obtains a share of the computing resources proportional to its (user-defined) priority, and, accordingly, is expected to achieve better performance. In offline scheduling, fairness is rarely addressed as it is usually supposed that all the submitted workflows belong to the same user. The notion of fairness partially overlaps with the notion of Service-Level Agreement (SLA), which is usually understood as a commitment between the end-user and the service provider regarding the quality of the provided service. Compared to fairness, SLA is a broader notion as it usually involves service availability guarantees, responsibilities for SLA violations, etc. [74] When using online policies, it can be problematic to achieve fairness and meet SLAs for individual workflows due to insufficient statistical information on historical workflow runs and limited time available for making scheduling decisions.

In this dissertation, as a system-oriented metric we often use the maximal utilization, which is defined as follows. First, the *instantaneous utilization* at any moment in time is the ratio of the number of currently occupied resources (which are running tasks) and the total number of available resources. When we refer to the utilization, we usually mean the *average utilization* on a certain time interval (e.g., the duration of an experiment), which is the average of the instantaneous utilization on that interval. Due to the dependencies among workflow tasks, the tasks may not be able to start their execution even though there are idle resources—the tasks have to wait to become eligible. In other words, workflow scheduling is not work-conserving. As a consequence, it will in general be impossible to achieve a utilization of 1.0, and the system will become unstable, i.e., when the queue of waiting workflows/tasks will grow without bounds, at utilizations lower than 1.0. We define the *imposed utilization* of a specific workload with some (stochastically) defined mix of workflows as the utilization the system will achieve under that workload, given also its arrival rate, if the system would be able to deal with it in a stable way. The imposed utilization can be computed from the average total runtime of the workflows, the arrival rate, and the size of the system. If the system is unstable, the utilization actually achieved will fall short of the imposed utilization. Finally, we define the *maximal utilization* (for a given workload of workflows and task placement policy) as the highest utilization that can be achieved, i.e., such utilization ρ_m so that for any imposed utilization ρ with $\rho < \rho_m$ the system is stable (not saturated), and for any imposed utilization ρ with $\rho > \rho_m$ the system

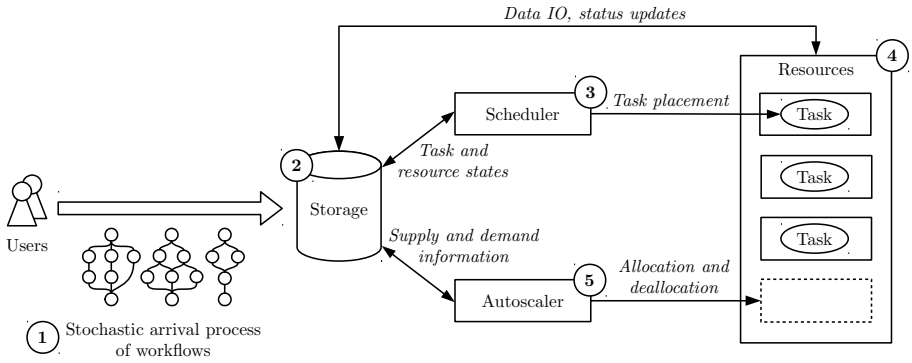


Figure 1.3: The general architecture of a workflow management system for online workflow scheduling.

is unstable (saturated).

Workflows are usually scheduled and executed by specialized workflow management systems. The general architecture of a typical workflow management system, which we also use in this dissertation, is presented in Figure 1.3. The presented system is supposed to be deployed in a large-scale homogeneous or heterogeneous computing system, such as a large cluster or a datacenter. The system is subject to an *arrival stream of workflows* (Component 1 in Figure 1.3). The workflows in the stream arrive according to some stochastic process. The *storage* (Component 2) stores the information on submitted workflows and their tasks. The *scheduler* (Component 3) places tasks onto the computing resources. The *computing resources* (Component 4) are capable of running workflow tasks. The optional *autoscaler* (Component 5) automatically controls the number of allocated resources. We only consider processors as the type of system resources that can be controlled by the scheduler, the autoscaler, or both.

1.2. Workflow Scheduling Challenges

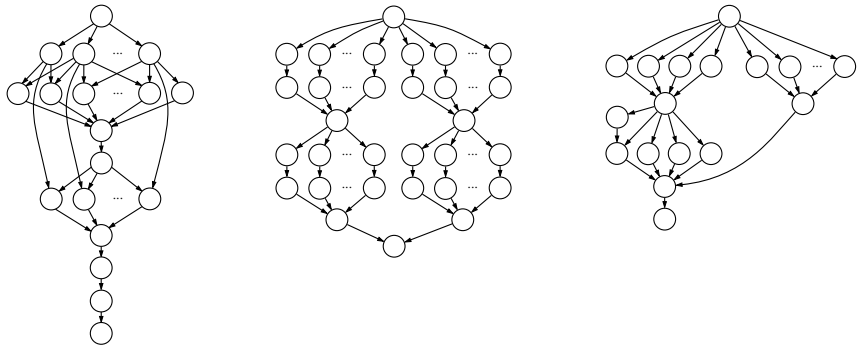
In this dissertation, we study the problem of *online* scheduling workloads of workflows which arrive over time in a distributed computing environment such as a cloud, a cluster, or a datacenter. We see a growing demand for online dynamic scheduling policies for workloads of workflows with support of multiple users, as the existing state-of-the-art policies cannot provide the required performance characteristics. While existing offline policies are not always directly applicable to workloads due to different performance goals, state-of-the-art online plan-based policies have potential scalability issues when dealing with sudden workload surges as their planning overhead limits the system and workload performance [118, 119, 160]. Partially, this is the reason why modern cloud services, which are the main target of the execution of such workloads of workflows, demonstrate significant performance variability especially from the multi-tenancy perspective [86, 105]. We identify and focus in this dissertation on three major challenges for online scheduling of workflows.

1. *How to realistically estimate the resource demand of a workflow?* Usually, workflows consist of segments with different parallelism and different interconnection types between tasks which affect the order how the tasks become eligible. Moreover, realistic task runtime estimates are not always available in advance and can only be obtained after a real workflow execution [43] or, in some cases, after a code analysis [99]. The user runtime estimates are often subjective [124] and thus less useful. The knowledge of resource demand is important for making good task placement and resource allocation decisions.
2. *How to assign tasks of workflows to resources to minimize average slowdown while achieving fairness?* A wrongly chosen task placement policy can easily degrade the performance of certain workflows in workloads and negatively affect the fair access of workflows to computing resources [32]. Fairness is very important in concurrent environments where multiple users simultaneously execute their workflows, as different users could have different numbers and types of workflows submitted. However, the efficiency of certain scheduling policies could depend on the system utilization, and fairness can be interpreted differently depending on user and service provider interests. For example, jobs can be prioritized based on the total number of submitted jobs by a user, based on the disposable budget, or on the number of tasks in the jobs, etc.
3. *How to allocate resources for workflows while meeting deadline and budget constraints?* Modern computing infrastructures make it possible to easily lease and release computing resources. Autoscaling decisions should be made wisely to minimize slowdowns and, accordingly, minimize deadline violations. In order to minimize incurred costs, the over-provisioning of resources during autoscaling should be minimized, and the use of leased resources by the placement policy should be efficient. Online scheduling makes it possible to react to changes in the number of allocated computing resources during the workload execution. Thus, preferably, the autoscaler and the scheduler should operate in tandem to achieve common optimization goals without counteracting each other.

1.3. Workflow Applications

The majority of extreme-scale workflows originate from the scientific domain. The Pegasus project [55] has done a great job on classifying and modelling many important scientific workflows. Figure 1.4 shows the structure of three typical scientific workflows from different fields. The Montage workflow [90, 152, 38, 92], created by NASA/IPAC Infrared Science Archive, is used in astronomy for processing telescope images to produce large custom mosaic images of the sky. Various operations can be applied to the input images, for example, they can be rotated, scaled, or their brightness can be adjusted. The geometry of the final mosaic depends on the geometry of the input images. The workflow has a complex structure, which is determined by the types of applied operations, and its size is determined by the number of processed images.

The LIGO Inspiral Analysis workflow processes the data from the detectors of the Laser Interferometer Gravitational Wave Observatory (LIGO) [22], and



(a) Montage (b) LIGO (c) SIPHT

Figure 1.4: The structure of the scientific workflows we use in our experiments.

searches for signals from binary inspirals—pairs of binary stars that are losing energy or binary black holes. The workflow successfully helped to detect in 2015 gravitational waves predicted earlier by general relativity. The experiment uses three laser interferometers and records approximately 1 TB of data per day. The LIGO workflow is extremely parallel and consists of many smaller (sub-)workflows combined into a single workflow. Each sub-workflow handles a block of time-frequency data from one of the detectors with a length of 2048 seconds. For each block, and for each detector, the workflow generates a set of template waveforms which are used to filter the data in blocks. By matching the data to each template, a binary inspiral signal can be detected by the high level of the signal-to-noise ratio. The task will produce a trigger which will be validated by comparing with triggers from other detectors.

The sRNA Identification Protocol using Highthroughput Technologies (SIPHT) workflow [108, 38, 92] is used to search for small untranslated bacterial regulatory sRNAs which are responsible, for example, for secretion and virulence of cells. The structure of SIPHT workflows stays almost the same between separate workflow instances, and only differs in the number of Patser [159] tasks which perform matrix-based pattern matching procedure. The number of Patser tasks depends on the input, more precisely on the number of transcription factor binding sites (TFBSs). However, the most computationally intensive tasks are those which compare sequence combinations.

The Large Hadron Collider is another prominent example of large-scale scientific workflows [39, 19]. Currently, it relies on a computing grid which consists of 170 interconnected computing centers. The Tier 0 computing center is located in CERN and processes the raw LHC data. The output from Tier 0 is consumed by twelve Tier 1 computing centers which split the data into smaller parts which are further processed by dependent computing centers down the grid. For each experiment these operations are controlled by a special workflow.

In 2017 there were 6.4 billion of IoT devices in the world [126]. With the appearance of cheap platforms such as the Raspberry Pi, esp8266, and Arduino, the number of IoT devices continues to increase and is expected to reach 20.8 billion by

Policy Name	Type	Year Published	Reference
HEFT	offline plan-based	2002	[155]
Hybrid.BMCT	offline plan-based	2004	[142]
OWM	online dynamic	2011	[77]
FDWS	online dynamic	2012	[29]
HR	online dynamic	2008	[169]

Table 1.1: An overview of task placement policies.

2020. Most of the IoT applications are represented as data transformation workflows. For example, the location data obtained from tracking devices installed on public transport and from mobile phones are pre-processed, filtered, and aggregated to calculate the road situation. Or smart thermostats installed throughout a building send temperature readings to the central heating controller which aggregates the data and optionally reports it to the energy supplier to estimate the energy consumption and calculate the efficiency. A similar example of a data transformation workflow is the content delivery networks which perform video transcoding [98] on-the-fly between the content providers, such as YouTube, and the clients. The key difference between data transformation workflows and scientific workflows is that the former can run the multiple chained tasks simultaneously by simply "pumping" the data through them. However, data transformation workflows can alternatively be decomposed into a set of classic workflows with sequentially running tasks.

Workflows are also common in data analytics frameworks, such as Hadoop [3]. For example, BTWorld [70] is a time-based big data analytics workflow for analyzing the evolution of the BitTorrent peer-to-peer network [45]. It uses the Hadoop distributed file system (HDFS) as a storage engine, Hadoop as an execution engine, and Pig Latin [130] as the high-level SQL-like language that compiles automatically into MapReduce jobs. New time-stamped records, representing snapshots of the global BitTorrent network, are periodically added to the continuously growing BTWorld data set. BTWorld relies on a logical workflow of seven types of SQL-like queries executed over the data records from the dataset. The total size of data files processed by the workflow by 2013 exceeded 14 TB. Currently, the data collected by BTWorld during its more than 8 years of operation represent one of the largest studies of peer-to-peer systems.

1.4. Workflow Task Placement Policies

In this section, we give an overview of relevant state-of-the-art workflow scheduling policies specifically created for task placement, following the classification in Figure 1.2. We only discuss here offline plan-based and online dynamic task placement policies, as, to the best of our knowledge, offline dynamic and online plan-based policies are mostly used in combination with resource allocation, and are further presented in Section 1.5. The policies discussed in this section are summarized in Table 1.1, and are used further in the dissertation for comparison with the novel proposed policies. We use several policies from each group to give an idea of their working principle. Additional information on more relevant policies is provided in the related work sections in each appropriate chapter.

Before describing the workflow task placement policies, we start with introducing some notions. Scheduling of workflows often operates with the notions of eligible set and level of parallelism. For a workflow, at any point in time before or during its execution, its *eligible set* (of tasks) is the set of non-completed tasks of which the precedence constraints have been satisfied. In other words, the eligible set is the set of all tasks that are currently running and those that are waiting but that could run if sufficient resources were available. More generally, we introduce the notion of the *generation-i eligible set* (or the *eligible set of generation i*) at any point in the execution of a workflow, which is a potential future eligible set. The generation-0 eligible set of a workflow is simply equal to its current eligible set. The generation-$(i+1)$ eligible set of a workflow contains all the tasks that will become eligible when (exactly and only) all the tasks from its generation-i eligible set have completed. It is important not to confuse eligible sets with levels in a workflow. The *levels* consist of the tasks that have the same distance from the entry task. However, the eligible sets of generation-i can differ from the levels, because paths of different lengths can exist in a workflow from the entry task to any other single task.

For a workflow that has not yet completed, we define its *Level of Parallelism* (LoP) as the maximum number of processors it may ever use at any future point in its execution, which is equal to the maximum number of tasks in any of its potential future eligible sets. Of course, the LoP of a workflow can only stay the same or decrease during its execution. The LoP can be computed exactly [79] or approximately [83].

The *upward rank* is often used to prioritize tasks in workflows based on their duration and proximity to the exit task. The upward rank requires task runtime estimates for all the resource types and communication overhead estimates for all the communication links to be known beforehand. Since the exit task has no successors, its upward rank simply equals to its average estimated runtime on all the resource types. For each task in a workflow its *upward rank* is recursively calculated, starting from the exit task, as the sum of the average estimated runtime of the task and the maximal sum (among all its immediate successors) of the upward rank of an immediate successor and the average communication overhead between the current task and that successor. A more detailed definition of the upward rank is provided in Section 3.3.1. The average estimated execution time is calculated for each task using the average speed of the processors in the system. The average estimated communication cost is calculated as the average communication start-up time plus the size of the data to be transmitted, divided by the average transfer rate between the processors. The length of the critical path of a workflow is equal to the maximum value of the upward rank among all its tasks. By workflow *length* we mean the length of its critical path.

Similarly, the *downward rank* of a task is recursively calculated starting from the entry task as the maximal sum, among all the immediate predecessors, of the downward rank of an immediate predecessor, the average estimated execution time of the predecessor, and the average communication overhead between the current task and that predecessor. For the entry task the downward rank is zero.

We now turn to the discussion of the policies. The Heterogeneous Earliest Finish

1

Time (HEFT) [155] is an offline plan-based policy proposed by Topcuoglu in 2002 and it is one of the most well-known heuristics for scheduling individual workflows. Given a workflow DAG with known task runtime estimates and communication delays between the tasks, this policy produces a task placement plan which is used to steer the workflow execution. HEFT uses upward rank to prioritize tasks. It adds the tasks to the plan in descending order of their upward ranks and assigns the tasks to processors that minimize their earliest finish times.

The Hybrid Balanced Minimum Completion Time (Hybrid.BMCT) [142] is an offline plan-based policy proposed by Sakellariou and Zhao in 2004 and consists of two parts: Hybrid and BMCT. The Hybrid part first sorts workflow tasks in descending order of their upward ranks, and then successively groups the tasks into eligible sets so that each set contains tasks with no direct dependencies between them. Each task can belong to a single eligible set only. For each workflow the eligible sets are numbered in the order of their creation, with the eligible set containing the entry node of the workflow having number one. The tasks from each eligible set are assigned to processors using the BMCT list-scheduling policy with the objective to complete the execution of all tasks as early as possible. In the beginning, BMCT plans the tasks to the processors minimizing their execution time. After the initial assignment, BMCT tries to optimize the plan by moving tasks between processors to minimize the overall makespan until no further improvement is possible.

The Online Workflow Management (OWM) [77] is an online dynamic policy proposed by Hsu, Huang, and Wang in 2011. It maintains a single joint eligible set containing only a single eligible task (if any) with the highest upward rank from every workflow present in the system. As long as there are eligible tasks in the system, the scheduler selects the task with the highest upward rank from the joint set and tries to assign it to a processor. If the idle processors have different speeds, the task is placed on the fastest one. If the processors have the same speed, the policy checks for the busy processor which will be idle first, whether the task has a lower estimated finish time on it rather than on any idle processors. If that the case, the task is postponed, otherwise, it is placed on any of the idle processors.

The Fairness Dynamic Workflow Scheduling (FDWS) [29] is an online dynamic policy proposed by Arabnejad and Barbosa in 2012. It maintains a single joint eligible set in the same way as OWM. However, within the joint set each task is prioritized based on the fraction of remaining tasks of its workflow and the workflow length.

The Rank Hybd (HR) [169] is an online dynamic policy proposed by Yu and Shi in 2008. The policy maintains a single joint eligible set of *all* the eligible tasks from all the workflows in the system. If the tasks in the joint set belong to different workflows, the scheduler selects the task with the lowest upward rank. If the tasks in the joint set are from the same workflow, the algorithm selects the task with the highest upward rank.

1.5. Workflow Resource Allocation Policies

In this section, we give an overview of different types of autoscalers for workflows—specialized policies targeting resource allocation. The resource allocation problem

Policy Name	Type	Year Published	Reference
DSCS	online plan-based	2011	[118]
PBTS	online plan-based	2011	[40]
Scheduling-First	online plan-based	2013	[119]
Scaling-First	online plan-based	2013	[119]
IC-PCP	offline plan-based	2013	[24]
DPDS	offline dynamic	2015	[116]
SPSS	offline plan-based	2015	[116]
DCCP	offline plan-based	2017	[33]
MIP	offline plan-based	2017	[164]

Table 1.2: An overview of resource allocation policies.

has attracted much attention in recent years due to the appearance of cloud computing with its on-demand resource provisioning model. The policies considered in this section are summarized in Table 1.2.

Autoscaling policies are usually designed to serve workloads and to operate in an online setup by leasing and releasing computing resources on-the-fly during execution [25]. The *demand* of resources is the minimal number of resources required for delivering the service according to SLA. The resource *supply* is the number of allocated (idle, booting, or busy) resources. In the context of autoscaling for workflows, the momentary demand is usually calculated based on the number of eligible and running tasks in all the workflows in the system. The goal of autoscalers is to match the supply and demand so that ideally, there are always enough resources to execute eligible tasks while not violating the SLA. Thus, when the supply exceeds the demand, the system is *over-provisioning*, and when the supply is lower than the demand, the system is *under-provisioning*.

Papers describing workflow scheduling policies often use overlapping approaches when addressing both task placement and resource allocation problems. Thus, it is often hard to draw a clear line to unambiguously classify the policies. The majority of autoscalers for workflows are plan-based, i.e., they are usually invoked periodically to create a resource allocation plan (and, often, a task placement plan) for the period between successive autoscaler invocations. At the same time, a large amount of research has been done for offline scheduling of a single workflow or a batch of workflows. Such offline policies, despite creating complete allocation and placement plans only once before the execution, use very similar techniques to the plan-based online autoscalers. Thus, even though authors of offline workflow resource allocation policies sometimes do not directly present these policies as potential autoscalers, we include them in this section (without calling them autoscalers) as they can be easily adapted to be used in a periodic autoscaling setup.

Additionally, we distinguish general and workflow-specific autoscalers. *General* autoscalers have been proposed for request-response applications, such as web-servers [26, 64], and can be potentially applicable for autoscaling workloads of workflows. *Workflow-specific* autoscalers have been developed specifically for autoscaling workflows, and, for example, are aware of the workflow DAG structure [118, 40]. In this section, we focus solely on the workflow-specific autoscalers.

The Dynamic Scaling-Consolidation-Scheduling (DSCS) policy [118] is an online plan-based autoscaler which combines scheduling and allocation approaches, published by Mao and Humphrey in 2011. DSCS is one of the first proposed autoscalers dedicated for workloads of workflows. The policy supports soft deadlines assigned on a per workflow basis by the user and supposes that the budget is unlimited. The main optimization goal of DSCS is to meet as many workflow deadlines as possible while minimizing the total operational cost. In the beginning, the policy groups workflow tasks that share the same predecessor and prefer the same resource type, and considers them as a single task. To improve the resource utilization, the policy tries to plan sequential execution of certain parallel tasks, if it does not lead to the deadline violation. Then DSCS distributes the per workflow deadline among its tasks proportionally to their runtimes on the most cost-efficient resources. If such assignment does not allow the workflow to finish by its deadline, DSCS tries to reduce the workflow makespan by placing certain tasks on faster but more expensive resources, which allow to shorten the maskespan at lowest increase in operational cost. The resources are allocated based on so-called "load vectors", where the load vector represents the number of resources of a certain type required to finish the task without violating its deadline. Finally, after resources are allocated, DSCS plans the tasks to the resources giving priority to the tasks with earliest deadlines.

The Partitioned Balanced Time Scheduling (PBTS) policy [40] is an online plan-based autoscaler proposed by Byun, Kee, Kim, and Maeng in 2011. The policy creates a resource allocation and task placement plan for each "time partition" which equals the billing period of the cloud provider, e.g., one hour. The goal of PBTS is to minimize the operational cost within a time partition and finish the workflow within its deadline. To minimize the operational cost, PBTS places the tasks on resources by allowing workflow tasks to be delayed (by slack time) if such a delay does not violate the workflow deadline. For that, PBTS, following the workflow structure, builds a task placement plan for each time partition to determine how the required resource demand is going to change. Then the policy distributes the slack time over time partitions to minimize the operational cost. Finally, PBTS approximates the required resource capacity within the target time partition and places the tasks on the resources.

The Scheduling-First policy [119] is another online plan-based autoscaler proposed by Mao and Humphrey in 2013. It is designed for autoscaling workloads of workflows in clouds with budget constraints. Each workflow has a user-assigned numeric priority, the budget is provided per autoscaling interval and is distributed among all the submitted workflows based on their priority. The policy iterates through the eligible tasks of all the workflows sorted in the descending order of their workflow priorities, and for each task it tries to allocate the fastest resource while there is enough budget. After that, the policy consolidates the resources by planning the execution of the remaining eligible and not-yet eligible tasks on the allocated resources.

The Scaling-First policy [119] is an online plan-based autoscaler proposed by Mao and Humphrey together with Scheduling-First. It first creates an independent per-workflow plan so that the number of resources in each plan could be bigger than the actual maximal number of available resources in the system. Then it calculates

the cost of each plan and scales it down to fit within the budget constraint. Finally, the policy allocates the resources based on the plan information and performs the resource consolidation.

The IaaS Cloud Partial Critical Paths (IC-PCP) policy [24] is an offline plan-based policy proposed by Abrishami, Naghibzadeh, and Epema in 2013. The policy is designed for planning a workflow execution in an IaaS cloud, and supposes that the workflow has a user-specified deadline and an unlimited budget. The optimization goal of the policy is to minimize the execution cost of the workflow, while completing the workflow before the deadline. IC-PCP finds the critical path in the workflow and distributes the workflow deadline among the tasks on the critical path in proportion to their minimum runtime. After that, each task on the critical path has a deadline assigned which is used to compute task deadlines for all of its predecessors. Finally, the policy plans tasks on the cheapest resources that allow to meet task deadlines.

The Dynamic Provisioning Dynamic Scheduling (DPDS) policy [116] is a offline dynamic autoscaler proposed by Malawski in 2015. It is designed to provide autoscaling for already present ensembles of scientific workflows *during the execution*, where each workflow has a user-assigned numeric priority. In a sense, DPDS is a transitional phase between a purely offline plan-based policies and online dynamic policies. DPDS supports cost- and deadline-constrained provisioning of resources of a single type only, where the deadline and the budget are provided for the whole ensemble. DPDS calculates the number of resources to provision so that the entire budget is consumed before the deadline. It allocates the calculated number of resources at the beginning of the ensemble execution. Then it leases or releases resources based on their utilization by the workflows according to given thresholds. To schedule workflow tasks, DPDS maintains a joint eligible set in the same way as in the aforementioned HR policy (see Section 1.4). The tasks from the set are placed on random idle resources according to the priority of the workflow to which they belong.

The Static Provisioning Static Scheduling (SPSS) policy [116] is an offline plan-based policy also proposed by Malawski in the same paper as DPDS. SPSS creates a plan for each workflow in the ensemble in priority order, and rejects any workflow that exceeds the deadline or budget. The goal of the policy is to finish each workflow by the deadline with the lowest possible cost to maximize the number of workflows completed within the given budget. SPSS distributes the workflow deadline to its individual tasks considering the slack time of the workflow, which is the additional amount of time that a workflow can extend its critical path while completing by the ensemble deadline. The slack time is distributed to each level of the workflow proportionally to the number of tasks in that level and the total task runtime in that level. The tasks are then planned on resources that minimize the execution cost and meet the task deadlines. A new resource is allocated if there are no slots available on the allocated resources that would allow the task to finish by its deadline. SPSS performs resource consolidation, similarly to Scheduling-First and Scaling-First policies, in order to increase the utilization of allocated resources.

The Deadline Constrained Critical Path (DCCP) policy [33] is an offline plan-based policy proposed by Arabnejad, Bubendorfer, and Ng in 2017. The policy has

the same goal as the IC-PCP policy. For each workflow task, DCCP calculates its distance to the exit task, which is the number of DAG edges on the shortest path to the exit task, and then groups the tasks with the same distance. Then the user-defined deadline is distributed among the levels of the workflow so that the levels with bigger task runtimes get a larger share of the workflow deadline. DCCP relies on the notion of Constrained Critical Path (CCP) [97], which denotes the subset of tasks laying on the critical path that are currently eligible. For finding the critical path, the policy utilizes modified upward and downward ranks that aggregate the communication overhead between a task and its successors and predecessors, instead of just selecting the maximum communication overhead. DCCP co-locates tasks from CCP that communicate within the same resource. The tasks are planned on resources that minimize their earliest completion time and do not exceed the level deadline.

The Mixed Integer Programming (MIP) policy [164] is an offline plan-based policy proposed by Wang, Xia, and Chen in 2017. The goal of MIP is to minimize the operational cost of using different types of cloud resources while ensuring the completion of the scheduled workflows by their deadlines. In contrast to all the considered heuristic policies, the MIP approach guarantees that the produced solution is optimal. The policy requires task runtime estimates and formulates the workflow scheduling problem using five MIP constraints: (i) Precedence constraints that require the tasks to be executed in the order specified by the workflow, (ii) Resource constraints that describe the requirements of each task to certain resource types, and set the limits to the maximal number of resources of each type that can be allocated, (iii) Non-overlapping constraints describe that each resource can run only a single task at a time, (iv) Ready time constraints that specify the time when each workflow can start its execution, and (v) Deadline constraints that require each workflow to finish by its deadline. The policy relies on the Gurobi MIP solver [14] for finding the solution. The policy models the decision space as a matrix of time slots vs. resources where each element is a binary decision variable which represents a task assignment. The precision of the solution depends on the time discretization used.

1.6. Workflow Management Systems

Computing workflows are usually executed by specialized workflow management systems. Even though such systems can provide different functionality, the fundamental properties inherent to most workflow management systems are: controlling task execution order in accordance with the workflow structure, and scheduling and placing of tasks on resources. Optionally, workflow management systems implement dedicated interfaces for data exchange between tasks, and resource management functionality.

To control the task execution order within a workflow, a workflow management system needs to know which tasks are eligible to start their execution. To this end, a workflow management system either periodically or in an event-driven manner monitors task statuses. Periodic monitoring means that the tasks are periodically polled by the workflow management system and their current status is checked. Event-driven approach means that a task sends a message to the workflow manage-

ment system when it changes its state, thus, potentially enabling faster triggering of descendants. Alternatively, a task can send messages directly to its descendants to trigger them. Various hybrid approaches are also possible, including the modification of inter-task dependencies during the workflow execution based on a certain criteria. Periodic monitoring is employed by most workflow management systems, e.g., Pegasus [17, 54], ASKALON [61], and by the more recent Apache Airflow [2]. Event-driven approach is common in serverless computing services, e.g., Amazon Lambda [1], IBM Cloud Functions [15], and Azure Functions [5], where serverless computing is a form of cloud computing enabling outsourcing of the operational logic to the service provider.

The eligible tasks are placed on resources according to a scheduling policy implemented by the workflow management system. Data can be exchanged between tasks either in peer-to-peer manner or centrally, e.g., through a shared file system. In any case, the workflow management system is aware about the exchanged data if that is important for determining task eligibility. Resource management is often performed by a specialized independent software layer, which, in some cases, can perform automatic allocation and deallocation of resources to react and adapt to the fluctuations in the number of eligible tasks to meet performance goals.

It is important to notice that the traditional approach, used by the majority of standalone workflow management systems, supposes that the user is responsible for deploying most of the software layers implementing the operational logic. In contrast, serverless services do this job for user, guaranteeing that the workflows are containerized, deployed, scheduled, provisioned, and available on demand, while only charging the user for the used resources. However, all the aforementioned workflow management systems and even the serverless computing services, require the user to first define a workflow which will stay in the system and will further be triggered either periodically or by an event. Thus, most of the existing systems, even if they are capable of processing workloads of workflows, are still designed with the offline approach in mind.

The Pegasus project [54, 55, 17] has developed a workflow management system for scientific workflows which first appeared in 2001. The development of Pegasus was motivated by the necessity to match the growing computational needs of the scientific community and the available cyberinfrastructures. Pegasus supports multiple execution engines, thus, the same workflows can be run on different physical infrastructures such as clusters, grids, and clouds. Based on the user-provided XML description of a workflow, Pegasus generates an executable workflow, i.e., the mapping of workflow tasks on the resources. The job scheduler controls the execution flow of individual workflow jobs according to the mapping. Pegasus allows local and remote execution of workflow tasks, possibly structured as a sub-workflow. A separate monitoring component provides information on running workflows, tasks and performance.

The ASKALON grid environment [61, 34] is designed specifically for execution of workflows in dedicated datacenters and computing grids. In ASKALON, the scheduler is responsible for processing the user-provided XML-based workflow specification and mapping workflow tasks onto resources. The scheduler uses HEFT as the primary scheduling policy, however, other policies are also supported.

The execution process is controlled by an enactment engine which resolves the data flow dependencies between tasks according to the mapping generated by the scheduler. The resources are controlled by the ASKALON resource manager which is responsible for discovering and reserving the resources based on requests from the scheduler. ASKALON also provides functionality for predicting task runtimes and data transfer times using the history of previous executions.

E-Science Central [75, 10] is a cloud computing platform and a service for execution of scientific workflows, data management, analysis and collaboration with support of private and public clouds. The platform provides a set of virtualized services that are available to the user as a Software-as-a-Service (SaaS) application through a Representational State Transfer (REST) interface. Workflows in e-Science Central can be defined either graphically or using an XML-based language. The platform allows users to upload their own services, representing workflow tasks, which are deployed on demand by a workflow enactment engine when they are required for workflow execution. The services exchange messages through a message execution engine. The workflow execution engine is responsible for creating a message plan which describes the order in which the services must be executed.

Apache Taverna [78, 4] is a workflow management system created for constructing and running workflows of services. It was originally designed for executing molecular biology workflows. Taverna workflows can be executed either locally or remotely using the Taverna Server service, which allows sharing workflows to a larger community of scientists. Different types of web interfaces are supported by services, constituting user workflows. The Freefluo enactment engine is responsible for executing the workflow based on the user-provided XML-definition, while the scheduling of workflow tasks and control of precedence constraints are driven by messages exchanged between workflow services.

Apache Airflow [2] is a workflow management system designed initially by Airbnb for automating data warehousing and analytics within the company. Airflow uses Python-based workflow descriptors which incorporate both inter-task dependencies and the task implementation code. Airflow uses a central scheduler which runs as a separate instance and makes decisions based on the status of workflows and tasks obtained from the database. Local and remote task execution are supported. The simplest way to distribute multiple workflow tasks over a set of Airflow workers is by using the Celery [7] asynchronous task queue. More elaborated execution models are supported with, for example, the Dask library [9] for parallel computing in Python to automatically parallelize individual workflow tasks.

1.7. Problem Statement

The research questions we address in this dissertation cover various aspects of task placement and resource allocation problems when dealing with the problem of online scheduling workloads of workflows in homogeneous or heterogeneous distributed computing environments. We identify the following research questions:

RQ1: What are appropriate policies for online scheduling of workflows without knowledge of task runtimes? Workflow scheduling policies often rely on the knowledge of task runtime estimates. However, in situations with unique arrivals, it is not always possible to obtain such estimates. When the task runtime estimates are

unknown, the workflows and workflow tasks can be still prioritized in different ways using alternative performance metrics, and various resource reservation schemes can be utilized.

RQ2: How do inaccurate task runtime estimates affect the performance of workflow scheduling algorithms? In situations when it is possible to obtain task runtime estimates, for example, when tracking and analyzing repeated workflow submissions, the quality of such estimates can vary. The impact of inaccuracy in task runtime estimates is rarely addressed by the designers of workflow scheduling policies. Usually, fully correct task runtime estimates are assumed. Inaccuracies in runtime estimates can have a different performance impact depending on statistical characteristics of workloads and the heterogeneity of the system.

RQ3: What is the performance of general versus workflow-specific autoscalers? Many general autoscalers which are agnostic to the controlled application types have been proposed. They are general because they mostly make their decisions using only external properties of the controlled system, e.g., workload arrival rates, or the output from the system, e.g., response time. At the same time, a variety of workflow-specific autoscalers have been proposed, which have access to the workflow structure, task runtime estimates, etc., and, supposedly, are able to better predict future resource demand.

RQ4: What are the performance benefits of feedback mechanisms in online scheduling of workflows? Online scheduling of workloads of workflows may benefit from using a feedback loop on performance metrics for making scheduling decisions. Various performance metrics can be calculated on-the-fly during the workflow execution. For example, if task runtimes are not available, the task throughput and past CPU time consumption can be analyzed, and, if task runtimes are available, the slowdown of the already finished part of the workflow can be determined. These approaches have not been previously applied to online workflow scheduling.

1.8. Research Methods

The research presented in this dissertation was supported by the Infrastructure Virtualization for e-Science applications (IV-e) project of the national Dutch COMMIT program [8]. E-Science is a computationally intensive science that is carried out in highly distributed computing environments where workflows are commonly used. We choose scientific workflows described by the Pegasus project as the main workload for our study. These workflows fit nicely within the considered e-Science paradigm, are publicly available and well defined, and are widely used by the research community.

For studying workflow scheduling policies, we use both simulations and real-world experiments. With the task placement policies in Chapters 2 and 3, we use simulations, as, in contrast to real-world experiments, simulations allow for testing the considered policies with a much larger number of configurations, e.g., under different system utilizations. For this purpose, we have developed a custom simulator based on the DGSim simulator [84], which was created in the Distributed Systems group of the Delft University of Technology.

For studying the autoscaling policies in Chapters 4 and 5, we execute real-world experiments with an actual workflow management system. Sacrificing the number

of parameters that can be evaluated, real-world experiments usually produce more representative results, as they, for instance, incorporate the possible overheads caused by the involved software layers.

For the experiments in Chapter 4, we extend the KOALA grid scheduler [62], also developed in the Distributed Systems group of the Delft University of Technology, by adding support to it for the execution of workflows defined in the XML format. In Chapter 5, we extend the code of the Apache Airflow workflow management system [2] to incorporate resource allocation. To speed-up the experiments and test more system configurations, in Chapter 5 we emulate resource allocation while using the actual Airflow [2] workflow management system. By such emulation we mean that the workflow management system continuously runs a set of resources, while marking the resources as allocated or deallocated instead of starting them up or shutting them down.

We run all our experiments on the DAS-4 multicluster system [35]. Using this private system minimizes the effects of background load, which is common in public clouds [86]. For the cloud experiments in Chapter 4, we rely on the OpenNebula [123] cloud computing platform deployed on DAS-4. In Chapter 5, to validate the experimental results obtained from the Airflow system against the optimal solution, we use a Mixed Integer Programming (MIP) optimization model and implement it in the popular Gurobi solver [14].

1.9. Dissertation Outline and Contributions

In this section, we present the structure of the dissertation and our contributions as the answers to the four research questions stated in Section 1.7.

Scheduling Workloads of Workflows with Unknown Task Runtimes. In Chapter 2 we answer RQ1 by proposing a family of four novel online workflow scheduling policies. The proposed policies include a greedy backfilling policy, which schedules any eligible task of any workflow in the system, and three policies that employ different forms of processor reservation. The main distinguishing feature of the four scheduling policies we propose is to what extent they are greedy in scheduling any task of any workflow in the queue versus to what extent they reserve processors for workflows towards the head of the queue in order not to unduly delay these workflows. To be able to make processor reservations, we propose a method for the realistic estimation of workflow level of parallelism. We simulate a homogeneous computing system where we execute the proposed policies with synthetic workloads of realistic workflows. As main metrics we use the average workflow slowdown and the maximal utilization that can be achieved. Our results show that even at moderate imposed utilizations, the greedy backfilling policy achieves better performance compared to the policies which use processor reservation. This chapter is based on our publication:

Alexey Ilyuskin, Bogdan Ghiţ, Dick Epema, "Scheduling Workloads of Workflows with Unknown Task Runtimes", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015.

The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows. In Chapter 3 we answer RQ2 by scheduling workloads of workflows with varying accuracy of task runtime estimates that are available to the scheduler. We implement four state-of-the-art online dynamic workflow scheduling policies, and propose two novel online dynamic policies, one of which addresses fairness, and the other one is our adaptation of the popular offline plan-based HEFT policy to the online plan-based case. Additionally, we describe two methods for automatic validation of system stability. We simulate a heterogeneous computing system and use two workloads, where the first workload consists of realistic workflows and the second one consists of random workflows. In our results, we can clearly see that the knowledge of task runtime estimates gives significant performance improvement in the average job slowdown for the considered dynamic online policies, but only at extremely high utilizations. Our fairness-oriented policy effectively decreases the variance of job slowdown and thus achieves fairness. The adapted HEFT policy demonstrates poor performance compared to dynamic online policies and brings extra complexity to the scheduling process. We conclude that at moderate utilizations, simpler backfilling-based policies that do not use task runtime estimates show comparable performance to more advanced policies. This chapter is based on our publication:

Alexey Ilyuskin, Dick Epema, "The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows", *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2018 (best paper award nomination).

