

Vrije Universiteit Amsterdam



Masters, Project Report

Understanding Datacenter Scheduler Programming Abstractions: Reference Architecture Design, Scheduler Analysis, and Cost Quantification



Author: Aratz Manterola Lasa (2690722)

<i>1st supervisor:</i>	Prof. dr. Alexandru Iosup
<i>daily supervisor, if different:</i>	Sacheendra Talluri
<i>2nd reader:</i>	Dr. Animesh Trivedi

July 4, 2023

Abstract

Datacenters are vital in our digital society, serving diverse sectors such as industry, academia, and public institutions. To efficiently manage resources, datacenters employ sophisticated schedulers with distinct capabilities accessed through their respective APIs. However, a lack of clarity exists regarding the offered programming abstractions and the reasons behind their selection, making it challenging to comprehend the differences and performance implications of these APIs.

In this work, we present a thorough investigation of programming abstractions provided by industrial schedulers, addressing their limitations and associated performance costs. Our study proposes a comprehensive reference architecture for scheduler programming abstractions. Firstly, we conduct an in-depth analysis of five prominent industrial schedulers to identify their programming abstractions: Kubernetes, SLURM, Apache Airflow, Condor, and Spark. Second, we analyze API extensions proposed in academic literature. Finally, we synthesize them into a unified reference architecture. Using this architecture, we analyze the APIs of industrial schedulers to uncover missing abstractions.

To assess the impact of these shortcomings, we evaluate the impact of extending a baseline scheduler APIs in three different ways. We add support for resource reservation, container migration, and storage metadata access. Our results demonstrate the performance trade-offs incurred by using simplified programming abstractions. Notably, we reveal that an API extension, such as container migration, can yield an 81% improvement in total execution time per task, underscoring the significance of addressing these limitations. Our findings enable schedulers to identify areas for improvement and provide valuable insights for future scheduler development.

To facilitate further research and reproducibility, all relevant software and data artifacts from our study are publicly available at <https://github.com/aratz-lasa/opensdc>.

This work not only enhances our understanding of programming abstractions in industrial schedulers but also serves as a reference for evaluating and advancing scheduler design and development in the future.

Contents

1	Introduction	6
1.1	Research questions	7
1.2	Research methodology	8
1.3	Stakeholders and use cases	9
1.4	Thesis contribution	11
1.5	Thesis structure	11
2	Background	12
2.1	Workload	12
2.2	Scheduling	13
2.3	Scheduling resources	13
2.4	Programming abstraction	13
2.5	Industrial schedulers	14
2.6	Related Work	15
3	Analysis of the programming abstractions of industrial schedulers	17
3.1	Overview	17
3.2	Methodology	17
3.3	Main requirements	18
3.4	Design principles	19
3.5	Aggregated programming abstractions analysis	19
3.6	Abstraction - Provision	20
3.7	Abstraction - Configure scheduler	27
3.8	Abstraction - Constraints	29
3.9	Abstraction - Quality of Service	36
3.10	Abstraction - Manage data	41
3.11	Abstraction - Communicate	45
3.12	Limitations	49
3.13	Summary	49
4	Reference architecture of scheduling programming abstractions	50
4.1	Overview	50
4.2	Methodology	50
4.3	Main requirements	51
4.4	Design principles	52
4.5	Emerging concepts from academia	53
4.6	Reference architecture	56
4.7	Validation through mapping of schedulers	61
4.8	Mismatch between industrial scheduler aggregated mapping and the reference architecture	69
4.9	Limitations	70
4.10	Summary	70
5	Experiments with the reference architecture	72
5.1	Selection of under-implemented scheduling APIs to experiment	72

5.2	Traces	74
5.3	Execution	75
5.4	General requirements	76
5.5	Extension 1: Reducing VM waiting times and slowdowns using reservations	76
5.6	Extension 2: Reducing VM total times using container migrations	90
5.7	Extension 3: Reducing Data Workflows execution times using metadata access	105
6	Conclusion	116

1 Introduction

Society’s increasing dependence on digital technologies and infrastructure has led to the widespread use of datacenters for deploying various services [14, 22]. Datacenters rely on sophisticated schedulers to efficiently manage resources and meet the demands of these services [18, 35]. For example, today, schedulers must offer capabilities to co-locate tasks with data [44] to reduce data transfer time between jobs over a network. Similarly, they also provide the ability to make reservations and submit tasks in advance [43] to ensure that there will be enough resources in the future to carry out the required computation. However, the programming abstractions¹ offered by these schedulers and their implications on performance and user control remains a topic of interest and research.

Schedulers play a crucial role in orchestrating the allocation and execution of tasks within datacenters. The interfaces they offer to users determine how much users can mold the orchestration process to support their application needs. Different schedulers provide varying levels of programmability and control to users [37, 48, 40, 21]. On one end of the spectrum, some schedulers provide restricted programming abstractions, aiming to minimize user input and tightly control the scheduling process. On the other end, schedulers offer more flexible interfaces, empowering users with greater control over resource allocation and job placement [37, 48]. For example, Tetrisched [43] allows users to make reservations, and the Energy-Credit Scheduler [27] provides an expressive API to the users for controlling energy consumption. This discrepancy raises several questions about the impact of design choices on performance, simplicity, and control that users can achieve.

The first question we raise about scheduler abstraction design is: **What programming abstractions are common in current schedulers?** Knowledge of programming abstractions in existing industrial schedulers informs designers of what is currently available to the users. The programming abstractions available in academic research schedulers can also suggest to designers which abstractions are necessary to incorporate the latest resource management techniques proposed by the research community.

The second question is: **What programming abstractions are sacrificed for simplicity?** Usually, academic schedulers offer a wide set of programming abstractions, allowing the users to customize several aspects of scheduler operational behavior. On the other hand, industrial schedulers usually implement a restricted subset for increased security and robustness [36]. In this work, we propose a comprehensive reference architecture for datacenter scheduler programming abstractions. This reference architecture provides a unified framework for analyzing and comparing existing schedulers, identifying similarities, differences, and potential shortcomings [25, 4]. By mapping various schedulers onto this reference architecture, we can identify missing abstractions in industrial schedulers.

The third question is: **What is the performance cost of the sacrificed abstractions?** Despite their security and robustness benefits, simpler abstractions have a performance cost. The performance cost is usually in the form of underutilized resources and slow-to-complete application jobs. To shed light on this issue, we conduct three experiments to quantify the performance costs of missing abstractions; we conduct experiments

¹We use programming abstraction and API interchangeably.

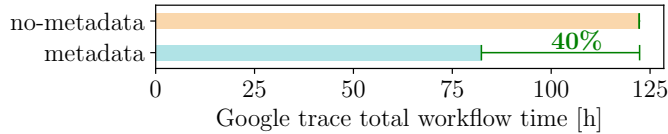


Figure 1: Performance penalty due to a missing programming abstraction: storage metadata access.

using real-world traces collected by major datacenter operators, such as Google and Microsoft. These experiments allow us to evaluate the impact of specific programming abstractions on task runtime, data workflows, and resource utilization. By quantifying the performance costs, we aim to provide schedulers with insights into the trade-offs associated with their programming abstractions and identify areas for improvement. Figure 1 depicts an exemplary result with the 85th percentile total execution time of tasks in a trace from Google [45]. We consider a scheduler that implements a crucial abstraction lacking in many industrial schedulers: metadata access to the data stored on datacenters’ object storage service (e.g., AWS S3). Comparing it against a scheduler lacking this abstraction, we observe a 40% reduction in total execution time when using the abstraction.

Our research contributes to the field by offering a comprehensive understanding of scheduler programming abstractions and their implications on performance and user control. By providing a reference architecture and quantitative analysis of missing abstractions, we enable schedulers to enhance their programming interfaces and optimize resource allocation to meet the demands of modern applications.

Establishing a common reference architecture brings several benefits. First, the reference architecture provides a common framework for analyzing and comparing existing industrial and academic schedulers. The comparison helps identify similarities, differences, and potential shortcomings, thus enabling the assessment of different implementations and design alternatives [4]. Second, it serves as a knowledge base for designing better schedulers that can meet the demands of modern applications by addressing shortcomings [17, 29, 8, 4]. Finally, establishing a common reference model reduces the risk of a scheduler being specialized to the current interface by providing a view of all possible programming interfaces. This helps avoid non-extensible designs that must be re-engineered at great development cost, as has been the case with Condor [40] and Borg [7] when the need for a new design arises.

In conclusion, this work aims to bridge the gap between scheduler design choices and their impact on performance and user control. By examining the programming abstractions offered by different schedulers, quantifying their performance costs, and providing a reference architecture, we provide valuable insights for improving scheduler designs and advancing the field of datacenter resource management.

1.1 Research questions

This section outlines the key research questions and the approach this thesis takes to address them. More precisely, we decompose the project into three research questions, that lead us to build and demonstrate the importance of a reference architecture:

RQ1 What are the programming abstractions of mainstream industrial sched-

ulers?

For building a reference architecture of scheduling programming abstractions and analyzing what costs they impose, it is fundamental first to understand the state-of-practice. For that, we study and model industry mainstream schedulers.

RQ2 What programming abstractions of scheduling are missing in mainstream industrial schedulers?

For designing and building schedulers, it is necessary to clearly understand what programming abstractions the scheduler could expose to their users. Or in other words, what is the potential functionality of a scheduler? We design a reference architecture for scheduling programming abstractions. Once the reference architecture is designed, we analyze the industrial schedulers and identify what programming abstractions are missing in the industry.

RQ3 What are the costs imposed by the missing programming abstractions of schedulers?

We hypothesize that existing schedulers have missing programming abstractions and sacrifice performance in exchange for simplicity. Therefore, based on the results from RQ2, we conduct experiments to prove that extending their existing APIs can increase the scheduler’s performance.

1.2 Research methodology

In this thesis, we approach the research questions through conceptual, technical, and experimental work. All the work is based on the following two high-level philosophies:

- *Everything we do in the thesis is reasoned and justified, and if there are alternative paths, we mention them and justify our choice.*
- *The problem and the solution evolve together since it is difficult to understand the problem until you start working on the solution.*

To address RQ1, we present an analysis of the programming abstractions of 5 mainstream schedulers. We first select five mainstream schedulers by consulting experts in the scheduling field. Next, we consult the official documents of their APIs, and methodically we go through the entire documents, listing the different components we find in them. All this study converges in an aggregated table where we list the main abstractions that we identify across the five schedulers, and we use it to compare them. It is important to note that the final result of the aggregate table evolves as we analyze the different schedulers. As mentioned, the solution and the problem evolve together during the thesis.

In RQ2, we design a reference architecture for scheduling programming abstractions and use it to identify shortcomings in industrial schedulers. For designing a reference architecture, we develop our own method. We do not aim to create an optimal methodology but rather to have a process that is justified and reasoned well enough. The method is based on analyzing the industrial schedulers, conducting a survey of research schedulers from academia, unifying the industrial and research schedulers, and finally, generating a reference architecture by applying intuition and creativity to the unified design. Once the

reference architecture is developed, we validate it by mapping the five industrial schedulers to the reference architecture. But most importantly, this mapping is used for identifying the shortcomings of the schedulers.

To address RQ3, we perform three experiments to prove that the shortcomings identified in RQ2 have performance costs for the schedulers. We select three shortcomings from mapping the industrial schedulers in the reference architecture and design an experiment for each. For every experiment, we first identify the requirements, define the system model to narrow the scope of the experiment, list the different implementation alternatives, specify the experiment’s configuration and design, and describe and implement prototype software to perform the experiments. Secondly, we run the experiment using the prototype software and gather all the necessary metrics and data. Finally, we present and analyze the experiment results from the metrics. This includes formatting, cleaning, ordering the experiment data, and offering an interpretation and reasoning of the results.

Everything we generate in this work is entirely open and designed so that anyone can reproduce it. We place a lot of emphasis on providing reproducible science.

1.3 Stakeholders and use cases

To answer the research question, it is necessary to identify the stakeholders and the use cases to address them. This way, we can identify the requirements for each research question based on the stakeholders and the use cases.

We identify five main stakeholders of this thesis:

- S1 Customers and end-users** do not make use of the thesis directly, but they are the ones that interact with the scheduler programming abstractions. So, they indirectly affect the schedulers’ design and their programming abstractions. These are the ones that need to use a scheduler to perform their tasks, such as analyzing data, deploying cloud services for their users, etc. Users expect schedulers to meet specific requirements or SLAs; when they are not satisfied, they are the most affected, and those who claim a lack of compliance. In the end, they are the ones who guide and make the design of scheduler APIs evolve based on their needs.
- S2 datacenter providers** are the ones that configure and sometimes even develop the schedulers and, consequently, their APIs. The datacenter providers are the owners of the resources that schedulers manage. Therefore, they are interested in the scheduler making efficient use of resources and increasing performance metrics such as utilization or reducing waiting times. Providers specify certain SLAs to their users, and to comply with them, they make use of schedulers and their programming abstractions.
- S3 Commercial scheduler builders** are the ones who design and build the schedulers. Those who design and choose the programming abstractions that the schedulers implement. Therefore, they are going to be the most important stakeholders since they are the ones that are going to make use of the results of the thesis. There are two main cases in which they might use this thesis, specifically the resulting reference architecture. First, when designing a new scheduler, identify which abstractions the scheduler could offer, depending on the needs the scheduler seeks to fill. Second,

the reference architecture would be useful for analyzing and improving an existing scheduler design, looking for gaps in the API.

S4 Scientists research new ways to improve the management of datacenter resources, and one of how they do this is through the design of schedulers. Therefore, it is essential that before coming up with a new design, they understand the set of functionalities that a scheduler could or should offer to users and what the API should look like.

S5 Students must learn what functionalities a scheduler offers to users since they are the future engineers and scientists who are going to design or make use of schedulers. Building a theoretical knowledge of schedulers and a practical experience of the problems schedulers solve is essential to acquire this understanding. Scheduling programming abstractions serve as introductory conceptual models and a high-level summary of the scheduling field.

Based on the stakeholders, we devise the following use cases:

UC1 Optimization of jobs Scheduler users can use the thesis to understand the various features that schedulers offer and how to leverage them to optimize the execution of their jobs. For example, by using the reference architecture, they may identify if their workloads have temporal patterns, they can use this insight by specifying temporal constraints when submitting tasks or jobs to the scheduler. RQ1 addresses this use case in Section 3 and RQ2 in Section 3.

UC2 Optimization of the utilization of datacenter resources Datacenter providers can use the thesis to decide what programming abstractions they can offer to their users to increase the efficiency in managing their resources. For example, using this thesis, they can learn to leverage user-level migrations through a callback abstraction and consequently efficiently reduce tenant interferences. This use case is addressed by RQ2 in Section 3 and exemplified by RQ3 in Section 41.

UC3 Development and design Developers of commercial schedulers, such as Kubernetes and Apache Airflow, can design and develop their solutions using the thesis. The thesis gives them a general overview of what their product could offer in terms of APIs, and then, based on their specific use case, they can choose the subset of programming abstractions they need to implement. This use case is addressed by RQ2 in Section 3 and exemplified by RQ3 in Section 41.

UC4 Shortcomings identification Developers of commercial schedulers like Kubernetes and Apache Airflow may need to improve their current solutions because their users request them or because they want to increase the features and quality of their products. In both cases, this thesis allows them to analyze their solutions by identifying missing gaps in their schedulers' programming abstractions. Consequently, they can leverage that analysis to recognize if any missing APIs could interest their users. This way, the thesis becomes a critical element of the evolution of commercial products. This use case is addressed and exemplified by RQ2 in Section 3.

UC5 Research Researchers can use the thesis to investigate new ways and contexts in which the different programming abstractions can be used to optimize performance,

energy efficiency, etc. In addition, the thesis allows them to conduct research and experiments more systematically through the resulting reference architecture. This use is exemplified by RQ3 in Sections 41.

UC6 Education Students can use the thesis to introduce themselves to schedulers since it allows them to learn about schedulers through their APIs or abstractions, facilitating the understanding of the context and importance of schedulers. RQ2 addresses this use case in Section 3.

1.4 Thesis contribution

This thesis has resulted in the following disseminated materials and developed software:

1. Articles submissions.

- Article submitted to a leading peer-reviewed journal of Middleware Systems, as the first author: A. Manterola Lasa, S. Talluri, A. Iosup, The Cost of Simplicity: Understanding Datacenter Scheduler Programming Abstractions, Middleware, December 2023.
- Article accepted in a leading peer-reviewed journal of Dutch Computer Systems, as the first author: A. Manterola Lasa, S. Talluri, A. Iosup, A Reference Architecture for Datacenter Scheduler Programming, CompSys, June 2023.
- Article accepted in a leading peer-reviewed journal of Performance Engineering, as the first author: A. Manterola Lasa, S. Talluri, A. Iosup, A Reference Architecture for Datacenter Scheduler Programming Abstractions: Design and Experiments (Work In Progress Paper), ICPE, April 2023.

2. Published open science artifacts.

- Publicly disseminated artifacts, following the FAIR principles for scientific data (Findable, Accessible, Interoperable, and Reusable), published on the Zenodo Open Science platform: <https://zenodo.org/record/7996281>
- Free and Open Source Software (FOSS) software artifacts published on GitHub, for inspection and reuse: <https://github.com/aratz-lasa/opendc>

1.5 Thesis structure

The remainder of the thesis is structured in the following way. In Chapter 2, we describe relevant background information, and in Chapter 3, we present an analysis of the requirements for this work. Next, in Chapter 4, we offer an analysis of programming abstractions of industrial schedulers. Chapter 5 is a reference architecture of scheduling programming abstractions and a critique of industrial schedulers. In Chapter 6, we design and justify the experiments for evaluating the costs of scheduler shortcomings. We present three different experiments in Chapter 7. Finally, in Chapter 8, we summarize the contributions of this thesis and propose future work that could emerge from this project.

2 Background

This chapter presents an overview of the topics and concepts related to scheduling and programming abstractions.

As distributed systems become more heterogeneous over time, scheduling is one of the most complex issues within distributed systems. Therefore, before presenting the analysis and the shortcoming of industrial schedulers, it is necessary to understand the critical concepts of scheduling and programming abstractions and the context of this work. To encapsulate the context and explain the central concepts, we present a set of system models: workload, scheduling resources, resource management and scheduling, and programming abstractions. Lastly, we introduce five industrial schedulers, which will be used throughout the thesis. Next, we explain each of them in detail.

2.1 Workload

The workload is executed using the resources the scheduler assigns to the user. There are several types of workload, and in this work, we assume that all of them fit the morphology of a workflow: a stream of jobs that are made up of one or several tasks, and there are dependencies in the precedence between the tasks.

In the context of scheduling, there are four primary types of workflows. These types have been defined in previous research, such as the work by Andreadis et al. [4]. In this study, the authors provide a clear and well-defined framework for understanding modern workflow management systems' different types of workflows. By adopting their definitions, we can ensure a common understanding of the different types of workflows, facilitating communication and collaboration among researchers and practitioners.

1. **Batch workflows** are workloads comprising several tasks with dependencies between them. Dependencies between tasks create precedence and can be visualized as a DAG.
2. **Bag-of-tasks** are jobs formed by several tasks without any dependency between them. Therefore, there is no precedence, and they can be executed arbitrarily.
3. **Long running tasks** run for a very long time and are usually inside a host such as a VM. These tasks are usually services offered by businesses, waiting for user requests.
4. **Managed jobs** are workloads where a manager coordinates all the tasks, such as Spark. Normally the manager tends to be a long-running task, while the tasks she coordinates tend to have a shorter duration.

The users request the scheduler to specify the requirements to execute the workload. Usually, the requirements are determined by the amount of CPU and memory. However, in some cases, other things, such as the start time, the dependencies between the tasks, the scalability of the resources, etc., are also specified. To submit the workload requirements, users interact with the APIs that schedulers offer, that is, with the programming abstractions.

2.2 Scheduling

A user submits a workload to use the resources through a central component, the scheduler [32, 7]. The scheduler takes care of several tasks: finding resources to assign to the workload based on the specified requirements, transferring the workload to the resources, starting the execution of the workload, managing the workload through its lifecycle (from placement to workload cleanup), and notifying to the user about lifecycle events.

Throughout the execution of a workload, the resource requirements of the workload and the number of available resources available to the scheduler can change. Therefore, the scheduler must adapt to changing workload requirements by increasing or decreasing dynamically allocated resources. It does this through a subcomponent called autoscaler [2]. At the same time, the scheduler must also preempt, recover, and migrate workloads when the amount of available resources changes. Depending on the priorities assigned to each workload and the type of workload, the scheduler can decide whether to preempt the workload or migrate it to another host in the datacenter.

Schedulers can be monolithic [28] and run in a single process that handles all tasks. They can be distributed where different tasks are split into other components, such as the autoscaler [2]. In the same way, the scheduler and its members can be replicated in several processes in parallel. Still, they must coordinate among themselves when assigning resources to the workloads. In addition, schedulers can be centralized [33], where a single entity implements the scheduler and dictates the policies and mechanisms, or it can be decentralized [40] so that several entities implement a scheduler. Each of them has different policies and mechanisms. When the scheduler is decentralized, the other instances must coordinate through a common protocol and sometimes use a central matchmaker.

2.3 Scheduling resources

The workloads are executed on top of the resources that the scheduler manages. Resources are physical machines, usually within a datacenter. In the datacenter, there are several clusters with several hosts each, and each host is a node in a rack. Hosts are heterogeneous, offering different amounts and types of resources. Each host virtualizes its resources in VMs or containers through a hypervisor and runs multiple independent workloads simultaneously.

In this work, we model the resources of a host as the combination of CPU, memory RAM, and storage. CPUs can have different frequencies and amounts of core. And memory and storage can have different sizes. We model the resource consumption in a discretized way, where the workload reports at each time step how many resources it has consumed, and the hypervisor consolidates the consumption of the different workloads through a fair-sharing policy.

2.4 Programming abstraction

Schedulers offer a set of programming abstractions for users to interact with. Programming abstractions are the API offered by schedulers and are the language by which the user submits workloads and modifies the workload’s requirements during the workload’s life cycle.

This is the central concept studied in this work. Programming abstractions are offered through a GUI, CLI, or a protocol such as HTTP. Users must identify themselves before performing any operation and provide a form of payment for the resource consumption of their workloads.

Programming abstractions can be imperative, declarative, or a mixture. Usually, when the language is imperative, programming abstractions offer programmability to perform a specific action on the workload. When declarative, they offer reconfigurability to alter the state of the workload. Besides programmability and configurability, schedulers offer observability over the underlying workload or resources. We explicitly exclude observability from this work, as it is not a direct interaction between the user and the scheduler. However, it is important to note that observability is an indirect form of exchange since it improves the user’s decision-making and consequently influences their use of programming abstractions. For example, you might decide to reduce the resource requirements of the workload if you find that utilization is very low.

2.5 Industrial schedulers

In Section 3, we analyze five industrial schedulers: Kubernetes, SLURM, Apache Airflow, Condor, and Spark. Therefore, we first introduce them and provide a short overview of each.

Kubernetes, also known as k8s, is a cluster management and container orchestration software for automating deployment, management, and scaling. In a typical scenario, Kubernetes is deployed on top of VMs in a datacenter. It is the most popular orchestration software, and it is used in all fields, from data analytics to machine learning applications. Kubernetes focuses on orchestration. However, orchestration and scheduling are interrelated. Orchestration leverages scheduling features for finer coordination and deployment. Consequently, Kubernetes exposes a rich set of scheduling programming abstractions to its users.

We analyze the programming abstractions by consulting <https://kubernetes.io/docs>.

SLURM is a cluster management and job scheduling software that runs on 60% of the TOP500 supercomputers as the resource manager [33]. It has three main functions: allocate access to resources to users, provide a framework for starting, executing, and monitoring jobs, and arbitrating contention to resources. Therefore, SLURM is considered a scheduler and exposes scheduling programming abstractions to their users for implementing these essential functions.

We analyze the programming abstractions by consulting <https://slurm.schedmd.com/documentation.html>.

Spark is a large-scale data analytics processing system that manages clusters with implicit data parallelism and fault tolerance. Spark’s architecture is based on the resilient distributed dataset (RDD), a collection of read-only data partitions distributed among machines on a cluster. Spark is not directly considered a scheduler, but it exposes scheduling programming abstractions to users for high performance on data analytics. The abstrac-

tions mainly focus on optimizing resource usage for data management, but they also offer more basic abstractions, such as provisioning and monitoring.

We analyze the programming abstractions by consulting <https://spark.apache.org/docs/latest/>.

Condor is a resource management system for compute-intensive jobs designed for the Grid. It can schedule within a single private cluster and in a global grid across multiple authorities and locations. Condor implements a powerful model where resource consumers are matched with resource owners, avoiding multiple queues for submitting jobs with different requirements. Despite not being the most popular scheduler commercially, it is one of the most potent and essential designs in the field due to its sophistication.

We analyze the programming abstractions by consulting <https://htcondor.readthedocs.io/en/latest/>, and [40].

Apache Airflow is a software for scheduling and monitoring workflows. It lets users specify tasks and their dependencies by constructing fine-grain DAGs, where each node represents a task. The workflows are not limited to a specific domain, but it allows to build of pipelines for anything, from Machine Learning models to data transfers. Both the definition of the DAG workflows and all the other functionalities related to the scheduling of the tasks are specified and controlled programmatically through Python code.

We analyze the programming abstractions by consulting <https://airflow.apache.org/docs/>.

2.6 Related Work

In the field of scheduling, various conceptual models and reference architectures have been proposed to understand the internal workings of schedulers. Schopf’s multi-stage model of the grid scheduling process [25], the Global Grid Forum [20], and the datacenter scheduler reference architecture [4] provide valuable insights into the overall scheduling process. However, these models primarily focus on the internal aspects and lack a detailed exploration of the external-facing components, specifically the programming interface.

Several conceptual models of APIs have been introduced for different computing environments, such as grid computing and cloud computing. Foster et al. presented a reference architecture for grid computing [17], and the National Institute of Standards and Technology (NIST) introduced models for cloud computing [29]. While these models offer valuable guidance for designing APIs within their respective domains, they do not address the specific API requirements of schedulers like Spark and Kubernetes. These schedulers have unique characteristics and demands that call for dedicated attention to their programming interface design.

Recent efforts have focused on developing schedulers that integrate multiple scheduling abstractions into a unified system. Projects like Ghost [23] and ESCHER [5] aim to provide advanced scheduling capabilities. Ghost enables users to have greater control over the scheduling process by delegating OS kernel scheduling decisions to them. It offers mechanisms to implement multiple scheduling policies, but it does not support different implementations of the scheduler mechanisms themselves. On the other hand, ESCHER

allows users to express arbitrary scheduling constraints as resource requirements, enabling fine-grained control over resource allocation. While ESCHER focuses on resource allocation given constraints, it does not address the modeling of complex interactions between users, resources, and the scheduler.

In contrast to the existing work, our research complements these efforts by specifically addressing the external-facing aspects of scheduling, emphasizing the design and implementation of programming abstractions in schedulers. By exploring the unique requirements and characteristics of schedulers like Spark and Kubernetes, we aim to provide a comprehensive reference architecture that facilitates efficient and user-friendly scheduling.

3 Analysis of the programming abstractions of industrial schedulers

What are the programming abstractions of mainstream industrial schedulers? To answer this question, it is necessary to model the state of the industry on scheduler APIs. The analysis will make us understand the current programming abstractions and let us study whether they are sufficient and what features they may be missing.

Based on our experience and consulting expert knowledge in the scheduling field, we identified and selected a group of 5 industrial schedulers. We consider the selected schedulers to represent the broader set of schedulers available in the industry. The group of schedulers is formed by: Kubernetes (v1.27) [3], SLURM (v23.02) [28], Spark (v3.4.0) [46], Condor (v10.4.3) [40], and Airflow (v3.3.0) [1]

3.1 Overview

First, we explain the methodology used for the analysis, the requirements, and the design principles. Next, we present a table where we unify and aggregate all the programming abstractions we identify in the industry and analyze which schedulers implement the different abstractions. This offers a clear view of which programming abstractions industrial schedulers implement and which ones they miss. Then, we explain how each scheduler implements the identified abstractions. In this way, we compare different alternatives to implement the same abstraction, and at the same time, we justify that the schedulers execute the aforementioned programming abstractions. Lastly, we explain the limitations of this analysis of industrial schedulers.

Our contribution is composed of multiple items:

1. We describe the methodology we follow to analyze the state of the industry (Section 3.2).
2. We analyze the requirements for the analysis (Section 3.3).
3. We specify the design principles for the analysis (Section 3.4).
4. We present an aggregated analysis of the industrial schedulers' programming abstractions (Section 3.5).
5. We explain the limitations of the analysis (Section 3.12).

3.2 Methodology

Before carrying out the analysis of scheduling programming abstractions, we define a methodology that we follow. We keep the process flexible because we are in the early stages of the work, and the problem and the solution are still evolving. However, we define a list of high-level steps that allow us to structure and limit the analysis:

1. **Analysis of requirements and design principles.** First, we identify the requirements of the reference architecture and the design principle by which we will guide

and evaluate both the design and the result. This allows us to justify and reason the reference architecture formally.

2. **List programming abstractions:** For each scheduler, we walk through the specifications of its APIs in the sources identified in the previous section. We list all the actions the API offers users, noting a short explanation and concrete examples.
3. **Create diagrams to the group and relate concepts:** For each scheduler, we convert the list of API actions to a diagram in which we group the related concepts.
4. **Unify concepts across schedulers:** We unify the different words and concepts identified in each scheduler to use the same terminology and groupings.
5. **Aggregate results:** We aggregate the different concepts and groupings identified in the schedulers to a single table, listing the categories and the images within each category.

3.3 Main requirements

To analyze the state of the industry of scheduling programming abstractions, we first identify the requirements that must be met. To specify the requirements. Below we list the functional and non-functional requirements that guide the analysis.

We identify two functional requirements:

FR1 Model existing industrial schedulers. For the thesis to impact the real world, working with popular schedulers in the industry is necessary. Since these are the market leaders and the ones that have the greatest impact, it is necessary to model the current mainstream schedulers since these models offer the user a clear understanding of the APIs offered by each scheduler in the industry and be able to compare and analyze them.

FR2 Expand the problem and the solution towards a reference architecture. In addition to offering a high-level overview of the abstractions provided by schedulers, this analysis is also a first step toward designing a reference architecture for scheduling programming abstractions. The problem and the reference architecture design are still evolving in this step. Therefore, the analysis of industrial schedulers should expand knowledge on how to design the reference architecture, in turn beginning to show where there may be shortcomings in industrial schedulers and which abstractions are relevant for identifying differences between schedulers.

We identify one non-functional requirement:

NR1 Simplicity. The analysis is aimed at a broad group of stakeholders, from university students to industrial computer engineers. Therefore, the study's results should be presented clearly so stakeholders can easily understand and interpret the analysis results.

3.4 Design principles

In addition, to carry out the specified steps in the methodology, we follow two design principles:

P1 Avoidance of concept types. The analysis serves to understand what existing schedulers offer in the industry. For that, it is essential to keep the analysis design simple and avoid creating different types of concepts when modeling the APIs. We do not differentiate the actions from the inputs that the actions receive, such as the lease action and the resource requirements specified in the lease. The two are essential concepts, so we keep the two separate but at the same level in the model. In addition, the analysis also serves to explore and build a vision for the reference architecture. That is why it is essential not to be tied to any specific framing or approach in this analysis and, to do so, not to model different types.

P2 Grouping of related concepts. Several concepts are related to each other. Therefore, to facilitate comprehension, related abstractions are grouped. This allows developers to use these groupings to design the scheduler architecture and students to simplify the establishment of their knowledge.

3.5 Aggregated programming abstractions analysis

In Table 1, we present the summary of the programming abstractions that industrial schedulers implement, identifying which abstractions each scheduler implements. Analyzing the industrial schedulers, we identify 21 programming abstractions and group them into six general categories context for the functionality of each abstraction. In the following sections, we describe the different abstractions and explain which schedulers implement them and how they do them.

Table 1: Aggregated analysis of scheduling programming abstractions from industrial schedulers.

Programming abstractions		Schedulers				
		Ku	Sl	Sp	Co	Ai
Provision	lease / release	●	●	●	●	●
	scale	●	○	●	○	○
	preempt	●	●	○	●	○
	recover	●	●	●	●	●
Configure scheduler	scheduling policy	●	●	●	○	○
	scheduling frequency	○	●	○	●	●
Constraints	space constraints	●	●	●	●	●
	time constraints	●	●	○	●	●
	affinities	●	○	○	○	○
	task dependencies	○	●	●	●	●
QoS	priorities	●	●	●	●	●
	group quotas	●	●	●	●	○
	exclusivity	●	●	○	○	○
Manage data	access input data	●	●	●	●	●
	replicate	○	○	●	○	○
	partition	○	○	●	○	○
	recover	○	○	●	●	○
Communicate	signal	●	●	○	●	○
	stdin	●	●	○	●	○
	callback	●	●	●	○	●

Legend: ●/○ = full/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow.

3.6 Abstraction - Provision

In this section, we present the abstraction for the provisioning of resources. This is the main and most basic abstraction of scheduling since it is by which resources are acquired and managed.

3.6.1 Lease / Release

Lease is the main abstraction within provisioning, which is the assignment and activation of resources. All schedulers offer this abstraction since it is the most basic scheduling operation. However, each scheduler provisions different execution units.

Kubernetes provisions containers encapsulated into pods and jobs by submitting a YAML file that describes the containers. Here is an example of a YAML file that describes a pod:

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: example-deployment
5  spec:
6    template:
7      metadata:
8        labels:
9          app: example
10     spec:
11       containers:
12         - name: example
13           image: example:1.28

```

A deployment named `example-deployment` is created in this example, indicated by the `.metadata.name` field. The `template` field contains the description of the pod. The Pods are labeled `app:example` using the `.metadata.labels` field. The Pod template's specification, or `.template.spec` field, indicates that the Pods run one container, `example`, which runs the `example` container image at version 1.14.2.

In case the file is saved as `example.yaml`, the command to submit the file and perform the provisioning is:

```
kubectl apply -f example.yaml
```

SLURM provisions jobs by running a CLI command and passing arguments to it. It does not require a file to describe what is being provisioned. Here is a specific example:

```
sbatch --job-name=example run.sh
```

In this example, the command `sbatch` submits a batch script to SLURM. the `job-name` parameter specifies a name for the job lease. The specified name and the job ID number will appear when querying running jobs on the system. Lastly, `run.sh` specifies the script's name that is submitted.

Spark provisions applications by running a CLI command. Here is a specific example:

```

./bin/spark-submit
--class org.apache.spark.examples.SparkPi
--master spark://207.184.161.138:7077
/path/to/examples.jar

```

In this example, the `class` argument specifies the entry point for the application that is `org.apache.spark.examples.SparkPi`. The entry point refers to the spark backend executable

that reads and runs the user application. The argument *master* specifies the Spark master URL for the cluster that is *spark://23.195.26.187:7077*. The master is the process that requests resources in the cluster and makes them available to the Spark backend. And lastly, the path */path/to/examples.jar* to a bundled jar, including the user application and all dependencies.

Condor submits a descriptive file to containers, VMs, and jobs. Here is a specific example:

```
1 executable = example
2 arguments  = SomeArgument
3
4 queue
```

In this example, it queues the program *example* with the arguments *SomeArgument* for execution somewhere in the pool. The *queue* statement tells Condor that the user is done describing the job and to send it to the scheduler queue for processing.

In case the file is saved as *submitexample*, the command to submit the file and perform the provisioning is:

```
condor_submit submitexample
```

Airflow provisions workflows by specifying a Python script. Here is an example of submitting a workflow:

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator
5
6 def example():
7     return 'Hello world from first Airflow DAG!'
8
9 dag = DAG('hello_world', description='Hello World DAG')
10
11 example_operator = PythonOperator(task_id='example_task',
12 ↪ python_callable=example, dag=dag)
13
14 example_operator
```

In case the file is saved as *submitexample*, the command to submit the file and perform the provisioning is:

```
airflow tasks run example\_operator example\_task
```

3.6.2 Scale

. Scale is the abstraction that specifies the resources scaling plans for automating how they respond to changes in workload or demand.

Kubernetes offers scaling abstractions via horizontal pods autoscalers, which allow users to specify scaling policies for containers via YAML configurations by setting the load at which the scaling is triggered. Here is a specific example:

```
1  apiVersion: autoscaling/v2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: example
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: example
10   minReplicas: 1
11   maxReplicas: 10
12   metrics:
13   - type: Resource
14     resource:
15       name: cpu
16       target:
17         type: Utilization
18         averageUtilization: 50
```

In this example, with this metric, Kubernetes will keep the average utilization of the pods in the scaling target at 60%. If the utilization moves up or down from the specified target, Kubernetes provisions more or fewer containers within the range specified by the *maxReplicas* and *minReplicas* arguments, that is, between 1 and 10.

Spark also allows users to specify scaling for the application. Still, instead of selecting a triggering load, users select the minimum and maximum resources that applications can vary and other secondary configurations, such as the initial set of resources. The scaling is specified through a configuration file. Here is a specific example:

```
./bin/spark-submit
--class org.apache.spark.examples.SparkPi
--master spark://207.184.161.138:7077
--conf spark.dynamicAllocation.enabled=true
--conf spark.dynamicAllocation.minExecutors=1
--conf spark.dynamicAllocation.maxExecutors=10
/path/to/examples.jar
```

In this example, three configuration parameters are set. First, the

spark.dynamicAllocation.enabled flag is set to *true*, indicating the use of dynamic resource allocation, which scales the number of executors registered with this application up and down based on the workload, where executors are the worker nodes that help in running individual tasks by being in charge of a given spark job. Then, the parameters *spark.dynamicAllocation.minExecutors* and *spark.dynamicAllocation.maxExecutors* specify the lower and upper bound for the number of executors.

SLURM, **Condor** and **Airflow** do not provide the scaling abstraction.

3.6.3 Preempt

Preempt is the abstraction that specifies the abortion of execution or assignment of a user scheduler, putting it back in the scheduler queue.

Kubernetes provides the preemption abstraction implicitly. When users submit containers based on their priorities (if any), Kubernetes may decide to preempt lower-priority pods. First, users create priority groups (classes), they specify the policy for preempting, and second, assign the priority class to the containers. Here is a specific example:

```

1  apiVersion: scheduling.k8s.io/v1
2  kind: PriorityClass
3  metadata:
4    name: high-priority-preempt
5  value: 1000000
6  preemptionPolicy: PreemptLowerPriority

```

This example creates a priority class with the name *high-priority-preempt*. The numeric priority value is set to *1000000* through the parameter *value*, and the *preemptionPolicy* parameter is set to *PreemptLowerPriority*, which specifies that pods of this *PriorityClass* will preempt lower priority pods if necessary. Then it is required to apply the priority class to the containers that are submitted:

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: example
5  spec:
6    containers:
7    - name: example
8      image: example:1,28
9    priorityClassName: high-priority-preempt

```

In this example, the container is assigned the priority class *high-priority-preempt* by setting the parameter *priorityClassName* so that when this container is submitted, it will preempt lower priority containers if there are no free resources.

SLURM offers preemption logic similar to Kubernetes. It allows preempting jobs based on the assigned priorities and the configuration. When a high-priority job is allocated resources already allocated to other jobs of lower priority, the low-priority jobs are preempted. Here is a specific example of the configuration:

```
1 PreemptType=preempt/qos
2 PreemptMode=REQUEUE
```

In this example, the *PreemptType* argument is set to *preempt/qos*, which specifies that job preemption occurs based on the numeric priority of the jobs. Also, the *PreemptMode* is set to *REQUEUE*, which specifies that the scheduler preempts jobs by queuing them if possible, and otherwise, it cancels them. Then, for the preemption to occur, the resources must be busy, and the user has to submit a higher-priority job than those currently submitted.

In case the file is saved as *slurm.conf*, the file must be placed in the special folder where SLURM looks up for configuration.

Condor offers preemption abstraction, but in a different way than Kubernetes and SLURM. Instead of implicitly implementing it, it does so explicitly, offering a command to preempt to the user. Here is a specific example:

```
condor_vacate_job 23
```

In this example, finds job 23 from the Condor job queue and vacate it from the host(s) where it is currently running. The job remains in the job queue and returns to the idle state.

Spark, and **Airflow** do not provide the preempt abstraction.

3.6.4 Recover

. Recover is the abstraction for recovering the execution of a job after a failure by restarting the execution or putting it back into the scheduler queue.

Kubernetes allows specifying restart on failure policies. Here is a specific example:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: example-deployment
5 spec:
6   template:
7     metadata:
8       labels:
9         app: example
10    spec:
11      restartPolicy: OnFailure
12    containers:
```

```

13     - name: example
14     image: example:1.28

```

In this example, the *restartPolicy* is set to *OnFailure*, which specifies that the container must be restarted automatically in case it exits with a failure code.

SLURM, instead of performing automatic restarts, it allows to requeue a failed job via a flag. It returns the failed task to the scheduling queue instead of restarting it on the same machine. Here is a specific example:

```

sbatch --job-name=example --requeue run.sh

```

In this example, the *requeue* flag specifies that the task must be submitted to the scheduling queue after it finishes.

Spark offers the same abstraction as Kubernetes; users can specify automatic restart policies in case of failures. Here is a specific example:

```

./bin/spark-submit
  --class org.apache.spark.examples.SparkPi
  --master spark://207.184.161.138:7077
  --supervise
  /path/to/examples.jar

```

In this example, the flag *supervise* is passed when submitting the app, and consequently Spark restarts the application automatically if it exits with a non-zero exit code.

Condor offers automatic restarts by specifying the maximum number of retries to be performed to run a failed execution. That is, unlike the other schedulers, it can restart on failure, but a limited number of times, and without being able to specify any policy. Here is a specific example:

```

1 executable = example
2 arguments  = SomeArgument
3
4 max_retries = 5
5
6 queue

```

In this example, the *max_retries* argument is set to 5, which tells Condor the maximum number of times to restart the job from scratch, if a job exits with a non-zero exit code.

Airflow is similar to Condor, it offers automatic restarts, by specifying the maximum number of retries to be performed to run a failed execution. Here is a specific example:

```

1  from datetime import datetime
2  from airflow import DAG
3  from airflow.operators.dummy_operator import DummyOperator
4  from airflow.operators.python_operator import PythonOperator
5
6  def example():
7      return 'Hello world from first Airflow DAG!'
8
9  default_args = {
10     'retries': 5,
11 }
12
13
14  dag = DAG('hello_world', description='Hello World DAG',
15     ↪ default_args=default_args)
16
17  example_operator = PythonOperator(task_id='example_task',
18     ↪ python_callable=example, dag=dag)
19
20  example_operator

```

In this example, the *retries* argument is set to 5 and is passed as *default_args* to the DAG, which tells Airflows the maximum number of times to restart the task, if a task exits with a non-zero exit code.

3.7 Abstraction - Configure scheduler

In this section, we explain the scheduler configuration abstraction. This abstractions family specifies the configuration of the behavior of the scheduler. This abstraction is necessary for the tuning of the scheduler to better adjust to the users' workload and the datacenter's environment.

3.7.1 Scheduling policy

. The configuration of the scheduling policy specifies the set of rules and objectives that guides the scheduler in its decisions. The policies promote or de-emphasize factors like business priorities and customer preferences when provisioning resources among simultaneous requests.

Kubernetes breaks the scheduling algorithm into different steps and allows configuring the policy for each of these steps using plugins. Here is a specific example:

```

1  apiVersion: kubescheduler.config.k8s.io/v1
2  kind: KubeSchedulerConfiguration
3  profiles:
4  - plugins:

```

```

5     score:
6         enabled:
7             - name: MyCustomPluginA
8             - name: MyCustomPluginB

```

In this example, two plugins *MyCustomPluginA* and *MyCustomPluginB* are submitted to the scheduler, which is used for scoring the candidate nodes. These plugins provide a score to each node that is a candidate to be provisioned for a container. The scheduler will then select the node with the highest weighted score sum.

In case the file is saved as *config.yaml*, the command to apply the scheduler configuration is:

```
kube-scheduler --config config.yaml
```

SLURM allows to choose between two different policies, backfill scheduling or strict priority order. Here is a specific example:

```

1 SchedulerType = sched/backfill

```

In this example, the argument *SchedulerType* is set to *sched/backfill*, which specifies the scheduler to use the backfill scheduling policy.

If the file is saved as *slurm.conf*, the file must be placed in the particular folder where SLURM looks up for configuration.

Spark only allows users to choose between two policies, fair scheduling or fifo. Here is a specific example:

```

./bin/spark-submit
--class org.apache.spark.examples.SparkPi
--master spark://207.184.161.138:7077
--conf spark.scheduler.mode=FIFO
/path/to/examples.jar

```

In this example, the configuration flag *spark.scheduler.mode* is passed with value *FIFO*, specifying the requests are processed in FIFO order.

Condor and **Airflow** do not provide the scheduling policy abstraction.

3.7.2 Scheduling frequency

. The configuration of the scheduling frequency specifies the frequency at which the scheduler is activated and processes the incoming requests.

SLURM allows users to configure the scheduling frequency through parameters of a configuration file. Here is a specific example:

```

1 sched_interval = 60

```

In this example, the *sched_interval* argument is set to 60, which specifies how frequently, in seconds, the main scheduling loop will execute and try to schedule all pending jobs.

If the file is saved as *slurm.conf*, the file must be placed in the special folder where SLURM looks up for configuration.

Airflow , like SLURM, allows users to specify the frequency through an argument of a configuration file.

```
1 [scheduler]
2 scheduler_idle_sleep_time = 5
```

In this example, the argument *scheduler_idle_sleep_time* is set to 5, which controls how long, in seconds, the scheduler will sleep between loops if there is nothing to do in the loop. i.e. if it schedules something, then it will start the next loop iteration straight away.

In case the file is saved as *airflow.cfg*, the file must be placed in the special folder where Airflow looks up for configuration (*in your \$AIRFLOW_HOME*).

Kubernetes and **Spark** do not provide the scheduling frequency abstraction.