An Experimental Performance Evaluation of Autoscalers. In Chapter 4 we answer RQ3 by experimentally evaluating five state-of-the-art general autoscalers and two novel workflow-specific autoscalers. Moreover, we present and refine performance metrics endorsed by the Standard Performance Evaluation Corporation (SPEC) [73] for assessing autoscalers, and define three approaches for comparing the autoscalers using the considered metrics. The experimental results show that general autoscalers can demonstrate comparable performance to workflow-specific autoscalers, if the former have access to workload statistics. The workflow-specific autoscalers, in turn, require task or workflow runtime estimates. As many workflows have deadline requirements on the tasks, we additionally investigate the effect of autoscaling on meeting workflow deadlines. Besides that, we look into the effect of autoscaling on the accounted and hourly-based charged costs, and evaluate performance variability caused by the autoscaler for various groups of workflow sizes. Our results highlight the trade-offs between the suggested policies, their impact on meeting the deadlines, and their performance in different operating conditions. This chapter is based on our two publications:

Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro V. Papadopoulos, Bogdan Ghiț, Dick Epema, and Alexandru Iosup, "An Experimental Performance Evaluation of Autoscaling Policies for Complex Workflows", *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2017 (best paper award nomination).

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)*, Vol. 3, No. 2, 2018.

1

Performance-Feedback Autoscaling for Workloads Workflows. In Chapter 5 we answer RQ4 when applied to the autoscaling problem. We compare two state-of-the-art online plan-based autoscalers with our novel online dynamic autoscaler. To make autoscaling decisions, this novel autoscaler analyzes historical task throughput and uses the not yet finished part of the workflow, instead of relying on task runtime estimates, as observing task throughput normally requires less effort than obtaining task runtime estimates. We emulate a heterogeneous cloud system with two independent users and use two workloads with different task runtime distributions. Overall, our approach shows lower time complexity and effectively minimizes workflow slowdowns compared to the state-of-the-art autoscalers. The usage of historical throughput information provides fairly accurate estimation of resource speeds when dealing with workloads with a long-tailed task runtime distribution. We additionally validate the experimental results by comparing the workflow slowdowns obtained from an actual workflow management system with an optimal solution obtained from solving the problem described as a mixed integer programming model. This chapter is based on our publication:

Alexey Ilyushkin, André Bauer, Alessandro V. Papadopoulos, Ewa Deelman, and Alexandru Iosup, "Performance-Feedback Autoscaling with Budget Constraints for Cloud-based Workloads of Workflows" (under review).

Conclusions. In the final Chapter 6 we summarize the main findings of this dissertation and provide suggestions for future work.

2

Scheduling with Unknown Task Runtimes

Workflows are important computational tools in many branches of science, and because of the dependencies among their tasks and their widely different characteristics, scheduling them is a difficult problem. Most research on scheduling workflows has focused on the offline problem of minimizing the makespan of single workflows with known task runtimes. The problem of scheduling multiple workflows has been addressed either in an offline fashion, or still with the assumption of known task runtimes. In this chapter, we study the problem of scheduling workloads consisting of an arrival stream of workflows without task runtime estimates. The resource requirements of a workflow can significantly fluctuate during its execution. Thus, we present four scheduling policies for workloads of workflows with as their main feature the extent to which they reserve processors to workflows to deal with these fluctuations. We perform simulations with realistic synthetic workloads and we show that any form of processor reservation only decreases the overall system performance and that a greedy backfilling-like policy performs best.

2.1. Introduction

Workflows are widely used for all kinds of computational and data analysis problems in many branches of science such as astronomy and bioinformatics. Because of the dependencies among their tasks and because of the diversity of their structures, sizes, and task runtimes, scheduling workflows efficiently on clusters and datacenters is a difficult problem. Most research on scheduling workflows focuses on the offline problem of minimizing the makespan of single workflows for which estimates of the task runtimes are known. In contrast, in this chapter we propose four scheduling policies for online scheduling workloads consisting of arriving workflows with unknown task runtimes, and we perform simulations to evaluate their performance.

The problem of minimizing the makespan of single workflows has been studied very extensively, usually assuming that the task runtimes are known. Well-known

approaches for task selection when resources become available are the HEFT policy [155] that schedules each workflow task on the processor that minimizes its finish time, and policies that try to optimize the schedule of the critical path of a workflow [100, 138, 23]. Scheduling multiple workflows has received some attention in the past. However, some of this work still considers the offline problem of minimizing the total makespan of a fixed set of workflows, without or with the added requirement of fairness [170]. Other work does consider the online problem with an arrival stream of workflows but translates the problem into scheduling and executing complete batches of workflows before considering later arrivals [76], or builds a single DAG from the DAGs representing a set of workflows [169]. In this chapter, we study the online problem of scheduling an arrival stream of workflows, which in addition to the problem of task selection involves the problem of workflow selection from which to pick tasks to run.

The workflows that are used in practice are still growing in size, complexity, as well as in the number of dependencies among their tasks [38, 92, 152]. Even though workflows are popular as an automation tool for e-Science experiments [132], obviously, the workloads of real clusters consist of jobs of different types in addition to workflows, such as parallel applications and bags-of-tasks. When workflows are run very often, either the user or the system may be able to derive reasonable estimates of the runtimes of tasks of workflows.

However, in this chapter, we take a step back and we address the fundamental research question: *What are appropriate policies for online scheduling workflows without having knowledge of task runtimes, and what is their performance in terms of the job slowdown as a function of the system utilization, and of the maximal utilization?*

When scheduling workflows from the queue of workflows that have been submitted but that have not yet completed, resources may be available while the workflows towards the head of the queue may not have tasks that are eligible to run. Thus, the main distinguishing feature of the four scheduling policies we propose is to what extent they are greedy in scheduling any task of any workflow in the queue versus to what extent they reserve processors for workflows towards the head of the queue in order not to unduly delay these workflows. Our policies range from a very strict reservation-based policy that guarantees no delay to the workflow at the head of the queue due to later workflows, to a greedy backfilling policy. The jobs for the simulations we generate based on real scientific workloads [38, 92, 132, 152]. In our performance evaluation we report the average job slowdown of the workflows and the maximal utilization—because workflow scheduling is not work-conserving due to the precedence constraints among workflow tasks, the system may become saturated for utilizations well below 1.0. For the implementation of the system model and the proposed scheduling policies we use an improved version of the DGSim discrete event simulator [85].

This chapter is organized as follows. In Section 2.2 we provide our problem statement. Section 2.3 motivates and presents our scheduling policies. Section 2.4 discusses our experimental setup and characterizes the synthetic workloads. In Section 2.5 we show and explain the obtained results. Section 2.6 contains a survey of related work. Finally, in Section 2.7 we present our conclusions.

2.2. Problem Statement

This section presents our model for the problem of scheduling workloads of workflows with unknown task runtime estimates and the performance metrics.

2.2.1. The Model

We consider large-scale computing systems such as large clusters and datacenters that are subject to an arrival stream of workflows. We only consider processors as the type of system resources that can be controlled by the scheduler. Every computing node in the system contains only one processor. Furthermore, we assume that the system is homogeneous, with identical processors and communication links between them. Since we focus on the computational properties of workflows, we assume that the data transfer times between computing nodes in our simulated system can be neglected. This is equivalent to the situation in a real system when all the tasks of a workflow write their results to a shared storage so that all the required data are immediately available for any workflow task when all of its dependencies are satisfied.

In this chapter we assume that each workflow task requires only one processor. All the considered workflow structures have a single entry node and a single exit node. We guarantee this by adding, if necessary, one or two artificial nodes with zero runtime.

Many workflow scheduling algorithms employ user estimates or predictions of task runtimes. It is well known that the estimates provided by users are usually quite inaccurate [124]. At the same time, the runtime prediction approaches are often relatively complex, do not work well for some situations, and can also be inaccurate. Finally, always new or unknown workflows can be submitted to systems. For all of these reasons, we suppose that the runtimes of the tasks of the workflows are unknown to the scheduler.

2.2.2. Performance Metrics

In order to compare the implemented scheduling policies, we define a set of metrics and baselines. Scheduling workflows is in itself not work-conserving as there may be idle processors in the system while there is no waiting task with all its dependencies satisfied. In addition, a property of several of our policies is that they reserve processors to workflows in order to deal with their fluctuating resource requirements. As a consequence, policies scheduling workloads of workflows may not be able to drive a system up to a utilization of 1.0. Therefore, we use the maximal utilization, which is defined in Section 1.1, as a system-oriented metric to assess the performance of workflow scheduling policies.

As a user-oriented metric to assess workflow scheduling policies we use the (average) slowdown, which is defined in steps in the following way:

- The *wait time* t_w of a workflow is the time between its arrival and the start of its first task.
- The *makespan* t_m of a workflow is the time between the start of its first task until the completion of its last task.

- The *response time* t_r of a workflow is the sum of its wait time and its makespan: $t_r = t_w + t_m$.
- The *slowdown* s of a workflow is its response time (in a busy system, when the workflow runs simultaneously with other workflows) normalized by its makespan t'_m in an empty system of the same size (when the workflow has exclusive access to all the processors): $s = t_r \,/\, t'_m$.

We additionally define the *execution time* t_e of a workflow which is the sum of the runtimes of all of the workflow tasks. The t_e metric is mostly useful for analyzing the workloads.

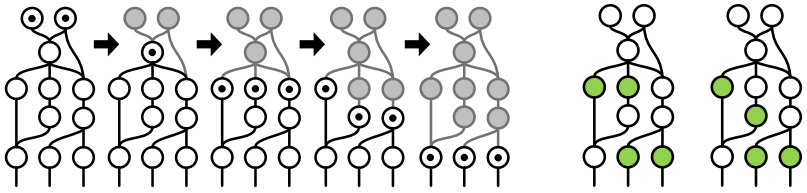
2.3. Scheduling Policies

In this section, we will describe the four policies for scheduling workloads consisting of workflows the simulation results of which we will show later in the chapter. Before doing so, we will present a novel method for calculating LoP with some relevant concepts, and will describe the way the scheduler manages the queue of waiting workflows.

2.3.1. Calculating the Level of Parallelism

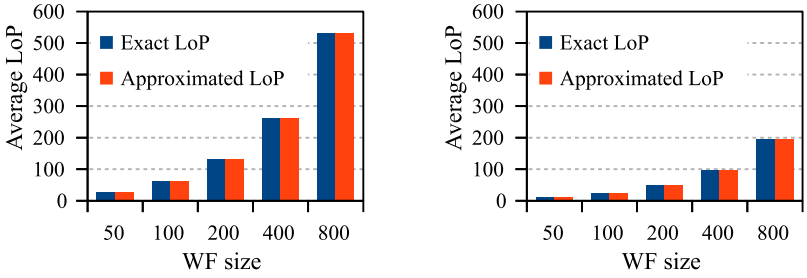
In some of our scheduling policies, we will use the LoP of the remaining part of a workflow consisting of the tasks that have not yet completed for deciding how many processors to reserve for it. Then, whenever a task of a workflow completes, the LoP of the remaining part of the workflow has to be recomputed, leading to a very large number of LoP recomputations. The DAG representing the non-completed part of a workflow is a sub-DAG of the original workflow DAG. Such a sub-DAG can have multiple entry nodes but will always have a single exit node. We say that a node a precedes (follows) node b in a DAG when there is a path of precedence constraints from a (b) to b (a). Two nodes a and b are said to be comparable (incomparable) if one (none) of them precedes or follows the other. With the "comparable" relation, a DAG can be considered as a *partially ordered set* (*poset*). The value of the LoP is equal to the width of this poset, which is defined as the cardinality of the maximum set of incomparable elements in the poset. There exist multiple algorithms [79] to compute the exact LoP using Dilworth's chain partition of the original DAG, but they require the creation of an additional comparability graph or a bipartite graph. Dilworth's theorem [58] represents the width of a poset as a partition of the poset into a minimum number of *chains*, where each *chain* is a path from a source to a sink in the directed comparability graph.

To avoid the construction of any auxiliary data structures, we will use a simple LoP approximation algorithm that calculates the LoP in an adequate time even for large workflow structures, and, as we will show, it does so with only a relatively small amount of underestimation of the actual LoP for many well-known workflow structures. Our approximation algorithm uses the size of the largest generation-i eligible set as the value of the LoP. To find the size of this set the algorithm employs tokens to simulate an execution "wave" in a DAG. Initially, the algorithm places tokens in the entry nodes of the DAG. Then in successive steps it moves these tokens to all the nodes all of whose parents already hold a token or were earlier

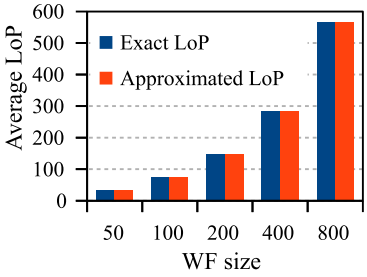


(a) Token propagation steps. (b) Two real LoPs.

Figure 2.1: An example of LoP approximation (a) versus the exact LoP (b).



(a) Montage. (b) LIGO.



(c) SIPHT.

Figure 2.2: A comparison of the exact and the approximation method for calculating the LoP for different workflow structures.

tokenized, until the exit node gets a token, as shown in Figure 2.1a. After each such step, the set of tokenized nodes is recorded. At the end of the algorithm, the size of the largest recorded tokenized set is the approximated LoP value. Note that in fact, these tokenized sets coincide with the eligible sets of different generations as defined in Section 1.4.

We have compared the results provided by this approximation algorithm with the exact LoPs for different workflow sizes for five popular workflow structures, LIGO, SIPHT, Montage, Cybershake, and Epigenomics. For each considered size of each workflow type, we generate 50 DAGs using the synthetic workflow generator by Bharathi et al. [16]. As can be seen from Figure 2.2, our method approximates the true LoP value extremely well. We provide the LoPs here only for LIGO, SIPHT,

and Montage, since Cybershake and Epigenomics show similar results. For both methods, the LoP values obtained for the 50 DAGs of each workflow type are all very close to the mean. Of course, there can be situations where our approach does underestimate the LoP, see for instance the simple example in Figure 2.1. However, since the algorithm works well for the selected workflows, which are quite popular and representative, we will use the LoPs computed by it when simulating policies that use LoP in their scheduling decisions.

From the results in Figure 2.2, we can conclude that the tested workflows have rather regular, but still different, structures. First of all, for all three workflow structures, the (average) LoP increases superlinearly (and especially for Montage and SIPHT, in a very strong way). Secondly, even for LIGO, but especially for the other two, the LoP is very large in relation to their total size. For LIGO, the LoP is slightly less than 200 for a workflow size of 800, but Montage and SIPHT already reach that LoP for workflows of size less than 400.

2.3.2. Queue Management and Task Selection

We assume that the scheduler maintains a single queue of waiting workflows. Every arriving workflow is appended to the tail of the queue, and the scheduler decides which tasks of which workflows in the queue are scheduled when resources become available. For all the policies we will consider, the scheduler is invoked when a task of a workflow completes, or when a new workflow arrives (possibly to a non-empty queue). The workflows are in principle processed in the order of their arrival, but multiple workflows can be partially in execution while the remainder of their tasks are still waiting (either for a lack of resources or because of precedence constraints between tasks), in which case we still regard them to be present in the queue. A workflow only leaves the queue when all of its tasks are finished.

When the scheduler is activated, it selects the workflows in the queue from which tasks are scheduled in a way dictated by the actual scheduling policy. As we assume that all workflow tasks require only one processor and that we do not have user estimates or predictions of task runtimes, after the scheduler has selected a workflow from which to start a task, it picks a task from the workflow's eligible set randomly. As the execution order of the tasks from the current eligible set may influence subsequent eligible sets, and so, the makespan of the workflow, this random task selection may not lead to the optimal schedule for the workflow, but without knowledge of task runtimes it is difficult to do better. Giving priority to tasks in the eligible set that would enable large numbers of other tasks might improve the makespan of a workflow, but this is highly dependent on the task runtimes.

2.3.3. The Strict Reservation Policy

The most classic and simple general queuing policy is FCFS. When scheduling workflows, the definition of FCFS is not completely straightforward. The idea behind our version of the FCFS policy for workflows can be summarized by the condition that the service to a workflow will only be influenced by the workflows ahead of it in the queue and never by workflows behind it in the queue. We enforce this condition by strictly reserving for any workflow in the queue sufficient resources

so that it will never be delayed by any later workflow—hence the alternative name is Strict Reservation (SR) policy.

With the SR policy, when the scheduler is invoked and the workflow at the head of the queue has fewer processors allocated to it than its LoP (which may be in use or idle), it allocates additional available processors to the workflow at the head of the queue until the workflow has LoP processors or until there are no more available processors. When at a later time the scheduler is invoked again while the workflow at the head of the queue has not yet been completed, the scheduler recomputes workflow's remaining LoP, keeps LoP processors allocated to the workflow, and releases any excessive processors. If any idle processors remain after LoP processors have been allocated to the workflow, the scheduler tries to schedule the next workflow in the queue—it does so in exactly the same way as if that workflow were at the head of the queue.

2.3.4. The Scaled LoP Policy

Of course, workflows may never attain their LoPs, and especially for large workflows, the SR policy may be very wasteful and lead to a low maximal utilization. Whereas the SR policy executes a workflow in the shortest possible time once it starts allocating processors to it, the wait times of the workflows with SR may be excessive. A straightforward solution to this problem is to reduce the reservation of a workflow to a number of processors that is lower than its LoP. Thus, the Scaled LoP (SLoP) policy with *scaling factor* $f, 0 \leq f \leq 1$, tries at all times to keep $f \cdot \text{LoP}$ processors allocated to a workflow. This means that if at some point the size of the eligible set of a workflow is smaller than $f \cdot \text{LoP}$, the scheduler will try to keep reserved a number of processors equal to the difference between these two values. If, however, at some point the size of the eligible set of a workflow exceeds $f \cdot \text{LoP}$, the SLoP policy will allocate any available processors to eligible tasks of the workflow. The SLoP policy behaves similar to the SR policy albeit with a lower reservation target, and in the boundary case when $f = 1$ it is equal to it. At the other extreme, as we will see below, our backfilling policy is in fact identical to the SLoP policy with scaling factor equal to 0.

2.3.5. The Future Eligible Sets Policy

The idea behind reserving processors for workflows is to reduce the delay in placing tasks in the eligible set. Rather than, when reserving processors, taking a worst-case perspective on the number of processors a workflow may ever need as we did in the SR policy, we may also try to look into the future of the execution of the workflows. Thus, the Future Eligible Sets (FES) policy with *depth* n tries to allocate to a workflow a number of processors that is equal to the size of largest eligible set of any generation from 0 through n. With our approximation of computing the LoP as presented in Section 2.3.1, the FES policy with depth ∞ coincides with the SR policy. At the other extreme, as we will see below, our backfilling policy is in fact identical to the FES policy with depth equal to 0.

2.3.6. The Backfilling Policy

Depending on the scaling factor and the depth, the SLoP and FES policies may be very wasteful of resources because of useless reservations. To completely do away with reservations and the waste of resources it entails, we now define the Backfilling (BF) policy that tries to allocate to any workflow any number of processors up to the size of its current eligible set. Thus, at every invocation, the scheduler scans the queue from head to tail and from each workflow it encounters it places as many tasks from the corresponding eligible set as it can. Thus, the BF policy is greedy as it allows to schedule any task from any eligible set of any workflow in the queue. As we assume that each task requires only one processor, there will only be idle processors in the system when all tasks in the eligible sets of all workflows in the queue are actually running. As already remarked above, the BF policy is a special case of the SLoP policy with scaling factor $f = 0$.

Our BF policy for workflows is similar to the backfilling policies that have been introduced for parallel jobs [149, 124], but our policy does not use runtime estimates. However, even so, with our BF policy for workflows and because of our assumption that each task requires one processor, starvation is intrinsically impossible—we always try to schedule as many tasks of a workflow as possible before considering the next workflow, thus always granting at least some resources to a workflow before allowing workflows later in the queue to receive resources. In contrast, unless special measures are taken, starvation is possible when backfilling parallel jobs (and when backfilling workflows with parallel tasks).

Compared to the SR policy, with BF, on the one hand workflows can be delayed by later workflows, thus increasing the makespans and so the job slowdowns. On the other hand, BF allows the workflows in the queue to start their execution earlier, thus decreasing the wait times and so the job slowdowns. From our evaluation we will see which of these two effects is stronger.

2.4. Experiment Setup

In this section, we present our simulation environment and we characterise the synthetic workloads we use to analyse the performance of our scheduling policies.

We have modified the DGSim simulator [84, 85] for cluster and grid systems to include our scheduling policies. The only resource modeled in our simulations is the processors. We assume that the workflows submitted to the simulated cluster arrive according to a Poisson process. The size of the homogeneous cluster we use in all of our simulations is 100. For our simulations, we select three representative types of workflows from different application domains, i.e., astronomy (Montage [90, 152, 38, 92]), physics (LIGO [139, 152, 38, 92]), and bioinformatics (SIPHT [108, 38, 92]). Montage builds mosaic images of the sky obtained from different telescopes. LIGO is used to process data from detectors of the Laser Interferometer Gravitational Wave Observatory (LIGO) [22] and its mission is to detect gravitational waves predicted by general relativity. SIPHT helps to search for small untranslated bacterial regulatory RNAs.

In Figure 1.4, we show the structure of the DAGs of the three workflow types. Montage has the most complicated structure and its size is determined by the number of processed images. A LIGO workflow usually consists of many smaller

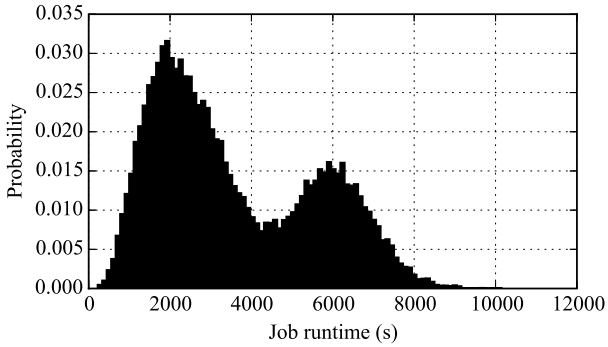


Figure 2.3: The distribution of the total workflow execution times in the workloads.

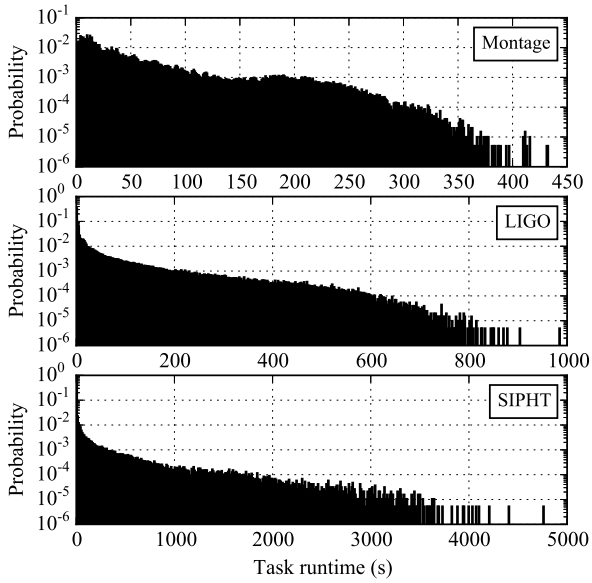


Figure 2.4: The distribution of the task runtimes for each workflow type (the horizontal axes have different scales, and the vertical axes are in log scale).

workflows combined into a single workflow. Similarly to LIGO, the SIPHT workflow combines smaller independent workflows, but with very similar structures. The workflow types are diverse not only in the structure of their DAGs, but also with respect to the processing requirements of the component tasks as we will see below. Furthermore, we have already analysed the maximum levels of parallelism they can achieve in Section 2.3.1.

We generate four workloads of 3,000 workflows each using the workflow generator [16] presented by Bharathi et al. [38]: one workload per workflow type, and an additional workload that mixes equal fractions of the three types. As with many other workloads in computer systems, in practice, workflows are usually small, but very large ones may exist too [132]. Therefore, in our simulations we

distinguish small, medium, and large workflows, which constitute fractions of 75%, 20%, and 5% of the workloads. We assume all workflows to have even sizes due to the limitations of the generator. The size of the small, the medium, and the large workflows is uniformly distributed on the intervals [30, 38], [40, 198], and [200, 600], respectively.

In order to obtain simulation results for the different workflow types that can easily be compared (especially when simulating the mixed workload), we use the same total execution time distribution for all three workflow types. This distribution is a two-stage hyper-Gamma distribution derived from the model presented by Lublin and Feitelson [112]. The shape and scale parameters (α, β) of two component Gamma distributions are set to (5.0, 501.266) and (45.0, 136.709), respectively. Their proportions in the overall distribution are 0.7 and 0.3. The average total execution time is one hour. Figure 2.3 visualizes this distribution. For every workflow, we normalize the task runtimes generated so that its total processing requirement is equal to the corresponding sample of the execution time distribution. In Figure 2.4, we show the distributions of the normalized task runtimes for each workflow type. The maximum task runtimes in Montage and LIGO are an order of magnitude smaller than the maximum task runtimes in SIPHT, but all three workflow types share the phenomenon that they are dominated by short tasks.

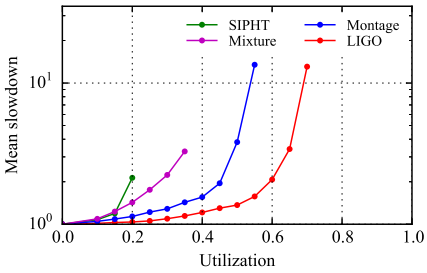
In our simulations, we vary the utilization starting at 0.05 with step size 0.05. If for some utilization the system did not become stable in the simulations, we will only show performance results up to that utilization. We will consider the highest utilization for which the system did become stable as the maximal utilization, defined in Section 1.1. We set the scaling factor f in our SLoP policy to 0.2, 0.8, and 0.9, and we evaluate the FES policy with depths 1, 2, and 10. For each experiment, we report the average job slowdown over three repetitions, and we only show results when the system is in steady state. Thus, when reporting performance results, to be on the safe side, we omit the performance information for the first 1,000 workflows in each simulation and for those workflows that have not completed their execution before the start of the last arriving workflow.

2.5. Experiment Results

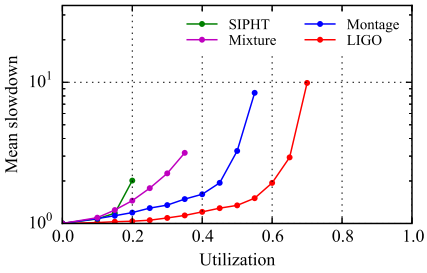
In this section we report our simulation results and their analysis. In Figure 2.5, we show for all the policies and for all four workload types the mean workflow slowdown as it depends on the utilization in the system. The last point in all curves is for the highest utilization for which the system is stable in the simulations. In Table 2.1, we summarize these maximal utilizations.

As a first general observation, we find that for all policies that do some form of reservation, the maximal utilization is low, or even very low. The worst is a maximal utilization of only 0.2 for the SR policy with the SIPHT workload. Apparently, reserved resources often remain idle, and the benefit of reservations for a short makespan do not balance their negative effect of having long wait times.

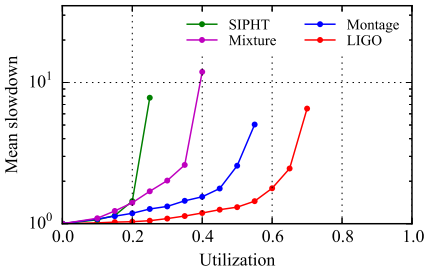
Our second general observation is that the performance both in terms of mean slowdown and maximal utilization varies considerably across the three workloads consisting of a single workflow type. However, the different workflow types almost always have the same relative performance. In particular, for all the policies except



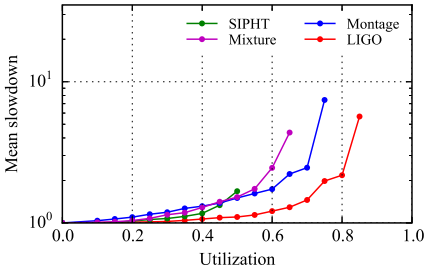
(a) SR.



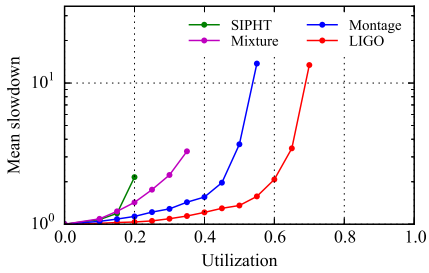
(b) SLoP, $f = 0.9$.



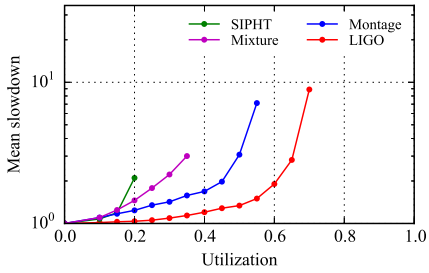
(c) SLoP, $f = 0.8$.



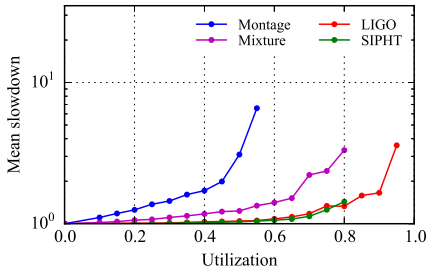
(d) SLoP, $f = 0.2$.



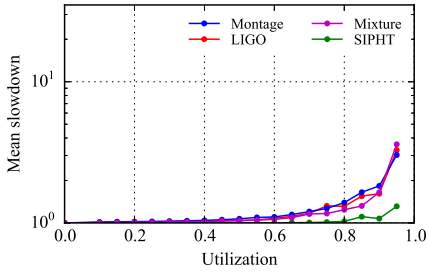
(e) FES, depth 10.



(f) FES, depth 2.



(g) FES, depth 1.



(h) BF.

Figure 2.5: The mean slowdown of workflows as a function of the utilization for the different policies and for each of the four workload types (the vertical axis is in log scale).

	Workload type			
Policy	Montage	LIGO	SIPHT	Mixture
SR	0.55	0.70	0.20	0.35
SLoP, $f = 0.9$	0.55	0.70	0.20	0.35
SLoP, $f = 0.8$	0.55	0.70	0.25	0.40
SLoP, $f = 0.2$	0.75	0.85	0.50	0.65
FES, depth 10	0.55	0.70	0.20	0.35
FES, depth 2	0.55	0.70	0.20	0.35
FES, depth 1	0.55	0.95	0.80	0.80
BF	0.95	0.95	0.95	0.95

Table 2.1: The maximal utilizations for the considered scheduling policies.

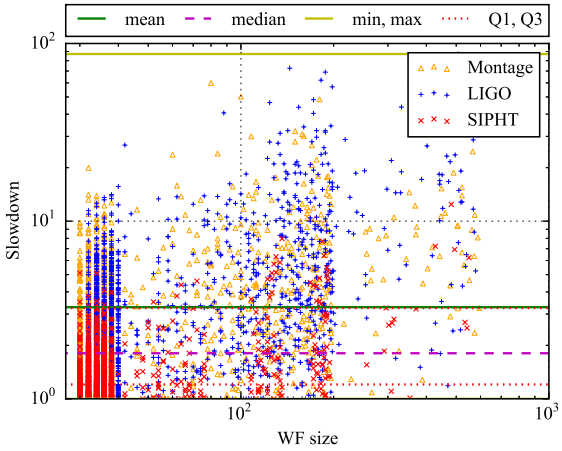
FES with depth 1 and BF, LIGO performs the best and SIPHT performs the poorest. This can be explained from the perspective that the number of reserved processors that is actually used is higher for LIGO than for the other workflow types. In contrast, SIPHT and Montage use smaller fractions of the reserved processors.

Third, as can be expected, the performance for the mixture workload is always somewhere in between the performance of the pure workloads. Except for the cases with the FES policy with depth 1 and the BF policy, it has a very low maximal utilization that is well below the maximal utilizations for the Montage and LIGO workloads.

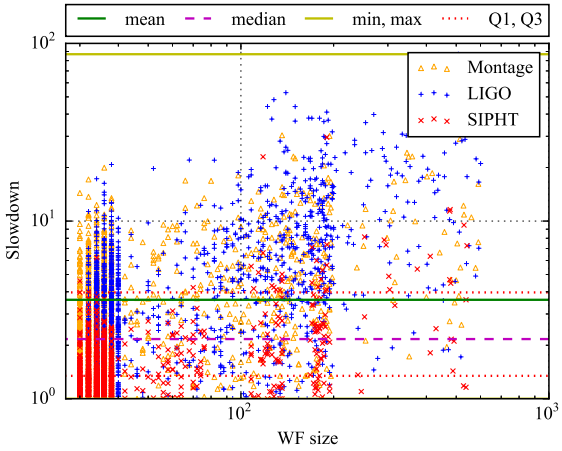
As to the performance of the SR and SLoP policies, decreasing the scaling factor f improves the performance as the curves for all workloads move to the right in the plots when going down from Figure 2.5a to Figure 2.5d. Going from a scaling factor of 1.0 (SR) through 0.9 and 0.8 to 0.2, the maximal utilization for the mixture workload increases from 0.35 (for scaling factors of 1.0 and 0.9) through 0.4 (for a scaling factor of 0.8) to 0.65 (for 0.2). Apparently, when decreasing the scaling factor, the SLoP policy decreases the number of idle but reserved processors.

Our experiments with the FES policy show that increasing the depth decreases the performance because the scheduler reserves ever more processors to each workflow. As Figure 2.5f shows, with a depth of 2 the FES policy already starts to behave like SR, and with a depth of 10 it has almost identical performance to SR. For a depth of 1 the FES policy goes closer to the BF policy, and an interesting effect can be observed. Whereas SIPHT for most policies exhibits the poorest performance, with FES with depth 1 it suddenly achieves a maximal utilization of 0.8. It means that with a depth of 1 the reservation size for SIPHT sharply drops. In contrast, Montage has the same maximal utilization with FES for depths 1, 2, and 10. The reason for this is the structure of the Montage workflow, which causes the scheduler to reserve quite a large number of processors even with a depth of 1.

Finally, the BF policy (which is identical to SLoP with a scaling factor of 0.0 and to FES with depth equal to 0) shows by far the best performance results. As can be seen from Figure 2.5h, it treats all the workflow types almost equally, and it is even stable at a utilization 0.95. Apparently, at extremely high utilizations there are always eligible waiting tasks to be found in the queue to start when a task finishes. Overall, we can conclude from Figure 2.5 that the SR policy is the



(a) SR at a utilization of 0.35.



(b) BF at a utilization of 0.95.

Figure 2.6: Scatter plots of the slowdowns versus the sizes of all workflows in the simulations for the mixed workload with the SR and BF policies at their maximal utilizations (both axes are in log scale).

worst and the BF policy is the best among our policies—reserving processors for workflows with workloads consisting solely of workflows is not a good idea!

In Figure 2.6, we show scatter plots with the slowdowns versus the sizes of all workflows in the simulations for the mixed workload with the SR and BF policies at their maximal utilizations. We show these plots for these two cases as they are the policies that are at the extremes of the spectrum of policies we consider. The most striking thing about these plots is not that they are very different, because in fact, they are not. The striking thing is that they are almost identical at such widely different utilizations (0.35 versus 0.95), exhibiting the huge advantage of using the BF policy. The variation of the density of dots in the horizontal direction is caused

	Mean wait time (s)			Mean makespan (s)		
Policy	S	M	L	S	M	L
Empty system	—	—	—	1136	429	130
SR	636	628	658	1162	484	263
SLoP, $f = 0.9$	584	578	645	1188	492	249
SLoP, $f = 0.8$	3195	3121	3329	1174	546	295
SLoP, $f = 0.2$	879	892	938	1291	597	297
FES, depth 10	637	632	652	1160	480	264
FES, depth 2	526	525	547	1189	503	278
FES, depth 1	596	631	581	1191	530	273
BF	703	707	694	1199	546	271

Table 2.2: The mean wait time and the mean makespan for the small (S), medium (M) and large (L) workflows with the mixed workload for the considered scheduling policies at their maximal utilizations, and in an empty system.

by the job size distribution in the used workloads as described in Section 2.4, with many small jobs of sizes below 40, and much smaller number of medium-sized jobs with sizes between 40 and 200, and a still smaller number of jobs with sizes between 200 and 600. Since we only generated even workflow sizes, it also explains the columns for small workflow sizes. None of our policies takes into account the size of a workflow when processing the queue. Still, in Figure 2.6 there are somewhat more outliers with slowdowns over 20 among the larger jobs than among the smaller ones, although the difference is not really significant. Among the outliers, most workflows are of the type LIGO, which has the smallest LoP; there are hardly any outliers of type SIPHT.

In Table 2.2, we present the mean wait time and the mean makespan for small, medium, and large workflows as defined in Section 2.4 with the mixed workload for all our policies at their maximal utilizations, and in an empty system. As expected, for every policy separately, the mean wait time does not depend on the workflow size. The longer mean wait times for the SLoP policy with scaling factor f equal to 0.8 and 0.2 are explained by the step size of 0.05 we use to vary the utilization in our experiments—with a smaller step size to detect the maximal utilization all the mean wait times in Table 2.2 would be in the same range. For all policies, the values of the makespans of three groups of workflows are relatively similar. This can be explained from the perspective that when the system is close to its maximal utilization, then despite the length of the queue, the scheduler considers only some set of workflows that are close to the head of the queue. The size of this set is related to the size of the system: the larger the system, the deeper the scheduler should inspect the queue for eligible workflow tasks. Another interesting observation is that the larger the workflows, the shorter their makespans. This effect is explained by our usage of the same total execution time distribution for all the workflow sizes, and by the fact that smaller workflows have lower levels of parallelism.

Furthermore, while for small and medium workflows the makespans at the maximal utilizations of the policies are almost equal to those in an empty system, large workflows then have makespans that are twice as large. Apparently, even in a

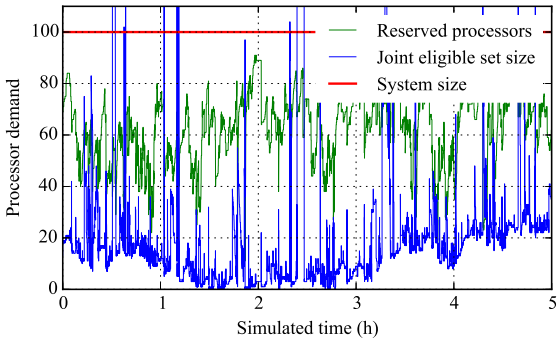


Figure 2.7: The dynamics of the joint eligible set size and the number of reserved processors during the simulation of the SR policy at a utilization of 0.35 during a 5-hour period.

busy system small and medium workflows can get a number of processors close to their LoPs, while large workflows suffer more from the presence of other workflows in the system.

Finally, we investigate in more detail how reservations limit the maximal utilization. Figure 2.7 shows the size of the joint eligible set, which comprises the tasks of the eligible sets of all workflows in the queue, and the number of reserved processors for the SR policy at its maximal utilization during a 5-hour period of the simulation. Obviously, the size of the joint eligible set can exceed the size of the system (in Figure 2.7, we crop these outliers higher than size 110) and is related to the queue size and the properties of workflows in the queue. The minimum of the two curves gives the number of wasted processors. Apparently, with SR, when the utilization is equal to 0.35, the system capacity spent on reservations approaches 0.65, and the system is saturated.