3.8 Abstraction - Constraints

. In this section we present the abstraction of constraints, which specify temporal and physical constraints for scheduling, absolute to the physical world, or relative to other scheduling events and concepts. These abstractions are necessary to identify the resources you want to provision and on which you want to run the user workloads.

3.8.1 Space constraints

. Space constraints specify physical constraints of a job. They can be hard constraints, which do not allow to schedule unless they are met, or soft constraints in which a best-effort is made to meet them.

Kubernetes allows users to specify the resource requirements. In Kubernetes, nodes have tags assigned, and then users can specify to schedule the containers only in nodes with specific labels. Here is a specific example:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: example
5 spec:
6   containers:
7     - name: example
8       image: example:1.28
9       resources:
10        limits:
```

```

11         nvidia.com/gpu: 1
12     nodeSelector:
13         accelerator: nvidia-tesla-p100

```

In this example, the container below selects nodes with the label *accelerator=nvidia-tesla-p100*, which specifies that the node where the container is provisioned must contain a specific graphics processing unit.

SLURM offers the ability to specify resource requirements through parameters of a CLI, and it can also be limited to specific nodes. Here is a specific example:

```

sbatch --job-name=example --constraint="gpu" run.sh

```

In this example, the parameter *constraint* is set to *gpu*, which specifies the job must be run on top of nodes that have a Graphics Processing Unit.

Spark offers the ability to specify the requirements of the application. Here is a specific example:

```

./bin/spark-submit
--class org.apache.spark.examples.SparkPi
--master spark://207.184.161.138:7077
--conf spark.executor.memory=2G
--conf spark.executor.cores=1
/path/to/examples.jar

```

In this example the configuration flag *spark.executor.memory* is set to *2G*, specifying that the amount of memory to use per executor process is 2 Gigabytes, and *spark.executor.cores* is set to *1*, specifying that the amount of cpu cores to use per executor process is one.

Condor offers great flexibility through list of attribute-value pairs called ClassAds, in which users can specify any arbitrary physical constraint. ClassAds are a flexible mechanism for representing the characteristics and constraints of machines and jobs in the Condor system. The following shows ten attributes, a portion of an example ClassAd:

```

1 executable = example
2 arguments  = SomeArgument
3
4 requirements = (Memory == 128) && (Cpus == 8)
5
6 queue

```

In this example, the argument *requirements* is set, so that the amount of memory and cpu number is specified. Condor allows to specify ClassAd boolean expressions, in which any desired feature can specified as a requirement.

Airflow the resources of the task are specified through code, and users can also specify where the task is executed. Here is a specific example:

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator
5
6 def example():
7     return 'Hello world from first Airflow DAG!'
8
9 default_args = {
10     'retries': 5,
11 }
12
13
14 dag = DAG('hello_world', description='Hello World DAG',
15 ↪ default_args=default_args)
16
17 example_operator = PythonOperator(
18     task_id='example_task',
19     python_callable=example,
20     dag=dag,
21     resources={
22         'request_memory': '5G',
23         'request_cpu': '4',
24     },
25 )
26 example_operator
```

In this example, the main task that is a `PythonOperator` sets the `resources` parameter, where the physical constraints are specified. In this case, the memory amount and the number of CPUs are specified. The parameter `request_memory` is set to `5G`, specifying that the amount of memory to use per executor process is five Gigabytes, and `request_cpu` is set to `4`, specifying that the amount of cpu cores to use per executor process is four.

3.8.2 Time constraints

. Time constraints specify the temporal constraints of a job, including recurrent patterns, start times, and maximum runtimes. They can be hard constraints, which do not allow to schedule unless they are met, or soft constraints in which the best effort is made to meet them.

Kubernetes offers cron jobs, which allow submission jobs along with a cron specification. Cron is a specification to run periodically at fixed times, dates, or intervals. This allows

users to specify recurring temporal patterns. Here is a specific example:

```
1  apiVersion: batch/v1
2  kind: CronJob
3  metadata:
4    name: example
5  spec:
6    schedule: "* * * * *"
7    jobTemplate:
8      spec:
9        template:
10         spec:
11           containers:
12             - name: hello
13               image: example:1.28
```

In this example, the CronJob manifest prints the current time and a hello message every minute. The `.spec.schedule` field is required. The value of that field follows the Cron syntax. For example, `0 0 13 * 5` states that the task must be started every Friday at midnight, as well as on the 13th of each month at midnight.

In case the file is saved as `example.yaml`, the command to submit the file and perform the provisioning is:

```
kubectl create -f example.yaml
```

SLURM allows to submit a request together with a desired start time instead of a recurring cron job. Here is a specific example:

```
sbatch --job-name=example --begin=2010-01-20T12:34:00 run.sh
```

In this example, the argument `begin` specifies the date and time the job must be provisioned and run.

Condor provides the abstraction of job deferrals, that is, the ability to specify a deferral time when users want to provide the resources. Here is a specific example:

```
1  executable = example
2  arguments  = SomeArgument
3
4  deferral_time = 1136138400
5
6  queue
```

In this example, the job's submit description file specifies in Unix epoch that the job will begin execution on January 1st, 2006, at 12:00 pm.

Airflow lets users specify cron jobs and limit runtimes, but also, users can specify specific dates and times to run their jobs without having to be recurring as in cron. Here is a specific example:

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator
5
6 def example():
7     return 'Hello world from first Airflow DAG!'
8
9 dag = DAG('hello_world', description='Hello World DAG')
10
11 default_args = {
12     'start_date': datetime(2015, 12, 1),
13     'schedule_interval': '@hourly',
14 }
15
16 example_operator = PythonOperator(task_id='example_task',
17     ↪ python_callable=example, dag=dag, default_args=default_args)
18
19 example_operator
```

In this example, the *start_date* argument is set to *2015-12-01* and is passed as *default_args* to the DAG, which tells Airflows the time at which the workflow must start running. Together with the starting date, the *schedule_interval* argument is set to *hourly*, specifying the workflow must be run every hour.

3.8.3 Affinities

. Affinities specify constraints relative to other jobs. In other words, a way to specify the characteristics of other scheduling units that run on the same physical resources. For example, that the HTTP server to be assigned to the same node where a certain database is running.

Kubernetes implements affinities through tags, so that users assign tags to their pods (containers), and later they can specify which tags the other pods of the node should have, as well as which tags to avoid. Here is a specific example:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4     name: example
5 spec:
6     containers:
7     - name: example
```

```

8      image: example:1.28
9      resources:
10         limits:
11             nvidia.com/gpu: 1
12  affinity:
13      podAffinity:
14          preferredDuringSchedulingIgnoredDuringExecution:
15              podAffinityTerm:
16                  labelSelector:
17                      matchExpressions:
18                          - key: server
19                            operator: In
20                            values:
21                                - http

```

In this example, the user specifies that the container must be deployed in a node that has other containers running, which have the tag *server* set to *http*. For that specifies the *key* and the *value* for the key that is looking for. The affinities are specified in the *affinity.podAffinity* field. This causes the container to run on the same node that the HTTP server runs on.

Condor has the potential of providing abstractions for affinities, due to the flexibility of the ClassAd. However, the default implementation it does not process affinity requests, since the machine does not collect and later expose the ClassAds of the running jobs. Therefore, it is not possible to specify constraints depending on the characteristics of other running jobs.

3.8.4 Task dependencies

. Task dependencies abstraction specifies the relationships in which a task or milestone relies on other tasks to be performed (completely or partially) before it can be performed.

Kubernetes does not offer the abstraction of task dependencies. The closest is the abstraction of a job, which creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate.

SLURM allows to specify, through a CLI parameter, the dependencies of the jobs that must be completed before. Here is a specific example:

```

sbatch --job-name=example --dependency afterok:20:21,afterany:23
run.sh

```

In this example, the dependency parameter is set, which specifies that the job can run only after a 0 return code of jobs 20 and 21 and the completion of job 23.

Spark provides the API for setting dependencies between tasks implicitly, since the code commands are executed sequentially, and this allows the generation of workflows. Here is a specific example:

```
1 text\_file = sc.textFile("hdfs://...")
2 counts = text\_file.flatMap(lambda line: line.split(" "))
3     .map(lambda word: (word, 1))
4     .reduceByKey(lambda a, b: a + b)
5 counts.saveAsTextFile("hdfs://...")
```

In this example, a few dataset transformations are performed to build a dataset of (String, Int) pairs called counts and then save it to a file. The reading and writing of the file is run as a single task, but the *flatMap*, *map* and *reduceByKey* operations are run in a distributed manner, so each of them are multiple tasks. Therefore, this code represents and acts as a workflow of several tasks.

Condor offers a framework called DagMAN, which allows users to explicitly define workflows in a configuration file, representing them as Directed Acyclic Graphs. Here is a specific example:

```
1 JOB A A.condor
2 JOB B B.condor
3 JOB C C.condor
4 JOB D D.condor
5 PARENT A CHILD B C
6 PARENT B C CHILD D
```

In case the file is saved as *example.dag*, the command to submit the workflow and perform the provisioning is:

```
condor_submit_dag example.dag
```

Airflow offers an explicit way to define DAGs to generate workflows with multiple task dependencies. But unlike Condor, Apache Airflow's DAGs are represented by code. Here is a specific example:

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator
5
6 def example():
7     return 'Hello world from first Airflow DAG!'
8
9 dag = DAG('hello_world', description='Hello World DAG')
```

```

11 first_task = PythonOperator(task_id='example_task_1',
    ↪ python_callable=example, dag=dag)
12
13 second_task = PythonOperator(task_id='example_task_2',
    ↪ python_callable=example, dag=dag)
14
15 third_task = PythonOperator(task_id='example_task_3',
    ↪ python_callable=example, dag=dag)
16
17 first_task >> [second_task, third_task]

```

In this example, three tasks are defined and then with the » operator the dependencies between them are specified. In this case, the second and third task are executed after first task's completion.

3.9 Abstraction - Quality of Service

. In this section we present the abstraction of the quality of service, which specifies the distinction of priorities for scheduling and execution of resources.

3.9.1 Priorities

. Priorities specify the levels of importance for jobs when it comes to assigning resources or having to reduce them.

Kubernetes provides priorities using `PriorityClass`, which assigns numeric values representing priorities to containers. Here is a specific example:

```

1  apiVersion: scheduling.k8s.io/v1
2  kind: PriorityClass
3  metadata:
4    name: high-priority
5  value: 1000000

```

In this example, a priority class is defined. *metadata.name* parameter defines the name of the priority, and the *value* parameter specifies the numeric value of the priority. After the `PriorityClass` is defined, we specify the `PriorityClass` in the pod's specification:

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: example-deployment
5  spec:
6    template:
7      metadata:
8        labels:
9          app: example

```

```

10 spec:
11   restartPolicy: OnFailure
12   containers:
13   - name: example
14     image: example:1.28
15     priorityClassName: high-priority

```

In this example, the *priorityClassName* is set to *high-priority*, which specifies that the pod is high priority.

SLURM , in the same way to Kubernetes, when submitting a task users can assign a numeric priority through the parameters. Here is a specific example:

```
sbatch --job-name=example --priority=5 run.sh
```

In this example, the priority argument is set to 5, which specifies the job has a numeric priority of 5.

Spark allows to group tasks into pools, and then assign a numeric priority to the pool, but it does not assign priorities to individual tasks. Here is a specific example:

```

1 <?xml version="1.0"?>
2 <allocations>
3   <pool name="pool-1">
4     <weight>1</weight>
5   </pool>
6   <pool name="pool-2">
7     <weight>2</weight>
8   </pool>
9 </allocations>

```

In this example, an XML file is created where two pools are defined. For each of the pools, the *weight* argument is defined, which controls the pool's share of the cluster relative to other pools. A weight of 2, for example, will get 2x more resources than other active pools. Then, the application is submitted with the configuration and assigning one of the pools:

```

./bin/spark-submit
  --class org.apache.spark.examples.SparkPi
  --master spark://207.184.161.138:7077
  --conf spark.scheduler.pool='pool-1'
  --conf spark.scheduler.allocation.file='conf.xml'
/path/to/examples.jar

```

In this example, the configuration flag *spark.scheduler.allocation.file* is passed to specify where the XML configuration file is placed, and the flag *spark.scheduler.pool* is set to *pool-1* so that the pool configuration is applied.

Condor allows to assign a numeric priority to the job in the ClassAd of the task that is submitted. Here is a specific example:

```
1 executable = example
2 arguments  = SomeArgument
3
4 priority = 5
5 queue
```

In this example, the argument priority is set to 15, which specifies the job has a numeric priority of 5.

Airflow similarly to SLURM and Condor, provides the priority abstraction, by allowing to assignment a weight to each task. Here is a specific example:

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator
5
6 def example():
7     return 'Hello world from first Airflow DAG!'
8
9 dag = DAG('hello_world', description='Hello World DAG')
10
11 example_operator = PythonOperator(
12     task_id='example_task',
13     python_callable=example,
14     dag=dag,
15     priority_weight = 5
16 )
17
18 example_operator
```

In this example, the parameter priority_weight with a value of 5 is passed to the task. This specifies the task has a numeric priority of 5.

3.9.2 Group quota

. Group quota abstraction specifies task grouping and partitioning of available resources among the different groups.

Kubernetes offers the concept of namespaces to group containers together, and then assign quotas to each namespace, limiting the total amount of resources they can use. Here is a specific example:

```

1  apiVersion: v1
2  kind: List
3  items:
4  - apiVersion: v1
5    kind: ResourceQuota
6    metadata:
7      name: pods-high
8    spec:
9      hard:
10       cpu: 100
11       memory: 200Gi
12     scopeSelector:
13       matchExpressions:
14       - operator : In
15         scopeName: PriorityClass
16         values: ["high"]

```

In this example, resource quotas are assigned to the pods with high priority. The *spec.hard.cpu* is set to 100, and *spec.hard.memory* is set to 200Gi, which specifies that resource quotas are limited to 100 CPU cores and 200Gi of memory. The *scopeSelector* specifies what pods the resource quota is applied to, and in this case, are high-priority pods.

SLURM , instead of limiting the total usage, partitions the available resources, assigning each task a partition. Here is a specific example:

```

1  PartitionName=partition-example
2  Nodes=node[1-4]

```

In this example, a partition named *partition-example* is created, and the nodes 1, 2, 3, and 4 are assigned. In case the file is saved as *slurm.conf*, the file must be placed in the particular folder where SLURM looks up for configuration.

After the partition is created, partitions are assigned to jobs by adding the *partition* parameter:

```

sbatch --job-name=example --partition=partition-example run.sh

```

Spark , tasks are grouped into pools, and then a minimum amount of resources is assigned to each pool. This way, the resources are partitioned, and the remaining available resources are assigned to the pools depending on their priorities. Here is a specific example:

```

1  <?xml version="1.0"?>
2  <allocations>
3    <pool name="pool-1">

```

```

4     <minShare>2</minShare>
5 </pool>
6 <pool name="pool-2">
7     <minShare>3</minShare>
8 </pool>
9 </allocations>

```

In this example, an XML file is created where two pools are defined. For each pool, the *minShare* argument specifies a minimum share (of several CPU cores) that the pool should have. Then, the application is submitted with the configuration and assigning one of the pools:

```

./bin/spark-submit
—class org.apache.spark.examples.SparkPi
—master spark://207.184.161.138:7077
—conf spark.scheduler.pool='pool-1'
—conf spark.scheduler.allocation.file='conf.xml'
/path/to/examples.jar

```

In this example, the configuration flag *spark.scheduler.allocation.file* is passed to specify where the XML configuration file is placed, and the flag *spark.scheduler.pool* is set to *pool-1* so that the pool configuration is applied.

Condor , provides the abstraction of group quotas, a mix between Kubernetes and SLURM. It creates different groups, each group is assigned a maximum resource usage, and the tasks are submitted to specific groups. Here is a specific example:

```

1 GROUP_NAMES = group_one, group_two
2 GROUP_QUOTA_group_one = 20
3 GROUP_QUOTA_group_two = 10

```

In this example, a pool with thirty slots is defined: twenty slots are owned by group one, and ten are owned by group two. The desired policy is that no more than twenty concurrent jobs run from the group, only ten from group two. It only matters that the proportions of allocated slots are correct.

After the group is created, groups are assigned to jobs by adding the *accounting_group* parameter:

```

1 executable = example
2 arguments = SomeArgument
3
4 accounting_group = group_one
5 queue

```

Airflow does not provide the preempt abstraction.

3.9.3 Exclusivity

. Exclusivity abstraction specifies the exclusive allocation of resources so that there are no jobs from different tenants using the same physical resources.

Kubernetes users can obtain CPUs exclusively by specifying that the container will use the "static" policy. However, this cannot be specified for every container. Instead, the configuration must be applied per node. Here is a specific example:

```
1 apiVersion: kubelet.config.k8s.io/v1beta1
2 kind: KubeletConfiguration
3 address: "192.168.0.8"
4 port: 20250
5 cpuManagerPolicy: "static"
```

In the example, the node on IP address 192.168.0.8 and port 20250 is configured to have a static cpu policy, which forces cpus on the node to be provisioned exclusively.

SLURM , similarly, allows to add of the *exclusive* CLI flag, so CPU cores are exclusively provisioned for every job. Here is a specific example:

```
sbatch --job-name=example --exclusive run.sh
```

Spark, **Condor** and **Airflow** do not provide the preempt abstraction.

3.10 Abstraction - Manage data

. In this section, we present the abstraction of data management, which specifies the management of the data users use while running their jobs.

3.10.1 Access input data

. Access input data specifies the abstraction for transferring initial input data into the running jobs.

Kubernetes abstracts the data transfer through an abstraction called volume, which maps files from the user file system into the container's file system. Here is a specific example:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: example
5 spec:
6   containers:
7   - image: example:1.28
8     name: example
9     volumeMounts:
```

```

10     - mountPath: /tmp/example
11       name: example-volume
12   volumes:
13     - name: example-volume
14       awsElasticBlockStore:
15         volumeID: "<volume id>"
16         fsType: ext4

```

In this example, an AWS Elastic Block Store (EBS) volume is transferred into the container file system path `/tmp/example`. The mount path is defined at `volumeMounts.mountPath`, and the AWS volume at `awsElasticBlockStore.volumeID`.

SLURM offers the `sbcast` command to send a file to all job nodes. Here is a specific example:

```
sbcast example /tmp/example
```

In this example, the file at path `/tmp/example` is sent to the job `example`.

Spark provides the input data access abstraction so that data transfers occur explicitly. In Spark, users can explicitly define a streaming data source for the job, such as Kafka, AWS S3, etc. Here is a specific example:

```

1 from pyspark.sql import SparkSession
2 spark = SparkSession.builder.getOrCreate()
3 foo = spark.read.parquet('s3a://example-file')

```

In this example, a spark application session is created, and then a parquet file is read from AWS S3 by executing the command `spark.read.parquet` and specifying the S3 file `s3a://example-file`.

Condor offers the `condor_transfer_data` command to transfer data to nodes, but users can also transfer data when submitting jobs by specifying input files and adding the `spool` flag. Here is a specific example:

```
condor_submit -file /tmp/example -spool example
```

In this example, the file at path `/tmp/example` is transferred to the node where the job `example` is submitted.

Airflow offers data transfer abstraction through tasks. In Airflow, the user can create a task that is in charge of the data transfer to other tasks. Here is a specific example:

```

1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator

```

```

5
6 dag = DAG('hello_world', description='Hello World DAG',
  ↪ on_failure_callback=task_failure_alert)
7
8 put_file = SFTPOperator(
9     task_id="test_sftp",
10    ssh_conn_id="ssh_default",
11    local_filepath="/tmp/example.txt",
12    remote_filepath="/tmp/example.txt",
13    operation="put",
14    dag=dag
15 )
16
17 def example():
18     return 'Hello world from first Airflow DAG!'
19
20 example_operator = PythonOperator(task_id='example_task',
  ↪ python_callable=example, dag=dag)
21
22 put_file >> example_operator

```

In this example, a task named *put_file* is created, which transfers a local file to a remote path using the SFTP protocol. Next, that task is placed as part of the workflow and is executed before the functions that will use the data.

3.10.2 Replicate

. Replicate specifies the abstraction for caching intermediate data of jobs.

Spark provides users with a `persist` command so that the intermediate data is persisted in the node and can be reused in other operations. Here is a specific example:

```

1 from pyspark.sql import SparkSession
2 spark = SparkSession.builder.getOrCreate()
3
4 df = spark.read.csv('s3a://example-file')
5 dfPersist = df.persist(StorageLevel.MEMORY_ONLY)

```

In this example, an S3 file is loaded from AWS S3 by running `spark.read.csv`, and then the file is persisted by calling `persist(StorageLevel.MEMORY_ONLY)`, where the argument `StorageLevel.MEMORY_ONLY` specifies that the file is persisted in memory instead of in disk or external storage.

Kubernetes, **SLURM**, **Condor** and **Airflow** do not provide the preempt abstraction.

3.10.3 Partition

. Partition specifies the abstraction for data partitioning between several nodes of the same job.

Spark provides explicit commands to partition the data being processed. But it also implicitly partitions the data when iterating and applying operations to an array of data without specifying it explicitly. Here is a specific example:

```
1 from pyspark.sql import SparkSession
2 spark = SparkSession.builder.getOrCreate()
3
4 df = spark.read.csv('s3a://example-file')
5 dfPartitioned = df.repartition(10)
```

In this example, an S3 file is loaded from AWS S3 by running *spark.read.csv*. Then the file is partitioned into ten parts by calling *repartition(10)*, where the resulting data is hash partitioned.

Kubernetes, **SLURM**, **Condor** and **Airflow** do not provide the preempt abstraction.

3.10.4 Recover

. Recover specifies the abstraction for the recovery of data after a job failure.

Kubernetes is a special case, as it offers API calls for checkpointing and data recovery but does not implement it. The user must implement all the logic behind the API call. Here is a specific example:

```
POST /checkpoint/{namespace}/{pod}/{container}
```

In this example, a checkpoint is requested through an HTTP Post request to a specific namespace, pod, and container.

Condor offers fault-tolerance through self-checkpointing jobs. Jobs can exit with a specific exit code, signaling the scheduler that a checkpoint has just been performed and what is the checkpoint file. This way, if the job fails later, the scheduler uses the most recent checkpoint and relaunches the job. Here is a specific example:

```
1 checkpoint_exit_code      = 85
2 transfer_output_files     = example.checkpoint
3 should_transfer_files     = yes
4
5 executable                = example.py
6 arguments                 =
7
8 queue
```

In this example, the checkpointing logic is configured. The configuration file commands Condor to transfer the file *example.checkpoint* to the submit node whenever the script exits with code 85. If interrupted, the job will resume from the most recent checkpoints. Once the configuration is set, the job that is submitted must take care of the checkpointing logic:

```

1  import sys
2  import time
3
4  value = 0
5  try:
6      with open('example.checkpoint', 'r') as f:
7          value = int(f.read())
8  except IOError:
9      pass
10
11 print("Starting from {0}".format(value))
12 for i in range(value,10):
13     print("Computing timestamp {0}".format(value))
14     time.sleep(10)
15     value += 1
16     with open('example.checkpoint', 'w') as f:
17         f.write("{0}".format(value))
18     if value%3 == 0:
19         sys.exit(85)
20
21 print("Computation complete")
22 sys.exit(0)

```

In this example, a Python script (*example.py*) is a toy example of code that checkpoints itself. It counts from 0 to 10, sleeping for 10 seconds at each step. It writes a checkpoint file and exits with code 85 at counts 3, 6, and 9. When done, it goes with code 0 to avoid checkpointing and restarting.

SLURM, **Spark**, and **Airflow** do not provide the preempt abstraction.

3.11 Abstraction - Communicate

. In this section, we present the abstraction for communication, which specifies the interactions between the user and the resources and between the user and the scheduler.

3.11.1 Signal

Signaling specifies the abstraction for sending signals to the running jobs.

Kubernetes does not provide a way to send signals to pods. The only way is to create a second pod that shares the process namespace with the pod and then sends a process

signal. Here is a specific example:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: example-2
5  spec:
6    shareProcessNamespace: true
7    containers:
8      - name: example-1
9        image: example:1.28
10     - name: example-2
11       image: example:1.28
```

This example enables process namespace sharing using the *shareProcessNamespace* field of *.spec*. After deploying the containers, you can connect to one of the containers and send a signal to the other.

SLURM allows sending process signals to scheduled tasks, including SIGTERM or SIGKILL, using the `scancel` command. Here is a specific example:

```
scancel --signal=KILL 1235
```

In this example, a SIGKILL signal is sent to all the tasks of job 1235.

Condor like in SLURM, you can send signals to the scheduled tasks using the `procd_ctl` command. Here is a specific example:

```
procd_ctl SIGNAL_PROCESS 9 1235
```

In this example, a signal with the number 9 is sent to job 1235.

Spark and **Airflow** do not provide the preempt abstraction.

3.11.2 Stdin

Stdin specifies the interaction with the running jobs through interactive input (stdin).

Kubernetes allows attaching stdin to containers by submitting them with the *interactive* flag, but also let's attach to an already running container using the `kubect1 attach` command. Here is a specific example:

```
kubect1 attach example
```

In this example, the user is attached to the standard input of a container named `example`.

SLURM allows users to attach to stdin of a running job using the `sattach` command. Here is a specific example:

```
sattach 15.0
```

In this example, the user is attached to task 0 of job 15.

Condor offers the ability to attach to stdin by submitting jobs with the `interactive` flag or via the `input` command. All the other schedulers do not provide this programming abstraction. Here is a specific example:

```
condor_submit -interactive submitexample
```

In this example, by specifying the *interactive* flags, the user submits a job and is attached to its standard input just after.

Spark and **Airflow** do not provide the stdin abstraction.

3.11.3 Callback

. The callback abstraction specifies executable code from the user executed asynchronously by the scheduler on certain events or conditions.

Kubernetes users submit callbacks for each of the container lifecycle events they are interested in so that the callback is executed when the pod changes state. A YAML document defines callbacks. Here is a specific example:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: lifecycle-demo
5  spec:
6    containers:
7    - name: example
8      image: example:1.28
9      lifecycle:
10       postStart:
11         exec:
12           command: ["/bin/sh", "-c", "echo Hello from the postStart
           ↪ handler > /usr/share/message"]
```

In this example, a callback is assigned via the *exec.command* parameter. In this specific case, it is specified that the callback is executed after the container has started, using *postStart*.

SLURM offers the ability to add callbacks to specific job events using the `strigger` command. Here is a specific example:

```
strigger --set --jobid=1234 --down --program=/tmp/callback
```

In this example, the program `/tmp/callback` is executed when any node allocated to job 1234 enters the DOWN state. That is to say when a node goes down.

Spark also offers callbacks, which can be defined in code and later set through a CLI flag. Here is a specific example:

```
./bin/spark-submit
—class org.apache.spark.examples.SparkPi
—master spark://207.184.161.138:7077
—conf spark.extraListeners=listener.ExampleListener
/path/to/examples.jar
```

In this example, a configuration flag `spark.extraListeners` is set to the value `listener`. `ExampleListener` specifies that a custom-made callback is passed to receive up-calls from events that happen during execution. The callback is defined in a code file named `listener`, and inside it, the listener is implemented as an object named `ExampleListener`.

Airflow allows users to assign callbacks to tasks so that they are activated when events occur in the task, such as failures. Here is a specific example:

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator
5
6 def task_failure_alert(context):
7     print(f"Task has failed, task_instance_key_str:
8         ↳ {context['task_instance_key_str']}")
9
10 def example():
11     return 'Hello world from first Airflow DAG!'
12
13 dag = DAG('hello_world', description='Hello World DAG',
14     ↳ on_failure_callback=task_failure_alert)
15
16 example_operator = PythonOperator(task_id='example_task',
17     ↳ python_callable=example, dag=dag)
18
19 example_operator
```

In this example, a function named `task_failure_alert` is executed whenever a workflow task fails. A python function is defined and passed to the dag object through the `on_failure_callback` argument for setting the callback.

Condor does not provide the preempt abstraction.

3.12 Limitations

First, the analysis of the industrial schedulers presents limitations due to the approach with which we carried it out. On the one hand, we want to offer a high-level understanding of the industrial scheduler APIs to understand what abstractions they provide and their differences. On the other hand, we also have the secondary objective of understanding scheduler APIs in general and taking the first steps toward designing a reference architecture. Therefore, we keep the analysis as flexible as possible so the design or the reference architecture approach does not condition us. Specifically, this is why we mix actions like *lease* with inputs like *physical constraints*. That is, we do not differentiate the user’s actions and the information she submits to perform the actions. Hence, this limits the understanding that the user generates about the abstractions.

Second, the analysis focuses more on the WHAT than on the HOW. That is to say, the objective of the analysis and the thesis, generally, is to model the content of the programming abstractions, not their form. This analysis gives concrete examples of how each scheduler implements the abstractions. However, we do this to justify that the scheduler implements the abstraction, not to analyze the differences in how they implement it. Hence, we do not offer a structured analysis of the form of the programming abstractions.

3.13 Summary

This section analyzes five mainstream industrial schedulers, identifying their main abstractions. The analysis is presented in an aggregated table, where we identify six abstraction categories: *provision*, *configure scheduler*, *constraints*, *quality of service*, *communicate*, and *manage data*. For each category, there are multiple sub-abstractions, and for each abstraction, we present a concrete example of how each scheduler implements it.

4 Reference architecture of scheduling programming abstractions

In this chapter, we will formally analyze the missing programming abstractions of the industrial schedulers. For that, we build a reference architecture, which allows us to map the schedulers into it, and thus identify which abstractions do not implement.

4.1 Overview

We identify missing abstractions of the industrial schedulers, build a reference architecture, and map the schedulers into it. A reference architecture is a conceptual model that provides a template solution for an architecture for a particular domain. In our case, we build a conceptual model that identifies the API schedulers that could potentially provide. This model allows mapping existing industrial scheduler APIs into it and consequently identifies shortcomings. But it also provides the potential to guide scheduler designs and offer a high-level overview of the scheduling field, helping with complexity and entry-level problems.

Our contribution is made up of multiple items:

1. We describe the methodology we follow to build the reference architecture (Section 4.2).
2. We analyze the requirements for the reference architecture (Section 4.3).
3. We specify the design principles for the reference architecture (Section 4.4).
4. We carry out a literature survey of academic schedulers (Section 4.5).
5. We present and describe the reference architecture (Section 4.6) .
6. We validate the reference architecture by mapping the industrial schedulers to the reference architecture (Section 4.7).
7. We identify missing programming abstractions of the industrial schedulers by mapping the industrial schedulers to the reference architecture (Section 4.7).
8. We explain the limitations of the reference architecture (Section 4.9).

4.2 Methodology

This section explains the methodology used to design and create the reference architecture. The methodology consists of the following steps:

1. **Analysis of requirements and design principles.** First, we identify the requirements of the reference architecture and the design principle by which we will guide and evaluate both the design and the result. This lets us formally justify and reason the reference architecture.
2. **Model real-world schedulers.** This step is already carried out in Section 3. In this step, the programming abstractions of real-world schedulers are modeled. For that, we identify the top mainstream schedulers in the industry, and we analyze their

APIs. After analyzing its APIs, we extract the main abstractions and we group them by categories. In this way, we identify the main current programming abstractions in the industry.

3. **Model emerging concepts from academia.** After modeling the real-world schedulers, we model the designs of emerging research in academia. The real-world schedulers offer models on current programming abstractions. However, they do not cover designs from emerging fields or alternative designs that are not yet established, such as IoT/Edge, energy efficiency, etc. Therefore, in order to build a more accurate reference architecture and identify shortcomings in industrial schedulers, we model the latest research concepts. In the literature, many articles offer new programming abstractions of schedulers, and it is impossible to model all of them. Therefore, we carry out a literature survey limiting the work to a representative and relevant subset of fifteen articles.
4. **Unify real-world and emerging concepts from academia.** Next, we extract, filter, generalize, and unify the abstractions found in the industry together with the ones found in academia, into a reference architecture. This step is the construction of the reference architecture. The step results in a visual and syntactic representation of a conceptual model that presents the scheduler programming abstractions, grouped into high-level concepts that facilitate reasoning and abstraction.
5. **Intuitively extend the reference architecture.** As a last step, we analyze the reference architecture and use our intuition to identify gaps and develop extensions. This step offers room for creativity and the idealization of the reference architecture. Current industry and academy designs for schedulers can model the programming abstractions of current and emerging schedulers. However, for the reference architecture to be future-proof and to drive innovation, we believe that it is necessary to go one step further, and it is essential to use intuition since intuition is the tool that allows modeling what does not yet exist.

4.3 Main requirements

To design a reference architecture, it is necessary to identify the requirements that must be met. To specify the requirements, the stakeholders and their use cases must first be identified, however, in this case, they are the same as those listed in section 1.3. Therefore, below, we only list the requirements that guide the design of the reference architecture.

R1 Comprehensibility. The most essential requirement is that when someone looks at the reference architecture, they can quickly get an understanding of what it is intended to offer. The reader should not make an excessive effort to identify the different components that make up the reference architecture, how they relate to each other, or what their high-level meaning is.

R2 Simplicity. The reference architecture is designed for various stakeholders, from university students to industrial computer engineers. Therefore, the design should be simple without losing context or utility. This will facilitate the adoption and use of the reference architecture. Quoting Leonardo di ser Piero da Vinci: *simplicity is the ultimate sophistication*.

- R3 Actionable.** The main driver of the reference architecture is that it is used in the real world, both as a learning resource about schedulers, as a scheduler design guide, as a shortcoming analysis tool, etc. Therefore, the design of the reference architecture always has to take into account whether the resulting work is actionable by the identified stakeholders.
- R4 Future-proof.** Technology constantly evolves, and new fields, such as edge computing, serverless, etc., are emerging. Consequently, both the resources that the schedulers assign to the workloads and the nature of the workloads change over the years, and it is impossible to predict what they will be like in 5 or 10 years. Therefore, it is important that the reference architecture is designed to be useful ten years from now, despite changes in the field. It is a great challenge, but other works, such as the Grid reference architecture, have achieved it, and all of them should serve as inspiration [17].

4.4 Design principles

For the design of the scheduling programming abstractions reference architecture, we identify four design principles, which serve as a guide and evaluation of it.

- P1 Separation of Objects from Actions.** Programming abstractions present what users can do with schedulers. Two of our main objectives are that the reference architecture is simple and comprehensive. To do this, it is important to make a clear distinction between the actions that are performed and the objects that are used as input to the actions. Actions represent what is going to be done, and objects how it is going to be done. This separation makes it easier for the reader to understand and build the mental model of the reference architecture.
- P2 Grouping of related actions.** Schedulers offer various actions to users, each serving a different purpose. However, several actions are related to each other. Therefore, to facilitate comprehension, related actions are grouped. This allows developers to use these groupings to design the scheduler architecture and students to simplify the establishment of their knowledge.
- P3 Avoidance of flavors within objects.** One of the main requirements of the reference architecture is that it be future-proof. For this, it is necessary that the reference architecture is not committed or coupled with any specific resource or workload. That is, it should not identify particular technologies, such as Virtual Machines, SSD storage devices, etc., since they may not exist in the future. In the same way, there are infinite ways to represent and form different objects. For all this, objects must be kept as high level as possible, avoiding concrete implementations and subtypes of objects.
- P4 Naming relationships between actions and objects.** The reference architecture must relate the objects with the actions. But it is not enough to relate them, for the work to be understandable and practical, it must identify the nature and context of the relationship. Therefore, the relationships between objects and actions must be named, and these names must be clear and concise.

4.5 Emerging concepts from academia

In this section, we model the emerging scheduler designs of the academy. The models serve to identify later shortcomings of industrial schedulers, unify them with them, and generate a reference architecture. We first present the methodology for selecting articles and then expose the chosen research model.

4.5.1 Literature survey methodology

We carry out the literature survey as a combination of a systematic literature survey and a snowballing search. Initially, the thesis supervisor recommends an initial subset of articles to generate the list of keywords needed for the literature survey. From these articles, using the snowballing search method, we find other articles and create a list of keywords that represents the field of scheduler designs. Afterward, we carry out a systematic literature survey until we have a list of fifteen articles. The systematic literature survey offers reproducibility and reduces the biased selection of articles. At the same time, the snowballing method facilitates the bootstrapping and collection of the key terms necessary for the systematic approach.

Through the snowballing approach, we obtain an initial set of articles, and then, we carry out the systematic literature survey as follows:

1. **Generate and apply a search query.** From the articles found through the snowballing method, we extract a set of keywords, and through them, we define a search query that we use to generate the candidate articles. Next, we show the query in SQL:

```
1
2  SELECT
3      venue, year, title, abstract, n_citations
4  FROM
5      publications
6  WHERE
7      lower(publications.title) LIKE ANY (array['%scheduling%', '%scheduler%']) OR (lower(publications.abstract) LIKE
8      ANY (array['%scheduling%', '%scheduler%']) AND
9      (lower(publications.abstract) LIKE ANY (array['%cloud%', '%cluster%', '%datacenter%', '%datacenter%', '%energy%', '%data%']) OR lower(publications.abstract) LIKE ANY (
10     array['%provided_by%', '%supplied_by%', '%aware%', '%interface%', '%api%', '%allows_user%']))
11  ORDER BY
12     year desc, n_citations desc, title
```

2. **Sort the search results by the number of citations normalized by year** Once we have the list of candidates, we order them so that the quality of the research and novelty are prioritized. The article order formula is the number of citations normalized by year. For that, we extend the SQL query above with the following:

```
1  ORDER BY
2     year desc, n_citations desc, title
```

3. **Apply an acceptance criterion to the search results** Once we sort the search list, we define which items represent the field we want to model. The acceptance criteria are straightforward: the article must present a scheduling programming abstraction or API.
4. **Select the first 15 articles while applying filtering criteria** Finally, we select the first 15 articles that meet the acceptance criteria, with one extra exception; there must be at least one article dealing with the following topics: energy efficiency, data management, and IoT. These topics are explicitly prioritized since we consider them essential due to global warming and the increasing data dynamism and heterogeneity resulting in an explosive expansion of connected devices.

4.5.2 Schedulers from academia

This section presents an overview of the schedulers found in the literature survey of academia.

Tetrished. [43] Tetrished is a scheduler that works with a reservation system and offers programming abstractions to lease release virtual machines and make reservations. But it also offers the possibility of specifying hard and soft time and space constraints, affinities between the different machines, and gang scheduling.

Rayon. [10] Rayon is a reservation scheduling system that allows to define reservations declaratively through a custom language called RDL. It also offers the ability to specify space and time, soft and hard requirements, task dependencies, and gang scheduling.

Alsched. [42] Alsched is a scheduler that handles hard and soft constraints using composable utility functions that users submit. Utility functions are defined as algebraic expressions, and in this way, Alsched can optimize the overall utility of tenants.

W-scheduler. [39] W-scheduler is a scheduler that schedules multi-objective requests applying the whale optimization algorithm (WOA). To handle multi-objective requests, the user specifies the resource requirements and their budget for each resource. The scheduler applies the optimization algorithm to optimize the budget cost simultaneously as the minimum makespan.

Energy-credit scheduler. [27] Energy-credit scheduler is a scheduler that focuses on improving energy efficiency through better packing of virtual machines. The scheduler performs machine energy consumption estimates by analyzing in-processor events. Then it schedules virtual machines based on the energy budget and fiscal interval of the budget that users submit.

Energy-aware scheduler. [26] Energy-aware scheduler is a scheduler that tries to handle multi-objective requests that balance execution time and energy consumption. To optimize scheduling, users submit an important energy-performance factor, and using that and a set of heuristic schedules, the user requests.

Computation and data decoupled scheduler. [34] In this article, they implement a scheduler to analyze the need to couple data movement and job scheduling. To analyze this, the scheduler receives from the user the specification of affinity constraints for scheduled

jobs on the nodes that have the data or on the nodes with the most negligible load. Aside from constraints, it also allows users to replicate user data.

Data grid. [9] This is one of the reference articles for data management, where they present the design principles for distributed management and analysis of large datasets. The article specifies two basic services for data management: user input data and meta-data access. On top of these two services, they present a third service: data replication management.

Mobility-aware scheduler. [6] Mobility-aware scheduler is a scheduler for fog computing, which optimizes the scheduling to choose the optimal location depending on the application requirements. To optimize scheduling, users submit a profile of their applications to determine if they are delay-tolerant, real-time, or need to run in a specific geolocation.

Delay-optimal scheduler. [49] Delay-optimal scheduler is a scheduler for edge computing that optimizes whether to run the application on a mobile device or offload to a nearby more powerful (MEC) server. The scheduler receives average delay estimates and power consumption constraints to optimize the decision and applies a Markov decision process.

Quasar. [13] Quasar is a scheduler that optimizes scheduling, trying to increase overall resource utilization while maintaining application performance. Users do not accept resource constraints such as CPU or memory to optimize scheduling. Instead, users should specify higher-level constraints such as throughput and latency.

Whiz. [19] Whiz is a scheduler offering a data-centric scheduling framework that tries to optimize applications' data computation. To optimize data computation, the scheduler offers the user primitives for accessing the intermediate data that the application generates.

Availability-on-Demand scheduler. [38] Availability-on-Demand scheduler is a scheduler that optimizes the availability of applications. To optimize the availability of applications, the scheduler offers an API to users through which they can specify their availability needs. They also offer different policies with which to configure the scheduler.

Paragon. [12] Paragon is a scheduler that reduces interference and increases the overall utilization of user applications. Paragon receives application profiles that specify interference tolerance and heterogeneity scores to optimize scheduling, then applies algorithms to decide which applications to co-locate together.

Cost and deadline constrained scheduler. [30] Cost and deadline-constrained scheduler optimizes scheduling for applications with budget and deadline constraints. It applies online or static algorithms to various user inputs to optimize scheduling. The scheduler receives two types of input; on the one hand, it receives the deadline and resource constraints of the application. On the other hand, it also receives application profiles specifying the runtime estimates and the execution priority.

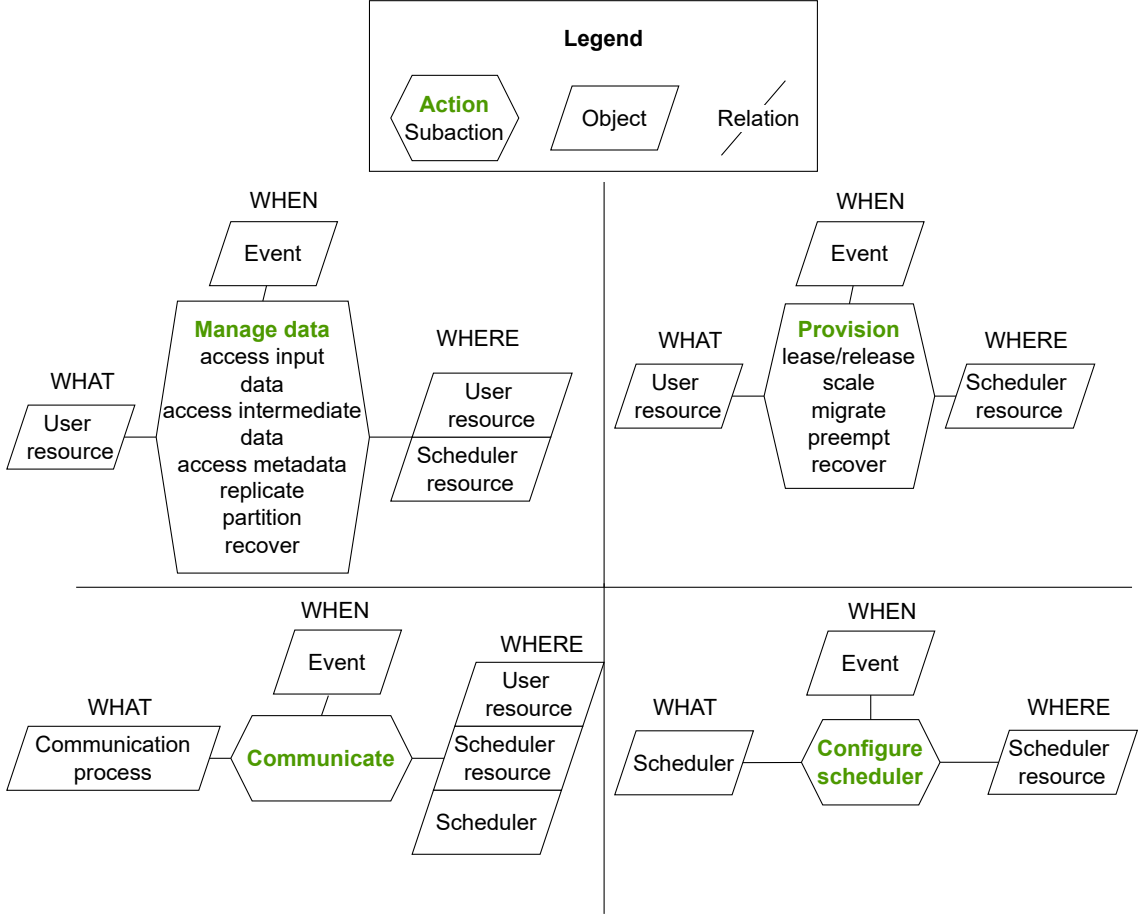


Figure 2: The visual representation of the reference architecture. The hexagons represent actions, the rectangles object, and the lines the relations between objects and actions.

4.6 Reference architecture

This section presents the reference architecture we generate from analyzing the industrial schedulers, research schedulers, other inspirational articles such as *A Reference Architecture for Datacenter Scheduling* [4], and intuition.

By modeling the industrial and academic schedulers, we identify state-of-the-industry the main components of the programming abstractions. These components could be objects like constraints, or actions like provisioning, which we do not distinguish. However, we distinguish objects from actions in the reference architecture, separating the WHAT from HOW it is done. The reference architecture is found in Figure 2. The reference architecture models the programming abstraction or scheduler APIs. The high-level approach of the model is based on listing the objects and actions that the APIs present and how the objects are related to the actions. The objects are the input that the actions receive. Each action must have four types of relationships: WHAT, WHEN, and WHERE, and for each relationship, there can be one or more objects. In this way, programming abstractions can be understood through the following syntactic structure: `<action> <object> IN <object>`

WHEN <object>, where the objects and actions are filled with the reference architecture.
For example:

```
Provision:Lease
UserResource<type:job, runtime:5d>
IN SchedulerResource<type:vm, cpu:2.4Ghz, memory:16Gb>
WHEN Event<day:31, month:12, year:2022, hour:00, minute:00>
```

And also:

```
Provision:Scale
UserResource<type:app>
IN SchedulerResource<type:vm, cpu:2.4Ghz, memory:16Gb>
WHEN Event<cpu.utilization:>80%>
```

In Listing 1, we define the syntax of our reference architecture formally using the Extended Backus–Naur Form [16], a family of metasyntax notations.

Listing 1: Reference Architecture EBNF Formal Syntax.