2.6. Related Work

There is an enormous amount of literature on the problem of scheduling workflows. Most of it concentrates on scheduling single workflows in order to minimize the makespan, for which many techniques have been proposed. For instance, the HEFT policy [155] computes the upward rank of a task as the length of the critical path from that task to the exit task in terms of the time it takes to process the tasks on the path, and schedules the task with the highest upward rank. Previously, Abrishami at el. [23] presented and analyzed a scheduling algorithm for workflows that recursively schedules partial critical paths. There are also several decent overview papers that present and discuss many algorithms for scheduling workflows [102, 101]. Invariably, all of these algorithms assume knowledge of task runtimes.

The problem of scheduling multiple workflows can be split up in the offline problem of scheduling a fixed set of workflows and the problem of online scheduling an arrival stream of workflows as we do in this chapter. Two approaches to the offline problem are executing batches with mixtures of multiple workflows [76], and building a single composite workflow from multiple workflows and then execute

it [170]. In the paper by Yu et al. [169], an online stream of workflows is considered, but the authors use a DAG composition approach and task runtime information to prioritize the tasks using HEFT. Also in the paper by Hsu et al. [77], a stream of arriving workflows is used and the ideas from the Yu's paper [169] are extended by considering the critical path for each workflow. In the work by Lee et al. [104], the Pegasus planner [54] and the DAGMan [67] batch workflow executor are used. In addition, a scheduling algorithm that allocates resources based on their runtime performance is proposed and real-world experiments are conducted using grid middleware over clusters. Despite the fact that they treat each workflow separately without composing a single DAG, their algorithm still uses task runtime information. In the paper by Singh and Deelman [145], a trace of a Teragrid cluster with applications representing (modified) Montage workflows is simulated. Different provisioning policies and priority schemes are considered, with a cap on the amount of resources that can be used by a single workflow.

A large body of work proposes backfilling as the main technique to improve the system utilization and reduce the job slowdowns in the area of parallel applications [124, 107, 141, 144, 89]. Despite the simplicity of the backfilling technique, there exist many variations of the algorithm. For instance, the number of reservations granted by the backfilling algorithm distinguishes two main strategies, conservative and aggressive. The former assigns to each job a reservation when it enters the system and moves smaller jobs forward in the queue as long as no delays are incurred on any of the previously queued jobs. The latter allows any job to be backfilled as long as it does not delay the first job in the queue. The number of queued jobs considered by the backfilling algorithm may have a significant impact on the overall performance. To maximize the utilization, dynamic programming may be employed to find the optimal packing of the jobs [144]. These aspects have been analyzed and incorporated in the Maui project which currently provides a real-world implementation of the general backfill algorithm [89]. The main feature distinguishing backfilling for parallel jobs and workflows is that with workflows, as soon as any number, however low, of resources are available, a workflow can make progress. So the concerns about not delaying workflows with backfilling are much less pressing.

2.7. Conclusion

In this chapter, we have presented a family of four policies for scheduling workloads consisting of arrival streams of workflows with unknown task runtimes. The main distinguishing feature of these policies is to what extent they reserve processors to workflows towards the head of the queue to deal with fluctuations in their level of parallelism. We have simulated these four policies in a cluster with synthetic workloads derived from popular real workflows with as metrics the average workflow slowdown and the maximal utilization that can be achieved. Our main conclusion is that any form of processor reservation for workflows without runtime estimates only decreases the overall system performance, leading to low or even to very low maximal utilizations.

3

The Impact of Task Runtime Estimate Accuracy

Workflow schedulers often rely on task runtime estimates when making scheduling decisions, and they usually target the scheduling of single workflows or batches of workflows. In contrast, in this chapter, we evaluate the impact of the absence or limited accuracy of task runtime estimates on slowdown when scheduling complete workloads of workflows that arrive over time. We study a total of seven policies: four of these are popular existing scheduling policies for (batches of) workloads from the literature, including a simple backfilling policy which is not aware of task runtime estimates, two are novel workload-oriented policies, including one which targets fairness, and one is the well-known HEFT scheduling policy for a single workflow adapted to the online workload scenario. We simulate homogeneous and heterogeneous distributed systems to evaluate the performance of these policies under varying accuracy of task runtime estimates. Our results show that for high utilizations, the order in which workflows are processed is more important than the knowledge of correct task runtime estimates. Under low utilizations, all policies considered show good results, even a policy which does not use task runtime estimates. We also show that our Fair Workflow Prioritization (FWP) policy effectively decreases the variance of job slowdown and thus achieves fairness, and that the plan-based scheduling policy derived from HEFT does not show much performance improvement while bringing extra complexity to the scheduling process.

3.1. Introduction

In workloads of modern computing systems, workflows are often used as a tool to drive complex computations, and their popularity continues to increase [52]. Many of these workflows are usually submitted to the system repeatedly so that (statistical) runtime estimates of their tasks can be derived [92, 135]; alternatively, runtime estimates can be provided by users [76]. However, the accuracy of runtime

estimates significantly depends on the employed estimation algorithm, or on the user—user runtime estimates can be very unreliable [63, 103]. Most previous work on scheduling workflows [155, 169, 29, 32] has assumed some (and often even perfect) knowledge of task runtimes. Moreover, often has been considered the *offline* problem of scheduling a single workflow or a batch of workflows (which are all initially present), or periodic submissions with a fixed interval. In most cases, the makespan has been used as the main metric.

However, workflows may be submitted to a system over time according to some arrival pattern, in which case job slowdown is a much more appropriate performance metric. Then, especially when workflows of widely different sizes are submitted, fairness becomes an issue, and an important goal is to reduce the variability of job slowdown. In this chapter, we investigate how the accuracy of task runtime estimates affects the quality of scheduling, and we address the issue of fairness in the *online* case of scheduling complete workloads of workflows. Moreover, we evaluate the system stability to know at which workflow arrival rates the system starts to uncontrollably accumulate waiting workflows. Besides that, we identify the maximal achievable system utilization which guarantees the stability.

We distinguish *dynamic* and *plan-based* policies. Dynamic policies make task placement decisions just-in-time when a processor becomes idle or a new task becomes eligible. Plan-based policies construct a full-ahead plan on every workflow arrival and strictly follow this plan to perform task placements between the workflow arrivals.

Task runtime estimates have been heavily used for different forms of task prioritization by various scheduling algorithms for single workflows and for batches of workflows. The most popular approaches [155, 142, 76, 32] include upward and downward ranking and different forms of list scheduling techniques. Workflow tasks are usually prioritized in ascending or descending order of their runtimes or by their proximity to the entry or exit task. The individual workflows are often prioritized based on the length of their critical path (a longest path from an entry to the exit task). In this situation, inaccurate runtime estimates can significantly affect the task and workflow ranking, as not only the length of the critical path can be affected but even a wrong critical path can be used. Knowing how the quality of estimates affects the performance helps to create better error-resilient scheduling policies and is useful when selecting policies which are less sensitive to incorrect estimates. In Chapter 2, we used a different approach where we scheduled an arriving stream of workflows without using any runtime estimates at all, and completely relied on the structure of the workflows when making scheduling decisions.

To study the influence of the accuracy of task runtime estimation on the performance, in this chapter, we study a total of seven scheduling policies for workloads of workflows, and we simulate their execution on two workloads of realistic workflows. Four of these are existing dynamic workflow scheduling policies, namely Greedy Backfilling (GBF), which came out best of all the policies we proposed in Chapter 2 when no runtime estimates are available, Online Workflow Management (OWM) [77, 32], Fairness Dynamic Workflow Scheduling (FDWS) [29, 32], and Rank_Hybd (HR) [169]. We also propose the simple Critical Path Prioritization (CPP) policy and, in order to address the issue of fairness, the Fair

Workflow Prioritization (FWP) policy. To check how existing plan-based scheduling algorithms can be applied when scheduling workloads of workflows, we have adapted the Heterogeneous Earliest Finish Time (HEFT) [155] policy to the online case. All policies except GBF require task runtime estimates for their operation.

The main research questions we address in this chapter and our contributions towards answering them, are:

1. *What are the appropriate policies for dynamic scheduling of workloads of workflows with known task runtime estimates?* We propose in Section 3.3 two novel dynamic workflow scheduling policies (CPP and the fairness-oriented FWP), implement four state-of-the art online dynamic policies, and adapt the popular HEFT policy to the online case.
2. *How do inaccurate task runtime estimates affect the performance and fairness?* We show in Section 3.5 the effect of incorrect task runtime estimates on the performance and fairness when scheduling workloads of workflows in a simulated computing environment.
3. *How does the knowledge of task runtime estimates help to improve the performance and achieve fairness, and how applicable are plan-based policies for online scheduling of workflows?* We demonstrate in Section 3.5.1 that the knowledge of task runtime estimates improves the performance at high system utilizations, and in Section 3.5.3 we show that the plan-based approach struggles to deal with workloads of workflows.

3.2. Problem Statement

This section presents our model for the problem of using runtime estimates for scheduling workloads of workflows and the performance metrics.

3.2.1. The Model

In this chapter, we consider both homogeneous and heterogeneous computing systems, in contrast to Chapter 2, where we only study homogeneous systems. For the rest, the system model, the arrival process, and the workflow definitions are the same as in Chapter 2. When the system is heterogeneous, we suppose that the execution time of a task on a processor is inversely proportional to the processor speed.

The runtime estimates are often extracted from historical runs, simulations of workflow executions, or even are obtained from users [63, 43]. However, the quality of such estimates can vary significantly depending on the estimation method. To study the effect of the quality of task runtime estimates on the system performance, we modify the perfect task runtimes obtained from the synthetic workload using a certain pre-defined *error factor* f_e. The error allows to either under-estimate or over-estimate the task runtimes. All of the evaluated schedulers are not aware of under- or over-estimation. They can only derive the error in task runtimes post factum by comparing the given runtime estimates with the actual task runtimes obtained during the execution (as in our FWP policy, Section 3.3.7). We use three methods to introduce the estimation errors:

- *Static error:* Here we multiply the runtime of every task of every workflow by the error factor f_e.
- *Random error I:* Here we multiply the runtime of every task within a single workflow by the same random error factor. This random error factor is independently generated for every workflow in the workload by drawing it from the uniform distribution on the interval $(0, f_e \times 2]$. So on average, task runtimes in the workload are under- or overestimated by a factor of f_e.
- *Random error II:* Here we multiply the runtime of every task by an individually generated random error. The runtime estimate of a task is computed by multiplying its original correct task runtime d_r by a random error factor drawn from the uniform distribution on the interval $(0, f_e/d_r \times 2]$. This is an extreme case of introducing an error, as it changes the distribution of task runtimes to a uniform distribution on the interval $(0, f_e \times 2]$. Thus, the scheduler operates with estimates which are very far from the original ones.

3.2.2. Performance Metrics

In this chapter, we use a set of metrics and baselines, similar to those defined earlier in Section 2.2.2. However, in contrast to Chapter 2, where task runtimes are unknown, in this chapter we suppose that task runtime estimates are available. Accordingly, we can calculate the critical path length c of a workflow using the perfect task runtimes obtained from the synthetic workload. We define the *slowdown* s of a workflow as its response time t_r (in a busy system, when the workflow runs simultaneously with other workflows) normalized by the length c of its critical path: $s = t_r \,/\, c$. The critical path length computation is originally presented in Section 1.4 and further explained in Section 3.3.1.

3.3. Scheduling Policies

In this section, we first provide a detailed definition of the upward rank computation, which is crucial for task prioritization in almost all of the scheduling policies we consider. Then we present GBF the greedy backfilling dynamic policy which does not use task runtime estimates at all, and five dynamic policies: CPP, OWM, FDWS, HR, and FWP, that require task runtime estimates. Dynamic policies make scheduling decisions whenever a new task becomes eligible or a processor becomes idle. Finally, we present the plan-based WHEFT policy which uses task runtime estimates to construct a full ahead execution plan on every workflow arrival; between arrivals, the execution is completely guided by the precomputed plan. We classify the considered policies by their distinctive properties in Table 3.1. The GBF policy is included in this chapter as it has shown good performance earlier in Chapter 2. The dynamic policies that require task runtime estimates have been selected based on the comparative study by Arabnejad et al. [32]. We choose HEFT as it is one of the most popular algorithms for workflow scheduling and it is often used as a reference [170].

Property	Policies						
	GBF	CPP	OWM	FDWS	HR	FWP	WHEFT
Proposed in this dissertation	+	+	−	−	−	+	+
Plan-based	−	−	−	−	−	−	+
Explicit job queue (FCFS)	+	+	−	−	−	−	−
Joint eligible set (one task per WF)	−	−	+	+	−	+	−
Joint eligible set (all eligible tasks)	−	−	−	−	+	−	−
Fairness-aware	−	−	−	+	−	+	−

Table 3.1: The distinctive properties of the considered policies.

3.3.1. The Upward Rank Computation

Further, we extend the less formal definition of the upward rank presented earlier in Section 1.4. For each task n_i in a workflow, the *upward rank* r_u is recursively calculated, starting from the exit task, using the following formula:

$$r_u(n_i) = \overline{e_i} + \max_{n_j \in S(n_i)} (\overline{c_{i,j}} + r_u(n_j)), \tag{3.1}$$

where $\overline{e_i}$ is the average estimated execution time of task n_i, $S(n_i)$ is the set of immediate successors of task n_i, and $\overline{c_{i,j}}$ is the average communication delay between tasks n_i and n_j. Since the exit task has no successors, its upward rank is just equal to its average estimated execution time. The average estimated execution time $\overline{e_i}$ is calculated for each task using the average speed of the processors in the system. The average estimated communication cost $\overline{c_{i,j}}$ is calculated as the average communication start-up time plus the size of the data to be transmitted, divided by the average transfer rate between the processors. The length c of the critical path of a workflow is equal to the maximum value of r_u among all the workflow tasks N:

$$c = \max_{n_i \in N}(r_u(n_i)). \tag{3.2}$$

In this chapter, we approximate LoP by dividing the total execution time t_e of a workflow by the length c of its critical path.

3.3.2. Greedy Backfilling

The simple *Greedy Backfilling* (GBF) policy is an application of greedy backfilling to workflow scheduling which we proposed in Chapter 2. This policy processes workflows in FCFS order, and does not require task runtime estimates for its operation. In GBF, on every invocation, the scheduler, starting from the head of the queue, selects the first workflow with a non-empty eligible set, randomly picks a task from it, assigns it to the first available fastest processor, and removes it from the set. It continues to do this until the eligible set of the workflow is empty or until there are no more idle processors. When the eligible set is empty but the system still has idle processors, the scheduler takes the eligible set of the next workflow in the queue, and so forth.

3.3.3. Critical Path Prioritization

Our *Critical Path Prioritization* (CPP) policy extends our GBF policy. In CPP, on every invocation, the scheduler, starting from the head of the queue, selects the first workflow with a non-empty eligible set, picks the task from it with the highest r_u, assigns it to the first available fastest processor, and removes it from the set. For the rest, the CPP scheduler is similar to GBF.

3.3.4. Online Workflow Management

The *Online Workflow Management* (OWM) policy [77, 32] maintains a single joint eligible set which contains only a single eligible task (if any) with the highest r_u from every workflow in the system. At every scheduler invocation, as long as the system has workflows with eligible tasks, the scheduler selects the task with the highest r_u from the joint set. If the idle processors have the same speed, OWM finds the busy processor which will become idle earlier than any other busy processor. If the estimated finish time of the selected task on that busy processor is smaller than EFT on any of the idle processors, the task is postponed (stays in the joint set) until the next scheduler invocation. Otherwise, the task is assigned to any of the idle processors. If the idle processors have different speeds, the task is assigned to the fastest idle processor.

3.3.5. Fairness Dynamic Workflow Scheduling

The *Fairness Dynamic Workflow Scheduling* (FDWS) policy [29, 32] maintains a single joint eligible set which is formed in the same way as in OWM. However, within the joint set each task of a workflow j is additionally prioritized with rank r_a (highest first) which considers the fraction of remaining tasks of the workflow and the length of its critical path. The additional rank r_a is defined as follows:

$$r_{a,j} = \left(\frac{m_j}{p_j} \cdot c_j \right)^{-1}, \tag{3.3}$$

where m_j is the number of unfinished (not yet eligible or eligible) tasks in workflow j, p_j is the total number of tasks in the workflow, c_j is the initial length of the workflow (at the moment of its arrival to the system). The first factor in the formula prioritizes workflows with lower fractions of remaining tasks, while the second factor in the formula gives priority to shorter workflows. There are two versions of the FDWS policy in the literature. The first version considers both idle and busy processors for task allocation. If the selected processor is busy the task is placed in its task queue. The second version considers only idle processors. In both cases the processor allowing the lowest estimated finish time for the task is selected. For better comparability with other considered policies, in this chapter we use the version of the FDWS policy [32] without per processor queues.

3.3.6. Hybrid Rank

The *Hybrid Rank* (HR, the original name is Rank_Hybd [169]) policy maintains a single joint eligible set of *all* the eligible tasks from all the workflows in the queue. On arrival of a workflow, the policy computes r_u for all its tasks. At every

scheduler invocation, if the tasks in the joint set belong to different workflows, the scheduler selects the task with the lowest r_u. If the tasks in the joint set are from the same workflow, the algorithm selects the task with the highest r_u. On the one hand, the HR policy tries to achieve fairness by allowing shorter workflows to start their execution earlier. On the other hand, during the execution of a workflow, the length of the remaining part of its critical path decreases as more tasks finish. Although HR could delay longer workflows just after their arrival, the policy gives them more preference when they are about to finish.

3.3.7. Fair Workflow Prioritization

We propose the *Fair Workflow Prioritization* (FWP) policy which is similar to OWM and FDWS in the way it forms the single joint eligible set, but which uses a different mechanism to compute task priorities to achieve even better fairness than FDWS. On every workflow completion, by averaging historical slowdowns of previously finished workflows, FWP computes the *target slowdown* which all the workflows in the system are supposed to experience. The workflows are prioritized based on their proximity to the target slowdown. The lower the current slowdown of a workflow than the target slowdown, the lower its priority, the higher the current slowdown than the target slowdown, the higher its priority.

FWP allows to achieve better fairness when scheduling multiple workflows simultaneously, as the acceleration of certain workflows is done at the cost of decelerating others. Thus, the number of possibilities to slow down a certain workflow is limited by the number of workflows present in the system. To achieve the same target slowdown, workflows with longer critical paths should be delayed more compared to workflows with shorter critical paths. If the system does not have enough concurrent workflows, the workflows with longer critical paths will experience lower slowdowns than the target. An alternative solution is to postpone certain workflows by periodically excluding their tasks from the joint eligible set and making them eligible for scheduling after a timeout. However, we keep this improvement for future work.

We calculate the *target slowdown* s_t based on the history of slowdowns s_i of K previously finished workflows by averaging them:

$$s_t = \frac{1}{K} \cdot \sum_{i=1}^{K} s_i. \tag{3.4}$$

When the history is empty, the system is initialized with $s_t = 1$. For workflow j, its *current slowdown* $\hat{s}_j$ is calculated as:

$$\hat{s}_j = \frac{\hat{t}_{r,j} + \hat{c}_j \cdot \xi}{c_j \cdot \xi}, \tag{3.5}$$

where $\hat{t}_{r,j}$ is the current residence time of workflow j from its arrival till now, $\hat{c}_j$ is the length of the critical path of the remaining part of the workflow (which is not yet running), c_j is the length of the workflow, and ξ is the correction coefficient which is required to cope with possibly incorrect estimates. Since FWP depends on critical path length to calculate $\hat{s}$, incorrect task runtime estimates could affect

the ranking. Thus, after the completion of each task, the policy stores its actual measured runtime d_m and its estimated runtime d_e, and computes a correction coefficient ξ using information about the M tasks that finished last:

$$\xi = \frac{\sum_{i=1}^{M} d_{m,i}}{M} \Big/ \frac{\sum_{i=1}^{M} d_{e,i}}{M}. \tag{3.6}$$

To prioritize the task from workflow j within the joint eligible set, FWP uses rank r_b which is calculated as:

$$r_{b,j} = \hat{s}_j - s_t, \tag{3.7}$$

where $\hat{s}_j$ is the current slowdown of workflow j and s_t is the target slowdown.

3.3.8. Workload HEFT

The *Workload HEFT* (WHEFT) policy is our adaptation of HEFT policy [155] for scheduling workloads of workflows. In addition to the *scheduler*, WHEFT uses a separate *planner* which maintains a global execution plan for all the workflows in the system. For each non finished task in the workflow, the plan defines the processor where and when the task should run. On every new workflow arrival a completely new global plan is created. Between the workflow arrivals the execution is completely guided by the scheduler using the plan.

Since the workload is an arriving stream of incoming workflows, to be able to apply HEFT it is required to combine the workflows in the system into one. For that, we use an *Alternating DAGs* approach proposed by Zhao and Sakellariou [170] as in the original paper it showed better performance compared to other approaches from the same group. To apply the Alternating DAGs approach, WHEFT planner first combines the workflows in the system by adding a single joint exit node. Then it computes upward ranks r_u for all the tasks within the new combined workflow. Further, using the Hybrid policy [142], WHEFT splits the combined workflow into levels where each level contains only independent tasks. The tasks within each level are grouped according to the original workflow where they belong to. WHEFT switches between the groups in a round robin manner to make a sorted list of tasks in (descending order of their r_u). The plan is created by sequentially traversing the levels and sequentially processing the sorted lists of tasks made from the groups by applying HEFT to them.

The WHEFT scheduler is called after the plan construction and after each task completion. The scheduler sequentially checks the plan and tries to assign non-running tasks from the plan to according processors. The tasks are checked for eligibility in the ascending order of their planned start times. If a task is not yet eligible, the scheduler proceeds to the next processor. When a task finishes, the scheduler removes it from the plan. The scheduler does not perform any task preemption. If, according to the plan, a certain task should be currently started, but the processor where it should run is still busy (as the plan could be incorrect due to erroneous runtime estimates), the task which occupies the processor runs until its completion.

3.4. Experiment Setup

In this section, we present the synthetic workloads we use to analyse the performance of our scheduling policies and we present our simulation environment.

3.4.1. Workloads

In our simulations, we use two workloads with an arrival process of workflows, and a batch of workflows that are all submitted simultaneously. Workload I mixes equal fractions of three representative types of workflows and is generated using the workflow generator [16] presented by Bharathi et al. [38]. The workflows are taken from different application domains, i.e., astronomy (Montage [90, 152, 38, 92]), physics (LIGO [139, 152, 38, 92]), and bioinformatics (SIPHT [108, 38, 92]).

Workload II solely consists of random workflows generated using an existing random DAG-generator created by Suter et al. [148]. The generator has four configuration parameters: *jump* sets the maximum number of workflow levels induced by the inter-task dependencies, *regular* specifies the regularity of the task distribution across workflow levels, fat specifies the width (LoP) of the workflow, and *density* specifies the numbers of dependencies between tasks of two consecutive workflow levels. The values for these parameters we use are selected uniformly from the following sets: $jump = 1, 2, 3$, $regular = 0.2, 0.8$, $fat = 0.2, 0.8$, and $density = 0.1$. We use only a single and relatively low value for *density* since for large workflows (with several hundreds of tasks and more), higher densities significantly increase the complexity of finding a critical path. More information on the parameterisation of the random DAG generator can be found in its code repository [148].

For the total workflow execution time t_e in Workload I, we use a two-stage hyper-Gamma distribution from the paper by Lublin and Feitelson [112], in the same way, as described in Section 2.4. Figure 3.1 visualizes this distribution. In Workload II we use the original total execution time distribution obtained from the generator, see Figure 3.2. In order to obtain simulation results for the different workflow types that can easily be compared, in both workloads we use the same average total execution time of one hour. For every workflow in both workloads, we normalize the generated task runtimes so that its total processing requirement is equal to the corresponding sample of the execution time distribution.

In Figures 3.1 and 3.2 we show the distributions of the task runtimes, the branching factors (the total number of inter-task links in a workflow divided by its size), and the approximated LoPs (t_e/c). The two workloads share the phenomenon that they are dominated by short tasks. However, the task runtimes in Workload I are an order of magnitude longer than in Workload II. At the same time, the range of the total job runtimes in Workload II is four times as large as in Workload I. Interestingly, Workload II also shows a higher diversity of branching factors but a twice smaller approximated LoP.

For each utilization level, both workloads consist of three unique sets of workflows with 3,000 workflows each, which allows us to perform three independent simulation runs with all the policies per utilization level. As with many other workloads in computer systems, in practice, workflows are usually small, but very large ones may exist too [132]. Therefore, in our simulations we distinguish small, medium,

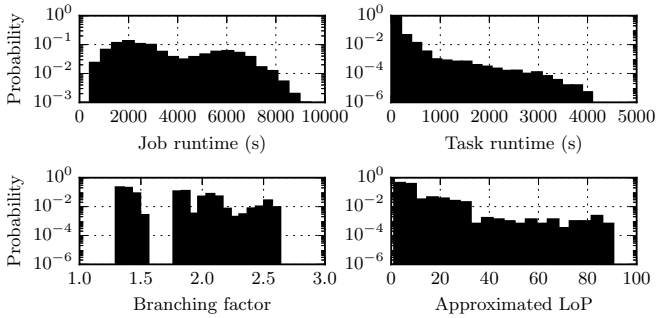


Figure 3.1: Statistical characteristics of Workload I. The vertical axes have a log scale.

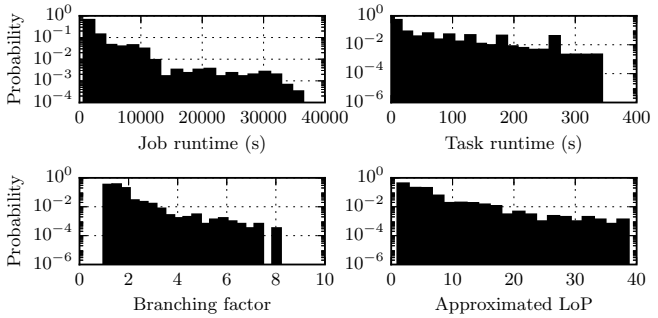


Figure 3.2: Statistical characteristics of Workload II. The vertical axes have a log scale.

and large workflows, defined by sizes that are uniformly distributed on the intervals [30,38], [40,198], and [200,600], respectively (all workflows are assumed to have even sizes). The small, medium, and large workflows constitute fractions of 75%, 20%, and 5% of the workload.

Finally, we use a batch of 1,000 workflows consisting of workflows from Workload I. Accordingly, the statistical characteristics of the batch are similar to those of Workload I.

3.4.2. Simulation Environment

We have modified the DGSim simulator [84, 85] for cluster and grid systems to include the workflow scheduling policies we consider. The size of the cluster we use in all of our simulations is 100 single-processor nodes. We vary the accuracy f_e of the estimates using the following values: 0.1, 2, 5, and 10. As in our model the communication overhead is not considered, the r_u values are computed only using the average estimated execution time. We suppose that after a task has been assigned to a processor, it runs there until its completion. For our FWP policy we choose values of $K = 300$ and $M = 1000$ (see Section 3.3.7).

We mostly focus on a homogeneous system where all the processors have an average processing speed of 1 workflow/hour. However, we also perform a set of experiments with a heterogeneous system with two equally sized groups of

processors: fast processors with an average processing speed of 1.5 workflow/hour and slow processors with an average processing speed of 0.5 workflow/hour.

For the majority of the simulations we use a system utilization of 98% since all the considered dynamic policies can handle such high utilizations (we show this later in Section 3.5). We only show results when the system is in steady state, i.e., when reporting performance results for workloads, we omit the performance information for the first 1000 workflows and last 1000 workflows in each simulation.

3.4.3. System Stability Validation

To be able to clearly distinguish situations when the system is or is not stable in a long-term perspective, we use two methods: the statistical system stability check proposed by Wieland et al. [166], as well as the Lyapunov drift theorem [128]. For every simulation run, we perform both stability tests. The system is considered stable if each method shows at least two stable results out of three.

According to the Wieland approach, we take the observed number of workflows in the system $N(t)$ (both queued and partially running) at every moment t, where $0 \leq t \leq \tau$ with τ the total duration of the simulation, and split the observations into b batches, where $b = 10$. Then for each batch $j = 2, \ldots, b$ we compute the time average number of workflows in the system within the batch as:

$$\hat{\lambda}_{N,j} = \int_{(j-1)\cdot\tau/b}^{j\cdot\tau/b} \frac{N(t)}{\tau/b} dt. \tag{3.8}$$

Then we compute the difference between the last and second batch observations: $\hat{\lambda}_N = \hat{\lambda}_{N,b} - \hat{\lambda}_{N,2}$, and compute the variance σ^2 of batch observations with $b-1$ degrees of freedom. We conclude whether the system is stable if

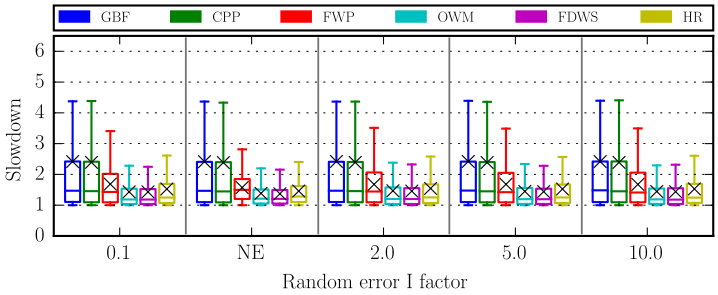
$$\frac{\hat{\lambda}_N}{\sqrt{2}\cdot\sigma} > t_{1-\alpha,b-2}, \tag{3.9}$$

where $t_{1-\alpha,b-2}$ is the $1-\alpha$ Student-T quantile with $b-2$ degrees of freedom. For the default values of $b = 10$ and $\alpha = 0.05$, $t_{1-\alpha,b-2} = 1.86$, thus, stability is rejected if $\hat{\lambda}_N > 2.63 \cdot \sigma$.

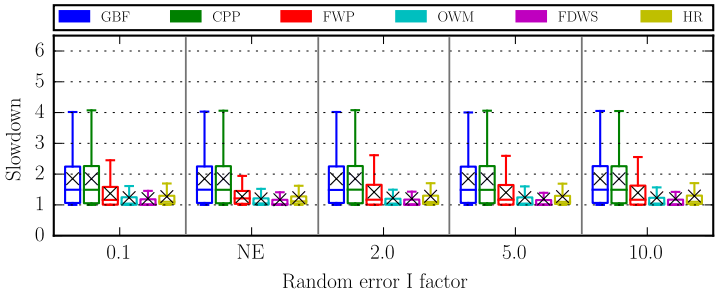
For the Lyapunov drift-based stability check we compute the mean Lyapunov drift throughout the simulation as follows: $\delta(t) = l(t) - l(t-1)$, where $l(t) = N(t)^2/2$. A low value of the mean Lyapunov drift after the initial transient indicates that the system converges and is stable. For our system we experimentally derive a threshold of 1 for the mean Lyapunov drift; if the mean drift exceeds 1, the system is considered unstable.

3.5. Experiment Results

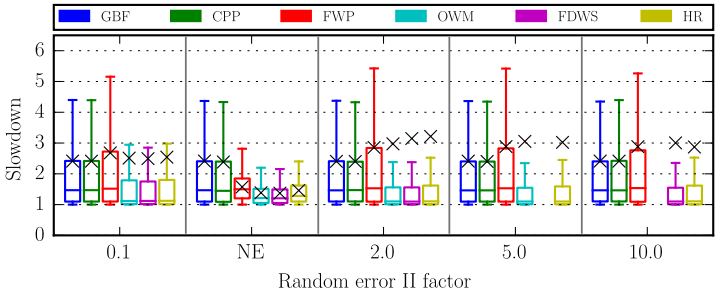
In this section, we present our experiment results. We first investigate how varying the error in task runtime estimates affects the slowdowns of workflows in homogeneous and heterogeneous systems for the six dynamic scheduling policies we consider. Then we show the performance of the plan-based WHEFT policy and the performance when scheduling batches of workflows.



(a) Random error I, Workload I.



(b) Random error I, Workload II.

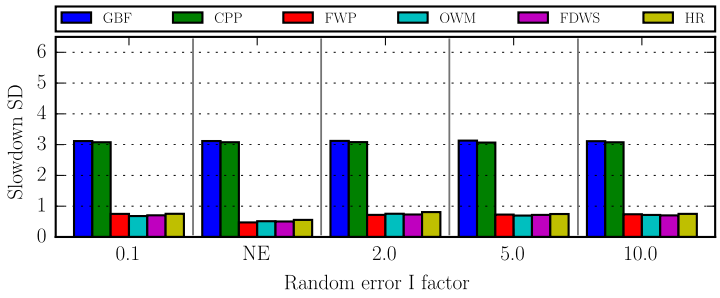


(c) Random error II, Workload I.

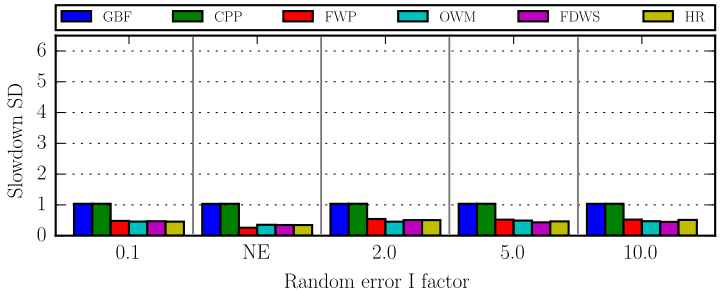
Figure 3.3: Slowdown versus the error factor at 98% system utilization in a homogeneous system. NE is No Error, means are marked with ×. Missing bars indicate unstable situations.

3.5.1. Performance of Dynamic Policies

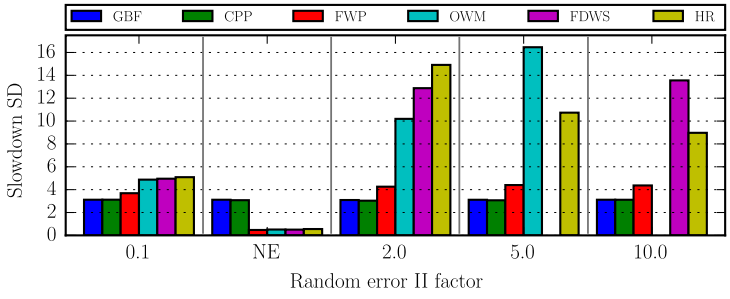
First, we compare the performance of the six dynamic policies we consider in a homogeneous system with the three methods for introducing estimation errors. All of these policies are able to achieve a 98% system utilization without destabilizing. Figures 3.3a and 3.3b show the workflow slowdown distribution and standard deviation versus the error factors in our two random error methods for both workloads. We do not show outliers in these figures and set the whisker boundaries within 1.5 times of the interquartile range. Changing the runtime estimates by a static factor does not affect the performance of any of the dynamic policies.



(a) Random error I, Workload I.



(b) Random error I, Workload II.



(c) Random error II, Workload I. The vertical axis has larger scale.

Figure 3.4: Slowdown standard deviation versus the error factor at 98% system utilization in a homogeneous system. NE is No Error. Missing bars indicate unstable situations.

The reason is that all the task upward ranks simultaneously scale in the same way, and that as a consequence, the order of selecting workflow tasks for execution is not affected. Therefore, we do not present the results for the Static error method as they are identical to the No Error results in Figures 3.3 and 3.4. Without an error, the mean number of workflows (fully or partially running and waiting) in the system throughout the experiment varies from 35 (CPP) to 40 (HR).

For the Random error I method, in Figures 3.3a and 3.3b we see that the performance of the policies is largely insensitive to the value of the error factor. We also find that the GBF and CPP policies, which both employ the FCFS principle,

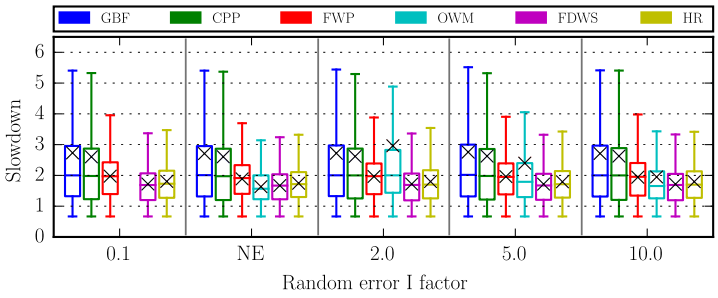
exhibit a much poorer mean slowdown and higher percentiles than the other policies. Moreover, the CPP policy exhibits only a slight decrease in the mean slowdown and the standard deviation with Workload I (see Figure 3.4a) over the GBF policy, which shows that the way in which it uses task runtime estimates is not effective. Notably, for our FWP policy the results for the No Error case are definitely the best. It also achieves lower values of the standard deviation with an error factor of 2.0 for Workload I, at the cost of an increased mean slowdown. In the other cases, FWP shows comparable or slightly higher standard deviation than the other policies, except GBF and CPP.

In contrast, using the Random error II method for varying the estimation error factor (see Figures 3.3c and 3.4c) does affect the slowdowns of the workflows in Workload I, creating many outliers and significantly increasing the mean slowdowns and standard deviations for all policies except GBF and CPP. OWM and FDWS even destabilize at high over-estimation factors, but stay stable at lower 97% utilization for all the error factors. However, our FWP policy shows the lowest values of the standard deviation compared to OWM, FDWS, and HR due to its correction mechanism for task runtime estimates (Figure 3.4c). At the same time, the Random error II method hardly affects Workload II and shows similar results as Random error I. Thus, we omit the results for Random error II with Workload II as they look identical to the Figures 3.3b and 3.4b. The statistical characteristics of the workloads (Figures 3.1 and 3.2) show the cause of this observation: Workload I has a much higher variability of task runtimes.

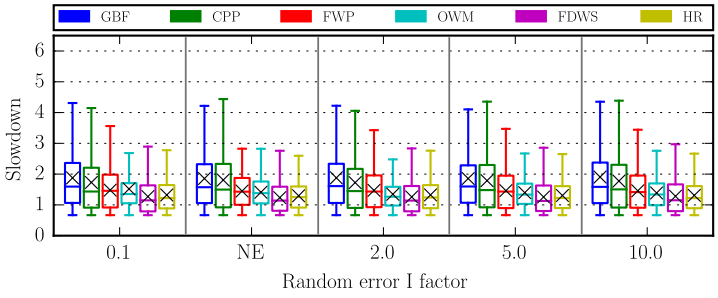
3.5.2. Effects of Heterogeneity

We conduct the same set of experiments with the dynamic policies as in Section 3.5.1 in a heterogeneous system and, analogously, we omit the results with static error factor. Interestingly, in a heterogeneous system the dynamic policies stay stable even at 99% imposed utilization with correct runtime estimates. Even though the average service rate of the heterogeneous system is the same as of the homogeneous system, the stream of arriving workflows does not split equally between two processor groups. There are two reasons for this: all the considered policies give priority to faster processors, and faster processors more often lead to scheduler invocations as they simply capable to process tasks faster. Compared to the homogeneous environment, the mean number of workflows in the heterogeneous system during the experiment without an error is higher and ranges from 38 (CPP) to 43 (HR).