```
<ReferenceArchitectureSyntax> ::= <Action>
    | (<UserResourceObject> | <SchedulerObject>
    | <CommunicationProcessObject>)
    | "IN"
    | (<UserResourceObject> | <SchedulerResourceObject>
    | <SchedulerObject>)
    | "WHEN" <EventObject>

<Action> ::= <ProvisionAction> | <ConfigureSchedulerAction>
    | <ManageDataAction> | <CommunicateAction>

<ProvisionAction> ::= 'Provision' ':'
    (
        'Lease' | 'Release'
        | 'Scale' | 'Migrate' | 'Preempt' | 'Recover'
    )

<ManageDataAction> ::= 'ManageData' ':'
    (
        'AccessInputData' | 'AccessIntermediateData'
        | 'AccessMetadata' | 'Replicate' | 'Partition'
        | 'Recover'
    )

<ConfigureSchedulerAction> ::= 'ConfigureScheduler'

<CommunicateAction> ::= 'Communicate'
```

```

<UserResourceObject> ::= 'UserResource ' <Metadata>

<SchedulerObject> ::= 'Scheduler ' <Metadata>

<CommunicationProcessObject> ::= 'CommunicationProcess '
    <Metadata>

<SchedulerResourceObject> ::= 'SchedulerResource ' <Metadata>

<Metadata> ::= '< ' '>'
    | '<' <MetadataMembers> '>'

<MetadataMembers> ::= <Pair>
    | <Pair> ' ,' <Members>

<Pair> ::= String ':' <Value>

<Array> ::= '[' ']'
    | '[' <Elements> ']'

<Elements> ::= <Value>
    | <Value> ' ,' <Elements>

<Value> ::= String
    | Number
    | <Object>
    | <Array>
    | true
    | false
    | null

```

4.6.1 Objects

Next, we define each of the objects and actions of the reference architecture.

The objects are the following:

- **Event:** objects in time or instantiations of object properties. Such as a concrete datetime (00:00 of 31st of December 2022) or an instantiation of a property like a metric reaching a number (CPU utilization is more significant than 80%).
- **User resource:** Representation of any input from the user. This includes execution units like a job, task, etc., and the data the execution units use as a file, environment variable, etc.
- **Scheduler resource:** Representation of resources owned by the scheduler and managed by the scheduler. Resources can be virtual machines, containers, storage systems, databases, etc.

- **Communication process:** Representation of the communication process, such as a signal, message, callback, etc.

4.6.2 Action - Provision

In this section, we present the abstraction for the provisioning of resources. This is the main and most basic abstraction of scheduling since it is by which resources are acquired and managed.

Lease/release specifies the activation and assignment of a valuable resource to a scheduler resource. For example:

```
Provision : Lease
UserResource<type:job , runtime:5d>
IN SchedulerResource<type:vm, cpu:2.4Ghz, memory:16Gb>
WHEN Event<day:31, month:12, year:2022, hour:00, minute:00>
```

Scale specifies the addition or deletion of scheduler resources. For example:

```
Provision : Scale
UserResource<type:app, ID:123>
IN SchedulerResource<type:VM, cpu:2.4Ghz, memory:16Gb>
WHEN Event<cpu.utilization:>80%>
```

Migrate specifies the migration of a resource to a different scheduler resource. For example:

```
Provision : Migrate
UserResource<type:app, ID: 123>
IN SchedulerResource<type: VM, cpu:2.4Ghz, memory:16Gb>
WHEN Event<oversubscription: > 20%>
```

Preempt specifies the abortion of execution or assignment of a user scheduler, putting it back in the scheduler queue. For example:

```
Provision : Preempt
UserResource<type:application , ID:123>
IN SchedulerResource<type: VM, cpu:2.4Ghz, memory:16Gb>
WHEN Event<time: now>
```

Recover specifies the recovery of a user resource after a failure, restarting the execution, or putting it back into the scheduler queue. For example:

```
Provision : Recover
UserResource<type: app, ID: 123>
IN SchedulerResource<type:VM, cpu:2.4Ghz, memory:16Gb>
WHEN Event<type:failure , return-code:>0>
```

4.6.3 Action - Configure scheduler

In this section, we explain the meta-scheduling abstraction. This abstraction family specifies the configuration of the behavior of the scheduler. This abstraction is necessary for the tuning of the scheduler to better adjust to the users' workload and the datacenter's environment.

4.6.4 Action - Manage data

In this section, we present the abstraction of data management, which specifies the management of the data users use while running their jobs.

Access input data specifies the access to data that user execution resources take as input. For example:

```
ManageData: AccessInputData
UserResource<type:file , src:/tmp/data>
IN SchedulerResource<type:VM, ID: 123>
WHEN Event<now>
```

Access intermediate data specifies the access to data that user execution resources generate during their runtime. For example:

```
ManageData: AccessIntermediateData
UserResource<type:stream , src:stdin>
IN SchedulerResource<type:SSD, space:150Gb>
WHEN Event<now>
```

Access metadata specifies the access to the user input and intermediate data information. For example, for accessing metadata on one VM about the files generated by another VM:

```
ManageData: AccessMetaData
UserResource<type:files , source:vm<id:123>>
IN SchedulerResource<type:type , src:/tmp/metadata ,
    dst:vm<id:124>>
WHEN Event<now>
```

Replicate specifies the replication or copying of the user input and intermediate data. For example:

```
ManageData: Replicate
UserResource<type:file , src:/tmp/file1>
IN SchedulerResource<type:HDD, src:/global/user1/file1>
WHEN Event<now>
```

Partition specifies the partitioning of the user input and intermediate data so that a subset of the data is placed in different places. For example:

```
ManageData: Partition
UserResource<type:file , src:/tmp/file1 , partitions:
    [partition<start:0 , stop:10 , step:byte> , partition<start:10>]>
IN SchedulerResource<SchedulerResource<type:HDD,
    src:/global/user1/file1/partitions>
WHEN Event<now>
```

Recover specifies the recovery of the user input and intermediate data after the failure of execution or the storage system. For example, for performing checkpoints every 10 minutes:

```
ManageData: Recover
UserResource<type:files , src:vm<id:123> , path:/tmp>
IN SchedulerResource<SchedulerResource<type: SSD,
    src: /global/user1/vm/123/checkpoint>
WHEN Event<interval: 10min>
```

4.6.5 Action - Communicate

In this section, we present the abstraction for communication, which specifies the interactions between the user and the resources and between the user and the scheduler. For example, for sending a signal to a running application:

```
Communicate
CommunicationProcess<type:signal , value:9>
IN UserResource<type:app , id:123>
WHEN Event<now>
```

And an example for setting a callback:

```
Communicate
CommunicationProcess<type:callback , src:callback.py>
IN UserResource<type:app , id:123>
WHEN Event<metric.cpu.utilization:>80%>
```

4.7 Validation through mapping of schedulers

A reference architecture must be able to model real-life schedulers since its main utility is to map items into the model accurately and analyze features and shortcomings. Therefore, in this section, we validate the reference architecture by mapping the five industrial schedulers we identify in RQ1.

We perform two mappings; the first is a detailed mapping of one of the schedulers to exemplify the mapping process. And the second is a generalized mapping of all the schedulers in the same table, validating the reference architecture.

Through the mapping, we respond to the question of RQ2 *What programming abstractions of scheduling are missing in mainstream industrial schedulers?*.

4.7.1 The mapping process

We consult each scheduler’s official documentation and source code and the articles and blogs we find online. Then, using these resources, for each component of the reference architecture, we identify if there is a complete match, partial match, or no match. The meaning of the match is different for objects than for model actions. In the case of actions, a complete match is when the scheduler offers the action. A partial match is when the action is offered in a limited way; that is, the action may only be offered at a specific moment in the lifecycle, e.g., it only allows to scale when the CPU utilization is more than 80% or when the parameters with which the action can be performed are limited, e.g., a service can only be scaled by adding VMs of the same type of resources. A no-match is when the scheduler does not offer the action. In the case of objects, a full match means that the scheduler accepts users to specify the object and that the object specification is flexible so that the scheduler does not have to know the user input in advance. For example, the user can add any metadata information. A partial match means that the scheduler allows the user to specify only a limited set of objects, and the user can only use inputs the scheduler knows in advance. For example, the user can only specify CPU constraints, not any other resource type. A no-match means that the scheduler does not allow to specify the object.

4.7.2 Exemplary mapping of Kubernetes

In this section, we describe and reason the mapping of Kubernetes to the reference architecture.

1. *Provision*: Two provisioning sub-actions are fully implemented: `lease/release` and `scale`². These actions are found in the analysis of Section 3.5. However, `recover`³ and `preempt`⁴ are not fully matched because Kubernetes limits the recovery of containers to simple restart policies, and the `preempt` is not performed via a direct command; instead, Kubernetes decides when and what to preempt based on assigned priorities. Lastly, the `migrate` action is a no-match because it is not implemented. The user cannot perform manual migrations on Kubernetes.

Regarding objects, when applying the `lease/release`, `scale`, `recover`, and `preempt` actions, the user can specify any user resource and scheduler objects. The scheduler resources are described through the resource constraints⁵ we identify in the analysis of industrial schedulers and the user resources through the abstraction of annotations⁶, which allows adding metadata to the user containers. However, the `event` object is

²<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale>

³<https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy>

⁴<https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption>

⁵<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#equality-based-requirement>

⁶<https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations>

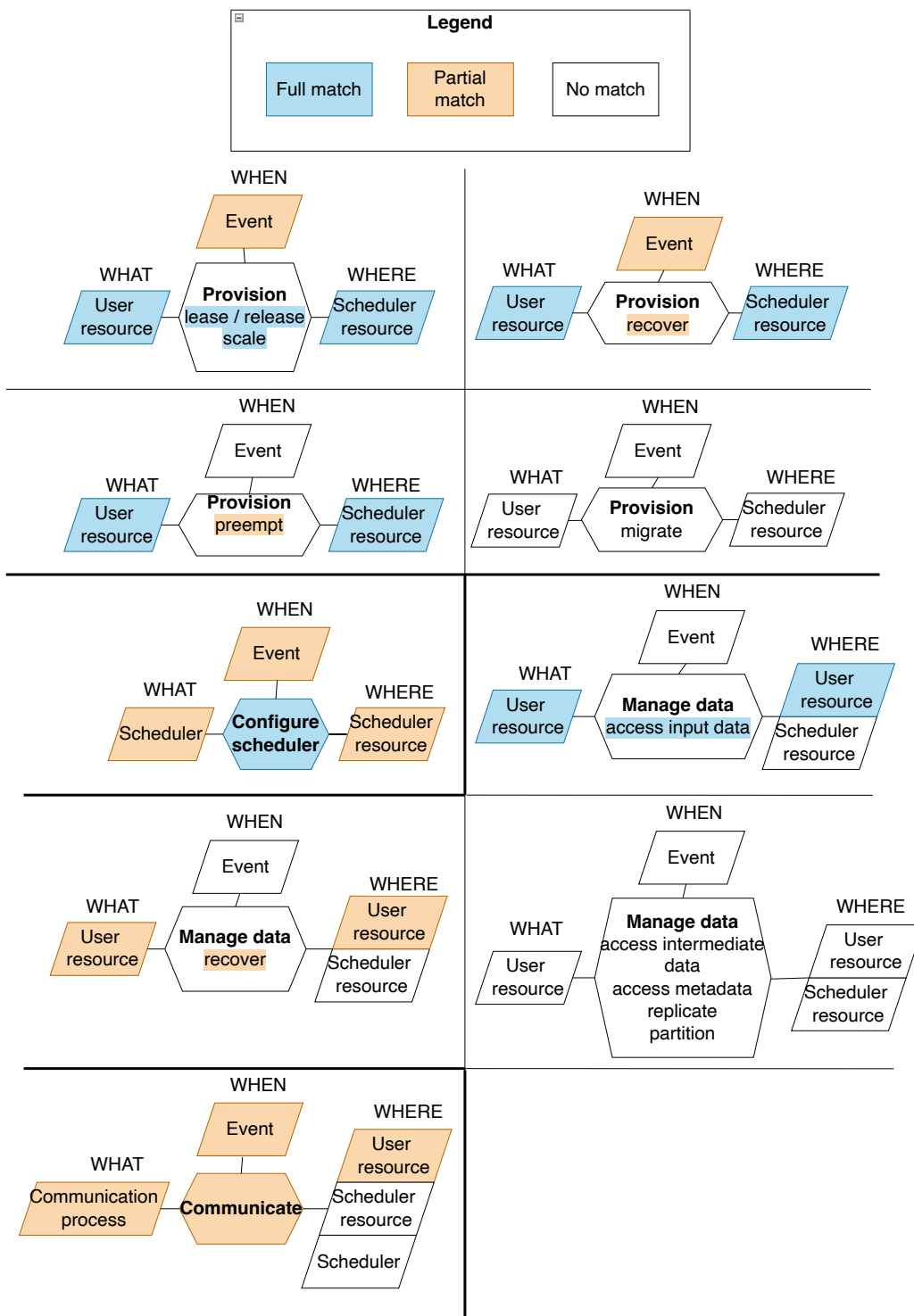


Figure 3: Kubernetes mapped to the reference architecture.

only partially implemented when doing `lease/release` and `scale` and `recover` and is not implemented for `preempt`. It is partially implemented because, in Kubernetes, the user can only `lease/release` using the cron format, which forces them to do recurring executions. The `scale` action only allows scaling based on metrics, not events such as specific dates. The `recover` action is limited to specific policies that include *always* and *onFailure*. Lastly, the `event` object is a no-match for `preempt` because the user cannot control when preemption occurs; Kubernetes decides internally when preemption should happen.

2. *Configure scheduler*: Kubernetes allows users to modify the scheduler configuration⁷ through so-called scheduler profiles specified by a YAML file. Moreover, it allows to deploy of multiple schedulers⁸. However, the scheduler configuration is limited to specific extension points⁹ and plugins¹⁰. Therefore, Kubernetes fully matches the `configure scheduler` action but partially matches the `scheduler` and `event` objects.
3. *Manage data*: Out of the six sub-actions of the management data action, Kubernetes only offers two: `access input data` and `recover`. On the one hand, regarding the access input data action, Kubernetes allows users to specify any input data via volumes¹¹, and the volumes are always attached to the containers that the user describes. Therefore the `user resource` object is fully matched. However, it does not specify when to access the input data nor allow storing and retrieving it from scheduler resources. Therefore, the `event` and `scheduler resource` objects are no-matches. On the other hand, the API it offers to `recover` is not through a YAML file but through an HTTP call with specific parameters¹² that partially lets to specify the `user resource` to recover. Therefore, the `user resource` object is partially matched, while `event` and `scheduler resource` objects are not.
4. *Communicate*: Kubernetes partially matches the `communicate` action since it allows to specify callbacks¹³ and attach the stdin¹⁴ to the containers. However, it does not allow sending generic messages or other communication processes, such as sending signals to containers through the API. Moreover, the callbacks are limited to specific events, and the stdin can only be attached synchronously; users cannot specify events to which stdin is attached. Lastly, only specific user resources can be specified when adding the callbacks and attaching the stdin; users cannot apply the action to multiple containers by grouping them based on specific characteristics. Therefore, `communication process`, `event`, and `user resource` objects are partially matched, while `scheduler resource` and `scheduler` are no-matched.

⁷<https://kubernetes.io/docs/reference/scheduling/config>

⁸<https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers>

⁹<https://kubernetes.io/docs/reference/scheduling/config/#extension-points>

¹⁰<https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins>

¹¹<https://kubernetes.io/docs/concepts/storage/volumes>

¹²<https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api>

¹³<https://kubernetes.io/docs/tasks/configure-pod-container/attach-handler-lifecycle-event>

t

¹⁴<https://kubernetes.io/docs/tasks/debug/debug-application/get-shell-running-container>

4.7.3 Exemplary flow of Kubernetes

This section presents an end-to-end example of a complete execution of Kubernetes, identifying which scheduling actions are performed and what objects are used. The flow example runs since the Kubernetes administrator configures the Kubernetes nodes and the cluster until the users run pods. Next, we list the steps and the specific action and object of the reference architecture:

1. The Kubernetes administrator setups the cluster; for that, it connects to a datacenter and provisions several Virtual Machines where the nodes are deployed. The command would be:

```
Provision : Lease
UserResource <type:vm, id:1>
IN SchedulerResource <type:vm, cpu:2.4Ghz, memory:16Gb,
    gpu:nvidia>
```

2. The Kubernetes manager setups automatic scaling of provisioned resources so that if the VMs consume more than 80% of the CPU power, a new VM is provisioned where a new K8s node is launched. The command would be:

```
Provision : Scale
UserResource <type:vm, id:i+1>
IN SchedulerResource <type:vm, cpu:2.4Ghz, memory:16Gb>
WHEN Event <cpu.usage: 80%>
```

3. The Kubernetes manager configures the cluster so that whenever the VM fails or stops running, it recovers by restarting the VM. The command would be:

```
Provision : Recover
UserResource <type:vm, id:1>
IN SchedulerResource <type:vm, cpu:2.4Ghz, memory:16Gb>
WHEN Event <exit.code:>0>
```

4. Apart from making sure that the user container restarts automatically when it fails, Kubernetes also performs health checks on its nodes to get metrics and understand the health and status of the nodes. The command would be:

```
Communicate
CommunicationProcess <type:metric, metrics:all>
IN UserResource <type:vm, id:1>
WHEN Event <metric: update>
```

5. Once the Kubernetes cluster is initialized, users start using it. The user deploys an application in a container on top of the Kubernetes cluster, passing a pod manifest. In the pod manifest, the user specifies the resources that the container needs, in this case, a GPU, but also that the container will be restarted every time it fails through the *restartPolicy*. Finally, the user specifies the input data they want to access from the provisioned pod. That is, the data the user wants to transfer to the initialization

of the pod. The data comes from a user path (host) under /data. The Pod manifest would be:

```

1  version: apps/v1
2  : Deployment
3  data:
4  me: example-deployment
5  :
6  mplate:
7  metadata:
8    labels:
9      app: example
10 spec:
11   restartPolicy: OnFailure
12   containers:
13   - name: example
14     image: example:1.28
15     resources:
16       limits:
17         nvidia.com/gpu: 1
18     volumeMounts:
19     - mountPath: /example-path
20       name: example-volume
21   volumes:
22   - name: example-volume
23     hostPath:
24       path: /data
25       type: Directory

```

And the commands would be:

```

Provision: Lease
UserResource<type:pod, id:1>
IN SchedulerResource<type:container,
    gpu:<type:nvidia, amount: 1>>

```

```

Provision: Recover
UserResource<type:pod, id:1>
IN SchedulerResource<type:container,
    gpu:<type:nvidia, amount: 1>>
WHEN Event<state: failed>

```

```

ManageData: AccessInputData
UserResource<type:folder, type:host, path:/data>
IN UserResource<type:pod, id:1>

```

6. Next, the user specifies a scaling policy so that when the container exceeds 90% CPU utilization, it automatically provisions a new pod. The manifest would be:

```

1  version: autoscaling/v2
2  : HorizontalPodAutoscaler
3  data:
4  me: example
5  :
6  aleTargetRef:
7  apiVersion: apps/v1
8  kind: Deployment
9  name: example
10 tricks:
11 type: Resource
12 resource:
13   name: cpu
14   target:
15     type: Utilization
16     averageUtilization: 90

```

And the command would be:

```

Provision: Scale
UserResource<type:pod, id:i+1>
IN SchedulerResource<type:container, gpu:<type:nvidia,
    units:1>>
WHEN Event<cpu.usage: 90%>

```

7. Finally, when the user's pod finishes execution, the user releases the container to make room for other users. The command would be:

```

Provision: Release
UserResource<type:vm, id:1>

```

4.7.4 Aggregated mapping of industrial schedulers

In this section, we repeat the mapping process for all the industrial schedulers and add the results in two tables. In Table 2, we map the actions and sub-actions, and in Table 3, we map the objects. For each action or sub-action, we specify if it is a complete, partial, or no match. And for each object, we specify if it is a complete, partial, or no match.

In the aggregated tables, we relate objects to several sub-actions simultaneously. In each sub-action, the objects can be specified in different ways, and therefore in some, the objects will be a whole match, and in others, partial or no match. Therefore, to decide to set an object as full, partial, or no match, we do the following: 1) for each sub-action, we see what type of match it is; 2) if it is full, we assign 100% if it is partial 50% and if it is none, 0%, 3) for each object we average the percentage of the actions that are a complete, partial match, and 4) if it is less than or equal to 33% we set it as no match if it is greater than 33% and less than 66% we set it as partial match, and if it is greater than or equal to 66% we set it as a full match.

Table 2: Full overview of programming abstraction actions of schedulers mapped to the reference architecture.

Action	sub-action	Schedulers				
		Ku	Sl	Sp	Co	Ai
Provision	lease / release	●	●	●	●	●
	scale	●	○	◐	○	○
	migrate	○	○	○	○	○
	preempt	◐	●	○	●	○
	recover	◐	◐	●	◐	◐
Configure scheduler		●	◐	●	●	●
Manage data	access input data	●	◐	●	●	●
	access intermediate data	○	○	◐	○	○
	access metadata	○	○	○	○	○
	replicate	○	○	●	○	○
	partition	○	○	●	○	○
	recover	◐	○	●	●	○
Communicate		◐	●	◐	◐	◐

Legend: ●/◐/○ = full/partial/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow.

Table 3: Full overview of programming abstraction objects of schedulers mapped to the reference architecture.

Action	Object	Schedulers				
		Ku	Sl	Sp	Co	Ai
Provision	user resource	●	◐	◐	◐	◐
	event	◐	◐	◐	◐	◐
	scheduler resource	●	●	◐	●	●
Configure scheduler	scheduler	◐	◐	●	●	●
	event	◐	◐	○	○	○
	scheduler resource	◐		○	○	○
Manage data	user resource	◐	◐	●	◐	●
	event	○	○	○	○	○
	scheduler resource	○	◐	○	◐	○
Communicate	communication process	◐	◐	◐	◐	●
	event	◐	◐	◐	○	◐
	user resource	◐	◐	●	◐	●
	scheduler resource	○	◐	●	○	○
	scheduler	○	◐	○	○	○

Legend: ●/◐/○ = full/partial/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow.

On the one hand, regarding actions of programming abstractions, the results indicate that the industrial schedulers have shortcomings and that several actions and components do not fully implement them. There is an obvious pattern: most schedulers implement four actions: `lease/release`, `configure scheduler`, `access input data`, and `communicate`. And yet, the `communicate` action is partially implemented. All others are either partially implemented or not implemented at all. The biggest shortcoming is the `manage data` action, where most sub-actions are not implemented. That is, industrial schedulers are not designed to manage and schedule the data used by users.

On the other hand, there is also a clear tendency in the objects where most of the objects for the `provision` and `configure scheduler` actions are fully implemented. While most objects for `manage data` are partially implemented, many are no matches. The schedulers capable of implementing the most significant number of objects are Kubernetes and Apache Airflow. Therefore, we conclude that the programming abstractions of the industrial schedulers are under-implemented.

In conclusion, in this section, we validated the reference architecture, being able to map all the industrial schedulers. Furthermore, we conclude that the mainstream industrial schedulers are unimplemented since a large part of the actions and objects of the reference architecture are partially implemented or not implemented at all.

4.8 Mismatch between industrial scheduler aggregated mapping and the reference architecture

The reference architecture we built has several mismatches with the analysis presented in section 3.5 of industrial scheduler abstractions; the two are not 1:1. In this section, we explain why there are differences between these two models. But before explaining the differences, it is essential to clarify that the analysis of industrial schedulers does not differentiate between action and object. Unlike the reference architecture, it mixes the different types of abstractions in the same table. Next, we will explain the differences by separately identifying the objects and actions they do not have in the joint.

On the one hand, three actions exist in the reference architecture and are not in the industrial scheduler table. The three actions are `migrate`, `accessIntermediateData`, and `accessMetadata`. Surprising as it may be, none of the schedulers offer an abstraction for migration. Some schedulers like Kubernetes perform migrations internally to optimize scheduling, but none offer an explicit API to users. We include it in the reference architecture since, based on our experience, it seems to be a necessary action in scheduling, which should not be implemented implicitly and internally but should be a main action. Afterward, none of the schedulers implements the abstractions to access intermediate data and metadata. Even so, we include them in the architecture reference because the most relevant academic articles present scheduling abstractions for data management [34] [9] identify these two APIs as essential.

On the other hand, several abstractions are in the industrial scheduler table but not in the reference architecture. The abstractions that do not appear are all the sub-abstractions of `configure scheduler`, all the `constraints` and `quality of service` abstractions, and

finally, all the sub-abstractions of `communicate`. All these abstractions that we find in the table of industrial schedulers but not in the reference architecture are objects. If we look closely, these abstractions are object nouns instead of action verbs. We do not have them in the reference architecture because instead of specifying exact implementations, we generalize the objects in a way that models as many existing and future scheduling objects as possible. However, in the table of the industrial scheduler, we identify the concrete abstractions we find. Generalizing has advantages and disadvantages, which we list in the limitations section of the reference architecture in 4.9. The sub-abstractions of `configure scheduler` are represented as objects of type `scheduler`, the `constraints` as objects of type `scheduler resource`, the `quality of service` as objects of type `user resource`, and the sub-abstractions of `communicate` as objects of type `communication process`.

Lastly, it is necessary to explain that while in the analysis of industrial schedulers, we only have full or no matches, when mapping the reference architecture, we add partial matches. Industrial scheduler analysis aims to identify and discover abstractions, so there is no necessity for differentiating between full and partial matches. However, the reference architecture mapping aims to understand what abstractions schedulers implement. That is why we introduce the partial match in the mapping.

4.9 Limitations

A reference architecture is limited because it is kept at a sufficiently high abstraction layer to map and represent all schedulers. Therefore, the abstraction layer cannot represent all the details that allow differentiating and comparing one scheduling API from another. It is essential to identify what these limitations are.

The limitations, like the reference architecture, are divided into two groups, the actions and the objects of the APIs. Regarding actions, the reference architecture identifies the primitive actions schedulers can offer. Our goal is that the actions are future-proof so that in 10 years, the actions do not need to be extended. However, it is impossible to know if any new field will develop or become a first-class citizen forcing schedulers to offer new actions in their API. Even so, we believe that future trends should not suppose an extension in the actions that the schedulers offer but in the definition of the objects.

The most important limitations are found in the objects. Our reference architecture has only five distinct objects, and we do not specify sub-objects for each. For example, one of the objects is the `SchedulerResource`, which allows the user to specify the characteristics of the resources she wants to assign. But our reference architecture does not differentiate between an API that offers VMs or Edge mobile devices. This is a limitation, but it is made to be future-proof since if there is one thing sure, the resource type is constantly changing. For example, now there is an increasing demand for resource schedulers on serverless platforms or edge devices, and in five years, nobody knows where the user jobs will be executed.

4.10 Summary

In this section, we identify the shortcomings of the industrial scheduler APIs, build a

reference architecture and map the schedulers to it. To build the architecture reference, we carried out a literature survey of academic schedulers, in which we selected and analyzed 15 articles using a systematic literature survey. Then, to generate the final version of the reference architecture, we combine it with the programming abstractions found in the industrial scheduler analysis and intuition. The reference architecture comprises five unique objects and four actions containing several sub-actions. The main abstractions of the industrial schedulers that are missing when mapping to the reference architecture are those related to data management and communication.

5 Experiments with the reference architecture

In the previous section, we mapped industrial schedulers into the reference architecture, and we identified under-implemented APIs. We state that schedulers prioritize simplicity in their programming models, and thus, they limit users' programmability. We hypothesize this simplicity has schedulers need to give their users greater programmability and to prove that schedulers must give greater programmability to their users to improve user-applications performance.

5.1 Selection of under-implemented scheduling APIs to experiment


In this section, we choose three different under-implemented programming abstractions in the scheduler APIs we will use to perform the experiments. The shortcomings are obtained from the mapping carried out in the previous section. Below we present the summary for each of the experiments, detailing: 1) a specific use case in which API scheduling is required, 2) how an ideal scheduler implements the abstraction, 3) the reason why one or more industrial schedulers cannot implement it, and 4) the extension they need to implement it.

Experiment 1: Reservations

- **Use-case:** Users make reservations of VMs; specifically, they request the scheduler to lease a vm at a specific datetime in the future, and they include an estimate of runtime for the scheduler to optimize the placement.
- **Ideal scheduler:** The user performs a lease action and submits a user scheduler object containing metadata about the expected runtime and an event object specifying the date and time in the future at which the virtual machine needs to be leased.
- **Industrial scheduler shortcomings:** Kubernetes does not provide a proper API for reservations; since it forces the users to submit recurring jobs, it does not allow them to execute them only once. SLURM allows specifying reservation dates, but it does not provide an abstraction for users to submit runtime estimates. Spark does not provide abstractions for reservations or runtime estimates. Condor, like SLURM, has the ability for future submissions but not for submitting estimates. Lastly, Airflow allows users both future submissions and estimates of runtime.
- **Extension:** Kubernetes must allow non-recurring future submissions. SLURM and Condor must allow users to submit runtime estimates, and Spark must allow users to submit future executions and runtime estimates. For allowing estimates, schedulers must be extended to fully implement the `user resource` objects and `event` objects for allowing future submissions.

Experiment 2: Migrations

- **Use-case:** When interferences occur in a physical machine, the scheduler requests the user to migrate or reduce part of the workload to another VM through a callback.
- **Ideal scheduler:** The user performs a `communicate` action specifying a communication process object: a callback, an `event` that identifies when there is interference,


Table 4: Diff on the mapping of the programming abstraction objects for Experiment 1.  represents the diff, the changes required for implementing the extension.

Action	Object	Schedulers				
		Ku	Sl	Sp	Co	Ai
Provision	user resource	●	●	●	●	◐
	event	●	◐	●	◐	●
	scheduler resource	●	●	◐	●	◐

Legend: ●/◐/○ = full/partial/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow.

and a useful resource object identifying the VM where the interference occurs.

- **Industrial scheduler shortcomings:** Kubernetes, SLURM, Spark, and Airflow allow users to submit callbacks, but none provide the object abstraction of a migration request within the callbacks. Unlike all the others, Condor does not provide an abstraction for users submitting callbacks.
- **Extension:** Kubernetes, SLURM, Spark, and Airflow must extend the API to include event objects that represent migrations, while Condor needs to extend its `communicate` action to accept callback `communication process` objects and also accept `event` objects that represent migrations.

Table 5: Diff on the mapping of the programming abstraction objects for Experiment 2.  represents the diff, the changes required for implementing the extension.

Action	Object	Schedulers				
		Ku	Sl	Sp	Co	Ai
Communicate	communication process	◐	◐	◐	●	●
	event	●	●	●	●	●
	user resource	◐	◐	●	◐	●
	scheduler resource	○	◐	●	○	○
	scheduler	○	◐	○	○	○

Legend: ●/◐/○ = full/partial/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow.

Experiment 3: Metadata Access

- **Use-case:** When the user has a bag-of-tasks workflow where each task processes a data object, the user requests metadata about each object to optimize the order in which the tasks are executed.
- **Ideal scheduler:** The scheduler internally records every object storage server’s congestion and returns the expected retrieval time based on the internal congestion metrics for every metadata request on objects.
- **Industrial scheduler shortcomings:** Kubernetes, SLURM, Spark, Condor, and

Table 6: Diff on the mapping of the programming abstraction actions for Experiment 3.

■ represents the diff, the changes required for implementing the extension.

Action	Subaction	Schedulers				
		Ku	Sl	Sp	Co	Ai
Manage data	access input data	●	◐	●	●	●
	access intermediate data	○	○	◐	○	○
	access metadata	●	●	●	●	●
	replicate	○	○	●	○	○
	partition	○	○	●	○	○
	recover	◐	○	●	●	○

Legend: ●/◐/○ = full/partial/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow.

Airflow, none provide API for accessing metadata.

- **Extension:** Kubernetes, SLURM, Spark, Condor, and Airflow must implement the access metadata abstraction.

A comprehensive overview of these experiments can be found in Table 7, which outlines the API extensions, parameters, traces, and metrics for each use-case.

5.2 Traces

For the experiments, we use real-world trace workloads. We chose traces from private and public cloud environments, which offer anonymized requests from VMs, plus aggregated metrics on resource utilization every 5 minutes. The chosen traces are Bitbrains Azure and Google. While Bitbrains and Azure traces are VM requests, Google traces are task requests. Generally, task requests are shorter in duration and consume fewer resources than VM requests. We include task requests to enrich our experiment and see how the results of experiments change with short-lived and small consumption requests. Even though our experiments are designed for VMs, it’s straightforward to convert them for task requests and CPU core reservations instead of VM reservations. Since the logic of experiments is the same for VMs as it is for tasks. Moreover, in the simulation of the experiment, we do not have to make any major changes.

Bitbrains is a private cloud provider operating mainly in the Dutch ICT market; the aggregate duration is one month with 1250 VMs. Finally, we also chose the Azure and Google traces, which are traces from Microsoft’s and Google’s public cloud providers, respectively. On the one hand, the Azure trace is very recent, from 2020. The original trace contains 2 million VMs, and the aggregate duration is approximately two and a half months. However, it is a very large trace compared to the other traces, requiring much time and execution power. Therefore, 1829 VMs from the original trace are sampled using the OpenDC sampling tool. On the other hand, the Google trace is from 2014, its original trace contains 17.8 million tasks, and the duration is approximately one month. This trace

Table 7: Summary of evaluation experiments.

Name	API extension	Parameters	Fixed parameters	Traces	Metrics
Reservation 5.5	User provided start time and resource estimates	Reservation ratio re-source utilization	Scheduling policy (EFT)	Azure Bitbrains Google	Waiting time slow-down
Migration 5.6	Container migration via orchestrator callbacks	Migration type over-subscription	Resource utilization (85%) FIFO policy	Azure Bitbrains Google	Execution time packing efficiency
Metadata access 5.7	Use storage subsystem busyness to order tasks	Metadata-aware task reorder policy	Resource utilization (80%)	Google and IBM combined	Buffer size total time

is very large, so we sample 1 million requests from the original trace in 2.5 days. We sample more requests than in Azure since the Google runtime is much smaller. This is because Google traces are not VM requests but task requests.

Table 8 summarizes the characterizations of these workloads.

Workload	VMs	Duration [Days]	VM duration		CPU cores		CPU capacity		Memory	
			[Days]	[Days]			[GHz]	[GHz]	[GBs]	[GBs]
			Mean	σ	Mean	σ	Mean	σ	Mean	σ
Bitbrains	1250	30	28	5	3.27	4.04	2.7	0.16	11.75	32.6
Azure	1829	30	2	6	2.48	2.28	2.5	0.0	5.8	10.16
Google	1M	2.5	0.0375	0.083	1.0	0.0	1.68	2.08	0.17	0.2

Table 8: Characteristics of the traces of the experiments

5.3 Execution

The reproducibility of the experiments is crucial for their validity and for other researchers to extend them. An experiment is reproducible if the methods are sufficiently well described and the artifacts available so that others running the same experiment will get identical results. We run the experiments on a personal laptop with an Apple M1 Max chip, 1TB SSD storage, and 32 GB memory. We run the experiments using OpenDC¹⁵, an open-source datacenter discrete event simulator developed by AtLarge, with multiple years of development and operation. Due to its discrete event model, OpenDC generates the same results regardless of the hardware used. The execution of each configuration lasts about 1-3 minutes, and the execution of each configuration has been repeated 20 times. All artifacts,

¹⁵<https://opendc.org/>

including the traces used in the experiment, are available in <https://github.com/aratz-lasa/opencv>.

5.4 General requirements

Each experiment has its requirements based on the extension they evaluate. However, there is a set of common non-functional requirements that all experiments have. We list them below:

NFR Make reproducible experiments

For an experiment to be reliable, it must be reproducible since today; there are notorious problems with the inability to reproduce scientific experiments. For this reason, it is necessary to publicly offer both the raw results of the experiments and the software artifacts used to carry them out.

NFR Make the experiment software artifacts reusable

Considering current programming standards, it is important to implement extensible and modular software artifacts so that it is easy to reuse or extend them for other projects.

NFR Provide experiments with different workloads.

To offer greater insight and reliability of the experiment, it is necessary to use different workloads. They must be different in terms of characteristics, such as the duration of each task, the resource requirements, the nature of the workload, etc.

5.5 Extension 1: Reducing VM waiting times and slowdowns using reservations

In the first experiment, we evaluate the use case where users want to make reservations. Still, some of the mapped programming abstractions of industrial schedulers cannot implement it. Therefore, we evaluate the benefits of providing reservation programmability in a scheduler in this experiment. We demonstrate that we can obtain higher performance on VM scheduling by using reservations. Reservation is a programming abstraction allowing users to submit scheduling requests that will be executed in the long term. That is, it allows users to provision resources ahead of time. Usually, reservation systems are used by applications running repetitive jobs that may know they will run a job several hours or days in advance. Studies show that many production jobs are long-running and are submitted periodically, which would benefit from a reservation system [10, 24].

In Section 4.7, with the mapping of industrial schedulers, we see that Kubernetes and Spark do not offer the ability to make reservations, while SLURM and Condor do not allow users to submit estimates. In addition, the schedulers that allow reservations do not leverage this capability to optimize scheduling. Most expose a cron-like programming abstraction so users can run a job periodically on a given schedule. But they do not apply any optimization algorithm to the reservations. In other words, reservations are used to facilitate user interaction, not to improve scheduling performance. Therefore, in this experiment, we extend a scheduler to provide reservation programmability to the

users. Then, we apply an optimization algorithm to improve scheduling key metrics such as waiting times and slowdowns.

5.5.1 Requirements

Before designing the experiment and the API extension, we identified specific functional requirements for the experiment, and we present them below:

FR Enable expression of reservations on datacenter scheduler APIs

The main objective of this experiment is to implement a datacenter API that offers the ability to make reservations. We hypothesize that it is necessary to offer this capability to obtain scheduling performance improvements, which is impossible without reservations. This occurs because reservations enable planning ahead of scheduling and greater insight into the workload. These two characteristics are the following two listed requirements.

FR Enable scheduling optimizations through plan-ahead mechanisms

Through reservations, the extended API allows the datacenter provider to plan ahead of schedule. In other words, to decide how and when you will provision the reservations. This offers a great opportunity to apply optimizations to the scheduling algorithm.

FR Enable scheduling optimizations through workload estimates

Reservations allow the ability to plan to be combined with greater insights about the workload. Users submit estimates of execution time or other workload characteristics, offering the greater potential to optimize scheduling. Datacenter providers can analyze and generate estimates of reservation requests using artificial intelligence techniques. However, not always accurate enough, and we believe that abstracting the user from specifying the estimates (if any) implies a sacrifice in performance.

5.5.2 System model

To evaluate the extension of a system with reservation programmability, we model VM scheduling in a datacenter. The datacenter has tenants, and there is a specific category of jobs that are long-running and periodically submitted [43], which are provisioned into VMs (❶ and ❷ in Figure 4). The VMs run for some time, and when they finish, they release the used resources. The physical machines of the datacenters are heterogeneous regarding resources; they have different combinations of CPU cores and frequencies. The scheduling requests specify the number of CPU cores, the frequency in MHz of the cores, and the amount of memory. That is to say, the lease action is limited to specifying resource requirements. Users consume 100% of the resources they have specified in the request. Therefore, the scheduler does not over-subscribe the physical machines. Otherwise, there will be interference and resource contention between the tenants.

To perform scheduling requests, the scheduler offers the following API to users:

- `lease(requirements)`: `vm`: the user passes a list of resource requirements, the provider boots up a virtual machine with those requirements, and returns the machine

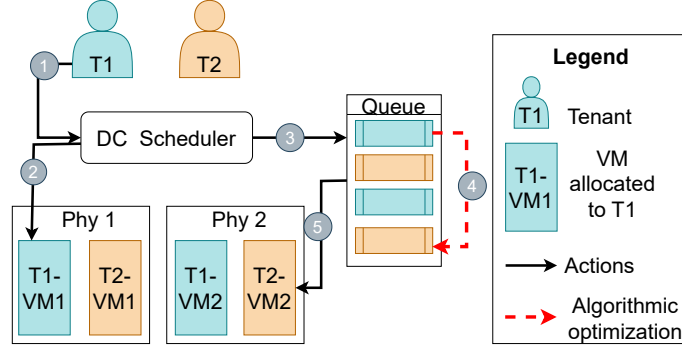


Figure 4: Reservations experiment system model.

to the user. The requirements are composed of CPU cores, CPU capacity, and memory.

- `release(vm)`: the user passes a machine, and the provider shuts it down.

5.5.3 Model extension

A large part of the jobs executed in clusters offers automated predictability systems that periodically submit them. Many of these are resource intensive and run for long hours. As these jobs are executed periodically, several investigations confirm that their runtime can be accurately predicted. In addition, users know in advance that they will have to run them. Therefore, users and datacenter providers can benefit from using reservations.

To enable reservations, we extend the system by modifying the `lease` action, including two additional parameters: runtime estimates and a specified provisioning time for future reservations. The runtime estimates are `user resource` objects, and the datetimes are `event` objects. When a user submits a reservation request, instead of immediately provisioning it, the scheduler adds the request to a reservation queue (3) alongside other pending reservations. During this time, the scheduler applies algorithmic optimizations to improve future provisioning (4). In our experiment, we employ a simple EFT scheduling policy [41] to optimize the reservation queue by prioritizing tasks with earlier estimated finish times, ensuring that resources are allocated efficiently and effectively. Tasks without reservation are scheduled according to the FIFO policy. Once the specified provisioning time arrives, the scheduler provisions the reserved resources into a VM (5), fulfilling the user’s reservation request. In Listing 5.5.3, we provide an example of the extension, showcasing the syntax for reservations.

```
Provision : Lease
UserResource <type:app, id:1,
runtime:1h>
IN SchedulerResource <type:vm, cores:8,
cpu-freq:2.4Ghz, memory:32Gb>
WHEN Event <day:11, month:12, year:2023>
```

The extended programming model offered by the scheduler is the following:

- `lease(requirements, datetime, estimate)`: `vm`: the user specifies the resource requirements of the virtual machine, the datetime of the submission for performing reservations to the future, and the runtime estimate that is used for providing insights for optimizing reservations. If the datetime is not empty, the scheduler stores the request in its reservation system and returns a VM object that has not yet been booted up. The future VM contains a callback to be notified when the machine is initialized. But if the datetime is not specified, the scheduler processes it as a regular lease action and immediately provisions the VM.

5.5.4 Alternatives

Prior to finalizing the implementation of the reservations API, we explored various alternatives to fulfill the system extension requirements. Next, we will briefly explain the alternatives and argue our chosen design.

Reservation without estimates. This is the simplest alternative for users. Users specify the requirements of their VMs and submit the requests to the datacenter. The requirements specify the resources they need and when to provision them. However, it does not include expected execution times or metrics such as average utilization. Therefore, the datacenter generates and applies machine learning techniques to analyze the request and generate a profile. This profile is necessary to apply optimizations to reservation scheduling.

Reservation with estimates. This is the alternative we chose to implement as the extension of the experiment. Apart from specifying the requirements, users also specify the estimate of the execution time. This implies greater implementation complexity for the user, but we consider the sacrifice of performance gains is not worth it in exchange for greater simplicity. Since users have business knowledge and have a greater facility to calculate estimates of their workload, this does not prevent the provider from applying techniques to analyze and classify the requests. The two can be combined.

5.5.5 Industrial schedulers

In this section, we explain how the industrial schedulers that we identified in Section 3 and 4 would implement the abstraction that we evaluated in this experiment and if they do not have it in their API, how they would implement it.

Kubernetes allows you to specify a date in the future using the cronjob. But it doesn't allow executing the job only once. Therefore, the user must delete the cronjob, or else the job has to have logic to auto-delete after completion.

```

1  apiVersion: batch/v1
2  kind: CronJob
3  metadata:
4    name: example
5    metadata:
6    annotations:
7    runtime: 1h

```

```

8 spec:
9   schedule: "0 0 29 8"
10  jobTemplate:
11    spec:
12      template:
13        spec:
14          containers:
15            - name: hello
16              image: example:1.28

```

In this example, the CronJob manifest executes the user task at `0 0 29 8`, which states that the task must be started on the 29th of August of this year. Apart from the reservation datetime, it also specifies the runtime of the application, which is one hour, by making use of annotations, which is for specifying metadata.

SLURM can implement reservations partially, as it offers the ability to specify a date-time in the future but does not allow the user to provide metadata about the runtime. Therefore, we show how SLURM would implement the reservation extension with its current API and a custom extension below.

In this example, the argument *begin* specifies the date and time the job must be provisioned and run.

```

sbatch --job-name=example --begin=2023-08-29T00:00:00
      --metadata=runtime:1h run.sh

```

In this example, the argument *begin* specifies the date and time the job must be provisioned and run. This parameter already exists in SLURM. However, the user specifies a second parameter named *metadata*, which specifies the job's expected runtime of one hour *runtime:1h*.

Spark cannot implement reservations among the industrial schedulers since it does not offer them. Next, we show how the scheduler could implement reservations in its API.