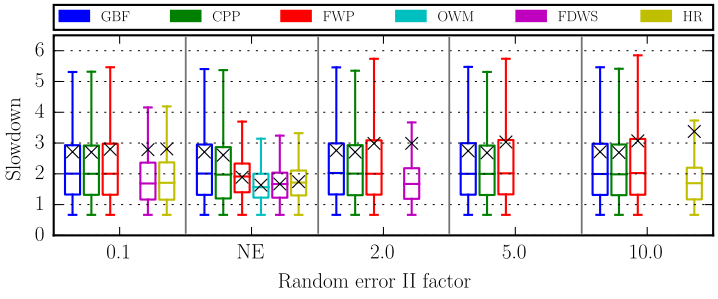
For comparability with the results in Section 3.5.1, in Figures 3.5 and 3.6 we show the results for the heterogeneous system at 98% utilization. Some policies, e.g., OWM, FDWS, and HR destabilize at certain error factors even more often as in the homogeneous system. However, similarly to the homogeneous system, all the considered dynamic policies are stable at 97% utilization for all the error factors. Comparing Figures 3.3 and 3.5, we can see that for all the policies their mean slowdowns increase in the heterogeneous system. At the same time, the values of standard deviation stay comparable to the homogeneous system, and only OWM perform much poorer and even destabilizes at error factor 0.1. The reason is that



(a) Random error I, Workload I.



(b) Random error I, Workload II.

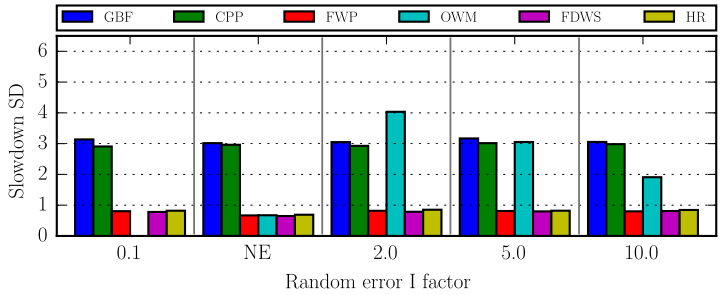


(c) Random error II, Workload I.

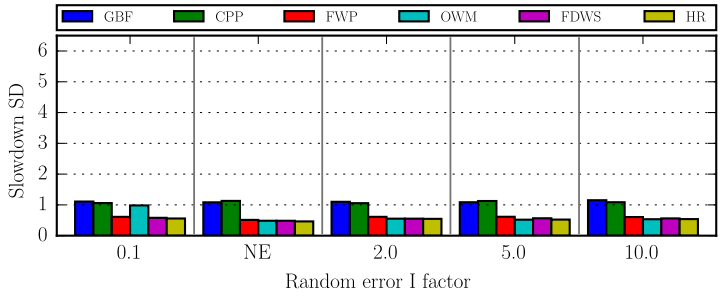
Figure 3.5: Slowdown versus the error factor at 98% system utilization in a heterogeneous system. NE is No Error, means are marked with ×. Missing bars indicate unstable situations.

OWM postpones tasks if there exist better placement in the future on a faster processor, and, of course, it is only applicable to the heterogeneous system. However, when task runtimes estimates are incorrect, OWM starts to make "mistakes" by unnecessarily postponing more tasks. Similar but even worse behavior can be observed in Figures 3.5c and 3.6c with Workload I and Random error II where OWM destabilizes for any error factor.

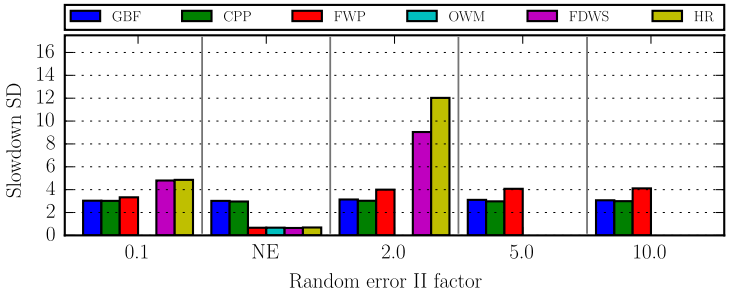
As in Section 3.5.1, all the policies stay stable with Workload II and only exhibit slightly higher mean slowdowns. Only in Figure 3.6b OWM shows an increase of standard deviation, however, without destabilizing. We do not present



(a) Random error I, Workload I.



(b) Random error I, Workload II.

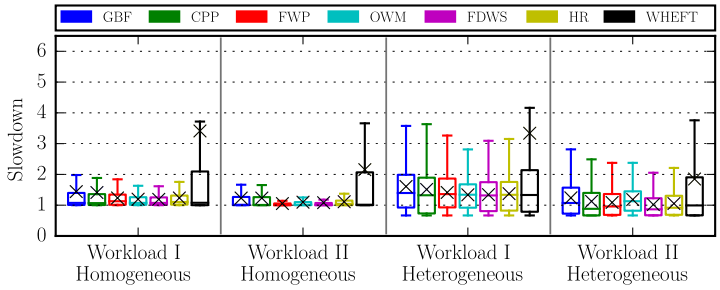


(c) Random error II, Workload I. The vertical axis has larger scale.

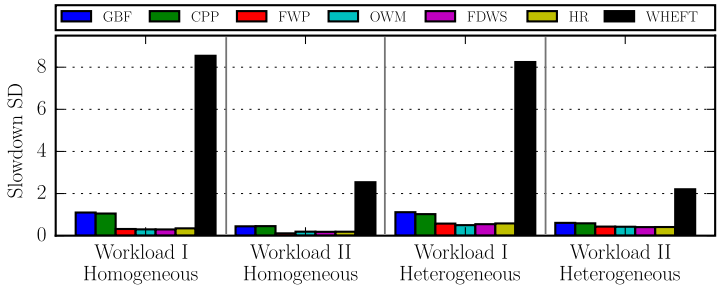
Figure 3.6: Slowdown standard deviation versus the error factor at 98% system utilization in a heterogeneous system. NE is No Error. Missing bars indicate unstable situations.

results for Workload II with Random error II as they are almost identical to Figures 3.5b and 3.6b with the only difference that OWM does not increase the standard deviation at error factor 0.1 and stays in line with FWP, FDWS, and HR.

Our FWP policy shows comparable performance to FDWS and HR while showing lower slowdown variability with Random error II and Workload I (Figure 3.6c) as FDWS and HR simply destabilize. Moreover, CPP policy performs better than GBF, showing that prioritizing tasks with higher upward rank has more effect in a heterogeneous system.



(a) Slowdown in various system configurations.



(b) Standard deviation of slowdown in various system configurations.

Figure 3.7: The performance of WHEFT in comparison with the dynamic policies at 97% imposed system utilization without estimation errors. Means are marked with ×.

3.5.3. Performance of Plan-based WHEFT

We include only a limited set of results with WHEFT as it simply turns out to be ineffective with workloads of workflows and brings extra complexity by requiring plan construction. Figure 3.7 shows a performance comparison of WHEFT with the dynamic policies at an imposed system utilization of 97%, as this is the maximum utilization at which WHEFT is stable in the No Error scenario. As a first conclusion from Figure 3.7, we find that a lower utilization decreases slowdowns and reduces the difference between (the interquartile ranges of) the dynamic policies compared to the results in Figures 3.3 and 3.5.

WHEFT is unstable for both error types and all error factors at 97% utilization. We investigated at which utilizations WHEFT stabilizes by decreasing the system utilization with steps of 10% in the presence of task runtime estimation errors. It turns out that WHEFT is only stable for all the considered error factors at a very low utilization of 40%.

The reason why WHEFT is so sensitive to estimation errors is that between workflow arrivals it is completely plan-driven and thus has less flexibility to cope with incorrect runtime estimates. If according to the plan a certain task should be currently scheduled to a processor, but it is not eligible due to the incorrectly calculated plan, WHEFT just skips it, leaving the processor idle. It thus creates a gap in the schedule and slows down the workflow to which the task belongs. Once

a task is wrongly placed in the plan due to incorrect estimates, it can only possibly be relocated to a better position when a new workflow arrives. While WHEFT postpones tasks "unintentionally", OWM postpones tasks on purpose in the hope that finally they will be scheduled on a faster processor. So the reason why WHEFT shows poor results is similar to why OWM becomes unstable in Section 3.5.2.

At a 97% utilization, our simulated system receives 97 workflows per hour (since the workflows have an average total execution time of 1 hour), which means that on average, the plan is recomputed 97 times per hour. So on average, every workflow task has a chance to be relocated to a better position 97 times during the workflow execution. Decreasing the utilization decreases the number of simultaneously scheduled workflows, but at the same time it decreases the number of possible task relocations.

Moreover, the original HEFT policy schedules tasks with higher upward ranks first. For this reason, WHEFT gives priority to longer workflows constituting the joint workflow. Accordingly, WHEFT postpones shorter workflows, which represent the majority of both our workloads. That increases the average slowdown and accumulates more workflows in the system, finally destabilizing it.

For plan-based policies in real-world non-simulated environments, the duration of the plan construction phase is crucial. Newly arrived workflows cannot start their execution until their tasks have been added to the plan. This can additionally increase job slowdown. In the considered simulated environment, from the perspective of workflows plan construction takes zero time, but in real systems it should be much smaller than the average workflow inter-arrival time. In our case, for 1000 simultaneously running workflows (see Section 3.5.4) the plan construction takes 40 minutes for a Python3 implementation running on a DAS-4 [35] node (2.4 GHz Intel E5620 CPU, 24 GB RAM). The planning time, however, can be reduced by using a tree structure (e.g., a k-tree [134]) to store the information about the gaps in the plan. It will decrease the time required to find an appropriate gap for a task at the cost of a higher memory consumption.

3.5.4. Performance of a Batch Submission

In this section, we investigate how the considered policies behave when handling a batch submission. Since this chapter mainly focuses on the analysis of workloads of workflows, we only perform a limited set of experiments with a single batch submission of 1000 workflows based on Workload I in a homogeneous system. Figure 3.8 shows the total schedule length in hours of this batch with variable Random error I. We do not show the results for Random error II as they are comparable. There is a small difference for under- and over-estimating situations, with GBF and CPP producing schedules that are longer by half an hour (5.5%) than the other dynamic policies.

We do not report slowdowns, as we suppose that batch submissions usually come from a single user who is only interested in minimizing the total schedule length rather than achieving fairness among the workflows in the batch. We can clearly see that WHEFT, indeed, constructs a shorter schedule than the dynamic policies. However, it is less resilient to errors as its scheduler postpones tasks which are not eligible or if a target processor is occupied (as in Section 3.5.3). The

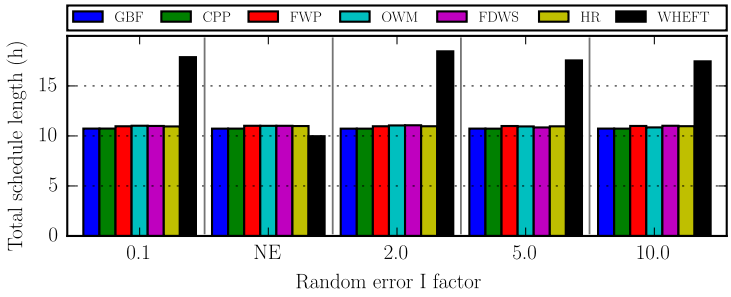


Figure 3.8: The total schedule length in hours of a batch submission based on Workload I with 1000 workflows in a homogeneous system. NE stands for No Error.

schedule length created by WHEFT matches the expected length of 10 hours, as we scheduled 1000 workflows with an average total execution time of 1 hour on 100 processors. Surprisingly, GBF and CPP policies produce shorter schedules than the other dynamic policies. Note, that GBF and CPP process workflows in the order in which they are defined in the batch, while all the other policies use various ways of ranking to prioritize workflows.

3.5.5. Fairness

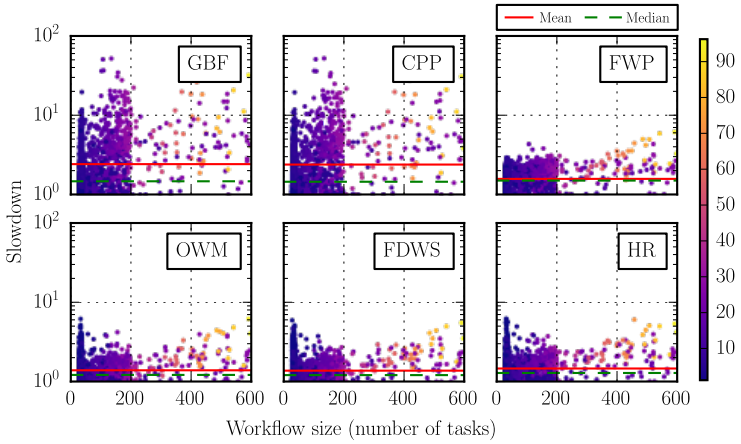
To demonstrate that our FWP policy allows to achieve better fairness when scheduling workloads of workflows, in Figure 3.9 we show scatter plots when running both workloads at 98% utilization in a homogeneous system without runtime estimation errors. We can clearly see that FWP reduces the number of outliers and moves the median slowdown closer to the mean than any other policy, while slightly increasing the mean. Similar behaviour is observed in the heterogeneous system (not shown).

Obviously, Workload I is more challenging for the dynamic policies as it contains more highly parallel workflows (see Figure 3.1). From one perspective, containing more tasks those workflows have more chances to adjust their ranks during the execution. From another perspective, among those highly parallel workflows some have very short critical paths, which means that their slowdowns, in case of any delay, increase much faster compared to relatively sequential workflows with long critical paths. These "short" but highly parallel workflows are the main reason why FWP does not show even better results with Workload I. None of the considered dynamic policies is able to completely remove such outliers. Including the level of parallelism when computing the rank in FWP might help to solve this problem.

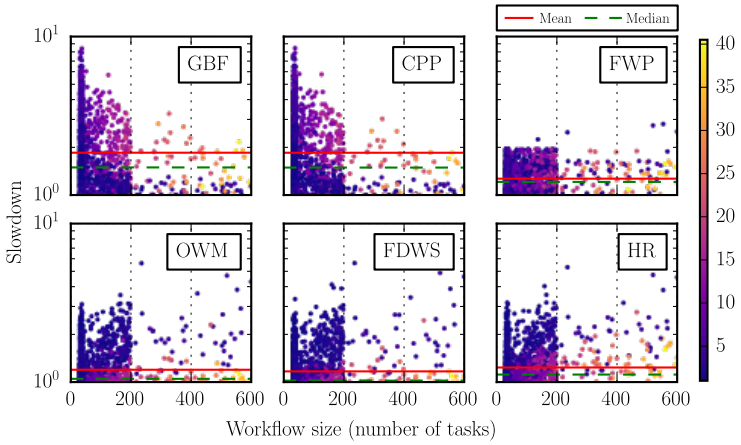
3.6. Related Work

Our work is a first study in the field which considers the influence of task runtime estimates on the quality of scheduling for a variety of workflow scheduling heuristics.

Yu ans Shi [169] use Poisson arrivals, but suppose perfect runtime estimates, and do not investigate slowdown variability. Hsu et al. [77] propose the original OWM algorithm and also use Poisson arrivals with a set of experiments dedicated to the



(a) Workload I.



(b) Workload II.

Figure 3.9: Scatter plots of slowdowns versus the sizes of workflows at 98% utilization in a homogeneous system without an error. The point color encodes the approximated LoP, the vertical axes have a log scale.

impact of inaccurate runtime estimates. Unlike us, that paper only considers one type of random uncertainty and compares OWM only with two other algorithms, including Rank_Hybd (HR) which we implement in this chapter. Moreover, the number of scheduled workflows in that paper is only 100, and the system utilization and stability are not taken into account.

In the paper by Arabnejad and Barbosa [30] the authors compare HEFT with FDWS and show that HEFT exhibits the poorest performance. They claim that they modified the original HEFT to use it in an online scenario, but do not clearly explain how. Moreover, the authors do not consider system utilization, just simply submitting relatively few workflows (50) with a fixed interval. In the recent paper by Arabnejad and Barbosa [31] which targets multi-QoS constrains the system

utilization is not considered either.

The slowdown-based fairness problem has been addressed before. Zhao and Sakellariou [170] proposed a plan-based policy which targets fairness using a variety of approaches. However, their algorithm has limited applicability for workloads of workflows as it is plan-based. Recently, Wang et al. [163] proposed fairness-aware dynamic FSDP algorithm which, however, does not clearly link the current slowdown and the target average slowdown which should be achieved, and recomputes workflow priorities on every new workflow arrival only. In contrast, our FWP policy recomputes priorities when a task becomes eligible or a processor becomes idle. An algorithm for fairness and granularity control for online scheduling of workflows is also addressed in a paper by Ferreira da Silva et al. [65]. Their approach, though, does not consider system utilization.

Among the algorithms which can operate without task runtime estimates we distinguish PRIO [117] which was successfully implemented in the DAGMan component of the Condor distributed job scheduler [153]. However, we do not include it in the comparison, as it tries to maximize the number of eligible tasks in hope to increase the throughput of the system, while in this chapter we focus on upward rank-based policies.

3.7. Conclusion

In this chapter, we have investigated the effect of incorrect task runtime estimates on the performance of dynamic and plan-based scheduling policies in the online scenario of scheduling workloads of workflows.

We can clearly see the benefit of knowing task runtime estimates as we do observe significant performance differences between the considered dynamic policies and large improvements in average job slowdown, but only at extremely high system utilizations. Similarly, the sensitivity to incorrect task runtime estimates increases at higher system utilizations. The order in which the workflows are processed is very important, as it allows to achieve a fairer distribution of slowdowns among workflows in the workload, as in our FWP policy.

Giving priority to workflows with longer critical paths, especially at extremely high utilizations, easily destabilizes the system if the workload has a majority of short jobs, as these short jobs start to accumulate. At lower utilizations, which are very common in real datacenters, simpler backfilling-based policies that do not use task runtime estimates are quite applicable and show comparable performance to more advanced fairness-oriented policies.

The plan-based WHEFT policy shows poor performance with workloads of workflows, but it does construct the shortest schedule for batch submissions. Moreover, WHEFT is quite unstable with incorrect task runtime estimates, running stably only at the relatively low utilization of 40%. We believe that even more complex policies like Hybrid.BMCT [142] would also suffer from this problem. Even though we do not exclude that plan-based approaches could achieve slowdowns comparable to those of dynamic policies, their planning overhead and implementation complexity do not seem to be worth it.

4

An Experimental Performance Evaluation of Autoscalers

Elasticity is one of the main features of cloud computing allowing customers to scale their resources based on the workload. Many autoscalers have been proposed in the past decade to decide on behalf of cloud customers when and how to provision resources to a cloud application based on the workload, utilizing cloud elasticity features. However, in prior work, when a new policy is proposed, it is seldom compared to the state-of-the-art, and is often compared only to static provisioning using a predefined Quality-of-Service target. This reduces the ability of cloud customers and of cloud operators to choose and deploy an autoscaling policy as there is often not enough analysis on the performance of the autoscalers in different operating conditions and with different applications. In this chapter, we conduct an *experimental* performance evaluation of autoscaling policies, using workflows—a popular formalism for automating distributed computations for applications with well-defined yet complex structures. We present a detailed comparative study of general state-of-the-art autoscaling policies, along with two new workflow-specific policies. To understand the performance differences between the seven policies, we conduct various experiments and compare their performance in both pairwise and group comparisons. We report both individual and aggregated metrics. As many workflows have deadline requirements on the tasks, we study the effect of autoscaling on workflow deadlines. Additionally, we look into the effect of autoscaling on the accounted and hourly-based charged costs, and evaluate performance variability caused by the autoscaler selection for each group of workflow sizes. Our results highlight the trade-offs between the suggested policies, how they can impact meeting the deadlines, and how they perform in different operating conditions, thus enabling a better understanding of the current state-of-the-art.

4.1. Introduction

Cloud computing emerged as a computing model where computing services and resources are outsourced on an on-demand pay-per-use basis. To make this model useful for a variety of customers, cloud operators try to simplify the process of obtaining and managing the provided services. To this end, cloud operators make available to their customers various autoscaling policies (*autoscalers*, *AS*), which are essentially parametrized cloud scheduling algorithms that dynamically regulate the amount of resources allocated to a cloud application based on the load demand and the Quality-of-Service (QoS) requirements typically set by the customer. Many autoscalers have been proposed in the literature, both general autoscalers for request-response applications [26, 42, 157, 64, 127] and autoscalers for more task- and structure-oriented applications such as workflows [25, 40, 60, 47, 136, 46]. The selection of an appropriate autoscaling policy is crucial, as a good choice can lead to significant performance and financial benefits for cloud customers, and to improved flexibility and ability to meet QoS requirements for cloud operators. Selecting among the proposed autoscalers is not easy, which raises the problem of finding a systematic, method-based approach to comprehensively evaluate and compare autoscalers. The lack of such an approach derives in our view from ongoing scientific and industry practice. For the past decade, much academic work has focused on building basic mechanisms and autoscalers for specific applications, with very limited work spent in comparisons with the state-of-the-art. In industry settings, much attention has been put on building cloud infrastructures that enable autoscaling as a mechanism, and relatively less on providing good libraries of autoscalers for customers to choose from [73, 133]. To alleviate this problem, in this chapter we propose and use the first systematic experimental method to evaluate and compare the performance of autoscalers using workflow-based workloads running in cloud settings as a use-case application.

Modern workflows have different structures, sizes, task types, run-time properties, and performance requirements, and thus raise specific and important challenges in assessing the performance of autoscalers: *How does the performance of general and of workflow-specific autoscalers depend on workflow-based workload characteristics?* Among the many application types, our focus on workflow-based workloads is motivated by three aspects. First, there is an increasing popularity [150, 152] of workflows for science and engineering [22, 90, 108], big data [109], and business applications [154], and by the ability of workflows to express complex applications whose interconnected tasks can be managed automatically on behalf of cloud customers [88]. Second, although general autoscalers focus mainly on QoS aspects, such as throughput, response-time and cost constraints, some autoscalers also take into account application structure [115]. One of the main questions we want to answer in this chapter is: *How does the performance of general and of workflow-specific autoscalers differ?*

Another interesting aspect that arise with workflow scaling is the effect of the autoscalers on workflow deadlines. Many of the workloads have deadlines, basically a bound on the tolerated time by the user between the workflow submission and when the computation results are available. Having enough capacity in the system for the workflows to finish their processing before the deadline can be severely

affected by the presence of autoscalers. For that we enforce per-workflow and per-workload deadlines which allow us to see: *How does each autoscaler affect deadline violations?*

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

Towards addressing the aforementioned questions, our contribution is threefold:

1. We design a comprehensive method for evaluating and comparing autoscalers (Sections 4.2–4.4). Our method includes a model for elastic cloud platforms (Section 4.2), a set of relevant metrics for assessing autoscaler performance (Section 4.3), and a taxonomy and survey of exemplary general and workflow-specific autoscalers (Section 4.4).
2. Using the method, we comprehensively and experimentally quantify the performance of 7 general and workflow-specific autoscalers, for more than 15 performance metrics (Section 4.5). We show the differences between various policy types, analyze parametrization effects, evaluate the influence of workload characteristics on individual performance metrics, and explain the reasons for the performance variability we observe in practice.
3. We also compare the autoscalers systematically (Section 4.8), through 3 main approaches: a pair-wise comparison specific to round-robin tournaments, a comparison of fractional differences between each system and an ideal system derived from the experimental results, and a head-to-head comparison of several aggregated metrics.

4.2. A Model for Elastic Cloud Platforms

Autoscaling is an incarnation of the dynamic provisioning problem that has been studied in the literature for over a decade [41]: many autoscalers in essence try to solve the problem of how much capacity to provision given a certain QoS, published state-of-the-art algorithms make different assumptions on the underlying environment, mode of operation, or workload used. It is thus important to identify the key requirements of all algorithms, and establish a fair cloud system for comparison.

4.2.1. Requirements

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

Since there are no perfect predictors, no ideal autoscaler exists. Thus, there is a need to have better understanding of the capabilities of the various available

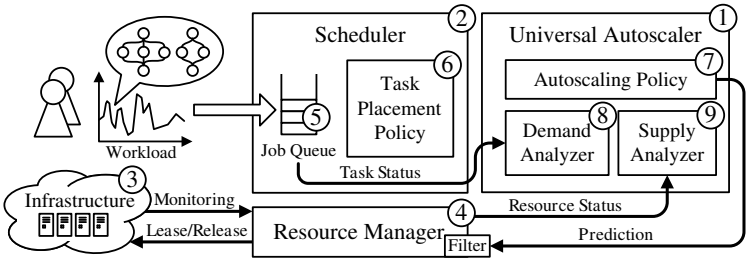


Figure 4.1: Elastic cloud platform.

4

autoscalers in comparison to each other. In this chapter, we classify autoscaling algorithms in two major groups: general and workflow-specific. Examples of general autoscalers include algorithms for allocating virtual machines (VMs) in datacenters. They are general because they mostly take their decisions using only external properties of the controlled system, e.g., workload arrival rates, or the output from the system, e.g., response time. In contrast, workflow-specific autoscalers base their decisions on detailed knowledge about the running workflow structure, task dependencies, and expected runtimes of each task [118]. Often, the autoscaler is integrated with the task scheduler [115].

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

4.2.2. Architecture Overview

Keeping the diversity of used cloud applications and underlying computing architectures in mind, we setup an elastic cloud platform architecture (Figure 4.1), which allows for comparable experiments by providing relatively equal conditions for different autoscaling algorithms and different workloads. The equal size of the virtual computing environment, which is agnostic to the used application type, is the major common property of the system in our model. We believe that our architecture represents modern elastic cloud platforms properly and reflects approaches used in many commercial solutions.

While our experiments should be valid for any cloud platform, we decided not to run any experiments on public clouds for two main reasons. First, scientific workflows have been shown to be cost-inefficient on public clouds [86, 168]. Second, as this chapter aims to fairly benchmark the performance of autoscalers, public clouds

will introduce variability due to the platforms as public cloud VM performance can vary considerably [86].

The core of our system is the autoscaling service (Component 1 in Figure 4.1) that runs independently as a REST service. The experimental testbed consists of a scheduler (Component 2) and a virtual infrastructure service (Component 3) which maintains a set of computing resources. A resource manager (Component 4) monitors the infrastructure and controls the resource provisioning. Users submit their workflows directly to the scheduler which maintains a single job queue (Component 5). The tasks from the queued workflows are mapped to the computing resources in accordance to the task placement policy (Component 6). The scheduler periodically calls the autoscaling service providing it with monitoring data from the last time period. We refer to this period as the *autoscaling interval*.

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

4.2.3. Workflow Applications and Deadlines

For our experiments, in this chapter we use complex workflows as our system workload. We rely on the same workflow job definitions as presented in previous chapters (e.g., in Section 2.2.1). Scheduling of workflows is often time critical and involves meeting deadlines for the processing times. For example, workflows for processing satellite data should handle the received information while the satellite makes a turn around the planet [137, 59]. Such workflows should finish before a new batch of information is received. Another example is the case of modern cloud services where the user pays per time slot and the deadlines are bounded to the lengths of these time slots [114]. We assume that deadlines for workflows are set on a per-workflow and per-workload basis. Per-workflow deadlines are unique for every workflow and are normally assigned based on user (statistical) estimates of the possible workflow runtimes. Per-workload deadlines are common when processing batches of workflows. In this case, a per-workload deadline applies to all the workflows in the workload. We consider soft deadlines which can be violated without affecting the execution of a workflow. Soft deadlines is a measure that can additionally reflect induced infrastructure costs for users.

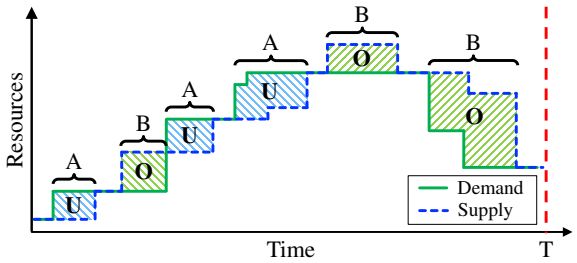


Figure 4.2: The supply and demand curves illustrating the under- and over-provisioning periods (A and B) quantified in the number of resources (areas U and O).

4.3. Performance Metrics for Autoscalers

We use both system- and user-oriented evaluation metrics to assess the performance of the autoscalers. The system-oriented metrics quantify over-provisioning, under-provisioning, and stability of the provisioning, all of which are commonly used in the literature [73, 26, 72, 28]. All the considered system-oriented metrics are based on the analysis of discrete supply and demand curves. The user-oriented metrics assess the impact of autoscaler usage on the workflow execution speed.

4.3.1. Supply and Demand

The resource *demand* induced by a load is understood as the minimal amount of resources required for fulfilling a given performance-related Service-Level Objective (SLO). In the context of our workflow model, a resource can only process one task at a time. We thus define the momentary demand as the number of eligible and running tasks in all the queued workflows. Extending the model to include resource sharing is trivial by using the average number of tasks processed by a resource instead.

Accordingly, the *supply* is the monitored number of provisioned resources that are either idle, booting or processing tasks. Figure 4.2 shows an example of the two curves. If demand exceeds supply, there is a shortage of available resources (under-provisioning) denoted by intervals A and areas U in the figure. In contrast, over-provisioning is denoted by intervals B and areas O.

4.3.2. Accuracy

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

$$a_U = \frac{1}{T \cdot R} \sum_{t=1}^{T} (d_t - s_t)^+, \tag{4.1}$$

$$a_O = \frac{1}{T \cdot R} \sum_{t=1}^{T} (s_t - d_t)^+, \tag{4.2}$$

where T is the time horizon of the experiment expressed in time steps, R is the total number of resources available in the current experimental setup, and $(x)^+ = \max(x, 0)$, i.e., only the positive values of x. The intuition behind the two accuracy metrics is shown in Figure 4.2. Under-provisioning accuracy a_U is equivalent to summing the areas U where the resource demand exceeds the supply normalized by the duration of the measurement period T. Similarly, the over-provisioning accuracy metric a_O is the sum of areas O where the resource supply exceeds the demand.

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

$$\overline{a}_U = \frac{1}{T} \sum_{t=1}^{T} \frac{(d_t - s_t)^+}{\max(d_t, \varepsilon)}, \tag{4.3}$$

$$\overline{a}_O = \frac{1}{T} \sum_{t=1}^{T} \frac{(s_t - d_t)^+}{\max(d_t, \varepsilon)}, \tag{4.4}$$

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

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

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

$$m_U = \frac{1}{T \cdot R} \sum_{t=1}^{T} u_t, \tag{4.5}$$

4.3.3. Wrong-Provisioning Timeshare

The accuracy metrics do not distinguish cases when the average amount of under- or over-provisioned resources results from a few large deviations between demand and supply or rather by a constant small deviation. To address this, the following two metrics provide insights about the fraction of time in which under- or over-provisioning occurs. As visualized in Figure 4.2, the following metrics t_U and t_O are computed by summing the total amount of time spent in an under- A or over-provisioned B state normalized by the duration of the measurement period. Letting $\operatorname{sgn}(x)$ be the sign function of x, the overall timeshare spent in under- or over-provisioned states can be computed as:

$$t_U = \frac{1}{T}\sum_{t=1}^{T}(\operatorname{sgn}(d_t - s_t))^+, \tag{4.6}$$

$$t_O = \frac{1}{T}\sum_{t=1}^{T}(\operatorname{sgn}(s_t - d_t))^+. \tag{4.7}$$

4.3.4. Instability of Elasticity

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

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

$$k = \frac{1}{T-1}\sum_{t=2}^{T}\mathbb{1}_{\operatorname{sgn}(\Delta s_t)>\operatorname{sgn}(\Delta d_t)}, \tag{4.8}$$

where $\mathbb{1}$ denotes the set indicator function, and where by over-provisioning trends we mean situations when supply increases while demand is stable or when supply increases while demand decreases or when supply is stable while demand decreases. Similarly, we define k' which shows the average fraction of time of under-provisioning trends:

$$k' = \frac{1}{T-1}\sum_{t=2}^{T}\mathbb{1}_{\operatorname{sgn}(\Delta s_t)<\operatorname{sgn}(\Delta d_t)}, \tag{4.9}$$

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

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

4.3.5. User-oriented Metrics

To assess the performance of autoscaling policies from the time perspective, we employ the (average) elastic slowdown as a main user metric together with a set of traditional metrics. In this chapter, we use capital letters for user- and cost-oriented metrics to distinguish them from the elasticity metrics.

The definition of the elastic slowdown relies on the definition of *response time* from Sections 2.2.2 and 3.2.2, denoted here by T_r. The execution time of a workflow is denoted here by T_e, and the workflow makespan is denoted by T_m. The *elastic slowdown* S_e of a workflow is its response time in a system which uses an autoscaler (where the workflow runs simultaneously with other workflows) normalized by its response time T_r' in a system of the same size without an autoscaler (where the workflow runs simultaneously with the same set of other workflows and where a certain amount of resources is constantly allocated): $S_e = T_r \,/\, T_r'$. In the ideal situation, where jobs do not experience slowdown due to the use of an autoscaler, the optimal value for S_e is 1. When S_e is less than 1, then the workflow is accelerated by the autoscaler. Additionally, we use the slowdown metric as defined in Section 3.2.2, but in this chapter we refer to this metric as *Schedule Length Ratio* (SLR) [29], to better distinguish it from the elastic slowdown S_e.

We also calculate the *average task throughput* $\overline{T}$ which is defined as the number of tasks processed per time unit. For each workflow, we define its *deadline proximity ratio* D_p as $D_p = T_r/D$, where T_r is the completion time of a workflow and D is the deadline. D_p is calculated for each workflow after the completion of its last task.

4.3.6. Cost-oriented Metrics

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

$$\overline{V} = \frac{1}{T} \sum_{t=1}^{T} s_t, \tag{4.10}$$

which reflects the *gain* of using an autoscaler. Though $\overline{V}$ expresses the average resource consumption, it does not show the incurred costs in relation to common cloud pricing models.

Therefore, we introduce special metrics to measure consumed CPU hours. When calculating the CPU hours, we distinguish between *accounted CPU hours* and *charged CPU hours*. Figure 4.3 illustrates this difference, where the light orange blocks represent the charged CPU hours and the light green blocks represent the accounted CPU hours. The *accounted CPU hours* H_j for VM j we define as:

$$H_j = \sum_{t=1}^{T} s_{t,j}, \tag{4.11}$$

where $s_{t,j}$ is the number of supplied resources at time t for VM j. Hence, H_j

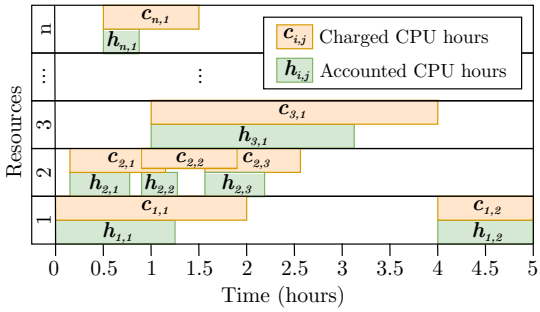


Figure 4.3: The difference between accounted and charged CPU hours.

4

represents the effective number of used CPU hours. We also compute the *average accounted CPU hours per VM* that is defined as:

$$\overline{H} = \frac{1}{R} \sum_{j=1}^{R} H_j, \tag{4.12}$$

where R is the number of VMs. Accordingly, we define the *accounted saving* $\tilde{H}$ as $\tilde{H} = \overline{H}_a / \overline{H}_n$, where $\overline{H}_a$ is the average number of CPU hours when all the resources are running throughout the experiment and $\overline{H}_n$ is the average number of accounted CPU hours when using an autoscaler. If $\tilde{H}$ is greater than 1, the autoscaler saves CPU hours, otherwise it uses the same or higher number of CPU hours than the no autoscaling scenario.

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

$$C_j = \sum_{t=1}^{T} \left\lceil \frac{\Delta_{t,j}}{P_{t,j}} \right\rceil K_{t,j}, \tag{4.13}$$

where T is the number of scaling events, $\Delta_{t,j}$ is the time elapsed between two events t and $t-1$ for VM j, $P_{t,j}$ is the charge period for VM j, $K_{t,j}$ is the charged cost for VM j, and $\lceil x \rceil$ denotes the ceiling function of a real number x. Notice that $P_{t,j}$ and $K_{t,j}$ can vary over time, but here we consider them as constants for the sake of simplicity. In the following, we selected $P_{t,j} = 60$ minutes, and $K_{t,j} = 1$. The *average charged CPU hours per VM* we calculate as:

$$\overline{C} = \frac{1}{R} \sum_{j=1}^{R} C_j, \tag{4.14}$$

where R is the number of VMs. Similarly to the accounted saving $\tilde{H}$, we compute the *charged saving* $\tilde{C}$ as $\tilde{C} = \overline{C}_a / \overline{C}_n$, where $\overline{C}_a$ is the average charged cost when all the resources are running throughout the experiment and $\overline{C}_n$ is the average charged cost when using an autoscaler. If $\tilde{C}$ is greater than 1 the autoscaler reduces costs, otherwise it leads to equal or higher costs than the no autoscaling scenario.

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

Table 4.1: The two-dimensional taxonomy of the considered autoscalers.

4.4. Autoscaling Policies

For evaluation, we select five representative general autoscalers and propose two workflow-specific autoscalers. We classify them using a taxonomy along two dimensions and summarize the survey of common autoscaling policies across these dimensions in Table 4.1. The taxonomy allows us to ensure the proper coverage of the design space. We identify four groups of autoscalers, which differ in the way they treat the workload information. The first group consists of general autoscalers Hist, Reg, and ConPaaS which require server-specific information and use historical data to make their predictions. The second group consists of React and Adapt autoscalers which also require server-specific information for their operation but do not use history to make autoscaling decisions. The last two groups use job-specific information (e.g., structure of a workflow) and also differ in a way they deal with the historical data: Plan needs detailed per task information while Token needs far less historical data and only requires a runtime estimate for the whole job. Further, we present all the autoscalers in more detail. When introducing each autoscaler we additionally indicate in the title of its section to which dimensions of the taxonomy it belongs.

4.4.1. General Autoscaling Policies

As different autoscalers exhibit varying performance, five existing general autoscalers have been selected. By a general autoscaler, we refer to autoscalers that have been published for more general workloads including multi-tier applications, but that are not designed particularly for workflow applications. The five autoscalers can be used on a wide range of scenarios with no manual tuning. We implement four state-of-the-art autoscalers that fall in this criteria. In addition, we acquire the source codes of one open-source state-of-the-art autoscaler. The selected methods have been published in the following years 2008 [157] (with an earlier version published in 2005 [156]), 2009 [42], 2011 [87], 2012 [26, 25], and 2014 [64]. The selected autoscalers are well-cited representatives of the autoscaler groups identified in the extensive survey by Lorido-Botran et al. [110].

General Autoscalers for Workflows

All of the chosen general autoscalers have been designed to control performance metrics that are still less commonly used for workflow applications, namely, request-response time, and throughput. The reason is that historically, workflow applications were rather big or were submitted in batches [152]. However, emerging workflow types require quick system reaction such as the usage of workflows in areas where they were less popular, e.g., for complex web requests, making the use of general autoscalers more promising.

The autoscalers aimed to control the response time are designed such that they try to infer a relationship between the response time, request arrival rates, and the average number of requests that can be served per VM per unit time. Then, based on the number of request arrivals, infer a suitable amount of resources. This technique is widely used in the literature [68, 110] due to the non-linearity in the relationship between the response time and allocated resources.

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

The React Policy (Server, Current)
Chieu et al. [42] present a dynamic scaling algorithm for automated provisioning of VM resources based on the number of concurrent users, the number of active connections, the number of requests per second, and the average response time per request. The algorithm first determines the current web application instances with active sessions above or below a given utilization. If the number of overloaded instances is greater than a predefined threshold, new web application instances are provisioned, started, and then added to the front-end load balancer. If two instances are underutilized with at least one instance having no active session, the idle instance is removed from the load balancer and shutdown from the system. In each case the technique *React*s to the workload change. For the remainder of this chapter, we refer to this technique as *React*. The main reason we are including this algorithm in the analysis is that this algorithm is the baseline algorithm in our opinion since it is one of the simplest possible workload predictors. We have implemented this autoscaler for our experiments.

The Adapt Policy (Server, Recent)
Ali-Eldin et al. [26, 25] propose an autonomous elasticity controller that changes the number of VMs allocated to a service based on both monitored load changes and predictions of future load. We refer to this technique as *Adapt*. The predictions are based on the rate of change of the request arrival rate, i.e., the slope of the workload, and aim at detecting the envelope of the workload. The designed autoscaler *Adapt*s to sudden load changes and prevents premature release of resources, reducing oscillations in the resource provisioning. *Adapt* tries to improve the performance in terms of number of delayed requests, and the average number of queued requests, at the cost of some resource over-provisioning.

The Hist Policy (Server, Long-term)
Urgaonkar et al. [157] propose a provisioning technique for multi-tier Internet applications. The proposed methodology adopts a queuing model to determine how many resources to allocate in each tier of the application. A predictive technique based on building *Hist*ograms of historical request arrival rates is used to determine the amount of resources to provision at an hourly time scale. Reactive provisioning is used to correct errors in the long-term predictions or to react to unanticipated flash crowds. The authors also propose a novel datacenter architecture that uses

VM monitors to reduce provisioning overheads. The technique is shown to be able to improve responsiveness of the system, also in the case of a flash crowd. We refer to this technique as *Hist*. We have implemented this autoscaler for our experiments.

The Reg Policy (Server, Long-term)
Iqbal et al. [87] propose a regression-based autoscaler (hereafter called *Reg*). The autoscaler has a reactive component for scale-up decisions and a predictive component for scale-down decisions. When the capacity is less than the load, a scale-up decision is taken and new VMs are added to the service in a way similar to *React*. For scale-down, the predictive component uses a second order regression to predict future load. The regression model is recomputed using the complete history of the workload when a new measurement is available. If current load is less than the provisioned capacity, a scale-down decision is taken using the regression model. This autoscaler was performing badly in our preliminary experiments due to two factors; first, building a regression model for the full history of measurements for every new monitoring data point is a time consuming task. Second, distant past history becomes less relevant as time proceeds. After contacting the authors, we have modified the algorithm such that the regression model is evaluated for only the past 60 monitoring data points.

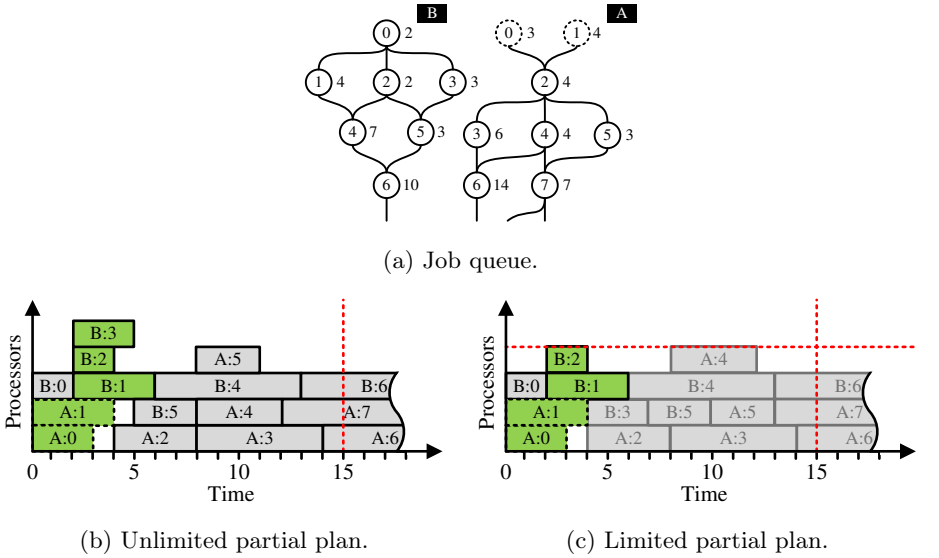
The ConPaaS Policy (Server, Long-term)
ConPaaS, proposed by Fernandez et al. [64]. The algorithm scales a web application in response to changes in throughput at fixed intervals of 10 minutes. The predictor forecasts the future service demand using standard time series analysis techniques, e.g., Linear Regression, Auto Regressive Moving Average (ARMA), etc. The code for this autoscaler is open source. We downloaded the authors' implementation.

4.4.2. Workflow-Specific Autoscaling Policies

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

The Plan Policy (Job, Long-term)
This autoscaler makes predictions by constructing and analyzing a partial execution *Plan* of a workflow. Thus, it uses the workflow structure and workflow task runtime estimates. The idea is partially based on static workflow schedulers [23]. On each call, the policy constructs a partial execution plan considering both workflows with running tasks and workflows waiting in the queue. The maximal number of processors which are used by this plan is returned as a prediction. The time duration of the plan is limited by the autoscaling interval. The plan is two-dimensional, where one dimension is time and another is processors (VMs).

The policy employs the same task placement strategy as the scheduler. In our case, the jobs from the main job queue are processed in the FCFS order and the tasks are prioritized in ascending order of their identifier. Each task of a workflow is supposed to be assigned with a unique numeric identifier using the classical breadth-first search algorithm, where the identifier of the entry task is 0. For



(a) Job queue.

(b) Unlimited partial plan.

(c) Limited partial plan.

Figure 4.4: The Plan autoscaling algorithm.

already running tasks, the runtimes are calculated as a remaining time to their completion. The algorithm operates as follows. On each call it initializes an empty plan with start time 0. Then it sequentially tries to add tasks in the plan in such as their starting times are minimal. The algorithm adds a task to the plan only if it is eligible or its parents are already in the plan. The plan construction lasts until there are no tasks which can be added in the plan or until the minimal possible task start time equals or exceeds the planning threshold (which is equal to the autoscaling interval), or until the processor limit is reached. If the processor limit is reached then this is returned as the prediction. Otherwise, the prediction is calculated as the maximal number of processors ever used by the plan within the planning interval.

Figure 4.4 shows an example of the operation of the algorithm. In Figure 4.4a, we show the job queue at the moment when the autoscaler is called. The queue contains two workflows A and B, where A is at the head of the queue. Each workflow task is represented by a circle with an identifier within it and runtime in time units on the right. Tasks A:0 and A:1 are already running, finished tasks are not shown. The autoscaling interval (a threshold) is equal to 15 time units and is represented by a vertical red dashed line. Figure 4.4b shows an example of an unlimited plan where the processor limit is not reached. In this case, the maximal number of processors used within the 15 time units interval is 5 which equals to the number of green rectangles in the figure (A:0, A:1, B:1, B:2, B:3). Figure 4.4c shows a plan where the number of available processors is limited by 4 (the horizontal red dashed line). In this case, the algorithm stops constructing the plan after placing task B:2 and returns the prediction, which simply equals to the maximal number of available processors (i.e., 4).

The Token Policy (Job, Recent)
The *Token* policy uses structural information of a DAG and does not directly consider task runtimes to make predictions and instead requires an estimated execution time of the whole workflow. It uses tokens to estimate the *Level of Parallelism* (LoP) of a workflow by simulating an execution "wave" through a DAG, as described in Section 2.3.1. The algorithm processes the workflows in the queue in the FCFS order. For each workflow, the number of token propagation steps is limited by a certain depth δ, which is defined as $\delta = (\Delta t \cdot N)/L$, where Δt is the autoscaling interval, N is the number of tasks on the critical path of the workflow, and L is the total duration of the tasks on the critical path of the workflow. Thus, the intuition is to evaluate the number of "waves" of tasks (future eligible sets) that will finish during the autoscaling interval. When δ or the final task of a workflow is reached, the largest recorded number of tokenized nodes is the approximated LoP value. The algorithm stops when the prediction value exceeds the maximal total number of available processors or when the end of the queue is reached. The final prediction is the sum of all of the separate approximated LoPs of the considered workflows.

The token-based algorithm does not guarantee the correct estimation of the LoP. The quality of the estimation depends on the DAG structure. In Figure 2.1a the estimated LoP of 3 is lower than the maximal possible LoP of 4 in Figure 2.1b. However, in Section 2.3.1, we show that this method provides meaningful results for popular workflow structures.

4.5. Experimental Evaluation

In this section, we present the workloads and the configuration of the cloud infrastructure we use for the experimental evaluation of the unified cloud system introduced in Section 4.2. To design our workloads, we use a set of representative scientific workflows. We take an experimental approach to evaluate chosen autoscaling algorithms with an extensive set of experiments in a virtualized environment deployed on our multi-cluster system.

4.5.1. Setup of Workflow-based Workloads

We choose three popular scientific workflows from different fields, namely Montage, LIGO, and SIPHT. The main reason for our choice is the existence of validated models for these workflow types. Montage [90] is used to build a mosaic image of the sky on the basis of smaller images obtained from different telescopes. LIGO [22] is used by the Laser Interferometer Gravitational-Wave Observatory (LIGO) to detect gravitational waves. SIPHT [108] is a bioinformatics workflow used to discover bacterial regulatory RNAs.

We generate synthetic workflows using the workflow generator by Bharathi et al. [16, 38]. Each workflow is represented by a set of task executables and a set of input files. We use two workloads: a primary Workload 1 and a secondary Workload 2 each consisting of 200 workflows of different sizes in the range from 30 to 600. Each workload contains an equal mixture of all of the three considered workflow types. As with many other workloads in computer systems, in practice, workflows are usually small, but very large ones may exist too [132]. Therefore,

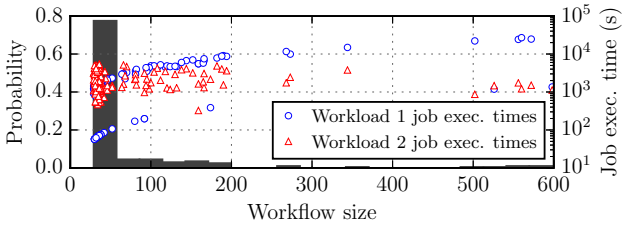


Figure 4.5: The distribution of job sizes in the workloads (histogram, left vertical axis) and the dependency between the job size and its execution time (points, right vertical axis). The right vertical axis is in log scale.

4

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

Table 4.2: Statistical characteristics of the workloads, σ stands for standard deviation.

in our experiments we distinguish small, medium, and large workflows, which constitute fractions of 75%, 20%, and 5% of the workload. The size of the small, the medium, and the large workflows is uniformly distributed on the intervals [30, 39], [40, 199], and [200, 600], respectively. The distribution of the job sizes in the workloads is presented in Figure 4.5. Figure 4.6 shows the distribution of task runtimes. Figure 4.7 shows the distribution of job execution times T_e in the workloads. Note, that the histogram of job execution times shows the *total execution time* of a job which is the sum of all the job's task runtimes. The job will be running this amount of time if and only if all of its tasks are executed sequentially. However, normally workflows have both parallel and sequential parts. Thus the job execution times reported in Figure 4.7 should not be confused with the actual observed makespans T_m of workflows running in a parallel system.

For Workload 1, we use the original job execution time distribution from the Bharathi generator. For Workload 2, we keep the same job structures as in Workload 1, but change the job execution times using a two-stage hyper-Gamma distribution derived from the model presented by Lublin and Feitelson [112]. The shape and scale parameters (α, β) for each Gamma distribution are set to (5.0, 323.73) and (45.0, 88.291), respectively. Their proportions in the overall distribution are 0.7 and 0.3. Table 4.2 summarizes the properties of both workloads.

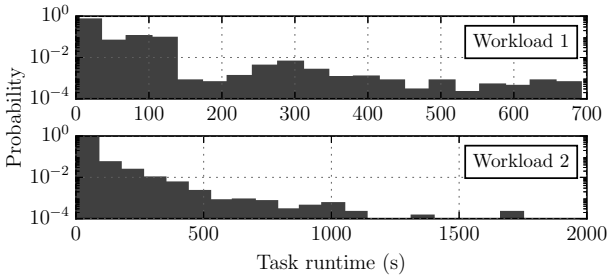


Figure 4.6: The distribution of task runtimes in the workloads (the horizontal axes have different scales, and the vertical axes are in log scale).

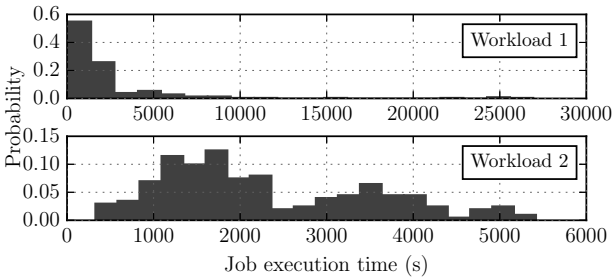


Figure 4.7: The distribution of job execution times in the workloads (all the axes have different scales).

4.5.2. Setup of the Private Cloud Deployment

To schedule and execute workflows, we use the experimental setup in Figure 4.1 (Section 4.2). The Koala scheduler is used for scheduling workflow tasks [62] on the DAS-4 [35] cluster deployed at TU Delft. Our cluster consists of 32 nodes interconnected through QDR InfiniBand with 8-core 2.4 GHz CPU and 24 GB of RAM each. As a cloud middleware, OpenNebula 5.4.6 is used to manage VM deployment, and configuration. The implementation and the VM images are available upon request. The REST interface between the scheduler and general autoscalers in our architecture makes it extendable and allows to use other autoscaling policies. The workflow-specific autoscalers are implemented within Koala, though other custom policies can also be added.

The execution environment for a single workflow consists of a single head VM and multiple worker VMs. The head VM uses a single CPU core and 4 GB of RAM, while each worker VM uses a single core and 1 GB of RAM. Tasks are then scheduled on the VMs. The workload generator and the workflow runners run on a dedicated node. The workflow runner coordinates the workflow execution by following the task placement commands from the scheduler. The runner is also responsible for copying files (task executables, input and output files) to and from the VMs in the virtual cluster. For data storage and transfer, we use Network File System (NFS). This implies that if the head VM and worker VM are located on

the same physical node, the data transfer time between them is negligible. In other cases, data transfer delays occur. The measured mean NFS write speed for 10 tests of transferring 1 GB is 280 MB/s. We use this value to calculate critical path lengths for the SLR metric.

Compared to job execution times, file transfer delays and the scheduling overheads are negligible. All tasks write their intermediate results directly to the shared storage reducing data transfer delays for all workflows. A task can run as soon as all of its dependencies are satisfied. The runner copies all input files for a workflow to the virtual cluster before starting the execution. Thus, the impact from file transfer delays between tasks on performance is negligible. Tasks are scheduled using *greedy backfilling* as it has been shown to perform well when task execution times are unknown a priori (Chapter 2). During the experiment, only the autoscaler has access to the information about job execution times and task runtimes. Note, that for all considered autoscalers, we do not perform any task preemption. If an autoscaler requires to stop a certain number of VMs, the scheduler only releases those VMs which are idle. The scheduler first releases the VMs which are idle the longest.

4.5.3. Experiment Configuration

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

We test with three different configurations in our experiments, where we change the value of the service rate parameter or the VM provisioning latency. The service rate in a request-response system is usually the average number of requests that can be served per VM. This parameter is either estimated online, e.g., using an analytical model to relate response time, as the one used in Hist [156], or offline [68, 56]. For a task-based workload, there are multiple options including using the mean task service time, the median task service time, or something in between.

In the first configuration, we assume that a VM serves on average 1 task per autoscaling interval, i.e., 2 tasks per minute. We derive this value by rounding the service rate calculated based on the mean task runtimes to the nearest integer (Table 4.2). This service rate allows us to perform additional comparison between general and workflow-specific autoscalers as the demand curves have the same dimension. In the second configuration, we use the median task runtime of Workload 1 which gives a service rate equal to 14 (also rounding to the nearest integer) tasks per autoscaling interval, i.e., 28 tasks per minute. The general autoscalers using the second configuration are marked with a star ($\star$) symbol. While in the first two configurations we guarantee that all the provisioned VMs are booted at the moment when the autoscaler is invoked, in the third configuration the VM booting time of 45 s exceeds the autoscaling interval of 30 s. This configuration is also used to test workflow-specific autoscalers. The autoscalers using the third configuration are marked with a diamond ($\diamond$).

For all the configurations and for both workloads the workload player periodically

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

We are aware that in computing clouds the job arrivals are often affected by the time of the day and various external influences resulting in burstiness [63]. Even for a datacenter which serves requests from all around the globe the intensity of job arrivals could vary as the distribution of world population is not even across different time zones, etc. We leave the experiments with other job arrival patterns to future work. However, the arrival of workflow tasks is non-Poissonian and depends on the workflow structure and on the distribution of task runtimes within a workflow. Thus, the diversity of the workflow structures, workflow sizes, and task runtime distributions which we use, allows us to suppose that our setup is representative. Furthermore, for the considered duration of the experiment of approximately 6.5 h, we can simply claim that our emulated system serves requests from multiple independent users.

4.5.4. Experiment Results

The main findings from our experiments are the following:

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

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

Table 4.3: Calculated *autoscaling metrics* for the main set of experiments with *Workload 1*. The diamond symbol ($\diamond$) marks the experiments where the VM booting time is longer than the autoscaling interval and service rate parameter is set to 1.0. The star symbol ($\star$) marks general autoscalers configured with service rate 14.0. All the other general autoscalers are configured with service rate 1.0. Best values in each column are highlighted in bold, except the No AS case.

Analysis of Elasticity

To show the trade-offs between the autoscalers, we use the metrics described in Section 4.3. While calculating the system-oriented metrics, we exclude periods where the demand exceeds 50 VMs, the total number of available VMs. Since system-oriented metrics are normalized by time, this approach does not bias the results.

The aggregated metrics for all experiments are presented in Table 4.3, Table 4.4, and Table 4.5. Considering the cases where VMs are booting faster than the autoscaling interval, Table 4.3 shows that the autoscalers under-provision between 1% (using Hist) and 8% (using Adapt) less resources from the demand needs. Hist's superior under-provisioning with respect to others comes at the cost of on average provisioning 7 times the actual demand, compared to 24% over-provisioning for Plan.

The React$^\diamond$ and Plan$^\diamond$ policies with longer booting VMs in Table 4.3 show slightly different results compared to the runs with faster booting VMs. We picked only these two policies to have one from each group of autoscalers. Both React$^\diamond$ and Plan$^\diamond$ tend to under-provision more when the VM provisioning time is longer. The job slowdowns in Table 4.4 are also higher. From Figure 4.8 we can clearly see that for ($\diamond$) autoscalers the supply curve is always lagging behind the demand. Thus, we can conclude that longer provisioning times decrease the number of available resources for the workload. We can also notice that the average number of idle VMs decreases for React$^\diamond$ and for Plan$^\diamond$ as the tasks more fully utilize provisioned VMs.

For the general policies configured with service rate 1.0 and for workflow-specific

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

Table 4.4: Calculated *user-oriented and cost-oriented metrics* for the main set of experiments with *Workload 1*. The diamond symbol ($\diamond$) marks the experiments where the VM booting time is longer than the autoscaling interval and service rate parameter is set to 1.0. The star symbol ($\star$) marks general autoscalers configured with service rate 14.0. All the other general autoscalers are configured with service rate 1.0. The metric S_e as well presented for small (S), medium (M), and large (L) job sizes. Best values in each column are highlighted in bold, except the No AS case.

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

Table 4.5: Calculated *autoscaling and user-oriented metrics* for the additional set of experiments with *Workload 2*. The metric S_e as well presented for small (S), medium (M), and large (L) job sizes. Best values in each column are highlighted in bold, except the No AS case.

policies in Table 4.4 and Table 4.5 job elastic slowdowns show low variability. We can conclude that the resources either significantly over-provisioned (Hist and ConPaaS) or already provisioned resources are underutilized (React, Adapt, Reg, Plan, and Token). The non-zero values of m_U metric in these cases confirm our assumption.

The Influence of Different Workloads
The difference is also visible between the two used workloads. While Workload 1 has the majority of short jobs, Workload 2 has a more equal distribution of job execution times and is thus less bursty. Elastic job slowdowns in both tables confirm this tendency. For Workload 2 they slightly increase (the Plan policy in Table 4.5 is an exception) while going from small to large job sizes. We do not run Workload 2 with service rate different from 1.0 as we expect that the trend will be the same as for Workload 1.

The system-oriented metrics do not vary much between the workloads. For example, compare React in Table 4.3 and Table 4.4 with React in Table 4.5. Only Hist over-provisions less while running with Workload 2 as can be explained by lower burstiness of the workload.

The Dynamics of Autoscaling
Figure 4.8 shows the system dynamics for each autoscaling policy while executing Workload 1. Some of the autoscalers have a tendency to over-provision resources (Hist and ConPaaS). The other policies appear to be following the demand curve more or less closely. Note, that the demand curve has different shape for each autoscaler as the autoscaling properties affect the order in which workflow tasks become eligible.

The workflow-specific Plan policy follows the demand curve quite well and shows results similar to general autoscalers React, Adapt, and Reg running with service rate of 1.0. However, if a policy follows the demand too close that increases job slowdowns, as seen in Table 4.4. This trade-off is intuitive. The best policy when it comes to reducing slowdown is to always have more capacity than needed as this will allow any task to run as soon as it becomes eligible.

The Influence of Service Rate Parameter on the Autoscaling Dynamics
The most noticeable differences in the results are between general autoscalers running with service rate 1.0 and with service rate 14.0, based on mean and median workload task runtimes, accordingly. Figure 4.9 shows general autoscalers running with the same Workload 1 as in Figure 4.8. The demand curves in these two figures look very different, except for ConPaaS and ConPaaS*. In addition, the supply curves do not follow the demand curves closely anymore. Although the service rate chosen is the median-based, it does not reflect the temporal properties of the workload when it comes to the length of running tasks. If longer jobs occur in parallel, a queue of tasks will build up resulting in enormous system slowdowns. This is clear from Table 4.4 where the slowdown between the two service rates is 10 to 15 times larger when using a larger service rate. The k' metric also increases for service rate 14.0 as the autoscalers need to estimate more while computing the next predicted supply value and thus the curves are not so well synchronized. On

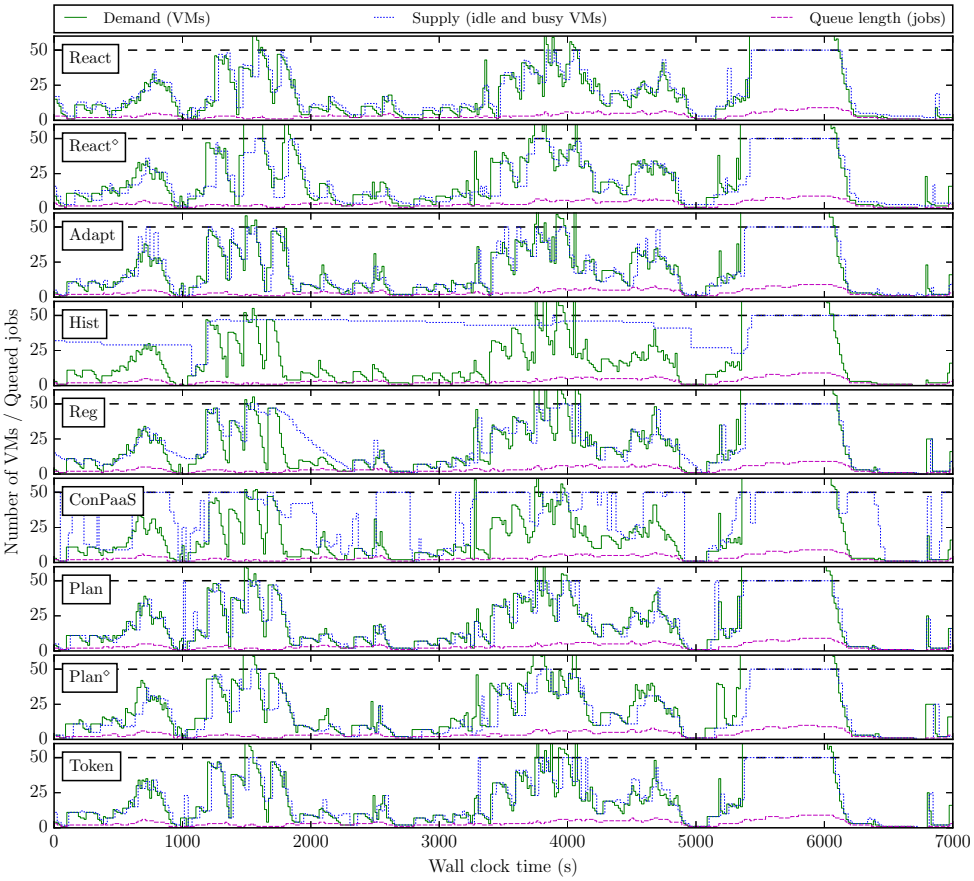


Figure 4.8: The experimental dynamics of five general autoscaling policies (Workload 1, service rate 1.0) and two workflow-specific policies during the cropped period of 7,000 s. The horizontal dashed line indicates the resource limit of 50 VMs. The diamond symbol (⋄) marks the experiments where the VM booting time is longer than the autoscaling interval.

the one hand, using a larger service rate significantly reduces the average charged CPU hours, potentially reducing the costs of operations for a user to almost one sixths of running with lower service rate.

The Trade-off Between Resource Usage and Performance

Here, we study the influence of the number of used VMs on the throughput. We evaluate only two user-oriented metrics: the throughput degradation in tasks per hour compared with the no autoscaler case and the number of used resources (VMs). The values of these metrics are plotted in Figure 4.10. For example, for React the throughput degradation of –24 tasks per hour contributes only to 1.16% of the hourly throughput. In Figure 4.10, we can see that $\overline{T}$ is definitely affected by $\overline{V}$. The variation of $\overline{T}$ depends on the properties of the workload such as task durations, the total number of tasks in the workload, and the number of tasks per job.

From these results we can conclude the following. Hist over-provisions quite

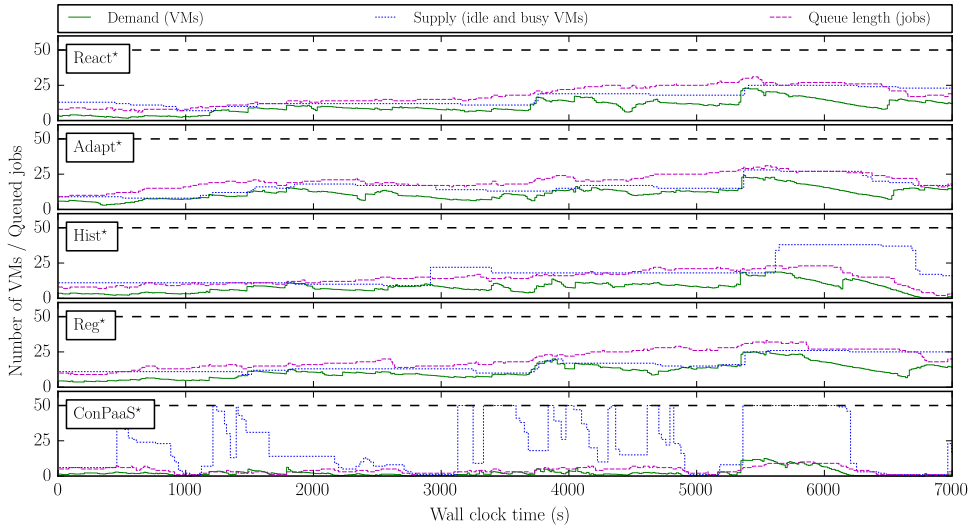


Figure 4.9: The experimental dynamics of all the five considered general autoscaling policies (Workload 1, service rate 14.0) during the cropped interval of 7,000 s. The horizontal dashed line indicates the resource limit of 50 VMs.

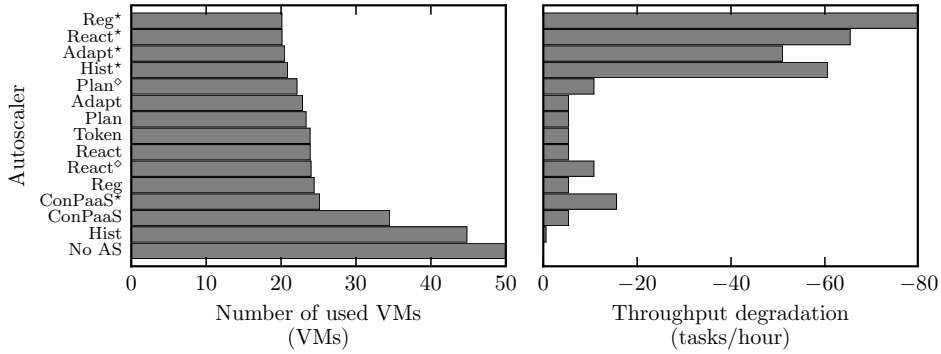


Figure 4.10: The average number of used VMs during the experiment and the average throughput degradation (compared with the no autoscaler case). All results are given for Workload 1.

a lot and achieves low throughput degradation. ConPaaS also over-provisions but the throughput is not much affected because its supply curve is very volatile. ConPaaS⋆ over-provisions less than ConPaaS as it "supposes" that the system needs less active VMs to process the same workload. Accordingly, the throughput degradation for ConPaaS⋆ is also bigger. Reg, React, and Adapt configured with service rate 1.0 as well as Token and Plan show almost similar results. Plan and Token policies show good balance between the number of used VMs and the throughput. Parametrization with the service rate of 14.0 decreases the performance by allocating less VMs. We can also see that longer booting VMs (React◇ and Plan◇) negatively affect the throughput.

4.5.5. Performance of Enforced Deadline-based SLAs

In this section, we analyze how enforced deadline-based SLAs perform for the considered autoscalers. We use Workload 1 and configure the general autoscalers with service rate 1.0. We look at two cases. In the first case, the deadlines for each workflow are set on per-workload basis, in the second case the deadlines for each workflow are calculated individually, thus set on per-workflow basis. In both cases, we use response times of workflows in a system without an autoscaler as the reference.

In the first case, for all the workflows we assign the same deadline which is based on the statistical characteristics of the whole workload. This approach imitates a situation where only high level statistics of the workload profile are available for calculating the deadlines. Although, in our case we use the whole workload, in other situations the size of the observed period (or the number of considered workflows) could be different and limited by the practicability, e.g., when the execution statistics are not available a priori and should be collected automatically on-the-fly [48]. We use three scenarios to assign per-workload-based deadlines:

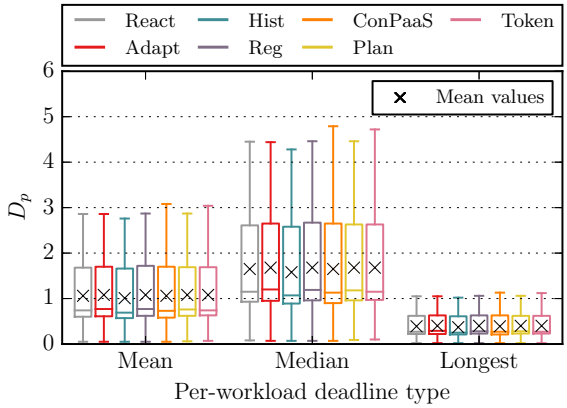
1. The longest response time of a workflow in the system without an autoscaler (1,247 s).
2. The mean response time of workflows in the system without an autoscaler (459 s).
3. The median response time of workflows in the system without an autoscaler (295 s).

Figure 4.11a shows deadline proximity ratios for all the three scenarios of enforced per-workload deadlines. Figure 4.11b shows the variability of deadline proximity ratios for small, medium, and large workflows for two selected autoscalers: React and Plan.

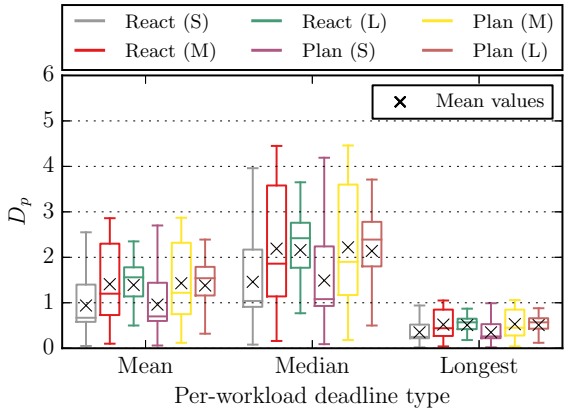
In the second case, we use per-workflow deadlines. Each workflow is assigned with a unique deadline which is equal to its response time in a system without an autoscaler. We vary the per-workflow deadlines by multiplying them by a certain error factor which we select from the following list of values [0.8, 0.9, 0.95, 1.0, 1.05, 1.1, 1.2, 1.3]. Note, that for the error factor 1.0 the deadline proximity ratio coincides with the elastic slowdown. By altering the deadlines, we can observe how the incorrectly set deadlines, due to inaccuracies in estimation methods [43, 63], affect deadline violations.

Figure 4.12 shows the variability of deadline proximity ratios for all the workflows in Workload 1. Figure 4.13 shows the variability of deadline proximity ratios between three considered groups of workflow sizes for React and Plan policies.

Varying per-workload deadlines using the similar approach as for per-workflow deadlines does not show significant difference in the results. Interestingly, Figure 4.11a shows low variability between different autoscalers within the same deadline type. The main reason is that the majority of jobs in Workload 1 are rather short. Thus, when the per-workload deadline is applied to each separate workflow, it leaves enough free space to allow the workflow to meet its deadline (despite the possible negative effects of elastic slowdown). Comparing different



(a) All autoscalers and all workflows.



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

Figure 4.11: The deadline proximity ratio for the three types of enforced per-workload deadlines for Workload 1.

scenarios of per-workload deadlines, we can definitely see that the per-workload deadline which is based on the longest response time, allows almost all workflows to finish before the deadline. The mean-based per-workload deadline also shows good results since mean deadline violation ratio is very close to 1. The median-based per-workload deadline is not the best solution as it increases deadline violations, shifts median deadline proximity ratio up to 1 and leads to higher variability in the deadline violations.

Comparing the deadline violations between React and Plan autoscalers in Figure 4.11b we can see that for the same job sizes, e.g., React (S) and Plan (S) deadline proximity ratios are almost identical. We can conclude that for per-workload deadlines the influence of autoscalers is less pronounced because each workflow has enough time to meet the deadline even despite the elastic slowdown.

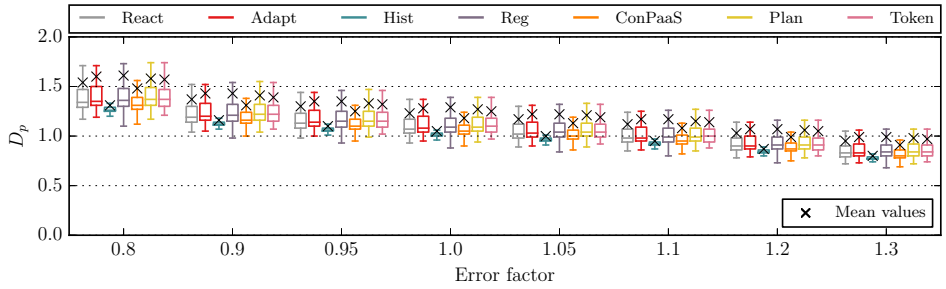


Figure 4.12: The effect of changing enforced per-workflow deadlines on the deadline proximity ratio (for all workflows in Workload 1).

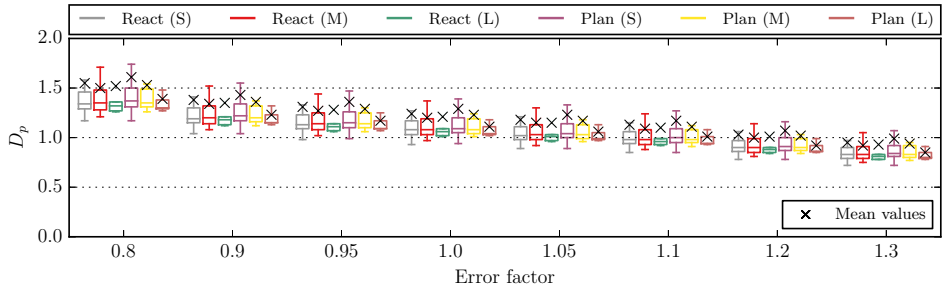


Figure 4.13: The effect of changing enforced per-workflow deadlines on the deadline proximity ratio for React and Plan autoscalers plotted separately for (S) small, (M) medium, and (L) large workflow sizes in Workload 1.

The general trend between different scenarios in Figure 4.11b is similar to Figure 4.11a, i.e., the Longest deadline scenario shows the best results and Median the worst. Table 4.6 additionally shows numerical variability characteristics for the selected per-workload and per-workflow scenarios, also for ($\diamond$) and ($\star$) autoscaler configurations.

Intuitively, the further the deadline, the lower the deadline violation ratio. Figure 4.12 confirms this proposition and indicates how varying per-workflow deadlines affects the deadline proximity ratio. Notably, the considered autoscalers show different variability in deadline proximity ratios. Comparing error factors 0.8 and 1.3, the variability increases when more deadlines are violated. However, the difference in variability of deadline proximity ratios between the autoscalers stays stable. For example, we can see that for all the error factors Adapt stably shows slightly higher variability in deadline violations and shows comparable to Reg mean deadline proximity ratios. Hist and ConPaaS which over-provision more exhibit better deadline proximity ratios.