```

./bin/spark-submit
--class org.apache.spark.examples.SparkPi
--master spark://207.184.161.138:7077
--conf spark.reservation.datetime=2024-01-20T12:34:00
--conf spark.reservation.estimate=1h
/path/to/examples.jar

```

In this example, a new configuration flag is created *spark.reservation.datetime*, which is set to `2024-01-20T12:34:00`, specifying that the application should start at a specific datetime. Moreover, a second flag is *spark.reservation.estimate* is created, which is set to `1h`, specifying that the application is expected to run for one hour. This is just one option for implementing reservation in Spark, but the flag's name, format, etc., can differ.

Condor implements reservations partially, as it offers the ability to specify a deferral time, that is, the ability to specify a future time when users want to provision the resources. Still, it does not allow the user to provide the expected runtime. Therefore, next, we show how Condor would implement the reservation extension with its current API and a custom extension.

```
1 executable = example
2 arguments  = SomeArgument
3
4 deferral_time = 1693267200
5 metadata_runtime = 1h
6
7 queue
```

In this example, the job's submit description file specifies in Unix epoch that the job will begin execution on August 29th, 2023, at 12:00 pm, through an already existing *deferral_time* parameter. However, the user specifies a second parameter named *metadata_runtime*, which specifies the job's expected runtime of one hour *1h*.

Airflow lets users specify crontabs and limit runtimes, but also, users can specify specific dates and times to run their jobs without having to be recurring as in crontab. Here is a specific example:

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator
5
6 def example():
7     return 'Hello world from first Airflow DAG!'
8
9 dag = DAG('hello_world', description='Hello World DAG')
10
11 default_args = {
12     'start_date': datetime(2023, 8, 29, 0, 0),
13     'end_date': datetime(2023, 8, 29, 1, 0),
14 }
15
16 example_operator = PythonOperator(task_id='example_task',\
17     python_callable=example, dag=dag, default_args=default_args)
18
19 example_operator
```

In this example, the *start_date* argument is set to *2023-08-29 at 00:00* and is passed as *default_args* to the DAG, which tells Airflows the time at which the workflow must start running. With the starting date, the *end_date* argument is set to *2023-08-29 at 01:00*, specifying that the application is expected to finish after running one hour.

5.5.6 Configuration and design of the experiment

We aim to conduct experiments that prove the limitations of a scheduler that does not provide reservation programming abstractions. Table 9 summarizes this extension’s configurations. Experiment configurations comprise three dimensions: trace, utilization, and reservation ratio. Below, each dimension and the different choices are explained, in addition to the metrics collected in the experiments.

Traces	Utilizations	Reservation ratios
Bitbrains	0.75 0.8 0.85	0.0 0.5 1.0
Azure	0.75 0.8 0.85	0.0 0.5 1.0
Google	0.75 0.8 0.85	0.0 0.5 1.0

Table 9: An overview of all the configurations of the reservation extension experiment.

Utilization and reservation ratio Apart from the trace, different combinations of resource utilization and reservation ratio are evaluated in the experiment.

For the experiment to be complete and valid, we experiment with different proportions of resource utilization. To do this, each trace’s average CPU and memory use has been calculated, and we generate an infrastructure topology that offers the desired utilization. We experiment with three different utilizations: 75%, 80%, and 85%.

Finally, we experiment with different reservation ratios. The reservation ratio is the proportion of VM requests submitted through the reservation programming abstractions, submitted in advance before wanting to boot up. We opted for three ratios: 0, 0.5, and 1.0 to keep the experiment and conclusions simple. 0 represents the neutral scenario in which reservation programmability is not available. 0.5 represents when half of the requests are made through reservations, and 1.0 is when all are reservations.

Metrics We apply the EFT heuristic to reservations to optimize the performance of the VMs. The performance of this experiment is defined by the metrics of waiting time and slowdown. Therefore, in this experiment, we will observe how the values of these metrics change in each of the configurations presented in table 9. The response time metrics and the use of a VM remain the same, regardless of the reservation programmability, since these metrics depend on the internal characteristics of the trace. However, we calculated the utilization to verify that the topology calculation was correct.

OpenDC allows exporting many metrics to analyze the simulation results. However, for our experiment, we only need a subset of them. Specifically, we only need the metrics to calculate the waiting time, the slowdown, and the utilization. We present the metrics we collect for the analysis of the experiment in Table 10.

5.5.7 Implementation of a Software Prototype

This section explains the prototype we develop to experiment. To experiment, we use OpenDC, an open-source datacenter discrete event simulator developed by AtLarge, with multiple years of development and operation. In the experiment, we extend OpenDC to

Name	Unit	Description
vm.id	-	Unique identifier of the VM
vm.provision time	Epoch (ms)	The instant at which the server was enqueued for the scheduler
vm.boot time	Epoch (ms)	The instant at which the server booted
vm.timestamp	Epoch (ms)	The timestamp of the current VM metric entry
machine.id	-	Unique identifier of the physical machine of the datacenter
machine.cpu utilization	-	The CPU utilization of the machine
machine.cpu count	-	The number of logical processor cores available for this machine
machine.timestamp	Epoch (ms)	The timestamp of the current physical machine metric entry

Table 10: The metrics that are recorded for the reservations extension evaluation.

include the reservation ability, add a new allocation policy, and optimize and fix bugs in the code. Below we explain the changes we make. The original code of the extension is available in <https://github.com/aratz-lasa/opendc/tree/master/opendc-experiments/reservation>.

Reservation service We implement our version of the `ComputeService` component to implement the reservation service. This component is responsible for carrying out the scheduling requests of the traces. Our extension includes the ability to simulate reservations. To simulate reservations, we extend the OpenDC compute service algorithm, which now has three extra steps: (1) splitting the trace into reservations and online (non-reservation) scheduling requests, (2) applying heuristics to reservation requests to optimize scheduling, and (3) combining and scheduling reservations and online requests. In the simulation, both reservation and online requests have a start time. In the case of reservations, the start time represents when the reservation has to be scheduled, while in non-reservations, it represents the arrival time of the request. For this reason, online requests must always be simulated by sorting them by start time since this is how the arrival of a request is simulated.

In Algorithm 1, we present the high-level algorithm that implements the reservation simulation logic. Based on the reservation ratio, the first step is to divide the trace into two groups, non-reservation (online) and reservation requests. For example, if the reservation ratio is 0.5, the trace is divided into two equal parts. One group simulates the requests that do not use the reservation service, and the other group simulates those that submit reservations. Those that do not use the reservation service are sorted on a First-Come-First-Service basis (in other words, by the start time) to simulate the arrival of requests. However, as reservations are submitted in advance, they are optimized by sorting them by the EFT heuristic. Once both groups have been put into sorted queues, the requests begin to be scheduled. Until both queues are empty, the heads of the reservations and no-reservation queues are queried, and the request with the earliest start time is popped. By looking only at the heads of the queues, we are simulating the decision of whether we

must schedule first a new online request that just arrived or the next reservation request that we optimize in advance. Note that we do not alter the EFT ordering applied to the reservations by looking only at the heads. Finally, we wait until the request has been scheduled, schedule the request, and check back the queues.

Once a request is scheduled, the OpenDC scheduler decides which physical machine to place the request on. It is important to understand the difference between our algorithm, which decides how to optimize reservations and whether to schedule an online request or a reservation first and the OpenDC scheduler, which decides which machine to place a request. This separation is due to the internal architecture of OpenDC, but in the real world, these two functions may be combined.

The ordering of reservations applying for the EFT heuristic works as follows: (1) we generate time intervals for the duration of the trace, (2) we obtain the reservation requests that their start time is within the interval, (3) we order the requests according to their expected finish time (start time plus the expected runtime), and finally (4) we append those requests into the reservation queue. The intervals are used, so requests that start very late but end very early do not block the other requests and increase their waiting times. So, the requests are ordered by the earliest finish time for each interval, not the entire trace duration. Also, it is important to explain that the intervals are generated by calculating the average start time frequency of the requests. In other words, how often a request must be scheduled based on their start times.

Secondary optimizations and bug fixes In addition to implementing the reservation service, we also extend OpenDC to implement a set of optimizations:

- A new allocation policy that prioritizes Virtual Machines that best fit the resource requirements. In other words, this policy evaluates that the physical machine would leave fewer resources available if the VM were assigned. We implement this policy to try to get a higher bin packing. This gives us more significant and precise control over the overall utilization throughout the experiment.
- A new scheduling mechanism that does not stop processing requests at the first failure. The current scheduling mechanism has a queue of scheduling requests, and as soon as a request fails to allocate, it stops scheduling. However, there are cases where a request cannot be allocated, but the next ones in the queue can. Therefore, we update the code to iterate over all requests instead of stopping on the first failed attempt.
- Export utilization metrics of the simulation. OpenDC calculates each machine's CPU utilization but does not export it. Therefore, we modify the code to be able to export resource usage.
- A bug fix for VMs simulation. There was a bug when simulating VMs, in which the CPU frequency of each core was miscalculated. The CPU frequency of each core was set to the aggregated frequency of all the CPU cores.

Algorithm 1: Reservation simulation algorithm

```
1 Function ReservationSimulation(trace, reservationRatio):
2   online, reservations = splitTrace(trace, reservationRatio)
3   online = sortByFCFS(online)
4   reservations = sortByHeuristicEFT(reservations)
5   for  $0 < \text{reservation.size}$  or  $0 < \text{fcfs.size}$  do
6     request = popNextEarliestStartTime(fcfs, reservations)
7     sleep(request.startTime)
8     schedule(request)
9
10 Function splitTrace(trace, reservationRatio):
11   shuffle(trace)
12   splitIndex = trace.size * reservationRatio
13   reservations = trace[0..splitIndex]
14   fcfs = trace[splitIndex..trace.size]
15   return fcfs, reservations
16
17 Function sortByHeuristicEFT(trace):
18   intervals = getTimeIntervals(trace)
19   traceEFT = List()
20   for interval in intervals do
21     intervalRequests = List()
22     for request in trace do
23       if interval.t0 < request.startTime and interval.stopTime < range.t1
24         then
25           intervalRequests.append(request)
26       sortedRequests = sortByStopTime(intervalRequests)
27       traceEFT.append(sortedRequests)
28   return traceEFT
```

5.5.8 Results

This section presents the experiment results for the Bitbrains, Azure, and Google traces. In Figures 5, 6 and 7, we present the waiting times and slowdowns for each combination of utilization and reservation ratio. The objective is to show the differences in scheduling performance between the configurations that use the reservations API and those that do not. On the upper left of the figures, we present the ECDF of the waiting times, and on the upper right, the ECDF of the slowdown, which is the sum of waiting time and execution time, normalized by running time. The intuition behind it is that the slowdown will also be reduced if the waiting time is reduced. Finally, on the lower middle, we present the average bars of slowdowns per configuration to learn how the results of the ECDF are reflected in averages.

Bitbrains In the case of Bitbrains, the waiting times and slowdowns are the same for different reservation ratios. In other words, a higher or lower percentage of reservation requests does not affect the waiting time or slowdowns. Similarly, the average slowdowns are practically the same with different reservation ratios. However, there is a clear trend regarding utilization, waiting times, and slowdowns. The higher the utilization, the higher the proportion of waiting times and slowdowns.

Azure Regarding Azure, in the waiting times, we see that in the 75% utilization, the proportions are the same for different reservation proportions. In the 80% utilization, when using reservations, the 50th percentile times increase around 2.5 hours (500%). However, when the utilization is 85%, lower times are obtained when reservations are used. In 85% utilization, times are reduced by 35 hours (43%), compared to 0% reservations. Similarly, the 60th percentile slowdown is increased 150% on 80% utilization, but the 50th percentile is reduced 70% on 85% utilization when using reservations API.

Moreover, there is a clear trend in all configurations where waiting time and slowdown tails increase as utilization increases. This is expected since the higher the utilization, the higher the probability that there will not be an instance with enough available resources. Also, it is essential to remember that we do not oversubscribe the instances in this experiment.

All these differences in the ECDF are projected in the average slowdown of the requests. In the 75% and 80% utilizations, the differences are not significant enough to be appreciated. However, at 85% utilization, as we see, the differences reach 25%, so they affect average slowdowns. When there are 0% reservation requests, the average slowdown is around 250; when using reservations, the slowdown drops 60 points, with 50% and 100% reservations, respectively.

Google . Finally, in the results of the Google trace, we see two clear trends in the slowdowns and waiting times for all configurations. The first is that the greater the utilization, the greater the waiting time of the requests, both at the beginning and in the tail of the requests. The second is that the ECDF results are the same at the beginning, but the use of reservations improves the results in the tail. In the 85th, 90th and 95th percentiles,

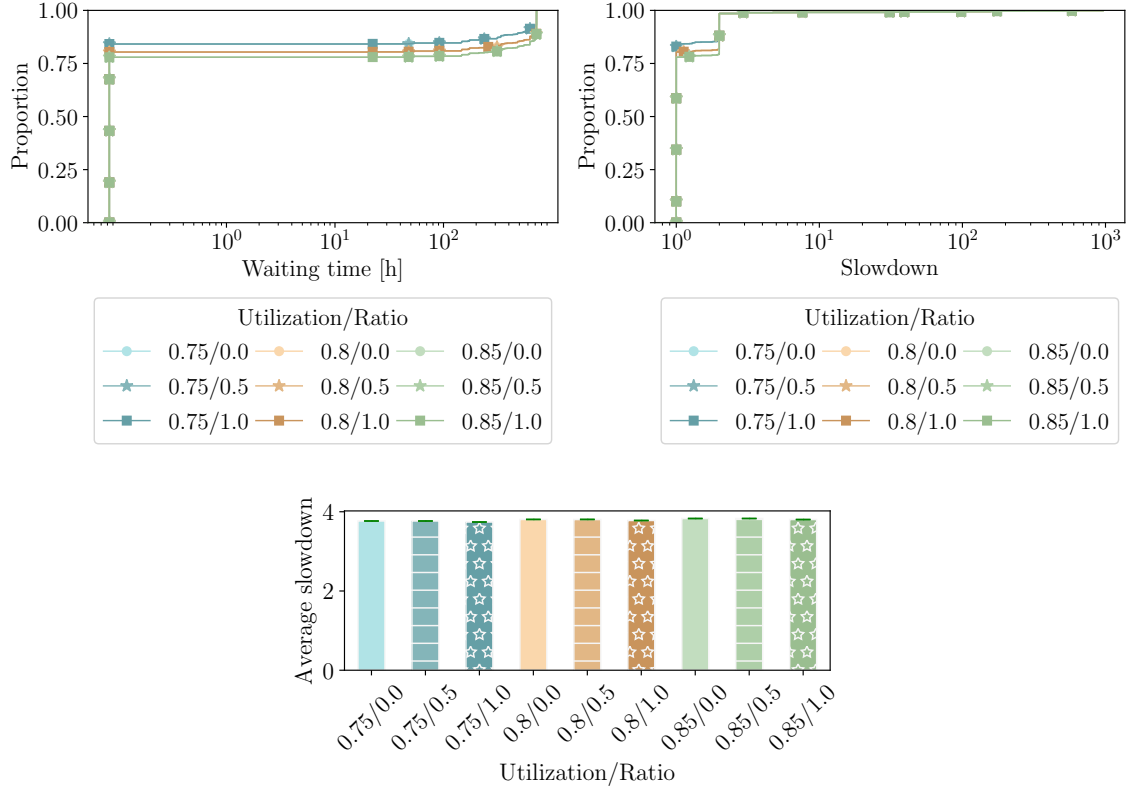


Figure 5: Waiting times (upper left) and slowdown (upper right) ECDF, and average slowdown (lower middle) of Bitbrains trace. Each line and bar represents a different `<Utilization>/<Reservation ratio>` configuration.

the 1 reservation ratio reduces the slowdown by 23% 38% and 53%, in 75%, 80% and 85% utilization, respectively.

When the reservation ratio is 0%, with the utilization of 75%, the average slowdown is around 1.7; with 80%, it is 2.3, and with 85%, it is 2.8. When going to a 50% reservation ratio, the results are practically the same. But with a 100% reservation ratio, the results improve by reducing the average slowdowns by 15%, 13% and 13%, respectively.

5.5.9 Discussion

Our main findings from this experiment are:

- MF1.1** In all traces except for Bitbrains, the slowdowns and waiting times are reduced by increasing the number of reservation requests.
- MF1.2** When the user does not know when she will need to provision resources or the expected runtime, the reservation system does not improve performance. Therefore, it is necessary to offer the reservations as programming abstractions since automating the reservation logic in the scheduler for every request is impossible.

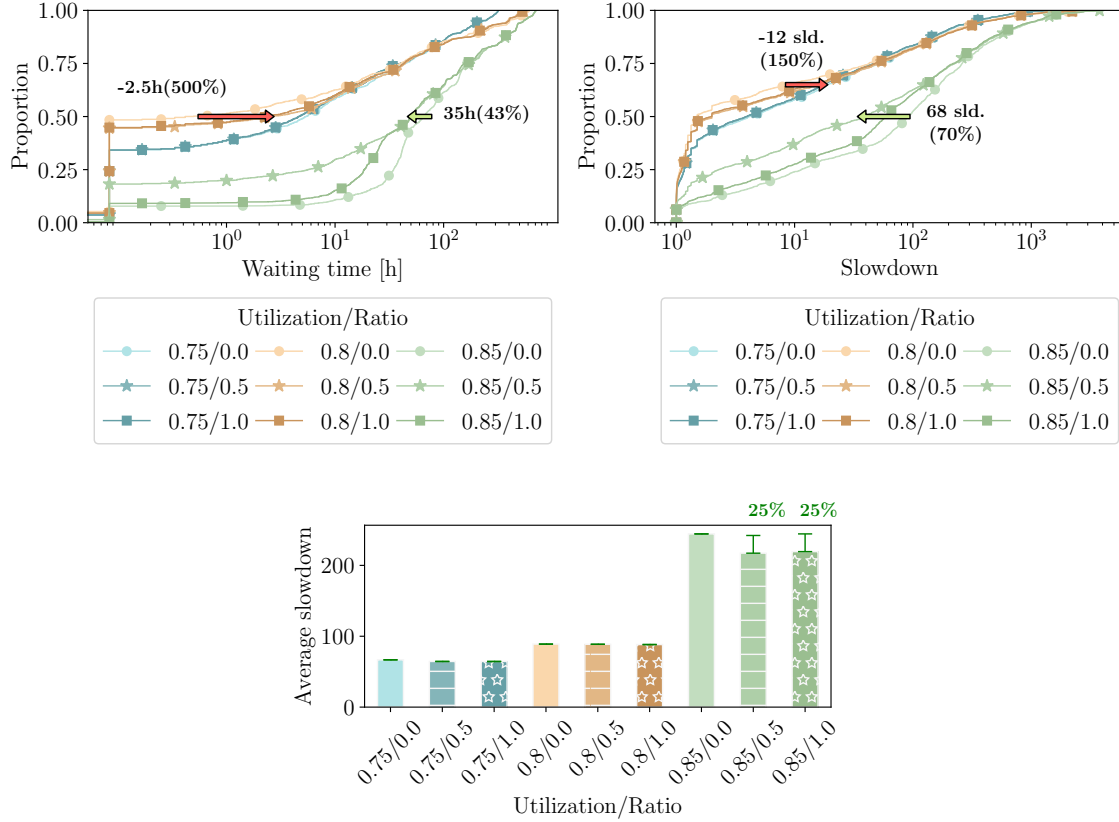


Figure 6: Waiting times (upper left) and slowdown (upper right) ECDF, and average slowdown (lower middle) of Azure trace. Each line and bar represents a different $\langle \text{Utilization} \rangle / \langle \text{Reservation ratio} \rangle$ configuration.

MF1.4 When implementing reservation programmability, it is necessary to consider the expected workload characteristics, the algorithm for optimizing reservations, and the interplay between them.

MF1.3 The lack of performance improvement differences in Bitbrains is because most requests run for the entire duration of the experiments.

This experiment aims to demonstrate that schedulers may sacrifice performance in exchange for simplicity if they do not offer reservation programmability to their users. The main takeaway from the results of this experiment is that in all configurations, except for Bitbrains, performance increases in terms of lower waiting times and slowdowns. This improvement is achieved by providing reservation programmability to the users. Reservations reduce slowdown by as much as 70% for the Azure trace, and 53% for Google trace, but not as much for Bitbrains. Moreover, the results are dependent on the durations of the tasks in the trace.

All these performance improvements are achieved using EFT, effortless and naive scheduling heuristic. However, several studies improve performance by implementing more com-

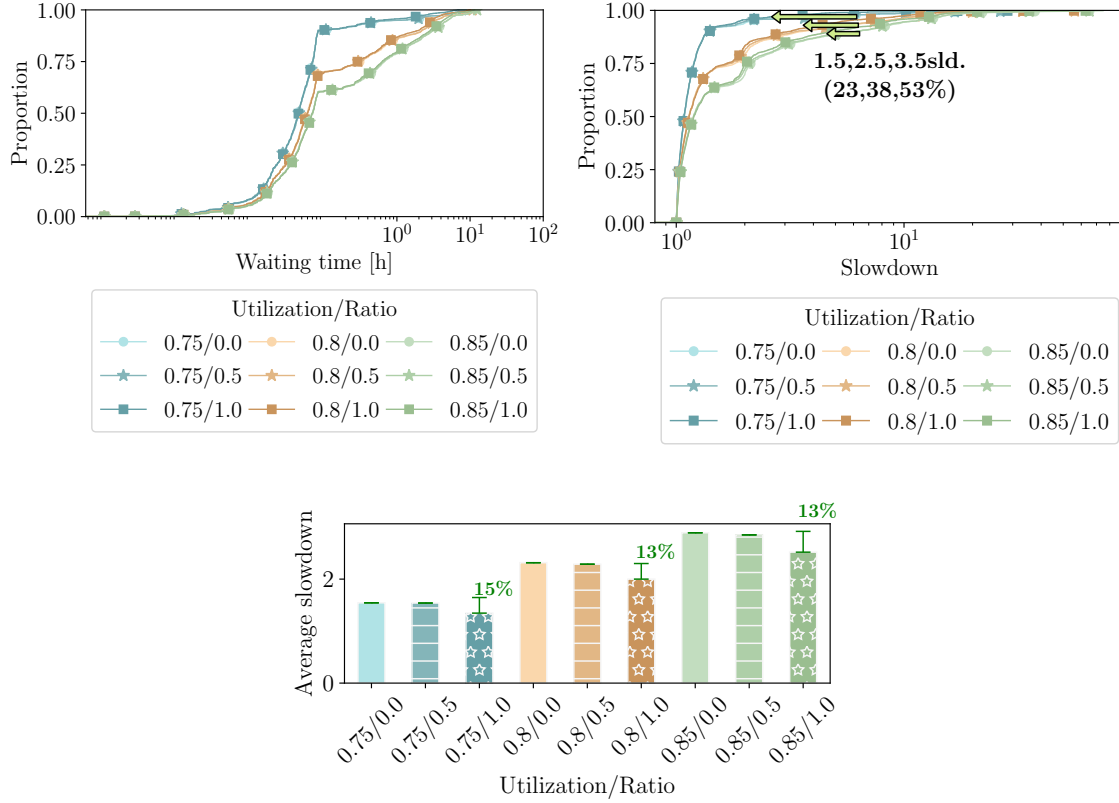


Figure 7: Waiting times (upper left) and slowdown (upper right) ECDF, and average slowdown (lower middle) of Google trace. Each line and bar represents a different $\langle \text{Utilization} \rangle / \langle \text{Reservation ratio} \rangle$ configuration.

plex algorithms such as MILP [43].