Comparing Figure 4.13 with Figure 4.11b we can notice that with per-workload deadlines more medium and large jobs do not meet deadlines while with per-workflow deadlines more smaller jobs do not meet their deadlines. This is due to the method for calculating per-workflow deadlines. Workload 1 has the majority of

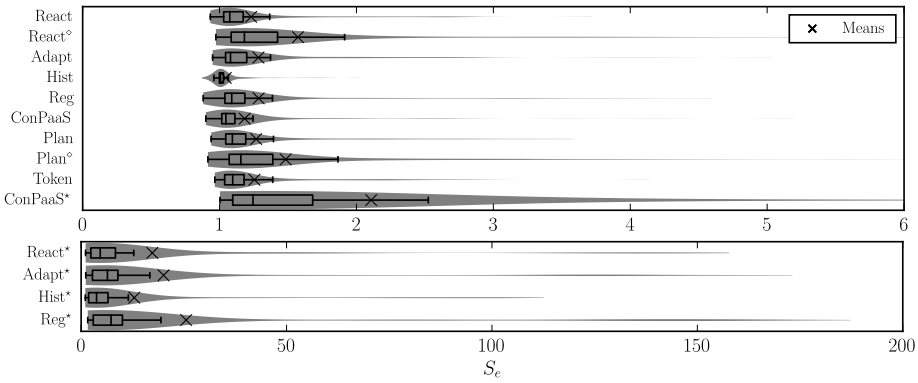


Figure 4.14: Variability of elastic slowdowns for all the workflows in Workload 1. Values in wider parts of shaded areas are more probable than those in narrower parts. The horizontal axes have different scales.

small workflows. Smaller workflows get smaller "safety margin" after multiplying their response time in a system without an autoscaler by the error factor.

4.6. Analysis of Performance Variability

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

4.6.1. Overall

Figure 4.14 shows variability of elastic slowdowns for all the workflows in Workload 1. From the shape of the violin plots we can conclude that all the distributions are long-tailed. For autoscalers configured with service rate 1.0 the distributions are unimodal, and for autoscalers React*, Adapt*, Hist*, and Reg* the distributions have second small peak on the right side of the S_e axis. Interestingly, heavy over-provisioning by Hist decreases elastic slowdown variability. Policies configured with service rate 14.0 show significantly higher variability and for this reason are plotted separately. ConPaaS, which over-provisions 2–3 times less than Hist, but still more than all the other autoscalers within the same configuration, shows variability comparable to e.g., React and Token. Increased VM booting time not only increases the mean value of the elastic slowdown but also doubles the variability. Adapt and Reg, while showing almost identical mean slowdowns, differ in the values which are below 1. Reg speeds up more workflows than Adapt does. The reason is probably related to the smoother downturns in the supply curve after demand decreases (Figure 4.8).

Additionally, in Table 4.6 we report numerical variability characteristics for all the policies with Workload 1: standard deviation (σ), skewness (γ_1), and kurtosis (γ_2) for S_e and for D_p for both deadline cases. For per-workload D_p we report only values for the scenario when the per-workload deadline is set based on the mean

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

Table 4.6: Additional *variability characteristics* of S_e and D_p metrics for per-workload and per-workflow cases for *Workload 1.* Best values in each column are highlighted in bold, except the No AS case. σ is standard deviation, γ_1 is skewness and γ_2 is kurtosis.

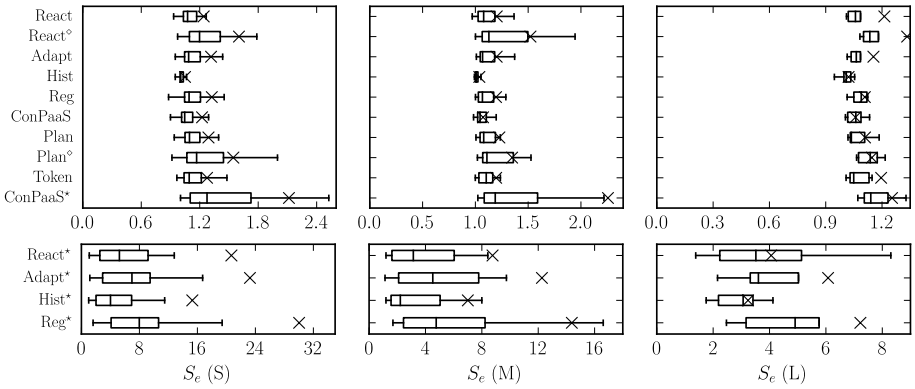


Figure 4.15: Variability of elastic slowdowns for (S) small, (M) medium, and (L) large workflow sizes in Workload 1. Means are marked with ×. The horizontal axes have different scales.

response time in a system without an autoscaler (Mean scenario, Section 4.5.5). For per-workflow D_p we present the situation when the error factor is 1.0. For both D_p cases we additionally report their mean values for all the policies with Workload 1. Note, that S_e means are reported in Table 4.4. Skewness allows us to see how symmetric the distribution is. The sign of skewness shows in which direction the distribution is tilted. Comparing slowdowns in Table 4.6 with Figure 4.14, we can observe that positive skewness basically means that the tail of the distribution is located on the right from its mean. Higher kurtosis shows that the distribution has infrequent extreme outliers. Negative kurtosis means that the distribution is more "flat" and has "thinner tails", e.g., a uniform distribution.

4.6.2. Performance Variability per Workflow Size

Figure 4.15 depicts variability of elastic slowdowns per size group for (S) small, (M) medium, and (L) large workflows in Workload 1. We do not show the distributions in Figure 4.15 as it is long-tailed and similar to Figure 4.14. To better show the interquartile ranges we do not plot the outliers.

For all the autoscalers configured with service rate 1.0 large workflows show the lowest elastic slowdown variability, while small workflows suffer more. This trend is similar to the differences in deadline violation ratios between the same groups of job sizes in Figure 4.11b and Figure 4.13.

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

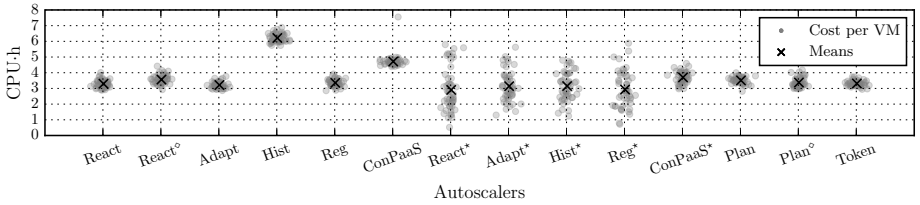


Figure 4.16: Accounted cost in CPU hours for all the considered autoscalers running Workload 1. For each policy, per VM costs are displayed with a slight random horizontal shift to avoid excessive superimposing.

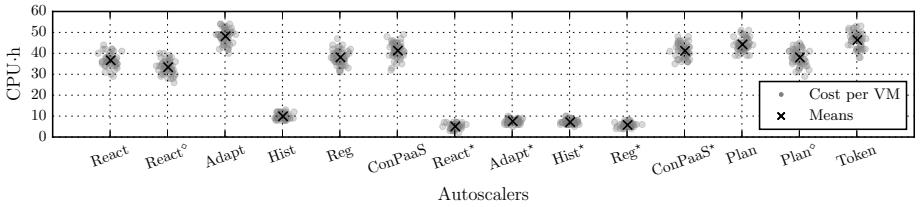


Figure 4.17: Charged cost in CPU hours for the autoscalers running Workload 1. For each policy, per VM costs are displayed with a slight random horizontal shift to avoid excessive superimposing.

4.7. Autoscaler Configuration and Charging Model

Configuring autoscalers in public clouds is key to cost savings. Understanding the pricing model and how an autoscaler can affect the total costs is important. To give an example, we compare accounted CPU hours, where the actual resource usage is paid for, and hourly-charged CPU hours that have a constant hourly, fixed-price scheme, e.g., like the one used by Amazon AWS for on-demand instances. Figure 4.16 shows accounted cost for each autoscaler and Figure 4.17 shows charged cost for each autoscaler.

Besides the average accounted and charged cost per VM, in Table 4.4 we analyze the accounted saving $\tilde{H}$ and charged saving $\tilde{C}$ in comparison to the scenario without autoscaling. While comparing average accounted CPU hours per VM $\overline{H}$ of the different autoscalers from Figure 4.16, it can be concluded that each autoscaling policy reduces the average accounted CPU hours per VM, i.e., the autoscalers e.g., React, Adapt, Reg, Plan, and Token have values between 3 and 4 CPU·h, whereas the no autoscaling scenario has 6.68 CPU·h. This result is more clear when we compute $\tilde{H}$. Here, the autoscalers Token and React use 2.02 times less VMs compared to the scenario where all VMs are running throughout the whole experiment. As the ($\star$) autoscalers adapt the system based on another resource demand, these autoscalers use less VMs than the others. Indeed, the achieved user metrics are worse compared to the other autoscalers.

Following the hourly pricing scheme, where the autoscalers start and stop VMs with no regard to the per-hour billing model, all autoscalers use more average charged CPU hours per VM $\overline{C}$ than the no autoscaling scenario (7 CPU·h), see Figure 4.17 or Table 4.4. This shows the danger of misconfiguring an autoscaler on

a public cloud. One can end up paying between 50% to 600% more than when not using an autoscaler at all.

4.8. Which Policy is the Best?

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

4.8.1. Pairwise Comparison

In this section, we rank the autoscalers using the pairwise comparison method [50]. In this method, for each algorithm we pairwise compare the value of each metric with the value of the same metric of all the other autoscalers. When a smaller value of a metric is better, for a pair of autoscalers A and B, autoscaler A accumulates one point if the value of this metric is lower than the value for autoscaler B. In case when bigger is better, autoscaler B gets the point. If both values are equal then both autoscalers get half point each. This method provides some ranking of the autoscalers, but its results do not fully capture the trade-offs as an autoscaler can get a point for being marginally better than another autoscaler in one metric, while being considerably worse than all of them in another metric.

We consider system metrics (a_U, a_O), (t_U, t_O), (k, k'), and user metrics S_e, the standard deviation of S_e, and $\overline{H}$. $\overline{T}$ and D_p are excluded since they correlate with S_e. We do not consider $\overline{a}_U$ and $\overline{a}_O$, as well as m_U due to redundancy with the selected accuracy metrics. The usage of $\overline{H}$ allows us to exclude $\overline{V}$ from the consideration. For all the metrics smaller value is better. The results of the comparison are given in Table 4.7. The bigger the number of points the better. The maximal number of points which each autoscaler can ever obtain is limited by the product of the number of considered metrics and the number of compared autoscalers.

4.8.2. Fractional Difference Comparison

In this section, we rank the autoscalers using the fractional difference method comparing all the autoscalers with an ideal case. For ideal case, we construct an empirical ideal system that achieves the best performance for all the metrics we consider. Note, this system does not exist in practice. Thus, the ideal system is a system which compiles all the optimal values for the considered metrics (here we use the same metrics as for Pairwise Comparison in Section 4.8.1), including the no autoscaler case. For each metric m_i, we compute its best value b_i which is

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

Table 4.7: The pairwise and fractional comparison, the aggregated elasticity and user metrics. The winners in each category (except No AS) are highlighted in bold.

either minimum or maximum value from the set of metric's values, depending on the metric (e.g., among our metrics only for $\overline{T}$ the biggest value is the best). For each autoscaler the score p for the metric j is computed as following:

$$p_j = \sum_{i=1}^{M} \frac{|m_i - b_i|}{\max(b_i, \varepsilon)}, \tag{4.15}$$

where M is the total number of considered metrics, and $\varepsilon > 0$, which is here set to $\varepsilon = 1$. The final score of an autoscaler is the average of all the individual p_j scores. The final score shows the fraction by which the autoscaler differs from the empirically established ideal system. Thus, the smaller the final score the better. The results of the comparison are given in Table 4.7.

4.8.3. Elasticity and User Metrics Scores

In this section, we aggregate elasticity and user metrics to scores as proposed by Fleming et al. [66]. As commonly done in the benchmarking domain, we first select a baseline as reference to compute metric ratios and then compute the averages of the metrics using an unweighted geometric mean. We choose the metric results with no active autoscaler as baseline. The unitless scores allow for a consistent ranking of autoscalers with 1 as a reference value. The larger the score, the better the rating. The scoring could be extended by user-defined weights.

For each autoscaler i we group the set of elasticity metrics based on the covered aspects into three groups: (i) accuracy as $a_i = a_{U,i} + a_{O,i}$, (ii) wrong provisioning timeshare $t_i = t_{U,i} + t_{O,i}$, and (iii) instability $\kappa_i = k_i + k'_i$. For the elasticity scores,

we do not consider $\overline{a}_U$ and $\overline{a}_O$, and m_U metrics to avoid redundancy in elasticity aspects. The user score comprises the average accounted CPU time of VM instances $\overline{H}_i$ and the elastic slowdown $S_{e,i}$ as metric ratios for each autoscaler i. The average throughput $\overline{T}$ is not considered as it is inversely dependent on the elastic slowdown S_e.

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

$$s_i = \left(\frac{a_b}{a_i} \cdot \frac{t_b}{t_i} \cdot \frac{\kappa_b}{\kappa_i} \right)^{1/3}, \tag{4.16}$$

$$u_i = \sqrt{\frac{\overline{H}_b}{\overline{H}_i} \cdot \frac{S_{e,b}}{S_{e,i}}}, \tag{4.17}$$

$$o_i = \sqrt{s_i \cdot u_i}. \tag{4.18}$$

The resulting ranking is presented in Table 4.7. Using the described metric aggregation approach and concerning the elasticity s_i and overall o_i scores, Plan outperforms the general autoscalers. The React policy shows the best results from a user perspective in u_i, while our Token policy and the general Adapt policy follow React with a small score difference of 0.01. Hist and ConPaaS perform slightly better than a system without an autoscaler in this context. Strong impact on the autoscalers has the service rate parameter ($\star$), a smaller impact can be observed for the experiments with longer provisioning time ($\diamond$).

4.9. Threats to Validity

The limitations of the study are mainly expressed in the constrained number of considered job types and autoscalers. Improvements can be achieved by adding extra workloads with different characteristics to ideally consider wider spectrum of major job types that benefit from autoscaling. For example, data analytics workflows, streaming workflow applications, and workflows requiring quick reaction time [161]. Additionally, it is possible to report the job slowdown per workflow type. To make the study more applicable to cloud environments, one can extend the set of workflow-related autoscalers with algorithms which consider job deadlines and costs [118, 47].

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

The experimental setup used in this chapter could also be improved. Despite the fact that our private OpenNebula environment is rather representative, the number of concurrent users in Amazon EC2 or Microsoft Azure is much higher than in our case. Thus, it would be beneficial to consider public clouds to capture possible performance effects which could arise there. In addition, avoiding interval-based autoscaling in real setups could improve the quality of predictions by reacting to changes in the demand more quickly. We parametrize general autoscalers (computed

service rate parameter) using the statistical properties of the whole workload as we have an access to this information. However, in the case when the workload properties are unknown different demand estimation methods can be used [147]. We do not analyze CPU utilization and RAM usage as for the considered workloads CPU and RAM information has low value as we primarily assign one task per VM and focus on performance characteristics from the perspective of job execution times.

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

4.10. Related Work

This chapter provides the first comprehensive comparative experimental study of autoscaling for workflows. We are unaware of any similar study in terms of the methodology taken, the number of policies compared, the number of performance metrics, and the size of experiments run. The importance of comparing different autoscaling algorithms has been recently discussed in the literature but mostly from a theoretical point of view [110, 133]. One exception is a tool that tries to utilize the differences between different autoscaling policies to achieve better QoS for customers by selecting a policy based on the workload [27]. That work, nevertheless, does not include any experimental comparison or deep analysis between the performance of the autoscalers as we do in this chapter.

The problem of scaling workflows has been studied in the literature with a focus on designing new autoscaling policies. Malawski et al. [115] discuss the scheduling problem of ensembles of scientific workflows in clouds while considering cost- and deadline-constraints. Mao et al. [119] optimize the performance of cloud workflows within budget constraints. They propose two algorithms, namely, Scheduling-First and Scaling-First. Cushing et al. [47] deal with prediction-based autoscaling of scientific data-centric workflows. Buyn et al. [40] try to achieve cost-optimized provisioning of elastic resources for workflow applications. The authors use the Balanced Time Scheduling (BTS) algorithm to calculate the minimal required number of resources which will allow to execute the workflow within a given deadline. Dörnemann et al. [60] consider scheduling of Business Process Execution Language (BPEL) workflows in the Amazon EC2 cloud. Their main findings include the methods to automatically schedule workflow tasks to underutilized hosts and to provide additional hosts in peak situations. The proposed load balancer uses the overall system load to take scaling decisions in contrast to other systems where

the throughput is more important. Heinis et al. [71] propose a design and evaluate the performance of a workflow execution engine with self-tuning capabilities.

4.11. Conclusion

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

5

Performance-Feedback Autoscaling for Workloads of Workflows

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.

5.1. Introduction

Workflows have been introduced to cloud workloads over a decade ago [53]. Today, the variety of workflow structures observed in modern cloud workloads and the diversity of cloud resource types require much more sophisticated resource

management and scheduling techniques [167, 52, 6]. *Autoscalers* [118] must not only allocate resources but also deallocate them, delivering results in time without resource waste and without exceeding preallocated budgets. They must further operate online, fulfilling dynamic requirements for multiple types of resources from a stream of multi-user requests. Meeting such demanding and dynamic Service-Level Agreements (SLAs) is not trivial, and poses important challenges to traditional single-workflow [101, 155] and multi-workflow [170, 24] schedulers, offline autoscalers for workflow ensembles [116, 33, 164], and online autoscalers [40, 119]. Online autoscalers have better performance scalability, but currently they often produce low quality demand estimations and thus could waste cloud resources [119], as we showed earlier in Chapter 4. Offline approaches use sophisticated planning techniques to produce high quality schedules and demand estimations, but lack the performance scalability needed to operate online in cloud settings [160, 82]. In this chapter, we aim to combine the qualities and avoid the drawbacks of previous approaches. To this end, we design and evaluate experimentally the Performance-Feedback Autoscaler (PFA), which we aim to make highly scalable while producing high-quality estimations.

Previously, the problem of autoscaling for workflows has been seen often from the perspective of just a single user who submits to the cloud a *batch* (an *ensemble*) of workflows. Usually, the workflows in the batch have previously known task runtimes, or good estimates obtained through code analysis, simulation, or from running on a reference system. This approach has been successfully adopted for executing batches of scientific workflows [155, 170, 116], which are well-studied [92] and have rather fixed patterns of execution [16], but can be too rigid for workflows in non-scientific domains [165]. Moreover, task runtime estimates have not been shown to be robust for batches in cloud settings, e.g., under multi-tenancy effects [91] and performance variability [120].

A more general approach assumes that the user submits a *workload* of workflows of different types as, for example, if the user runs an application serving many other, diverse customers. Many cloud-based services, such as Airbnb (rentals), Twitter (communication), and Netflix (video streaming), use this approach [131].

Autoscaling workloads of workflows is fundamentally different from autoscaling batches of workflows. Normally, for both the cloud user has a budget, to be used to run the workflows before their individual deadline. This is challenging because cloud resources are heterogeneous, have different costs, and act as performance black boxes. But autoscaling workloads does not typically aim to minimize the average makespan, as autoscaling batches typically does. Because, in a workload, workflows arrive in the system dynamically as a stream, it is important to minimize the workflow *slowdown*, as this leads to predictable service performance. Further, to keep their service sustainable under varying workload demand, cloud users are very interested to reduce resource waste and thus reduce operational costs.

Many existing autoscalers for workflows operate *offline* [116, 33, 164]. Given a batch of workflows, they create full autoscaling and task placement plans, which are then executed by the workflow management system. *Online* autoscalers, for when workflows stream over time forming a workload, are relatively rare [40, 119]; the few online autoscalers mostly use a *plan-based* approach, as they create a partial

plan to follow during an entire *autoscaling interval*. Although online plan-based approaches have showed promise in the past decade, we know now that they lead to *unscalable* time complexity when applied to workloads of workflows [160].

Autoscalers that do not scale well with the workload can negatively affect the stability of the system. Shorter autoscaling intervals, that are more in line with the current trend of fine-grained billing [140], further complicate the problem, leaving even less time for making autoscaling decisions. Moreover, plan-driven task placement may slow down the execution of newly arrived workflows, as new tasks wait to be added to the plan. In this case, the plan-based, computationally intensive autoscalers are not beneficial and should be substituted by simpler and faster, dynamic approaches.

We envision further improving autoscaling for workloads of workflows by combining concepts inherent to general autoscalers [26] and to workflow-aware autoscalers [119]. From general autoscalers, we aim to adopt the *principle of performance-feedback*, to derive and analyze runtime statistics during the execution. For example, instead of deriving task runtime estimates [43], we can use task throughput, which is easier to observe. From workflow-aware autoscalers, we can adopt *token-based techniques for estimating the expected resource demand*, which are less computationally intensive (Chapters 2 and 4). Overall, the main research questions addressed in this chapter, and our contributions toward answering them, are:

1. *How to minimize workflow slowdowns within the budget constraint with unknown in advance task runtime estimates when autoscaling cloud resources for workloads of workflows?* We propose in Section 5.3.3 a novel, online, dynamic Performance-Feedback Autoscaler (PFA) that uses the resource task throughput information and a token-based estimator.
2. *Does the autoscaling policy found when answering Question 1 has lower time complexity than the state-of-the-art plan-based online autoscalers?* In Section 5.5, through real-world experiments, we show that PFA answers this question favourably by outperforming two state-of-the-art, plan-based, online autoscalers.
3. *How far is the performance and scalability of the policy found in Question 1 from the optimal solution?* We compare in Section 5.6 all the considered autoscalers with the optimal solution obtained from a Mixed Integer Programming model.

5.2. 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.

5.2.1. Autoscaling Model

We consider a public cloud computing system which is a subject to an arriving workload of workflows. The workflow model is the same as in the previous chapters.

The workload consists of multiple independent sub-workloads each belonging to an independent user.

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 a per-user basis. The separate *scheduler* is responsible for placing tasks onto the allocated resources. In this chapter, as well as in Chapter 4, 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.

5.2.2. Performance Metrics

The system is constantly monitored by its users and operators, who assess its performance for a set of metrics commonly used in autoscaling settings. In this chapter, we mostly rely on metrics defined earlier in previous chapters.

User- and System-Oriented Metrics

As a main user-oriented metric we use *slowdown* which is defined in the same way and using the same notations as in Section 3.2.2. We also consider the *monetary cost per autoscaling interval* as a user-oriented metric. By monetary cost we understand the total cost of 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*.

Elasticity-Oriented Metrics

To evaluate the performance of the considered autoscalers, we take the elasticity into account. We extend the system model from Chapter 4, where we allow each resource to run only a single workflow task at a time. Thus, we rely on the same demand and supply definitions as in Section 4.3, and use the same under- and over-provisioning accuracy metrics a_U, a_O, and under- and over-provisioning timeshare metrics t_U, t_O. However, since in this chapter we consider multiple users, the supply and demand are additionally defined per user. From the resource heterogeneity perspective, we do not define the demand per resource type, as we suppose that task resource preferences are unknown a priori to the scheduler.

5.3. 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 [119] and designed specifically for workloads of workflows. The relevance of these autoscalers is supported by the recent survey [111].

5.3.1. Planning-First Autoscaler

The Planning First (PLF) [119] 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:

1. Distribute the user budget among the workflows based on their priority.
2. Perform initial supply prediction by determining the number of each resource type to allocate within the budget constraint.
3. Consolidate the budget left after the initial supply prediction.
4. Allocate the resources according to the predicted supply.
5. Create an execution plan for the upcoming autoscaling interval.
6. 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 resource 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 demand estimation. 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 allocates the resources according to the predicted supply.

In the fifth step, the policy performs so-called resource consolidation which basically means the creation of an execution plan on the already allocated (at the moment of the autoscaler invocation) and newly allocated resources (after the fourth 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.

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.

As the original paper [119] relies on simulations, many very important details, that are crucial when implementing the policy in a real system, are missing or imprecise. Further, we provide our interpretation of the resource consolidation step. When constructing the execution plan, PLF processes workflows in a 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.

5.3.2. Scaling-First Autoscaler

The Scaling First (SCF) [119] 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:

1. Perform initial supply prediction by creating a per-workflow execution plan without limiting the number of resources.
2. Scale the initial prediction to fit within the budget constraint, and consolidate the remaining budget.
3. Allocate the resources according to the predicted supply.
4. Create an execution plan for the upcoming autoscaling interval.
5. 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 explain 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, which for SCF also means the creation of an execution plan on the allocated resources for the upcoming autoscaling interval. 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 the 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.

5.3.3. 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. We based the idea of PFA on observations on the performance of general and workflow-specific autoscalers from the literature [116, 160] and on our own observations from the previous chapters.

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 six steps:

1. Determine the resource type ratios from historical throughputs.
2. Determine the number of resources (the supply) that can be allocated within the user budget with the obtained ratios.
3. Estimate future resource demand using the token propagation approach and historical throughput information.
4. Scale down or inflate the profile-based supply to match the token-predicted demand.
5. Allocate the predicted number of resources.
6. Deallocate idle resources that are staying idle the longest and approaching the end of their billing period.

We detail steps 1–6 of PFA in the remainder of this section. Table 5.1 summarizes all the symbols used in the description of the autoscaler. To explain the algorithms employed by each PFA stage, we use a symbolic notation rather than pseudocode, because pseudocode descriptions are known to under-specify (obscure) explanations to the point where schedulers cannot be reproduced in practice [122].

	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 total 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

Table 5.1: Symbols used for the PFA autoscaler.

Determining the Resource Profile

The first two steps of the autoscaler, based on historical task throughputs and user budgets, derive for each user the initial resource profile: the resource type ratio and the number of resources of each type. PFA relies on two alternative mechanisms for smoothing out possible short-term throughput fluctuations: Moving Average (MA) and Exponentially Weighted Moving Average (EWMA).

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

$$\tau_{i,j}(t) = \begin{cases} \frac{c_{i,j}(t)}{n_{i,j}(t)}, & \text{if } n_{i,j}(t) \neq 0, \\ 0, & \text{otherwise,} \end{cases} \tag{5.1}$$

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:

$$\hat{\rho}_{i,j}(t) = \begin{cases} \frac{\tau_{i,j}(t)}{\sum_{r \in \mathcal{R}} \tau_{r,j}(t)}, & \text{if } \sum_{r \in \mathcal{R}} \tau_{r,j}(t) \neq 0, \\ 0, & \text{otherwise,} \end{cases} \tag{5.2}$$

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:

$$P_{i,j}(t) = \left\{ \hat{\rho}_{i,j}(t-k) : \sum_{r \in \mathcal{R}} \hat{\rho}_{r,j}(t-k) > 0, \forall k \in [0, m] \right\}. \tag{5.3}$$

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

$$\rho_{i,j}(t) = \begin{cases} \frac{1}{|P_{i,j}(t)|} \cdot \sum\limits_{k \in P_{i,j}(t)} k, & \text{if } \sum\limits_{k \in P_{i,j}(t)} k > 0, \\ \frac{1}{|\mathcal{R}|}, & \text{otherwise,} \end{cases} \tag{5.4}$$

where $|\mathcal{X}|$ denotes the cardinality of a set $\mathcal{X}$. If any resource has zero historical throughput then 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:

$$\rho_{i,j}(t) = \begin{cases} \alpha \cdot \rho_{i,j}(t-1) + (1-\alpha) \cdot \hat{\rho}_{i,j}(t), & \text{if } \hat{\rho}_{r,j}(t) > 0, \forall r \in \mathcal{R}, \\ \frac{1}{|\mathcal{R}|}, & \text{otherwise,} \end{cases} \tag{5.5}$$

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:

$$\nu_{i,j}(t) = \frac{q_i \cdot \rho_{i,j}(t)}{\sum\limits_{r \in \mathcal{R}} \big(q_r \cdot \rho_{r,j}(t)\big)}. \tag{5.6}$$

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:

$$\hat{\mu}_{i,j}(t) = \left\lfloor \frac{b_j \cdot \nu_{i,j}(t)}{q_i} \right\rfloor, \tag{5.7}$$

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:

$$\tilde{\mu}_j(t) = \sum_{r \in \mathcal{R}} \hat{\mu}_{r,j}(t). \tag{5.8}$$

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 Sections 2.3.1 and 4.4.2. 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 have 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 only 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 gives the approximated LoP $\lambda_j(t)$, and the total number of token movements $\theta_j(t)$ gives the total 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:

$$T_j(t) = \left\{ \tau_{i,j}(t-k) : \sum_{r \in \mathcal{R}} \tau_{r,j}(t-k) > 0, \forall k \in [0,m], \forall i \in \mathcal{R} \right\}, \tag{5.9}$$

With MA, the TBA lookup depth is computed as:

$$\zeta_j(t) = \begin{cases} \left\lceil \frac{1}{|T_j(t)|} \cdot \sum\limits_{k \in T_j(t)} k \right\rceil, & \text{if } \sum\limits_{k \in T_j(t)} k > 0, \\ \infty, & \text{otherwise,} \end{cases} \tag{5.10}$$

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:

$$\sigma_j(t) = \begin{cases} \left\lceil \theta_j(t) \cdot |T_j(t)| \cdot \left(\sum_{k \in T_j(t)} k \right)^{-1} \right\rceil, & \text{if } \sum\limits_{k \in T_j(t)} k > 0, \\ \lambda_j(t), & \text{otherwise.} \end{cases} \tag{5.11}$$

With EWMA, the TBA lookup depth is computed as:

$$\zeta_j(t) = \begin{cases} \left\lceil \alpha \cdot \zeta_j(t-1) + (1-\alpha) \cdot \frac{\sum_{r \in \mathcal{R}} \tau_{r,j}(t)}{|\mathcal{R}|} \right\rceil, & \text{if } \sum\limits_{r \in \mathcal{R}} \tau_{r,j}(t) > 0, \\ \infty, & \text{otherwise.} \end{cases} \tag{5.12}$$

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

$$\sigma_j(t) = \begin{cases} \left\lceil \theta_j(t) \cdot |\mathcal{R}| \cdot \left(\sum\limits_{r \in \mathcal{R}} \tau_{r,j}(t) \right)^{-1} \right\rceil, & \text{if } \sum\limits_{r \in \mathcal{R}} \tau_{r,j}(t) > 0, \\ \lambda_j(t), & \text{otherwise.} \end{cases} \tag{5.13}$$

Scaling Down or Inflating the Profile
In the fourth step, we scale down or inflate, if necessary, the profile-based resource supply to match the predicted resource demand $\sigma_j(t)$ and produce corrected values $\mu_{i,j}(t)$. 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 $\hat{\mu}_{i,j}(t)$ to produce the corrected values $\mu_{i,j}(t)$:

$$\mu_{i,j}(t) = \left\lceil \frac{\sigma_j(t)}{\tilde{\mu}_j(t)} \cdot \hat{\mu}_{i,j}(t) \right\rceil . \tag{5.14}$$

However, if $\tilde{\mu}_j(t)$ is lower than the token-predicted demand $\sigma_j(t)$, we instead inflate the resource profile 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 corrected $\mu_{i,j}(t)$ 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 increase 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.

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. When de-allocating the idle resources, PFA gives priority to those that approach the end of their billing interval.

Task Placement
The PFA autoscaler can operate with various independent task placement policies. Both 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. Thus, 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.

5.4. Experiment Setup

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

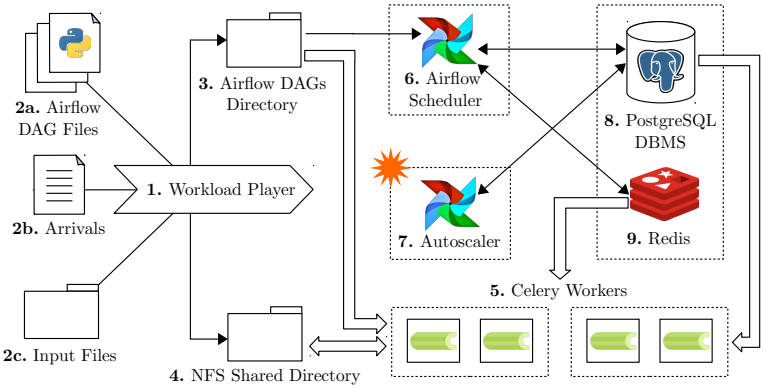


Figure 5.1: System architecture.

5

5.4.1. Apache Airflow Deployment and Configuration

Our setup is based on the Apache Airflow [2] workflow management system (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 [11]. The architecture of our system is presented in Figure 5.1. All the components of the system are deployed on the DAS-4 supercomputer [35] with the following characteristics. Head node: Intel Xeon X5650 @ 2.67 GHz CPU, 49 GB RAM, 18 TB HDD. 32 compute nodes: Intel Xeon E5620 @ 2.40 GHz CPU, 24 GB RAM, 2 TB 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 5.1) 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 a 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 of 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 [7] distributed task queue (version 4.1.1) with Redis [18] (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.

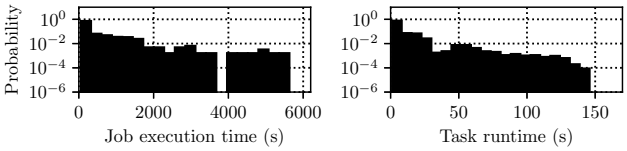
The Airflow Scheduler (Component 6) is responsible for placing eligible tasks for execution to the resources (Celery workers). The default Airflow scheduler is an online dynamic scheduler as it simply sends the eligible tasks in the order of their priority to the single Celery queue which is monitored 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 a 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.

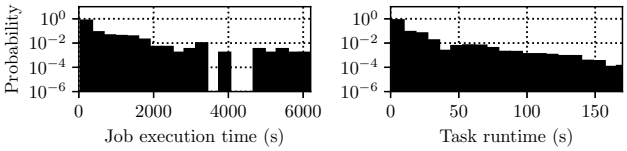
The Workload Player, Airflow Scheduler, and Autoscaler are running on individual worker nodes. The PostgreSQL database management system is co-located with the Redis in-memory database on the head node.

5.4.2. Billing Setup

We configure the system with two resource types: Small and Large, that have different costs. Further, we use the generic currency sign ¤ when referring to monetary costs. An instance of the Small resource type costs 1¤ per billing interval, while a Large instance costs 5¤ per billing interval. The system is configured to



(a) Workload I.



(b) Workload II.

Figure 5.2: Statistical characteristics of the workloads. The vertical axes have a log scale.

5

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 [140]. 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.

5.4.3. 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 [90] is used to build a mosaic image of the sky on the basis of smaller images obtained from different telescopes. LIGO [22] is used by the Laser Interferometer Gravitational-Wave Observatory (LIGO) to detect gravitational waves. SIPHT [108] 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 [16, 38]. 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

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 5.2: Characteristics of Workloads I and II.

nearest integer. We guarantee that the minimal task runtime is 1 second and the minimal files size is 1 KB. 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 5.2 presents task runtime and job runtime distributions of the workload. The details of each workload are summarized in Table 5.2.

5.5. Experiment Results

In this section, we present our experiment 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 5.3. The full set of software and computational artifacts used to obtain the presented results is available online [81, 80]. Our results show that our PFA

Section	Budget Configuration	PFA Configuration	WL
§ 5.5.2	eq. 60¤, 80¤, 100¤, 120¤; diff. 120¤ & 80¤	$m = 10, 20, 30$; $\alpha = 0.7, 0.8, 0.9$	I
§ 5.5.3	eq. 60¤, 100¤, 120¤; diff. 120¤ & 80¤	$m = 10$; $\alpha = 0.7$	I
§ 5.5.4	eq. 60¤, 100¤, 120¤; diff. 120¤ & 80¤	$m = 10$; $\alpha = 0.7$	I
§ 5.5.5	eq. 120¤	$m = 10$	I
§ 5.5.2	eq. 100¤	$m = 10$; $\alpha = 0.7$	II

Table 5.3: Experiment configurations.

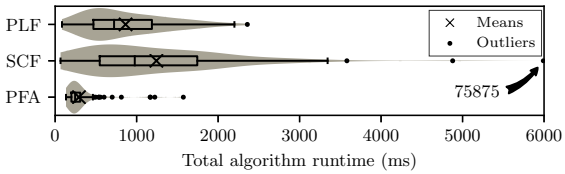


Figure 5.3: Variability of total runtimes for all the considered autoscalers.

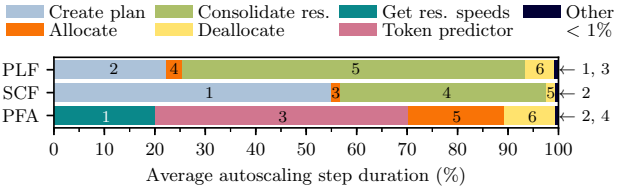


Figure 5.4: Average duration of each algorithm step within the total algorithm runtime for each considered autoscaler. The superimposed numbers represent corresponding algorithm steps.

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.

5.5.1. Algorithm Performance

Figure 5.3 shows the variability of total algorithm runtimes executed at every autoscaler invocation. The runtime data were collected using the setup described in Section 5.4. 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 workflow management system during sudden demand surges.

Figure 5.4 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.

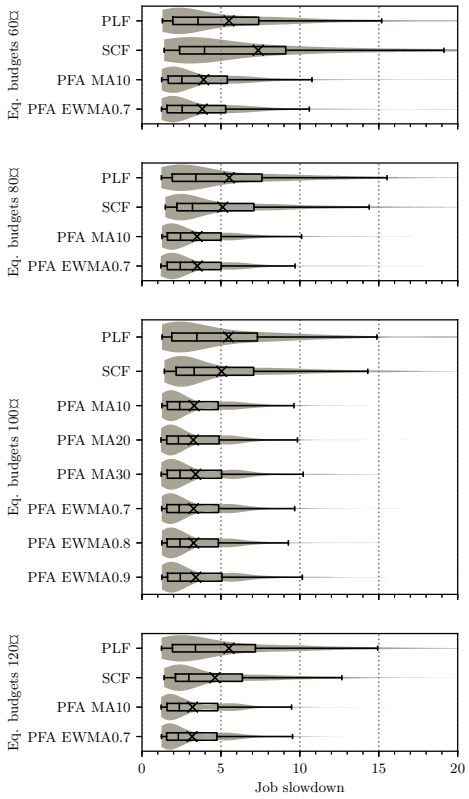


Figure 5.5: Variability of job slowdowns for all the studied autoscalers with equal budgets of 60¤, 80¤, 100¤, and 120¤ when running WL I. PFA autoscaler was executed with different smoothing methods. Means are marked with ×.

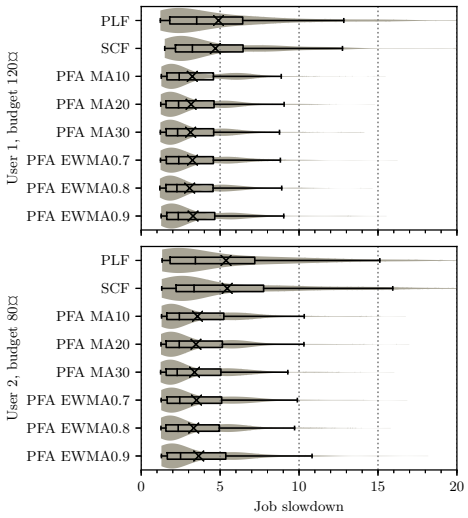


Figure 5.6: Variability of job slowdowns for all the studied autoscalers with different budgets for each user when running WL I. PFA autoscaler was executed with different smoothing methods. Means are marked with ×.

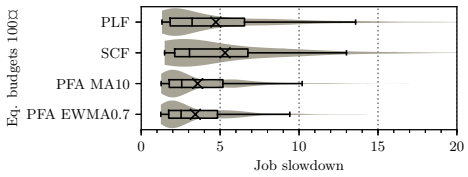


Figure 5.7: Variability of job slowdowns for all the studied autoscalers with equal budgets for both users when running WL II. PFA autoscaler was executed with MA depth 10 and EWMA with pole value 0.7. Means are marked with ×.

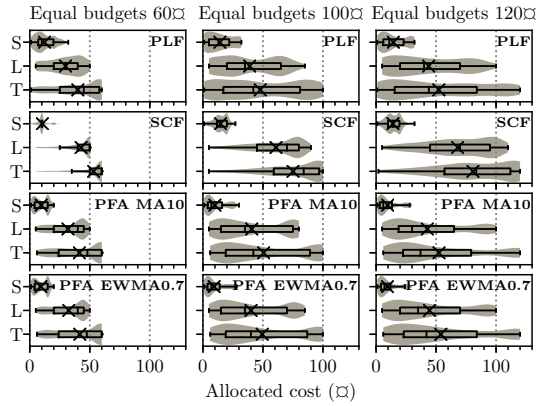


Figure 5.8: Variability of monetary cost for allocated resources per billing interval for User 1 for the studied autoscalers with equal budgets of 60¤, 100¤, and 120¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

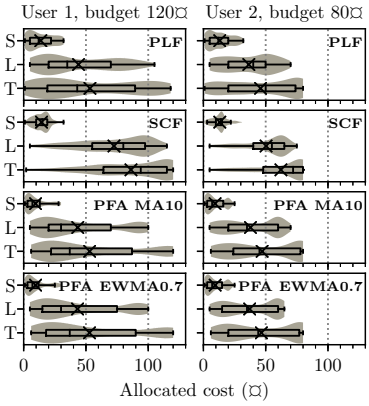


Figure 5.9: Variability of monetary cost for allocated resources per billing interval for each user for the studied autoscalers with different budgets when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

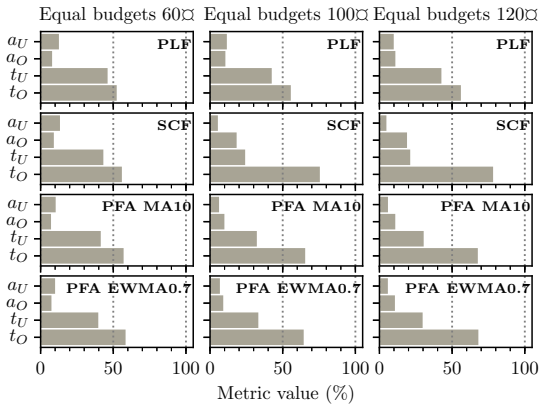


Figure 5.10: Elasticity metrics for User 1 for the studied autoscalers with equal budgets of 60¤, 100¤, 120¤ when running WL I. For all the metrics lower values are better.

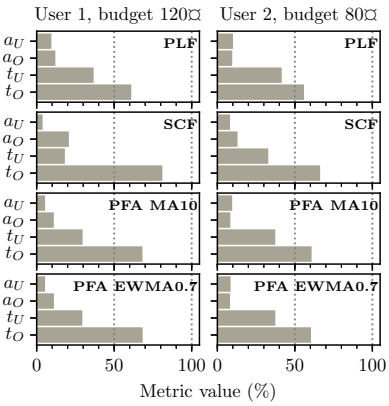


Figure 5.11: Elasticity metrics for each user for the studied autoscalers with different budgets 120¤ and 80¤ when running WL I. For all the metrics lower values are better.

5.5.2. Workload Performance

Further, we analyze the job slowdowns to investigate how the considered autoscalers affect the workload performance from the end-user perspective. Figure 5.5 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 budgets 100¤ as a baseline, as then each user can at maximum allocate 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 5.6 shows job slowdown variability when 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 5.7 presents job slowdowns for the configuration with equal budgets 100¤ for Workload II. The observed trend is the same as in Figures 5.5 and 5.6. The tasks in WL II on average run faster on the Large resource type than on the Small resource type (see Table 5.2). Thus, we can conclude, that all the considered autoscalers can successfully operate with workloads where tasks "prefer" a specific resource type.

From Figures 5.8 and 5.9 we can see that no autoscaler exceeds 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 5.8 we omit the results with the equal budgets 80¤, and from Figure 5.9 we omit the results for PFA with the configurations MA $m = 20, 30$, and $\alpha = 0.8, 0.9$.

5.5.3. Elasticity Performance

Figure 5.10 shows the considered elasticity metrics for the configurations with equal budgets 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 5.11 shows elasticity metrics for both users 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.

In Figure 5.10, 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 equal budgets 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 equal budgets 60¤, SCF spends less time over-provisioning, but still shows on average the worst over-provisioning accuracy of 8.5%.

In contrast, PLF with equal budgets 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 equal budgets 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 equal budgets 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- or over-provisioning with lower accuracy. The same trends can be observed for the configuration with different budgets in Figure 5.11. 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 workloads of workflows than the plan-based autoscalers.

5.5.4. 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 show how effectively the resources are utilized, and how many resources are actually allocated. Figure 5.12 presents the percentage of busy resources for the configurations with equal budgets of 60¤, 100¤, and 120¤ for User 1. Figure 5.14 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 5.13 shows the percentage of busy resources for both users for the configuration with different budgets of 120¤ and 80¤ for User 1 and User 2, accordingly. Figure 5.15 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 5.5.2. 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.

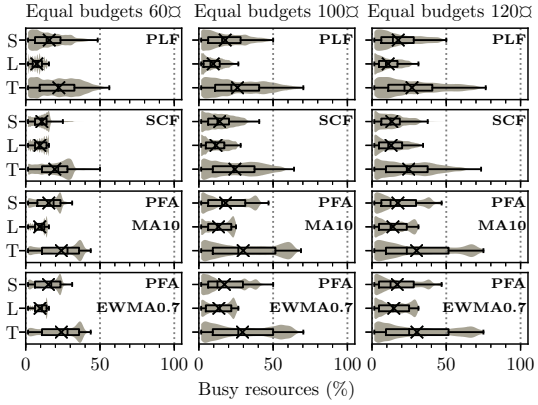


Figure 5.12: Variability of busy resources for User 1 for the studied autoscalers with equal budgets of 60¤, 100¤, 120¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

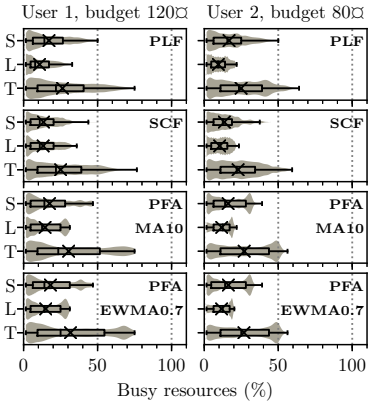


Figure 5.13: Variability of busy resources for each user with different budgets 120¤ and 80¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

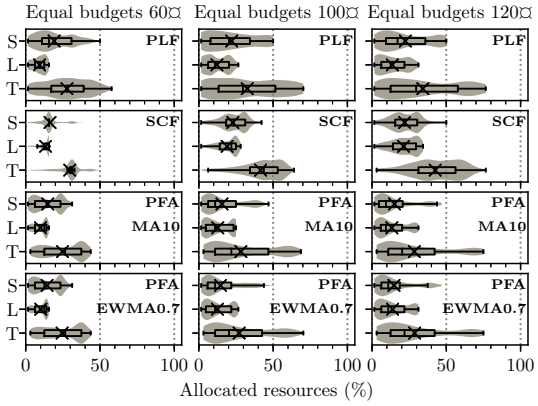


Figure 5.14: Variability of allocated resources for User 1 for the studied autoscalers with equal budgets of 60¤, 100¤, 120¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

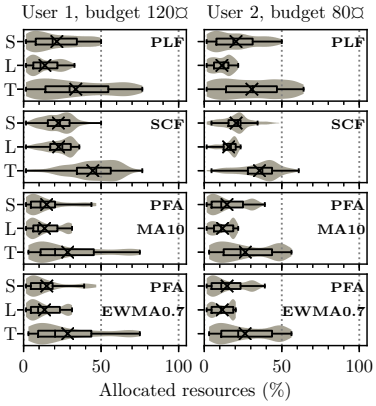


Figure 5.15: Variability of allocated resources for each user with different budgets 120¤ and 80¤ when running WL I. For Small and Large resource types, and in Total. Means are marked with ×.

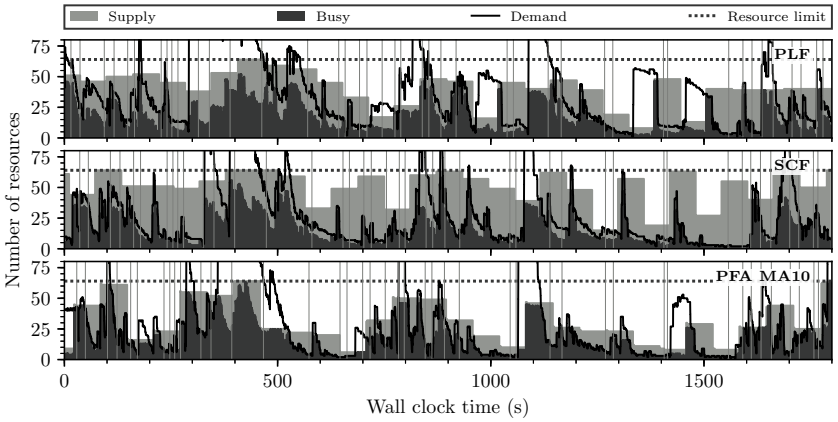


Figure 5.16: The dynamics of autoscaling on a cropped interval of 1,800 seconds for both users with equal budgets 120¤. Vertical lines indicate workflow arrivals.

5

5.5.5. Autoscaling Dynamics

We further study the dynamics of the obtained Airflow traces to better understand the performance differences between the autoscalers. Figure 5.16 shows the snapshots of autoscaling dynamics on a cropped interval of 1,800 seconds for both users with equal budgets 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 1,350 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., the spike at the time around 1,425 seconds which is, however, 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. This is exactly what makes the autoscaling for workflows that challenging. 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 5.3. That is why, to minimize the possible effect of such drifting, we run three independent subsets of workflows within the workload.

$\mathcal{T}$	The set of time slots $\mathcal{T} = \{1, 2, \ldots, T\}$
$\mathcal{W}$	The set of workflows $\mathcal{W} = \{1, 2, \ldots, W\}$
$\mathcal{S}$	The set of workflows tasks $\mathcal{S} = \{1, 2, \ldots, S\}$
$\mathcal{M}$	The set of billing intervals $\mathcal{M} = \{1, 2, \ldots, M\}$
$\mathcal{V}$	The set of resources $\mathcal{V} = \{1, 2, \ldots, V\}$
L	Number of time slots per billing interval
B	Budget per billing interval
A_w	Arrival time of workflow w
C_w	Critical path length 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
$x^t_{j,k}$	Binary variable, equals one if task j starts at time t on resource k
y^m_k	Binary variable, equals one if resource k is active on billing interval m
z^m_k	Integer variable, determines the number of active time slots for resource k on billing interval m
u^t_w	Binary variable, equals one if workflow w finishes at time t

Table 5.4: Symbols used for the MIP model.

5.6. 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. [164] to incorporate budget constraints, while following the similar notation, and implement the model in the Gurobi [14] solver (v. 8.0.1). The symbols used for describing the MIP model are presented in Table 5.4. 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.

5.6.1. Mixed Integer Programming Model

The MIP model presents time as a set $\mathcal{T} = \{1, 2, \ldots, T\}$ of discrete time slots of equal duration, where T is the furthest time horizon. The time slots are grouped into M billing intervals. Each billing interval consists of L time slots. The set of billing intervals is denoted by $\mathcal{M} = \{1, 2, \ldots, M\}$, 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 $\mathcal{W} = \{1, 2, \ldots, W\}$, where each workflow contains tasks. All the tasks in all the workflows are represented by the set $\mathcal{S} = \{1, 2, \ldots, S\}$, 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 C_w of its critical path, and earliest possible completion time D_w, so that $D_w = A_w + C_w - 1$. The model also defines a set of computing resources $\mathcal{V} = \{1, 2, \ldots, V\}$.

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 $x^t_{j,k}$, where $x^t_{j,k} = 1$ if task i is scheduled to run on resource k starting at time slot t, and $x^t_{j,k} = 0$ otherwise. Each task should start only once, which we specify as follows:

$$\sum_{k \in \mathcal{V}} \sum_{t \in \mathcal{T}} x^t_{j,k} = 1, \forall j \in \mathcal{S}. \tag{5.15}$$

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

$$\sum_{t=(m-1)\cdot L+1}^{m \cdot L} \sum_{j \in S} \sum_{r=\max(1,t-R_{j,k}+1)}^{t} x^r_{j,k} = z^m_k, \forall k \in \mathcal{V}, \forall m \in \mathcal{M}. \tag{5.16}$$

Let the binary variable y^m_k denote the active/idle state of each resource k on billing interval m, with $y^m_k = 1$ if some tasks are assigned on the resource, and $y^m_k = 0$ 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:

$$y^m_k = \min(1, z^m_k), \forall k \in \mathcal{V}, \forall m \in \mathcal{M}. \tag{5.17}$$

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:

$$\sum_{j \in \mathcal{S}} \sum_{r=\max(1,t-R_{j,k}+1)}^{t} x^r_{j,k} \leq 1, \forall k \in \mathcal{V}, \forall t \in \mathcal{T}. \tag{5.18}$$

The precedence constraints are formulated as follows:

$$\Big(\sum_{k \in \mathcal{V}} \sum_{t \in \mathcal{T}} t \cdot x^t_{i,k} - \sum_{k \in \mathcal{V}} \sum_{t \in \mathcal{T}} (t + R_{j,k}) \cdot x^t_{j,k} \Big) \cdot l_{i,j} \geq 0, \quad \forall i, j \in \mathcal{S}. \tag{5.19}$$

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

$$\sum_{k \in \mathcal{V}} \sum_{t \in \mathcal{T}} t \cdot x^t_{j,k} \geq A_w, \forall w \in \mathcal{W}, \forall j \in w. \tag{5.20}$$

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:

$$h_w(t) = \begin{cases} 1, & \text{if } t \leq D_w, \\ D_w - t, & \text{otherwise.} \end{cases} \tag{5.21}$$

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:

$$\sum_{t \in \mathcal{T}} u_w^t = 1, \forall w \in \mathcal{W}. \tag{5.22}$$

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

$$\sum_{k \in \mathcal{V}} \sum_{t \in \mathcal{T}} (t + R_{j,k} - 1) \cdot x_{j,k}^t \leq \sum_{t \in \mathcal{T}} t \cdot u_w^t, \forall w \in \mathcal{W}, \forall j \in w. \tag{5.23}$$

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

$$\sum_{k \in \mathcal{V}} P_k \cdot y_k^m \leq B, \forall m \in \mathcal{M}. \tag{5.24}$$

Finally, we can formulate the profit maximization objective:

$$\begin{gathered} \max \sum_{w \in \mathcal{W}} \sum_{t \in \mathcal{T}} h_w(t) \cdot u_w^t \\ \text{s.t. } (5.15)(5.16)(5.17)(5.18)(5.19)(5.20)(5.22)(5.23)(5.24). \end{gathered} \tag{5.25}$$

5.6.2. 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 tasks 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

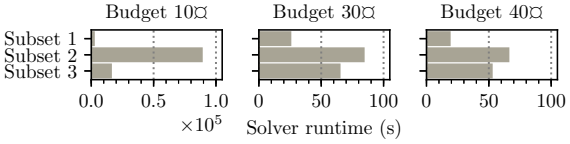


Figure 5.17: Solver runtimes for all three subsets of workflows with different budget constraints. In the left plot the x axis has much larger scale than in the other two plots.

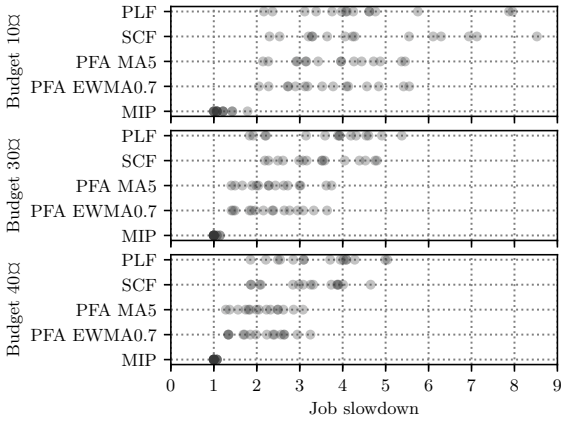


Figure 5.18: The variability of job slowdowns for different budgets for Airflow-based and MIP results.

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 5.17 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 [13] that can significantly affect the performance of the solver. To somehow automate this process, Gurobi even provides a parameter tuning tool [12]. 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. [164]. 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 5.18, 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 5.18 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 workflow management system itself. However, the general trend is similar to Figures 5.5, and 5.6 where our PFA autoscaler shows better workload performance than the plan-based autoscalers.

5.7. Related Work

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

The Dynamic Scaling Consolidation Scheduling (DSCS) [118], Partitioned Balanced Time Scheduling (PBTS) [40], IaaS Cloud Partial Critical Paths (IC-PCP) [24], Deadline Constrained Critical Path (DCCP) [33], Dyna [171], and Partition Problem-based Dynamic Provisioning and Scheduling (PPDPS) [146] autoscalers combine scheduling and allocation approaches, and, in contrast to the approach used in this chapter, 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 Dynamic Provisioning Dynamic Scheduling (DPDS) [116] 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) [116] is an offline autoscaler that creates a plan for each workflow in the ensemble, and rejects workflows that exceed the deadline or budget. BAGS [140] 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 perform simulations when evaluating the proposed algorithms. Versluis et al. [160] perform comprehensive analysis of different autoscalers for workloads of workflows. Overall, this study emphasizes the need for autoscalers that can cope with workloads of workflows, but neither proposes 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 [111] on cost and makespan-aware workflow scheduling in cloud provides a good overview of the current scheduling and autoscaling trends for workflows.

5.8. Conclusion

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.

6

Conclusion

In this dissertation, we have studied the problem of online scheduling of workloads of stochastically arriving workflows, both from the task placement and the resource allocation perspectives. We have mostly focused on designing new scheduling policies, but we have also adapted existing state-of-the-art policies designed for scheduling a single workflow or batches of workflows to the case of online workflow scheduling. For new policies, we have kept their implementation effort and their applicability in production systems in mind. Our research methods include a wide set of simulation-based and real-world experiments on the DAS-4 multicluster to thoroughly evaluate the studied policies. Additionally, we have used a mixed integer programming approach to validate our real-world experimental results versus the optimal solution. Below we present our conclusions, and our suggestions for future work.

6.1. Conclusions

We present the following conclusions based on the three workflow scheduling challenges (Section 1.2) and the four research questions addressed in this dissertation (Section 1.7):

1. Greedy backfilling is the best policy for scheduling workloads of workflows with unknown task runtimes compared to policies that employ processor reservation for workflows in the queue (Chapter 2). Any form of processor reservation for workflows without runtime estimates only decreases the overall system performance, leading to low maximal achievable utilizations.
2. Under realistic system utilizations backfilling-based policies allow achieving good performance even without task runtime estimates (Chapters 2 and 3). More complex scheduling policies are beneficial only at extremely high system utilizations and for scheduling batches of workflows.
3. Inaccurate runtime estimates negatively affect the slowdowns experienced by the workflows in the workload, but their effect is more substantial with the

increase of the imposed system utilization (Chapter 3). In general, at high system utilizations the knowledge of task runtime estimates allows to achieve significant improvements in the average workflow slowdowns.

4. The performance-feedback mechanism used by our novel Fair Workflow Prioritization (FWP) task placement policy to monitor workflow slowdowns helps to achieve a fairer distribution of slowdowns among workflows in the workload (Chapter 3). FWP shows a lower standard deviation of the workflow slowdowns compared to the state-of-the-art policies without the feedback mechanism. The order in which the workflows are processed by a task placement policy is very important, as an incorrectly chosen prioritization method can easily destabilize the system (Chapter 3).

5. When adapted to a plan-based online scenario, the relatively simple offline HEFT scheduling policy underperforms (Chapter 3). We believe that even more complex policies [142] would also suffer from this problem. In general, online plan-based policies are more vulnerable to scalability issues during workload surges, and are harder to implement and parallelize compared to dynamic online policies with similar performance.

6. It is possible to realistically estimate the resource demand of a workflow in terms of the number of required processors when the task runtimes are unknown by using our novel token-based algorithm for approximating the level of parallelism (Chapters 2, 4, and 5). We have thoroughly evaluated the algorithm both in simulations and in real-world experiments, and it shows its effectiveness when applied to popular scientific workflow structures.

7. Although workflow-specific autoscalers have the privilege of knowing the workflow structure in advance, it is possible for properly configured general autoscalers without such knowledge to achieve similar performance (Chapter 4). The correct choice of an autoscaler and its parameters significantly depends on the application type. To assist the end-user in choosing an autoscaler and its parameters, we have proposed comprehensive multilateral ranking methods for comparing autoscalers which showed their applicability for comparing general and workflow-specific autoscalers.

8. The novel feedback-based online dynamic Performance-Feedback Autoscaler (PFA) that uses task throughput and the workflow structure to make resource allocation decisions outperforms online plan-based policies for online scheduling of workflows (Chapter 5). PFA has better applicability potential due to its good scalability when dealing with possible demand surges, and good end-user and system-oriented characteristics compared to the plan-based state-of-the-art policies. The feedback mechanism in the novel autoscaler analyzes historical task throughput. Observing task throughput normally requires less effort than obtaining task runtime estimates, while allowing for fairly accurate estimation of resource speeds when dealing with workloads with a long-tailed task runtime distribution.

9. Obtaining an optimal solution for a workflow scheduling problem with a mixed integer programming approach can be used to find how much the

results obtained with other methods, e.g., real-world experiments, differ from the optimal solution (Chapter 5). However, the mixed integer programming approach has a high time complexity, a large number of parameters, and unpredictable performance, which makes it not suitable for scheduling workloads of workflows.

6.2. Suggestions for Future Work

Scheduling workloads of workflows still has many open questions that need to be addressed. In this section, we present some future research directions.

1. In Chapter 2, we have reserved resources for each workflow based on its expected level of parallelism with a manually set lookup depth. This method did not show substantial performance benefits. It was expected that at lower system utilizations processor reservation would help to reduce workflow slowdowns, as the system would have enough resources to compensate for capacity losses due to reservations. However, taking into consideration the results obtained with the feedback-based autoscaler in Chapter 5, it would be beneficial to study the effect of other reservation methods, for example, by dynamically adjusting the lookup depth of the token-based parallelism approximator on a per-workflow basis.
2. In Chapter 3, we have proposed a workflow scheduling policy that targets fairness and uses historical slowdowns for making task placement decisions. We have defined fairness as the minimization of the slowdown variability among the workflows in the workload. As future work, while using the same fairness definition, it would be interesting to investigate other workflow prioritization methods to allow even short and extremely parallel workflows to experience comparable slowdowns. Additionally, the fairness can be redefined, for example, to link the expected slowdowns to certain job types or user-defined priorities.
3. As we have shown in Chapter 3, the performance of task placement policies that rely on task runtime estimates depends on the type of the runtime estimation error and the system utilization. For this reason, it would be interesting to consider other error types, for example, by assuming more error variability for shorter tasks, as has been observed by Feitelson [63]. Moreover, the effect of incorrect task runtime estimates on the studied dynamic policies can additionally be validated in a real computing environment with different system utilizations.
4. With all the autoscaling policies in Chapters 4 and 5, we have used greedy backfilling as a task placement policy. The main reason for this choice was our focus on the autoscaling performance which we did not want to contaminate with the effects caused by different task placement policies. However, there could be potential performance benefits from using different combinations of task placement and resource allocation policies that can be further investigated. Moreover, the metrics proposed in Chapter 4 can be extended to address the effects of task placement policies.

5. In Chapter 5, we used a pre-configured history lookup depth and smoothing factor for the exponentially weighted moving average of historical throughputs. For future work, to make the proposed Performance-Feedback Autoscaler even more autonomous, it would be useful to automatically configure the signal smoothing. To add support for various application types, it seems reasonable to consider, instead of task throughput, other application-specific metrics.
6. In this dissertation, we have focused on computationally intensive workflows and, thus, we have only considered processors as the type of system resources that can be controlled by the scheduler. However, many other workflow types are currently evolving that require other resource types for their operation. Therefore, it would be logical to continue this work by assessing the performance of I/O-intensive and memory-bound workflows and develop the appropriate scheduling policies. Besides that, in all the chapters, we have used Poisson arrivals for the workflow jobs within the workload. Alternatively, other arrival patterns can be investigated. Furthermore, to investigate the workflow scheduling problem from another perspective, various workload modifications can be explored (e.g., the modification of arrival patterns) that would help to achieve better performance results without changing the scheduling policy. Finally, various scheduling parallelization attempts can be made to investigate the possible performance benefits of decentralizing scheduling decisions.

Bibliography

[1] Amazon Lambda. https://aws.amazon.com/lambda.

[2] Apache Airflow. https://airflow.apache.org.

[3] Apache Hadoop. https://hadoop.apache.org.

[4] Apache Taverna. https://taverna.incubator.apache.org.

[5] Azure Functions. https://azure.microsoft.com/en-us/services/functions.

[6] Camunda BPM: Workflow and decision automation platform. https://camunda.com.

[7] Celery: Distributed task queue. http://docs.celeryproject.org.

[8] COMMIT IV-e: e-Infrastructure Virtualization for e-Science applications. https://www.commit-nl.nl/projects/e-infrastructure-virtualization-for-e-science-applications.

[9] Dask: Scalable analytics in Python. https://dask.org.

[10] e-Science Central. https://www.esciencecentral.org.

[11] Google Cloud Composer: A fully managed workflow orchestration service built on apache airflow. https://cloud.google.com/composer.

[12] Gurobi reference manual: Parameter tuning tool. https://www.gurobi.com/documentation/8.1/refman/parameter_tuning_tool.html.

[13] Gurobi reference manual: Parameters. https://www.gurobi.com/documentation/8.1/refman/parameters.html.

[14] Gurobi: The state-of-the-art mathematical programming solver. http://www.gurobi.com.

[15] IBM Cloud Functions. https://cloud.ibm.com/functions.

[16] Pegasus: Synthetic workflow generators. https://github.com/pegasus-isi/WorkflowGenerator.

[17] Pegasus workflow management system. http://pegasus.isi.edu.

[18] Redis: In-memory data structure store. https://redis.io.

[19] Updated workflows for new LHC. http://newscenter.lbl.gov/2016/02/22/updated-workflows-for-new-lhc.

[20] G. Aad, T. Abajyan, B. Abbott, J. Abdallah, S. A. Khalek, A. A. Abdelalim, O. Abdinov, R. Aben, B. Abi, M. Abolins, et al. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. *Physics Letters B*, 716:1–29, 2012.

[21] B. P. Abbott, R. Abbott, T. D. Abbott, F. Acernese, K. Ackley, C. Adams, T. Adams, P. Addesso, R. X. Adhikari, V. B. Adya, et al. GW170817: Observation of gravitational waves from a binary neutron star inspiral. *Physical Review Letters*, 119:161101, 2017.

[22] B. P. Abbott, R. Abbott, R. Adhikari, J. Agresti, P. Ajith, B. Allen, R. Amin, S. B. Anderson, W. G. Anderson, M. Arain, et al. Search for gravitational waves from binary inspirals in S3 and S4 LIGO data. *Physical Review D*, 77:062002, 2008.

[23] S. Abrishami, M. Naghibzadeh, and D. H. J. Epema. Cost-driven scheduling of grid workflows using partial critical paths. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 23:1400–1414, 2012.

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

[25] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *ACM Workshop on Scientific Cloud Computing (ScienceCloud)*, 2012.

[26] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *IEEE Network Operations and Management Symposium (NOMS)*, 2012.

[27] A. Ali-Eldin, J. Tordsson, E. Elmroth, and M. Kihl. Workload classification for efficient auto-scaling of cloud resources. Technical report, Umeå University, Lund University, 2013.

[28] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems (TOCS)*, 19:483–518, 2001.

[29] H. Arabnejad and J. Barbosa. Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2012.

[30] H. Arabnejad and J. G. Barbosa. Multi-workflow QoS-constrained scheduling for utility computing. In *IEEE International Conference on Computational Science and Engineering (CSE)*, 2015.

[31] H. Arabnejad and J. G. Barbosa. Multi-QoS constrained and profit-aware scheduling approach for concurrent workflows on heterogeneous systems. *Future Generation Computer Systems (FGCS)*, 68:211–221, 2017.

[32] H. Arabnejad, J. M. G. Barbosa, and F. Suter. Fair resource sharing for dynamic scheduling of workflows on heterogeneous systems. In *IEEE International Conference on High Performance Computing, Data and Analytics (HiPC)*, 2014.

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

[34] ASKALON Team and T. Fahringer. ASKALON grid environment: User guide. http://www.askalon.org/documents/ASKALONUserGuide-final.pdf, 2015.

[35] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *IEEE Computer*, 49:54–63, 2016.

[36] R. Barga and D. Gannon. Scientific versus business workflows. In *Workflows for e-Science, Scientific Workflows for Grids*, pages 9–16. Springer, 2007.

[37] L. S. Baumann and R. D. Coop. Automated workflow control: A key to office productivity. In *National computer conference*, 1980.

[38] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi. Characterization of scientific workflows. In *Workshop on Workflows in Support of Large-Scale Science*, 2008.

[39] I. Bird. Computing for the large hadron collider. *Annual Review of Nuclear and Particle Science*, 61:99–118, 2011.

[40] E. Byun, Y. Kee, J. Kim, and S. Maeng. Cost optimized provisioning of elastic resources for application workflows. *Future Generation Computer Systems (FGCS)*, 27:1011–1026, 2011.

[41] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. *ACM SIGOPS Operating Systems Review*, 35:103–116, 2001.

[42] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *IEEE International Conference on e-Business Engineering (ICEBE)*, 2009.

[43] A. M. Chirkin, A. S. Z. Belloum, S. V. Kovalchuk, M. X. Makkes, M. A. Melnik, A. A. Visheratin, and D. A. Nasonov. Execution time estimation for workflow scheduling. *Future Generation Computer Systems (FGCS)*, 75:376–387, 2017.

[44] W. Clark, W. N. Polakov, and F. W. Trabold. *The Gantt chart: A working tool of management*. Ronald Press Company, 1922.

[45] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[46] E. D. Coninck, T. Verbelen, B. Vankeirsbilck, S. Bohez, P. Simoens, and B. Dhoedt. Dynamic auto-scaling and scheduling of deadline constrained service workloads on IaaS clouds. *Journal of Systems and Software (JSS)*, 118:101–114, 2016.

[47] R. Cushing, S. Koulouzis, A. S. Z. Belloum, and M. Bubak. Prediction-based auto-scaling of scientific workflows. In *ACM International Workshop on Middleware for Grids, Clouds and e-Science (MGC)*, 2011.

[48] R. F. Da Silva, G. Juve, M. Rynge, E. Deelman, and M. Livny. Online task resource consumption prediction for scientific workflows. *Parallel Processing Letters*, 25:1541003, 2015.

[49] G. B. Dantzig and M. N. Thapa. *Linear programming 1: Introduction*. Springer, 2006.

[50] H. A. David. Ranking from unbalanced paired-comparison data. *Biometrika*, 74:432–436, 1987.

[51] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *ACM Communications*, 51:107–113, 2008.

[52] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. S. Vetter. The future of scientific workflows. *The International Journal of High Performance Computing Applications (IJHPCA)*, 32:159–175, 2018.

[53] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: The Montage example. In *ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2008.

[54] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. C. Laity, J. C. Jacob, and D. S. Kat. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13:219–237, 2005.

[55] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, and R. K. Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems (FGCS)*, 46:17–35, 2015.

[56] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices*, 49:127–144, 2014.

[57] Y. Demchenko, C. Blanchet, C. Loomis, R. Branchat, M. Slawik, B. I. Zilci, M. Bedri, J. Gibrat, O. Lodygensky, M. Zivkovic, and C. de Laat. Cyclone: A platform for data intensive scientific applications in heterogeneous multi-cloud/multi-provider environment. In *IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, 2016.

[58] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.

[59] Q. Dishan, H. Chuan, L. Jin, and M. Manhao. A dynamic scheduling method of earth-observing satellites by employing rolling horizon strategy. *The Scientific World Journal*, page 304047, 2013.

[60] T. Dornemann, E. Juhnke, and B. Freisleben. On-demand resource provisioning for BPEL workflows using Amazon's elastic compute cloud. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2009.

[61] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. L. Truong, A. Villazón, and M. Wieczorek. ASKALON: A development and grid computing environment for scientific workflows. In *Workflows for e-Science, Scientific Workflows for Grids*, pages 450–471. Springer, 2007.

[62] L. Fei, B. Ghiț, A. Iosup, and D. H. J. Epema. KOALA-C: A task allocator for integrated multicluster and multicloud environments. In *IEEE International Conference on Cluster Computing (Cluster)*, 2014.

[63] D. G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation.* Cambridge University Press, 2015. pp. 399-489.

[64] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling web applications in heterogeneous cloud infrastructures. In *IEEE International Conference on Cloud Engineering (IC2E)*, 2014.

[65] R. Ferreira da Silva, T. Glatard, and F. Desprez. Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions. *Concurrency and computation: Practice and experience (CCPE)*, 26:2347–2366, 2014.

[66] P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *ACM Communications*, 29:218–221, 1986.

[67] J. Frey. Condor DAGMan: Handling inter-job dependencies. Technical report, Department of Computer Science, University of Wisconsin, 2002.

[68] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)*, 30:14:1–14:26, 2012.

[69] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and parallel Databases*, 3:119–153, 1995.

[70] T. Hegeman, B. Ghiț, M. Capota, J. Hidders, D. Epema, and A. Iosup. The BTWorld use case for big data analytics: Description, MapReduce logical workflow, and empirical evaluation. In *IEEE International Conference on Big Data*, 2013.

[71] T. Heinis, C. Pautasso, and G. Alonso. Design and evaluation of an autonomic workflow engine. In *IEEE International Conference on Autonomic Computing (ICAC)*, 2005.

[72] N. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *International Conference on Autonomic Computing (ICAC)*, 2013.

[73] N. Herbst, R. Krebs, G. Oikonomou, G. Kousiouris, A. Evangelinou, A. Iosup, and S. Kounev. Ready for rain? A view from SPEC research on the future of cloud metrics. Technical report, arXiv:1604.03470, SPEC Research Group, Cloud Working Group, 2016.

[74] N. R. Herbst, S. Kounev, A. Weber, and H. Groenda. BUNGEE: An elasticity benchmark for self-adaptive IaaS cloud environments. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2015.

[75] H. Hiden, P. Watson, S. Woodman, and D. Leahy. e-Science Central: Cloud-based e-Science and its application to chemical property modelling. Technical report, CS-TR-1227, School of Computer Science, Newcastle University, 2011.

[76] A. Hirales-Carbajal, A. Tchernykh, R. Yahyapour, J. L. González-García, T. Röblitz, and J. M. Ramírez-Alcaraz. Multiple workflow scheduling strategies with user run time estimates on a grid. *Journal of Grid Computing*, 10:325–346, 2012.

[77] C.-C. Hsu, K.-C. Huang, and F.-J. Wang. Online scheduling of workflow applications in grid environments. *Future Generation Computer Systems (FGCS)*, 27:860–870, 2011.

[78] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: A tool for building and running workflows of services. *Nucleic acids research*, 34:729–732, 2006.

[79] S. Ikiz and V. K. Garg. Online algorithms for Dilworth's chain partition. Technical report, Parallel and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, University of Texas at Austin.

[80] A. Ilyushkin, A. Bauer, A. V. Papadopoulos, E. Deelman, and A. Iosup. Computational Artifacts for Performance Feedback Autoscaling Experiments with Workloads of Workflows in Apache Airflow. https://doi.org/10.5281/zenodo.2635573, 2019.

[81] A. Ilyushkin, A. Bauer, A. V. Papadopoulos, E. Deelman, and A. Iosup. Software Artifacts for Performance Feedback Autoscaling Experiments with Workloads of Workflows in Apache Airflow. https://doi.org/10.5281/zenodo.2635571, 2019.

[82] A. Ilyushkin and D. Epema. The impact of task runtime estimate accuracy on scheduling workloads of workflows. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2018.

[83] A. Ilyushkin, B. Ghiț, and D. Epema. Scheduling workloads of workflows with unknown task runtimes. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015.

[84] A. Iosup, O. Sonmez, and D. Epema. DGSim: Comparing grid resource management architectures through trace-based simulation. In *European Conference on Parallel Processing (Euro-Par)*, 2008.

[85] A. Iosup, T. Tannenbaum, M. Farrellee, D. Epema, and M. Livny. Interoperating grids through delegated matchmaking. *Scientific Programming*, 16:233–253, 2008.

[86] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2011.

[87] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems (FGCS)*, 27:871–879, 2011.

[88] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur. Oozie: Towards a scalable workflow management system for Hadoop. In *ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET)*, 2012.

[89] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2001.

[90] J. C. Jacob, D. S. Katz, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M. Su, T. A. Prince, and R. Williams. Montage: A grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computer Sciences and Engineering (IJCSE)*, 4:73–87, 2009.

[91] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, and C. Kim. EyeQ: Practical network performance isolation for the multi-tenant cloud. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[92] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems (FGCS)*, 29:682–692, 2013.

[93] L. V. Kantorovich. Moi put' v nauke (My way in science). *Uspehi Matematicheskih Nauk*, 42:183–213, 1987.

[94] L. V. Kantorovich. *Mathematical-Economic Articles*. Nauka: Novosibirsk, 2011.

[95] J. E. Kelley Jr and M. R. Walker. Critical-path planning and scheduling. In *IRE-AIEE-ACM computer conference*, 1959.

[96] A. Kembhavi. Big data in astronomy and beyond. In *Data Science Landscape*, pages 59–66. Springer, 2018.

[97] M. A. Khan. Scheduling for heterogeneous systems using constrained critical paths. *Parallel Computing*, 38:175–193, 2012.

[98] D. K. Krishnappa, M. Zink, and R. K. Sitaraman. Optimizing the video transcoding workflow in content delivery networks. In *ACM International Conference on Multimedia Systems (MMsys)*, 2015.

[99] M. Kuhnemann, T. Rauber, and G. Runger. A source code analyzer for performance prediction. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[100] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 7:506–521, 1996.

[101] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing (JPDC)*, 59:381–422, 1999.

[102] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31:406–471, 1999.

[103] C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely. Are user runtime estimates inherently inaccurate? In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2004.

[104] K. Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. Fernandes, and G. Mehta. Adaptive workflow processing and execution in Pegasus. *Concurrency and Computation: Practice and Experience*, 21:1965–1981, 2009.

[105] P. Leitner and J. Cito. Patterns in the chaos—a study of performance variation and predictability in public IaaS clouds. *ACM Transactions on Internet Technology (TOIT)*, 16:15:1–15:23, 2016.

[106] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000.

[107] D. A. Lifka. The ANL/IBM SP scheduling system. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1995.

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

[109] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *ACM symposium on Cloud computing (SoCC)*, 2010.

[110] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12:559–592, 2014.

[111] P. Lu, G. Zhang, Z. Zhu, X. Zhou, J. Sun, and J. Zhou. A review of cost and makespan-aware workflow scheduling in clouds. *Journal of Circuits, Systems and Computers*, 28:1930006, 2018.

[112] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing (JPDC)*, 63:1105–1122, 2003.

[113] S. Ma, A. Ilyushkin, A. Stegehuis, and A. Iosup. ANANKE: A Q-learning-based portfolio scheduler for complex industrial workflows. In *IEEE International Conference on Autonomic Computing (ICAC)*, 2017.

[114] M. Malawski, K. Figiela, M. Bubak, E. Deelman, and J. Nabrzyski. Scheduling multilevel deadline-constrained scientific workflows on clouds based on cost optimization. *Scientific Programming*, page 680271, 2015.

[115] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. In *ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2012.

[116] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. *Future Generation Computer Systems (FGCS)*, 48:1–18, 2015.

[117] G. Malewicz, I. Foster, A. L. Rosenberg, and M. Wilde. A tool for prioritizing DAGMa jobs and its evaluation. *Journal of Grid Computing*, 5:197–212, 2007.

[118] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2011.

[119] M. Mao and M. Humphrey. Scaling and scheduling to maximize application performance within budget constraints in cloud workflows. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.

[120] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, R. Ricci, and A. Klimovic. Taming performance variability. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[121] E. R. Marsh. The harmonogram of Karol Adamiecki. *Academy of management Journal*, 18:358–364, 1975.

[122] J. T. Meehean. *Towards Transparent CPU Scheduling*. PhD thesis, University of Wisconsin-Madison, 2011.

[123] D. Milojičić, I. M. Llorente, and R. S. Montero. OpenNebula: A cloud management tool. *IEEE Internet Computing*, 15:11–14, 2011.

[124] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 12:529–543, 2001.

[125] S. Murray, V. Bahyl, G. Cancio, E. Cano, V. Kotlyar, D. F. Kruse, and J. Leduc. An efficient, modular and simple tape archiving solution for LHC Run-3. In *Journal of Physics: Conference Series*, volume 898, page 062013, 2017.

[126] M. Nardelli, S. Nastic, S. Dustdar, M. Villari, and R. Ranjan. Osmotic flow: Osmotic computing + IoT workflow. *IEEE Cloud Computing*, 4:68–75, 2017.

[127] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas. Dependable horizontal scaling based on probabilistic model checking. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2015.

[128] M. J. Neely, E. Modiano, and C. Li. Fairness and optimal stochastic control for heterogeneous networks. *IEEE/ACM Transactions On Networking (TON)*, 16:396–409, 2008.

[129] N. M. O'boyle, A. L. Tenderholt, and K. M. Langner. Cclib: A library for package-independent computational chemistry algorithms. *Journal of computational chemistry*, 29:839–845, 2008.

[130] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data*, 2008.

[131] M. Oppitz and P. Tomsu. Cloud computing. In *Inventing the Cloud Century*, pages 267–318. Springer, 2018.

[132] S. Ostermann, R. Prodan, T. Fahringer, A. Iosup, and D. Epema. On the characteristics of grid workflows. In *CoreGRID Symposium Euro-Par*, 2008.

[133] A. V. Papadopoulos, A. Ali-Eldin, K. Årzén, J. Tordsson, and E. Elmroth. PEAS: A performance evaluation framework for auto-scaling strategies in cloud applications. Tail response time modeling and control for interactive cloud services. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 1:15:1–15:31, 2016.

[134] H. P. Patil. On the structure of k-trees. *Journal of Combinatorics, Information and System Sciences (JCISS)*, 11:57–64, 1986.

[135] I. Pietri, G. Juve, E. Deelman, and R. Sakellariou. A performance model to estimate execution time of scientific workflows on the cloud. In *IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2014.

[136] M. Pundir, M. Kumar, L. M. Leslie, I. Gupta, and R. H. Campbell. Supporting on-demand elasticity in distributed graph processing. In *IEEE International Conference on Cloud Engineering (IC2E)*, 2016.

[137] M. Rahman, X. Li, and H. Palit. Hybrid heuristic for scheduling data analytics workflow applications in hybrid cloud environment. In *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2011.

[138] M. Rahman, S. Venugopal, and R. Buyya. A dynamic critical path algorithm for scheduling scientific workflow applications on global grids. In *IEEE International Conference on e-Science and Grid Computing (e-Science)*, 2007.

[139] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. Scheduling data-intensive workflows onto storage-constrained distributed resources. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2007.

[140] M. A. Rodriguez and R. Buyya. Budget-driven scheduling of scientific workflows in IaaS clouds with fine-grained billing periods. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 12:5:1–5:22, 2017.

[141] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan. Scheduling of parallel jobs in a heterogeneous multi-site environment. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2003.

[142] R. Sakellariou and H. Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[143] A. Shade and T. K. Teal. Computing workflows for biologists: A roadmap. *PLoS biology*, 13:e1002303, 2015.

[144] E. Shmueli and D. G. Feitelson. Backfilling with lookahead to optimize the packing of parallel jobs. *Journal of Parallel and Distributed Computing (JPDC)*, 65:1090–1107, 2005.

[145] G. Singh and E. Deelman. The interplay of resource provisioning and workflow optimization in scientific applications. *Concurrency and Computation: Practice and Experience*, 23:1969–1989, 2011.

[146] V. Singh, I. Gupta, and P. 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.

[147] S. Spinner, G. Casale, F. Brosig, and S. Kounev. Evaluating approaches to resource demand estimation. *Performance Evaluation*, 92:51–71, 2015.

[148] F. Suter and S. Hunold. DAGGen: A synthetic task graph generator. https://github.com/frs69wq/daggen.

[149] D. Talby and D. G. Feitelson. Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In *IEEE International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, 1999.

[150] D. Talia. Clouds for scalable big data analytics. *IEEE Computer*, 46:98–101, 2013.

[151] F. W. Taylor. Scientific management. *The Sociological Review*, 7:266–269, 1914.

[152] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science, Scientific Workflows for Grids*. Springer, 2007.

[153] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and computation: Practice and experience (CCPE)*, 17:323–356, 2005.

[154] S. Tilloo. Running arbitrary DAG-based workflows in the cloud. http://www.ebaytechblog.com/2016/04/05/running-arbitrary-dag-based-workflows-in-the-cloud, 2017.

[155] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 13:260–274, 2002.

[156] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *ACM SIGMETRICS Performance Evaluation Review*, 2005.

[157] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3:1:1–1:39, 2008.

[158] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup. Serverless is more: From PaaS to present cloud computing. *IEEE Internet Computing*, 22:8–17, 2018.

[159] J. van Helden. Regulatory sequence analysis tools. *Nucleic Acids Research*, 31:3593–3596, 2003.

[160] L. Versluis, M. Neacsu, and A. Iosup. A trace-based performance study of autoscaling workloads of workflows in datacenters. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2018.

[161] N. Vydyanathan, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. A duplication based algorithm for optimizing latency under throughput constraints for streaming workflows. In *IEEE International Conference on Parallel Processing (ICPP)*, 2008.

[162] P. Waibel, S. Videnov, M. Borkowski, C. Hochreiner, S. Schulte, and J. Mendling. Process simulation for machine reservation in cloud manufacturing. In *IEEE International Conference on Industrial Informatics (INDIN)*, 2018.

[163] Y. Wang, S. Cao, G. Wang, Z. Feng, C. Zhang, and H. Guo. Fairness scheduling with dynamic priority for multi workflow on heterogeneous systems. In *IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, 2017.

[164] Y. Wang, Y. Xia, and S. Chen. Using integer programming for workflow scheduling in the cloud. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2017.

[165] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos. Fog orchestration for Internet of Things services. *IEEE Internet Computing*, 21:16–24, 2017.

[166] J. R. Wieland, R. Pasupathy, and B. W. Schmeiser. Queueing-network stability: Simulation-based checking. In *Winter Simulation Conference*, 2003.

[167] F. Wu, Q. Wu, and Y. Tan. Workflow scheduling in cloud: A survey. *The Journal of Supercomputing*, 71:3373–3418, 2015.

[168] K. Yelick, S. Coghlan, B. Draney, and R. S. Canon. The Magellan report on cloud computing. Technical report, LBNL-5376E, U.S. Department of Energy, Washington D.C., USA, 2011.

[169] Z. Yu and W. Shi. A planner-guided scheduling strategy for multiple workflow applications. In *IEEE International Conference on Parallel Processing-Workshops (ICPP-W)*, 2008.

[170] H. Zhao and R. Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *IEEE International Parallel and Distributed Processing Symposium*, 2006.

[171] A. C. Zhou, B. He, and C. Liu. Monetary cost optimizations for hosting workflow-as-a-service in IaaS clouds. *IEEE Transactions on Cloud Computing*, 4:34–48, 2016.

Summary

A workflow is a universal abstraction for representing complex activities consisting of multiple interconnected tasks. Workflows are often used to describe and orchestrate computations at different scales. The growing applicability of such computing workflows has led to the development of appropriate algorithms designed for scheduling and executing workflows efficiently. Workflow scheduling can be seen from the task placement and resource allocation perspectives, where task placement controls the mapping of tasks to computing resources, and resource allocation controls the number of dynamically available computing resources. Due to the usage patterns of earlier workflow applications in clusters and grids, most of the existing workflow scheduling algorithms were designed to process statically present sets of workflows. Such offline algorithms often fail to address the challenges introduced by modern online applications of workflows, for example, in computing clouds where diverse workflows from different users arrive stochastically forming a workload. This provides a variety of opportunities for the development of new scheduling policies to meet the challenges of online scheduling.

In this dissertation, we identify and address three key challenges that are characteristic to the online scheduling of workloads of workflows in modern distributed computing systems, such as clusters and clouds. The first challenge is realistic estimation of the resource demand of a workflow, as it is important for both task placement and resource allocation. Second, is the efficient placement of workflow tasks to minimize average workflow slowdown while achieving fairness. A wrongly chosen task placement policy can easily degrade the performance and negatively affect the fair access of workflows to computing resources. Third, is the automatic allocation (autoscaling) of computing resources for workflows while meeting deadline and budget constraints. Computing clouds make it possible to easily lease and release resources. Such decisions should be made wisely to minimize slowdowns and deadline violations, and to efficiently use the leased resources to reduce incurred costs.

To address these challenges, we propose novel online scheduling policies and investigate applicability of relevant state-of-the-art workflow scheduling policies to the online scenario. For new policies, we keep in mind their implementation effort and their suitability for production systems. We experimentally evaluate these workflow scheduling policies by conducting a wide set of simulation-based and real-world experiments on a private multicluster computer. Additionally, we use a Mixed Integer Programming (MIP) approach to validate our real-world experimental results versus the optimal solution obtained with a MIP solver.

In Chapter 1, we summarize workflow scheduling approaches and identify main workflow scheduling challenges from both task placement and resource allocation perspectives. We also provide an overview of state-of-the-art workflow scheduling policies and modern workflow managements systems. Furthermore, we present

research questions that aim to address the identified workflow scheduling challenges, the research methods used in this dissertation, the dissertation outline, and our key contributions.

In Chapter 2, we address the problem of online scheduling of workloads of workflows with unknown task runtime estimates. For this we propose a family of four novel online workflow scheduling policies which differ in to what extent they reserve processors for the workflows towards the head of the queue. To be able to make processor reservations, we propose a method for the realistic estimation of workflow level of parallelism. Our results show that even at moderate imposed utilizations, the greedy backfilling policy leads to lower average workflow slowdowns compared to the policies which use processor reservation.

In Chapter 3, we focus on scheduling workloads of workflows with varying accuracy of task runtime estimates that are available to the scheduler. We implement four state-of-the-art online dynamic workflow scheduling policies, and propose two novel online dynamic policies, one of which addresses fairness, and the other one adapts a popular offline plan-based policy for the online plan-based scenario. In our results, we can clearly see that the knowledge of task runtime estimates gives significant performance improvement in the average job slowdown, but only at extremely high utilizations. At moderate utilizations, simpler backfilling-based policies outperform more advanced policies. The plan-based online policy demonstrates poor performance compared to dynamic online policies. Our fairness-oriented policy effectively decreases the variance of job slowdown and thus achieves fairness.

In Chapter 4, we experimentally evaluate five state-of-the-art general autoscalers and propose two novel workflow-specific autoscalers. Moreover, we present and refine performance metrics endorsed by the Standard Performance Evaluation Corporation (SPEC) for assessing autoscalers, and define three approaches for comparing the autoscalers using the considered metrics. The experimental results show that general autoscalers can demonstrate comparable performance to workflow-specific autoscalers, if the former have access to workload statistics. The workflow-specific autoscalers, in turn, require task or workflow runtime estimates. We additionally investigate the effect of autoscaling on meeting workflow deadlines. Our results highlight the trade-offs between the suggested policies, their impact on meeting the deadlines, and their performance in different operating conditions.

In Chapter 5, we propose a novel online dynamic autoscaler that employs a feedback mechanism for making autoscaling decisions. For that our autoscaler analyzes historical task throughput and uses the not yet finished part of the workflow, instead of relying on task runtime estimates, as analyzing the task throughput requires less effort than obtaining task runtime estimates. Our approach shows lower time complexity and effectively minimizes workflow slowdowns compared to two state-of-the-art autoscalers. The usage of historical throughput information provides fairly accurate estimation of resource speeds when dealing with workloads with a representative long-tailed task runtime distribution. We additionally validate the experimental results by comparing the workflow slowdowns obtained from an actual workflow management system with an optimal solution.

Finally, in Chapter 6, we present the main conclusions of this dissertation and we provide suggestions for several future directions.

Samenvatting

Een workflow is een universele abstractie voor het weergeven van complexe activiteiten die bestaan uit meerdere onderling verbonden taken. Workflows worden vaak gebruikt om berekeningen op verschillende schaal te beschrijven en te organiseren. De groeiende toepasbaarheid van zulke workflows heeft geleid tot de ontwikkeling van geschikte algoritmen om ze efficiënt uit te voeren. Workflow scheduling kan vanuit twee perspectieven worden bestudeerd: Ten eerste vanuit task placement, waarbij taken zo geschikt mogelijk aan computer resources worden gekoppeld, en ten tweede vanuit resource allocatie, waarbij men dynamisch computer resources beschikbaar stelt. Aangezien in het verleden workflow applicaties voornamelijk waren gericht op clusters en grids, zijn de scheduling algoritmes uit deze tijd gericht op statische verzamelingen workflows. Zulke zogenaamde offline algoritmes zijn niet altijd even geschikt voor gebruik in de moderne online toepassing van workflows, bijvoorbeeld in cloud netwerken. Daarin starten de diverse workflows van verschillende gebruikers op verschillende tijdstippen, en vormen zo een stochastische workload van workflows. Dit biedt de mogelijkheid om nieuwe scheduling policies te ontwikkelen die in staat zijn om te gaan met de uitdaging van online scheduling.

In dit proefschrift worden drie sleuteluitdagingen geïdentificeerd en behandeld die karakteristiek zijn voor workloads van workflows in moderne gedistribueerde computersystemen, zoals clusters en clouds. De eerste uitdaging is het realistisch inschatten van de resource-eisen van een workflow. Dit is belangrijk voor zowel task placement als resource allocatie. De tweede uitdaging is het efficiënt plaatsen van workflow-taken met als doel het minimaliseren van de gemiddelde slowdown van workflows en het eerlijk verdelen van de resources. Een verkeerd gekozen task placement policy kan de prestaties sterk beïnvloeden en kan een nadelig effect hebben op hoe eerlijk de computer resources verdeeld worden onder de verschillende workflows. De derde uitdaging is het automatisch toewijzen van computer resources ("autoscaling") bij een beperkt budget en met de aanwezigheid van deadlines. Computer clouds maken het mogelijk om gemakkelijk computer resources te huren en weer vrij te geven. Zulke beslissingen moeten op een verstandige manier worden gemaakt om de slowdown en het aantal schendingen van deadlines te minimaliseren, en om de computer resources efficiënt te huren zodat de kosten binnen de perken blijven.

Om deze uitdagingen aan te gaan worden nieuwe online scheduling policies voorgesteld, en wordt de geschiktheid van relevante moderne workflow scheduling policies voor het online scenario onderzocht. Voor de voorgestelde nieuwe policies wordt de moeite om ze te implementeren en hun geschiktheid voor productiesystemen in acht genomen. De workflow scheduling policies worden experimenteel getoetst met een breed scala van simulaties, en met echte experimenten op een multicluster computersysteem. We gebruiken mixed integer programming om de experimentele resultaten te vergelijken met een theoretisch gezien optimale oplossing.

In hoofdstuk 1 vatten we workflow scheduling methoden samen en identificeren we uitdagingen voor workflow scheduling vanuit het perspectief van task placement en resource allocation. Ook wordt er een overzicht gepresenteerd van state-of-the-art workflow scheduling policies en moderne workflow management systemen. Bovendien worden de onderzoeksvragen geformuleerd om de geïdentificeerde uitdagingen aan te gaan, en worden de in dit proefschrift gebruikte methoden, een overzicht van het proefschrift, en de belangrijkste bijdragen gepresenteerd.

In hoofdstuk 2 wordt het probleem behandeld van de online scheduling van workloads van workflows zonder de beschikbaarheid van schattingen van de executietijd van taken. Hiervoor worden vier nieuwe online workflow scheduling policies gepresenteerd die verschillen in de mate waarin ze processoren reserveren voor de workflows vooraan in de wachtrij. Om in staat te zijn processoren te reserveren wordt een methode voorgesteld die een realistische inschatting kan maken van het niveau van parallellisme in workflows. De resultaten hiervan laten zien dat zelfs bij een matige bezettingsgraad de greedy backfilling policy leidt tot een lagere gemiddelde workflow slowdown in vergelijking met de policies die gebruik maken van processorreservering.

In hoofdstuk 3 ligt de focus op de scheduling van workloads van workflows met gebruik van schattingen van de executietijden van taken van variërende precisie. Hier worden vier state-of-the-art online dynamische workflow scheduling policies geïmplementeerd, en worden twee nieuwe online dynamische policies voorgesteld. Eén daarvan richt zich op eerlijke toewijzing, de tweede is geïnspireerd op populaire offline plan-based policies voor het online plan-based scenario. In de resultaten kan duidelijk worden gezien dat beschikbaarheid van informatie over de lengte van taken een significante prestatieverbetering tot gevolg heeft in de gemiddelde job slowdown, maar alleen bij extreem hoge bezettingsgraden. Bij matig gebruik presteren simpele backfilling-based policies beter dan dynamische online policies. De eerlijkheid-georiënteerde policy reduceert de variantie in job slowdown en is dus inderdaad eerlijker.

In hoofdstuk 4 worden er vijf state-of-the-art generieke autoscalers geanalyseerd en worden er twee nieuwe workflow-specifieke autoscalers voorgesteld. Daarnaast worden de prestatiemetrieken aanbevolen door de Standard Performance Evaluation Corporation (SPEC) voor het beoordelen van autoscalers beschreven en verfijnd. Ook worden er drie manieren voorgesteld om autoscalers te vergelijken aan de hand van deze metrieken. De experimentele resultaten laten zien dat generieke autoscalers ongeveer hetzelfde presteren als workflow-specifieke autoscalers indien deze laatste toegang hebben tot workload-statistieken. De workflow-specifieke autoscalers vereisen schattingen van de executietijden van taken van workflows. Daarnaast wordt het effect van autoscaling op het behalen van workflow deadlines onderzocht. De resultaten laten goed de trade-offs tussen de verschillende policies, hun impact op het behalen van deadlines, en de prestaties in verschillende omstandigheden zien.

In hoofdstuk 5 wordt een nieuwe online dynamische autoscaler geïntroduceerd die gebruik maakt van een terugkoppelingsmechanisme. Deze autoscaler analyseert historische doorvoersnelheden van taken in plaats van gebruik te maken van schattingen van de executietijden van taken. Dit is voordelig, aangezien het minder

moeite kost om doorvoersnelheden te analyseren dan vooraf voor elke taak een schatting van zijn executietijd te maken. Deze benadering van het probleem heeft een lagere tijdscomplexiteit en minimaliseert de workflow slowdown in vergelijking met twee state-of-the-art autoscalers. Het gebruik van historische doorvoersnelheden biedt een redelijk accurate inschatting van resource snelheden bij workloads met een representatieve lange-staart verdeling van de executietijden van taken. Hiernaast worden de experimentele resultaten gevalideerd door de workflow slowdown verkregen met een hedendaags workflow management systeem te vergelijken met een optimale oplossing.

Tot slot worden in hoofdstuk 6 de algemene conclusies van dit proefschrift gepresenteerd en worden er suggesties gedaan voor vervolgonderzoek.

Acknowledgements

This dissertation would never happen without the help and support of numerous people who surrounded me through all these years.

First, I would like to thank my promotor Dick Epema for trusting me and giving the opportunity to do science. Dick, I extremely appreciate your patience, discipline, and high standards for research. Among your many lessons, you taught me how to be a clear communicator and an independent researcher. It was a great pleasure working with you while sharing a broader view on solving applied problems.

I am grateful to my second promotor Alexandru Iosup. Alexandru, you inspired me to work with autoscalers and greatly expanded my scientific network by introducing me to the SPEC research activities. Our meetings within the AtLarge team helped me to learn from other peer researchers—that was extremely useful for making this dissertation done.

I am also thankful to Henk Sips, as he was involved in my admission to the PDS group. Henk, even though we did not work together, thank you for your support during the PhD process and your advice on research in general.

I would like to thank the defense committee members: Koen Langendoen, Paola Grosso, Boudewijn Haverkort, Ewa Deelman, Radu Prodan, and Eelco Visser, for spending time on reading and evaluating my dissertation. Your useful comments and suggestions helped to improve this work.

I was extremely lucky working together with Bogdan Ghiț. Bogdan, I really appreciate that you significantly helped me at the early stages of my PhD in both technical and textual parts of my papers.

Special thanks go to Ahmed Ali-Eldin, Alessandro Vittorio Papadopoulos, Nikolas Herbst, André Bauer, and Samuel Kounev for our fruitful cooperation through the SPEC research group. Our joint activities enriched this dissertations and gave me valuable experience working on large projects together.

I greatly enjoyed working with Alexandru Uță, Laurens Versluis, Sietse Au, and Shenjun Ma, our collaborations helped me to explore new topics in resource autoscaling. I additionally thank Laurens for pointing me out to the MIP paper which gave me new ideas for one of the dissertation chapters.

Johan Pouwelse, your endless enthusiasm has always inspired me. I am very grateful for your help in my career and all the conversations we had at our group.

I am thankful to Yuri Demchenko for giving me the opportunity to join the Cyclone project at the SNE group of the University of Amsterdam.

To my office mates Riccardo Petrocco, Adele Lu Jia, Erwin van Eyk, Akbar Mostafavi, Amin Rezaeian, and Vadim Bulavintsev, thank you for the great working atmosphere and for all the nice chats.

To my colleagues, Otto Visser, Ana Varbanescu, Vincent van Beek, Dimitra Gkorou, Lucia d'Acunto, Jie Shen, Niels Zeilemaker, Mihai Capotă, Elric Milon,

Lipu Fei, Ernst van der Hoeven, Paul Brussee, Martijn de Vos, Giorgos Oikonomou, Tim Hegeman, Jesse Donkervliet, Stefan Hugtenburg, and Maria Voinea, thank you for contributing to the social side of my days at the (P)DS group. I will never forget all the countless pizza evenings and game nights we had together.

I would like to thank Aleksandra Kuzmanovska for sharing her insights on research with me and the moral support. It was also a great time we spent in the USA with the VU guys at CCGrid. I wish you all the best in your career.

Kees Verstoep, Paulo Anita, and Munire van der Kruyk, thank you for the excellent technical support. Without your help, many experiments in this dissertation simply would not happen. I am also grateful to all the secretaries Ilse Oonk, Rina Abbriata, Shemara van der Zwet, and Kim Roos for the administrative support.

I would probably never have come to the Netherlands to complete this dissertation if Ilia Vialshin would not have decided to start his own PhD in Nijmegen. Ilia, I am very happy knowing you for so many years and thank you for being such a great friend and crazy chemist.

Vallín García Cruz, you are my first friend in the Netherlands, and you definitely made my PhD time here unforgettable. Thank you for having such a good heart and for always being willing to help.

Gleb Polevoy, I really enjoyed all our profound philosophical conversations that gave me insight into many topics from a completely new perspective. Your ideas helped me to shape this dissertation and enriched my stay in the Netherlands.

Dmitry and Yulia Kononchuk, it has been a great pleasure knowing you, and thank you a lot for making this PhD time in the Netherlands full of nice gatherings with great thorough discussions.

I would like to thank all the Russian-speaking friends whom I met in Delft: Natalia Vtyurina, Maria Zamiralova, Emil Gallyamov, Irina Gallyamova, Alexander Nagui, Ivan Koryakovskiy, Mikhail Belonosov, Nick Gaiko, Sergey Bezrukavnikov, Inna Medvedeva, Mikhail Davydenko, Aleksandra Uuemaa, Tatiana Kozlova, Oleg Guziy, Nikita Lenchenkov, Vladimir Kovalenko, and Ilya Popov, for making my PhD journey so bright and full of amazing memories.

Dear Natalia Vdovenko, Polina Platonova, Evgenia Platonova, Ekaterina Balabanyuk, Nikita Dubov, and Tatiana Kosovets: The way I met all of you is simply incredible, and I truly enjoyed spending time with you. Thank you a lot for your positive energy and your warm characters.

Many thanks go to Delfts Studenten Muziekgezelschap Krashna Musika for all the great music performances we had together. Even though the word "krashna" does not exist in any Slavic language, somehow the official legend still says that it means *beautiful*. Let's believe so. It was indeed a great pleasure to be a part of such a beautiful society while making beautiful music.

Dena Kasraian, I promised that you will be the first in the Krashna list, and you have definitely deserved that, being the first person to introduce me to the group. I genuinely appreciate it, and wish you all the best in your professorship! Jaime Junell, thank you for coping with me and for all the lovely musical collaborations we had together. Ranko Tošković, I appreciate your directness and your Montenegrin sense of humour. Daan van Dijk, I am so happy knowing you and being able to discuss various scientific topics with you. Moreover, thank you for translating

the summary of this dissertation. Christina Widmer, you are simply the cherry on the top of the Krashna cake, I am glad meeting you there. Arturo Alvarado Briasco, thank you for your support, patience and understanding for many years. Susana Pedraza de la Cuesta, thank you for sharing with me your excitement about fishkeeping and various forms of automation. Jay van der Berg, thank you for all the intricate dance moves. Encarna Micó Amigo, I adore your sunny nature and your positive attitude to everything. Tommaso Mannucci, Maarten Gramsma, Nathan Zuiderveld, Joris Jonk, Sophie Armanini and many more, I really appreciate that I met all of you. I am grateful for all the music experiences we had together.

Tanya Tsui, thank you for being a great example of work-life balance and for being such an amazing housemate. I always enjoy philosophizing on science and life with you, which helped me to reflect on the last years of my PhD. Fabian Geizer and Kristen David, I am glad meeting you and spending time together. David Méndez Sevillano and Jelle Duijndam, thank you for being so open-minded and for all the exciting discussions we had. Matija Lovrak, Rohit Kacker, Christopher Rose, and Javier Fernández de la Fuente, our PhD Start-up group has added lots of bright moments to my PhD life. Guido Sluijsmans, I am thankful for the Dutch life crash-course. Peter Novák, thank you for all your scientific ideas, views on life and politics, and for introducing me to Apache Airflow.

Maria Vostrikova, you are simply my best university friend ever, and I am really happy having such a great soulmate. Alexandra Novikova, Andrey Andreev, Alexandra Vanchurina, Anna Fishbein, Maria Yarovaya, Ekaterina Melnikova, Elena Gorina, Dmitriy Paramoshkin, Olesia Paramoshkina, Dmitry Vdovkin, and Ilia Moltyaninov, thank you for being such great friends through all these years. I really appreciate your support that helped me to complete this work. Special thanks go to Andrey for creating the cover art for this dissertation. Many thanks to Elena Vadimovna Syulaeva and Ekaterina Petrovna Karpova (Mescheryakova) for teaching me English.

Fortunately, not all the people in this world speak English, so the rest of the acknowledgements are in Russian.

Благодарности

В первую очередь я хотел бы выразить сердечную благодарность родителям, сестре Арине и всей семье за безусловную и полную поддержку на протяжении всей моей жизни. Без вашей помощи я бы никогда не написал эту работу.

Кроме того, я очень благодарен своему школьному учителю информатики Татьяне Степановне Игнатьевой за доброту и привитый интерес к алгоритмам и программированию. Отдельное спасибо Галине Анатольевне Солодовой, научившей меня математике. Также я хотел бы отдать должное своим университетским преподавателям, в частности, Наталье Алексеевне Валиулловой, Ольге Анатольевне Шевченко, Светлане Дмитриевне Николайчук и Юлии Николаевне Копрянцевой за отличное образование, послужившее, по-сути, основой для этой диссертации.

Я особенно признателен Сергею Владимировичу Шибанову, который, благодаря своему бесконечному энтузиазму, стал моим проводником в науку.

Curriculum Vitae

Alexey Sergeyevich Ilyushkin (Алексей Сергеевич Илюшкин) was born in Kuznetsk, Penza oblast, USSR, on 21 January 1988. He graduated cum laude with an MSc degree in Software Engineering (Software for Computing Machinery and Automated Systems) from Penza State University, Russia, in 2009. In his thesis, supervised by Dr. Sergey Shibanov, Alexey focused on designing and implementing a complex application server for a distributed statistics-processing automated information system. During the whole period of his studies at Penza State University, Alexey was the recipient of the federal scholarship of the Russian Federation. From 2007 he has been participating in the research at the Department of Software Engineering and Computer Applications of Penza State University, which led to a number of publications in Russian. Alexey also co-supervised several undergraduate students.

After his graduation, Alexey worked as a software engineer in the Scientific Production Company Krug in Penza, where he obtained experience designing and implementing industrial Supervisory Control and Data Acquisition (SCADA) systems. After that, he worked in the Moscow-based SaaS company Moe Delo where he was a part of the team that developed a custom search engine.

In 2013, Alexey joined the Distributed Systems group of Delft University of Technology as a PhD candidate. His research focused on workflow scheduling and autoscaling of resources in large computing systems like clusters, multiclusters, and clouds, and was guided by Prof. Dick Epema and Prof. Alexandru Iosup. During his PhD studies, Alexey collaborated with researchers from Umeå University, Sweden, University of Würzburg, Germany, and Mälardalen University, Sweden through the Cloud Research Group of the Standard Performance Evaluation Corporation.

In 2015, Alexey received the best poster award at the ICT.OPEN conference. His papers were published in a number of international scientific conferences, and two of his papers were nominated for the best paper award in 2017 at ACM/SPEC ICPE and in 2018 at IEEE/ACM CCGrid conferences. Alexey was selected to participate at the 4$^{\text{th}}$ Heidelberg Laureate Forum in 2016. At Delft University of Technology, Alexey was also involved in teaching activities as an assistant in Cloud Computing, Distributed Computing Systems, and Distributed Algorithms courses.

In 2017, for half a year Alexey worked as a postdoctoral researcher at the Systems and Networking group of University of Amsterdam in the Horizon 2020 EU Cyclone project, which resulted in one co-authored publication. In 2018, Alexey joined as an evaluation committee member of the Artifact Evaluation track of the ACM/SPEC ICPE conference.

Alexey is currently employed as a scientific software engineer for Science and Technology B.V. in Delft, which he joined in 2018. One of the major projects where Alexey is at this moment involved is the development of prototype data processors for the ESA Sentinel-5 Earth observation satellite.

List of Publications

1. **A. Ilyushkin**, A. Bauer, A. Papadopoulos, E. Deelman, A. Iosup, Performance-Feedback Autoscaling with Budget Constraints for Cloud-based Workloads of Workflows. *Under review.*

2. A. Bauer, V. Lesch, L. Versluis, **A. Ilyushkin**, N. Herbst, S. Kounev, Chamulteon: Coordinated Auto-Scaling of Micro-Services. In *Workshop on Data Science of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Dallas, Texas, USA, 2019.

3. **A. Ilyushkin**, and D. Epema, The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Washington D.C., USA, 2018.

4. **A. Ilyushkin**, A. Ali-Eldin, N. Herbst, A. Bauer, A. Papadopoulos, D. Epema, A. Iosup, An Experimental Performance Evaluation of Autoscalers for Complex Workflows. In *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, Vol. 3, No. 2, 2018.

5. A. Uță, S. Au, **A. Ilyushkin**, A. Iosup, Elasticity in Graph Analytics? A Benchmarking Framework for Elastic Graph Processing. In *Proceedings of the 20th IEEE International Conference on Cluster Computing (CLUSTER)*, Belfast, UK, 2018.

6. **A. Ilyushkin**, A. Ali-Eldin, N. Herbst, A. Papadopoulos, B. Ghiț, D. Epema, A. Iosup, An Experimental Performance Evaluation of Autoscaling Policies for Complex Workflows. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, L'Aquila, Italy, 2017.

7. M. Slawik, C. Blanchet, Y. Demchenko, F. Turkmen, **A. Ilyushkin**, C. de Laat, C. Loomis, CYCLONE: The Multi-Cloud Middleware Stack for Application Deployment and Management. In *Proceedings of the 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Hong Kong, 2017.

8. S. Ma, **A. Ilyushkin**, A. Stegehuis, A. Iosup, Ananke: a Q-Learning-Based Portfolio Scheduler for Complex Industrial Workflows. In *Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC)*, Columbus, Ohio, USA, 2017.

9. A. Ali-Eldin, **A. Ilyushkin**, B. Ghiț, N. Herbst, A. Papadopoulos, and A. Iosup, Which Cloud Auto-Scaler Should I Use for my Application?: Benchmarking Auto-Scaling Algorithms. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*, Delft, the Netherlands, 2016. *Poster paper.*

10. **A. Ilyushkin**, B. Ghiț, and D. Epema, Scheduling Workloads of Workflows with Unknown Task Runtimes. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Shenzhen, China, 2015.

11. **A. Ilyushkin**, and D. Epema, Towards a Realistic Scheduler for Mixed Workloads with Workflows. In *Doctoral Symposium of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Shenzhen, China, 2015.

12. Шибанов С.В., Мезенков А.А., Шевченко О.А., **Илюшкин А.С.**, Принципы организации и функционирования активных пакетов для обмена информацией и конфигурирования распределенных приложений. *Известия высших учебных заведений. Поволжский регион. Технические науки.* 5–18, 2013.

13. **Илюшкин А.С.**, Мезенков А.А., Шибанов С.В., Павлов А.С., Применение технологии плагинов в активных информационных системах, *Международный сборник научных трудов «Математическое и программное обеспечение систем в промышленной и социальной сферах» I*, 65–71, 2011.

14. Шибанов С.В., **Илюшкин А.С.**, Мезенков А.А., Скоробогатько А.А., Программная платформа для построения адаптивных приложений информационных систем, *Труды Международного симпозиума «Надежность и качество» I*, 267–272, 2011.

15. Шибанов С.В., **Илюшкин А.С.**, Шевченко О.А., Макарычев П.П., Обмен информацией в распределённых информационных системах с использованием активного пакета, *Труды Международного симпозиума «Надежность и качество» I*, 288–292, 2010.

16. **Илюшкин А.С.**, Шибанов С.В., Шевченко О.А., Концепция активного пакета для распространения данных в распределенных системах, *Материалы конференции «Технологии Майкрософт в теории и практике программирования»*, 158–160, 2010.

17. **Илюшкин А.С.**, Шибанов С.В., Шевченко О.А., Система исполнения активного пакета в узлах распределенной системы, *Материалы конференции «Технологии Майкрософт в теории и практике программирования»*, 165–167, 2010.

18. Шибанов С.В., Казакова Е.А., Апаров М.И., **Илюшкин А.С.**, Архитектура метаданных в автоматизированной информационной системе «Прокуратура-статистика» как основа разработки и сопровождения, *Труды Международного симпозиума «Надежность и качество» II*, 298–301, 2008.

This dissertation addresses three key challenges that are characteristic to the online scheduling of workloads of workflows in modern distributed computing systems.

The first challenge is the realistic estimation of the resource demand of a workflow, as it is important for making good task placement and resource allocation decisions. Usually, workflows consist of segments with different parallelism and different interconnection types between tasks which affect the order how the tasks become eligible. Moreover, realistic task runtime estimates are not always available.

The second challenge is the efficient placement of workflow tasks on computing resources for minimizing average workflow slowdown while achieving fairness. A wrongly chosen task placement policy can easily degrade the performance and negatively affect the fair access of workflows to computing resources.

The third challenge is the automatic allocation (autoscaling) of computing resources for workflows while meeting deadline and budget constraints. Computing clouds make it possible to easily lease and release resources. Such decisions should be made wisely to minimize slowdowns and deadline violations, and to efficiently use the leased resources to reduce incurred costs.

To address these challenges, this dissertation proposes novel scheduling policies for workloads of workflows and investigates the applicability of relevant state-of-the-art policies to the online scenario. For new policies, implementation effort and suitability for production systems are kept in mind. The considered workflow scheduling policies are experimentally evaluated by conducting a wide set of simulation-based and real-world experiments on a private multicluster computer. Additionally, a Mixed Integer Programming (MIP) approach is used to validate the obtained real-world experimental results versus the optimal solution from a MIP solver.

ISBN 978-94-6366-228-4

TU Delft

Propositions

accompanying the dissertation

Scheduling Workloads of Workflows in Clusters and Clouds

by

Alexey Sergeyevich Ilyushkin

1. Dynamic online scheduling of workflows has lower complexity while delivering better end-user and system performance compared to plan-based online scheduling (this thesis).
2. Feedback mechanisms in scheduling workloads of workflows yield performance improvements (this thesis).
3. When scheduling workloads of workflows, task runtime estimates are only useful at extremely high system utilizations (this thesis).
4. The combination of general-purpose and workflow-specific autoscaling methods gives better performance than using these approaches separately (this thesis).
5. Algorithm designers should think more of the engineers who will implement their ideas.
6. Blindly following popular trends in the society for obtaining funds and attracting attention to research unnecessarily contributes to the growth of undesirable trends.
7. A less formal approach to scientific publications would help to conduct research faster and to reduce unnecessary costs.
8. The intangibility of software hides not only its beauty and complexity but also the resource and energy consumption it induces.
9. The time required for doing a PhD project has remained surprisingly stable over many decades despite all modern tools and equipment.
10. Low-hanging fruit becomes overripe faster.

These propositions are regarded as opposable and defendable,
and have been approved as such by the promotors
Prof. dr. ir. D.H.J. Epema and Prof. dr. ir. A. Iosup.