Furthermore, this experiment demonstrates that using reservations improves performance and that offering it as a programming abstraction is necessary. The user does not always know a VM’s start time and expected running time in advance. Therefore, the scheduler cannot internally implement the reservation logic without exposing programmability to the user. In our experiments, reservation ratios of 0 and 0.5 simulate this type of neutral case.

The differences in performance improvement between the traces are due to different factors related to the implemented EFT heuristic and the characteristics of each trace. However, Bitbrains does not show any significant performance improvement at all. This is because the start times are much less spread out than in the other traces. Most requests are collapsed at the beginning of the trace, and the VM runtimes are the same. This is because, in the original trace, most traces are already running from the beginning and run for the entire duration. Therefore, the benefits of the EFT heuristic are reduced significantly. So, this exemplifies that the schedulers must take into account both the characteristics of the workload, as well as the algorithm we use to optimize reservations when offering reservation

programmability to users.

5.6 Extension 2: Reducing VM total times using container migrations

In the second experiment, we evaluate the use case where the scheduler wants to reduce oversubscription and interferences between tenants using container migrations. Still, some of the mapped programming abstractions of industrial schedulers cannot implement it. Therefore, in this experiment, we want to demonstrate the benefits obtained in datacenter scheduling by leveraging container migrations. Users underutilize resources, and consequently, datacenter providers oversubscribe resources. This allows them to obtain greater utilization of the resources in the datacenter. However, this can lead to a resource being oversubscribed and users experiencing interference with each other. Moreover, many other phenomena cause interferences despite not being oversubscribed. To reduce interference in those cases, datacenter providers could migrate VMs or let the interference continue. However, another alternative is to let tenants know about the oversubscription and interferences so that they can take action to reduce the workload running on those VMs. For example, if users run a K8s cluster on top of their provisioned VMs, they could migrate out pods from oversubscribed VMs. Since pods are smaller than VMs, the migrations will be more efficient, as well as the packing. Moreover, the user has the business-logic context to make the best decisions on reducing the workload, such as what pods should be migrated or stopped.

In the analysis of the mainstream schedulers, we see that none of the schedulers let users know about the oversubscription and interferences. Mainly, the information they offer is about the resource consumption of the VMs, not the underlying physical resources. So, they do not have a way of knowing whether their VMs are oversubscribed or low-performing. Therefore, in this experiment, we implement a scheduling extension where users are notified when their VMs are oversubscribed, and consequently, they perform container migrations. This way, we show that in some scenarios, better scheduling performance is achieved by offering users programmability that allows them to get callbacks to implement container migrations.

5.6.1 Requirements

Before designing the experiment and the API extension, we identified specific functional requirements for the experiment, and we present them below:

FR Enable expression of oversubscription on datacenter scheduler APIs

The main requirement of this experiment is to have datacenter schedulers express oversubscription to users. Our hypothesis is based on the fact that current datacenters abstract users from oversubscriptions. This offers greater simplicity but sacrifices performance improvements that can only be obtained with the user's collaboration since users have the necessary knowledge about workload and business, which allows for taking optimal actions.

FR Enable container migrations.

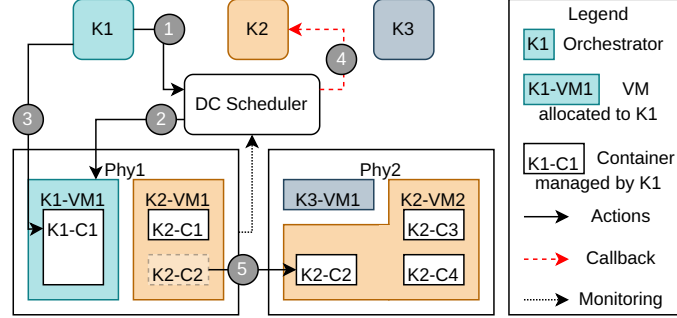


Figure 8: Container migrations experiment system model.

Specifically, with this experiment, we enable container migrations. container migrations are performed for tasks running inside provisioned virtual machines rather than the virtual machines themselves. In this way, we hope that the migration capacity and the packing of tasks will be greater when there are oversubscriptions in the datacenter.

FR Provide security and privacy.

By enabling oversubscription expression in the datacenter, the provider offers information about the underlying resources and possible aggregated data about other tenants. This has the potential to generate security breaches over. Therefore, the API must be able to enable the oversubscription expression while not offering compromised information about the underlying resources.

5.6.2 System model

To conduct this experiment, we will simulate the oversubscription scenario within a datacenter environment. In the datacenter, multiple tenants engage in leasing and releasing virtual machines, as illustrated by ① and ② in Figure 8. Each tenant establishes a Kubernetes cluster (**K1**) on the leased virtual machines, deploying multiple batch tasks within the Kubernetes environment. The Kubernetes cluster consists of several nodes (**K1-VM1**) representing virtual machines, where application containers are launched (③). The number of nodes varies across clusters, reflecting the unique workload requirements of each tenant. This Kubernetes layer operates as a secondary scheduling layer atop the datacenter scheduling, with tenants provisioning virtual machines and deploying Kubernetes nodes to establish their clusters. Once the task load subsides, the virtual machines are released, and the Kubernetes cluster is dismantled.

However, the utilization of resources within Kubernetes clusters can sometimes be sub-optimal, leading to underutilization. To address this, the datacenter provider resorts to resource oversubscription in order to maximize performance. Consequently, the aggregate virtual resource allocation on a physical machine may exceed the underlying physical resources. When a Kubernetes cluster experiences high load under such conditions, interference and reduced performance may occur among tenants sharing the same physical machine. In such cases, the datacenter scheduler may attempt to migrate virtual machines

to alternative physical machines, aiming to mitigate interference and improve overall performance.

The datacenter provides users with a simplified API, consisting of the following operations:

- **lease(requirements): vm:** Users specify their resource requirements, and the provider provisions a virtual machine that meets those specifications. The API call returns the allocated virtual machine to the user. Resource requirements typically include CPU cores, CPU capacity, and memory.
- **release(vm):** Users release a previously leased virtual machine by passing it as a parameter to the API call, and the provider shuts down the specified machine.

5.6.3 Model extension

In this experiment, we expand on the model described in the previous section by introducing the capability for the datacenter scheduler (④) to initiate callbacks to the user when the underlying resources are oversubscribed. The user includes a callback function named **requestUserMigration**, which the scheduler invokes. The callback function receives a target virtual machine and the amount of CPU capacity that is oversubscribed as arguments. In response, the orchestrator migrates (⑤) selected containers (specifically pods in this case) and provides the amount of CPU capacity to be reduced through container migrations. Migration has a cost proportional to the size of the VM migrated [11], so this approach aims to improve task performance by reducing the migration size, as pods are smaller than nodes. Consequently, virtual machines achieve better resource packing, leading to reduced interference and improved performance. In Listing 5.6.3, we provide an example of the extension, showcasing the syntax for migrations:

```
Communicate
CommunicationProcess<type:callback,>
    <url:orchestratorhost/callback>
IN UserResource<type:app, id:1>
WHEN Event<interference:10%>
```

The extended programming model offered by the scheduler includes the following functions:

- **communicate(callback, event):** This function allows the user to specify a **callback** and an associated **event** that triggers the callback, which in this case is a migration event. The scheduler receives and stores the callback and associated event, ready to be executed when oversubscription occurs and user migrations need to be performed.
- **requestUserMigration(vm, cpuCapacity) migratedCpuCapacity:** In the user-submitted callback function, the datacenter scheduler provides the oversubscribed virtual machine (**vm**) and the amount of CPU capacity (**cpuCapacity**) that needs to be migrated. The user performs necessary calculations and returns the amount of CPU capacity from the requested migration that will be migrated.

5.6.4 Alternatives

Prior to finalizing the implementation of the migrations API, we explored various alternatives to fulfill the system extension requirements. Next, we will provide a brief overview of the alternative approaches considered and present the rationale behind our chosen design.

Transparent utilization. The simplest alternative is for the datacenter provider to offer live metrics of the underlying physical machines. Users can view the utilization of the physical machine when provisioning a virtual machine, allowing them to determine if the machine is oversubscribed and if there are interferences with other tenants. This approach enables tenants to perform container migrations when they detect oversubscription or implement optimizations as resource usage grows. However, this model raises privacy and security concerns since users gain direct access to information about the underlying resources and can take actions that may conflict with the interests of the datacenter provider.

Oversubscription notification. Another alternative to prevent users from accessing live metrics of the physical machine directly is to replace them with notifications. The datacenter provider sends notifications to users when the underlying physical machine is oversubscribed or experiencing interference. Users receive information only when there is an oversubscription, prompting them to take necessary actions. However, this alternative can make coordination between tenants challenging, similar to the transparent utilization approach. Each tenant independently decides whether to perform container migrations, which may lead to scenarios where all tenants migrate simultaneously or none of them migrate while expecting others to migrate.

Oversubscription callback. The third alternative offers a balance between security and coordination among users. Instead of receiving notifications, users provide a callback function to the provider. This callback function receives the amount of CPU capacity that the provider requests the user to migrate, and in response, the user specifies the amount of CPU capacity that will be migrated. This bidirectional communication between the user and the datacenter allows the provider to act as a coordinator among the users. Implementing this alternative requires more complexity from the user’s perspective, but we believe that the potential performance gains outweigh the added complexity. Therefore, we have chosen to implement this third alternative as the model extension for the experiment.

5.6.5 Industrial schedulers

In this section, we explain how the industrial schedulers that we identified in Section 3 and 4 would implement the abstraction that we evaluated in this experiment and if they do not have it in their API, how they would implement it.

Kubernetes users submit callbacks for each of the container lifecycle events they are interested in, allowing the execution of a callback when the pod changes its state. Next, we show how the user would specify the migration callback of her application.

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: lifecycle-demo
5  spec:
6    containers:
7    - name: example
8      image: example:1.28
9      lifecycle:
10       migration:
11         exec:
12           command: ["/tmp/migration.prog"]

```

In this example, a callback is assigned via the *exec.command* parameter. In this specific case, it is specified that the callback is executed when the scheduler requires a migration. After the container has started, use *postStart*. Although Kubernetes provides an abstraction for adding callbacks, there is currently no built-in lifecycle state specifically for migrations. Therefore, Kubernetes would need to extend its API to include a migration state and support container migrations effectively.

SLURM offers the ability to add callbacks to specific job events using the **strigger** command. Below we present how the user would specify the migration callback of her application.

```

strigger --set --jobid=1234
         --migration --program=/tmp/migration.prog

```

In this example, the program */tmp/migration.prog* is executed when the scheduler requires migrations by specifying the *-migration* flag. So, SLURM already provides an abstraction for adding callbacks but does not include a migration state to activate the callbacks. Therefore, SLURM would require an API extension to support container migrations.

Spark also provides the ability to add callbacks, which can be defined in code and set through a CLI flag. Here is a specific example:

```

./bin/spark-submit --class org.apache.spark.examples.SparkPi
  --master spark://207.184.161.138:7077
  --conf spark.extraListeners=listener.MigrationListener
  /path/to/examples.jar

```

In this example, a configuration flag *spark.extraListeners* is set to the value *listener*. *MigrationListener* specifies that a custom-made callback is passed to receive up-calls when migrations are required from the scheduler. The callback is defined in a code file named *listener*, and inside it, the listener is implemented as an object named *MigrationListener*. In Spark, there is already the abstraction to add callbacks as listeners. However, the

current Spark API lacks a method specifically for receiving migration requests. To enable container migrations, Spark would need to extend its API by adding an interface method for receiving migration requests and implementing the necessary migration logic.

Condor is the only industrial scheduler among the ones analyzed that does not support callbacks. Consequently, it cannot provide container migrations based on the proposed alternative extension. Next, we show how the scheduler could implement callbacks in its API.

```
1 executable      = example
2 arguments      = SomeArgument
3 callback.migration = /tmp/callback.prog
4
5 queue
```

In this example, the callback is implemented in Condor by adding a new parameter to the submission file *callback.migration*, which is set to */tmp/callback.prog* that specifies where the program that performs the container migrations is located. The implementation of callbacks in Condor can take different forms, including using a binary or defining the callback code directly in the submission file.

Airflow allows users to assign callbacks to tasks, triggering their execution when specific events occur, such as failures. Next, we present how Spark could use callbacks to implement container migrations.

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator
5
6 def task_migration(context):
7     vm = context['vm']
8     cpu_capacity = context['cpu_capacity']
9     def example():
10         ret 'Hello world from Airflow DAG!'
11
12 dag = DAG('hello_world', description='Hello World DAG',
13         on_migration_callback=task_migration)
14
15 example_operator = PythonOperator(task_id='example_task',
16         python_callable=example, dag=dag)
17
18 example_operator
```

In this example, a function named *task_migration* is executed whenever the scheduler requires container migrations. A python function is defined and passed to the dag object through the *on_migration_callback* argument for setting the callback. While Airflow

already offers an abstraction for adding callbacks, it currently lacks a way to specify callbacks for migrations. To enable container migrations, Airflow would need to extend its API by introducing a new parameter, such as `on_migration_callback`, which would receive migration requests and facilitate the implementation of container migrations.

5.6.6 Configuration and design of the experiment

Our objective is to perform a series of experiments that highlight the limitations of a scheduler lacking container migrations programming abstractions. The configurations for this extension are summarized in Table 11. These experiment configurations encompass three key dimensions: trace, container migrations, and oversubscription ratio. In the following sections, we will delve into each dimension and discuss the various choices available, along with the metrics collected during the experiments.

Traces	container migrations	Oversubscription ratios
Bitbrains	true false	3.0 4.0 5.0
Azure	true false	3.0 4.0 5.0
Google	true false	3.0 4.0 5.0

Table 11: An overview of all the configurations of the oversubscription extension experiment.

In addition to the traces, container migrations, and oversubscription ratios, there is another dimension to consider: the utilization level and the number of Kubernetes clusters. For our experiments, we set these two dimensions to a fixed value of 85% utilization and 5 Kubernetes clusters. This means that we calculate the underlying physical topology of the physical machines based on the average CPU and memory usage observed in each trace. We then set the CPU capacity and CPU core amounts to 85% of their maximum values, reflecting the desired utilization level. This fixed configuration allows us to maintain consistency in the experiments and evaluate the impact of other dimensions, such as traces, container migrations, and oversubscription ratios, on the performance metrics.

Container Migrations and Utilization Ratio in addition to the trace analysis, we evaluate various combinations of container migrations and oversubscription ratios to gain a comprehensive understanding of their impact on system performance.

To ensure the experiment’s completeness and validity, we explore different oversubscription ratios. The oversubscription ratio represents the ratio between the number of virtual CPUs provisioned and the available physical CPUs. For instance, if the oversubscription ratio is set to 4.0 and there are 2 physical CPUs, the corresponding Kubernetes clusters will have $2 \times 4.0 = 8$ CPUs. We conduct experiments using three oversubscription ratios: 3.0, 4.0, and 5.0.

Furthermore, we examine the effects of container and datacenter migrations to simulate beneficiary and neutral scenarios. When container migrations are enabled, pods are migrated upon detecting oversubscription, while nodes/VMs are migrated when disabled. If interference persists despite pod migrations, the datacenter initiates subsequent node mi-

grations to mitigate performance degradation. By exploring different migration strategies, we gain insights into their respective impacts on system performance.

Metrics In this experiment, our objective is to enhance performance through migrations, as we anticipate that migrations will improve resource packing. To evaluate the effectiveness of migrations, we need to gather three types of metrics: time improvement metrics, resource packing metrics, and migration-related metrics.

The time improvement metrics enable us to analyze the performance enhancement in terms of execution time and waiting time. By examining these metrics, we can assess the impact of migrations on reducing task completion time and overall job waiting time.

The resource packing metrics allow us to understand whether the performance improvements are a result of better resource utilization and allocation. These metrics provide insights into how effectively resources are utilized and how well the system is able to pack tasks onto available resources.

Additionally, the migration-related metrics enable us to investigate the effectiveness of migrations in achieving better resource packing. These metrics help us analyze the number and frequency of migrations performed, as well as the impact of migrations on resource allocation and interference reduction.

While OpenDC provides a wide range of available metrics for analyzing simulation results, we have selected a subset of metrics specifically tailored to our experiment. The chosen metrics, presented in Table 12, provide us with the necessary data to conduct a comprehensive analysis of the experiment and evaluate the effectiveness of container migrations in improving performance.

5.6.7 Implementation of a Software Prototype

In this section, we provide an overview of the software prototype we developed for conducting the experiment. The prototype is built upon OpenDC, an open-source datacenter discrete event simulator created by AtLarge with several years of development and operation. To incorporate the necessary functionalities for our experiment, we extended OpenDC to support a second layer of scheduling, perform VM migrations, and execute container migrations within the second layer. The original code of the extension can be found at <https://github.com/aratz-lasa/opendc/tree/master/opendc-experiments/studying-apis/migrations>.

Second Layer of Scheduling To simulate a second layer of scheduling, similar to Kubernetes running on top of a datacenter, we made modifications and additions to various components of OpenDC.

Firstly, we expanded the internal representation of virtual machines (VMs) to include metadata about the Kubernetes pods running on them. This enhancement enables us to model second-layer tasks and perform scheduling across VMs. Subsequently, we extended the `ComputeService` component to manage this metadata effectively, allowing for

Name	Unit	Description
vm.id	-	Unique identifier of the VM
vm.provision time	Epoch (ms)	The instant at which the server was enqueued for the scheduler
vm.boot time	Epoch (ms)	The instant at which the server booted
vm.stop time	Epoch (ms)	The instant at which the server stopped
vm.timestamp	Epoch (ms)	The timestamp of the current VM metric entry
machine.id	-	Unique identifier of the physical machine of the datacenter
machine.cpu utilization	-	The CPU utilization of the machine
machine.cpu count	-	The number of logical processor cores available for this machine
machine.timestamp	Epoch (ms)	The timestamp of the current physical machine metric entry
migrations.success	-	The number of migrations that successfully reduced oversubscription
migrations.failure	-	The number of migrations that are not able to reduce oversubscription
migrations.improvement	-	The amount of CPU capacity that is causing interference is migrated
migrations.penalty	Time duration (ms)	The amount of penalty in task durations migrations cause

Table 12: The metrics that are recorded for the migrations extension evaluation.

operations such as adding, deleting, and moving second-layer pods. Furthermore, we implemented new schedulers that ensure pods are exclusively executed on nodes in their corresponding Kubernetes cluster. To achieve this, we extended the metadata and schedulers to consider resource consumption per Kubernetes node rather than per VM. Additionally, we developed a modified version of the `ComputeServiceHelper` component, which determines when tasks are submitted to the datacenter and when they are removed, thereby establishing the underlying physical topology.

Oversubscription Detection and Migrations The most complex part of our software development efforts was implementing the mechanisms for detecting oversubscription and performing migrations at both the VM and pod levels.

To detect oversubscription, we extended the metadata to track the tasks running on each machine and their respective CPU capacity consumption. We integrated this detection mechanism into the scheduling process, so that whenever a new task is scheduled, oversubscription is analyzed, and migrations are initiated if necessary.

For simulating migrations, we implemented the logic to suspend task execution on one VM and resume it on another. This involved stopping the work of a task and launching it on a new VM. We also developed a customized version of the `ComputeService` component, which incorporated all the necessary logic for executing both container and VM migrations.

The migration procedures for containers and VMs are similar.

To provide a clear understanding of the implementation details, Algorithm 2 presents the complete pseudocode for resolving any implementation-related doubts.

Secondary Optimizations and Bug Fixes In addition to implementing migrations, we introduced several optimizations and bug fixes to enhance the functionality and accuracy of OpenDC:

- We addressed a bug related to simulating interferences between virtual machines. We discovered that the rate was not being updated correctly, causing tasks to continue running at their initially requested rate. To rectify this, we made the necessary adjustments.
- We improved the implementation of the `ComputeServiceHelper` component. The default implementation in OpenDC launches tasks as VMs and waits for completion. However, instead of waiting for VM completion, we modified the component to calculate the task’s ideal runtime and stop the machine after that specific duration. As a result, we developed a custom `ComputeServiceHelper` that sets a listener on the machine and waits until it has finished executing the workload.
- We added new metrics to ensure accurate calculations of time and migration-related statistics. Although OpenDC already provides metrics for boot time and provisioning time, it

lacks a metric for recording stop time. Therefore, we introduced this metric. Additionally, for our experiment, we included additional metrics to measure the number of migrations performed and evaluate the resulting improvements or penalties.

5.6.8 Assumptions

When conducting experiments of this nature, it is essential to make certain assumptions to manage the experiment’s complexity effectively. Below, we outline the key assumptions made in this experiment:

- Each task is assigned a specific CPU count, capacity, and utilization percentage for the requested resources. However, it is assumed that the workload for each task remains constant throughout the entire execution.
- Migrations are associated with penalties to simulate the time required to migrate a VM from one machine to another. The penalty duration is determined based on the amount of requested memory, with an extension of 1 minute for every 4 GB of memory requested.
- The results of the experiments are limited to the steady state of the traces. This is necessary because the traces have a predefined time limit. If the graphs were plotted from the beginning to the end of the execution, there would be a tail at the end where tasks continue to be executed, but no new tasks are submitted. This tail does not accurately represent a real-world scenario, as tasks continuously arrive in an actual system. To address this, we focus on the steady state of the obtained results. The

Algorithm 2: Migrations simulation algorithm

```
1 Function Migrate(host, oversubscription):
2   if isPodMigrationsActivated then
3     migrated = migratePods(host, oversubscription)
4     oversubscription -= migrated
5   migrated = migrateNodes(host, oversubscription)
6   return oversubscription - migrated
7
8 Function migratePods(host, oversubscription):
9   migrated = 0
10  nodes = nodesSortedByCpuCount(host)
11  for node in nodes do
12    migrated += requestUserMigration(node, oversubscription-migrated)
13    if migrated  $\geq$  oversubscription then
14      return migrated
15  return migrated
16
17 Function requestUserMigration(node, oversubscription):
18   migrated = 0
19   pods = podsSortedByCpuCount(node)
20   for pod in pods do
21     nodes = getClusterNodes()
22     for node in sortByLessRemainingCpus(nodes) do
23       machine = nodeToMachine[node]
24       if !isOversubscribed(machine) then
25         migrated += migratePod(pod, node)
26         if migrated  $\geq$  oversubscription then
27           return migrated
28   return migrated
29
30 Function migrateNodes(host, oversubscription):
31   migrated = 0
32   nodes = nodesSortedByCpuCount(host)
33   for node in nodes do
34     machines = getCandidateMachines(node)
35     for machine in sortByLessRemainingCpus(machines) do
36       if !isOversubscribed(machine) then
37         migrated += migrateNode(node, machine)
38         if migrated  $\geq$  oversubscription then
39           return migrated
40   return migrated
41
```

steady state is defined as the time range between the submission of the first task and the submission of the last task, plus a delta. In this experiment, we set the delta as 5% of the average task duration.

5.6.9 Results

In Figures 9, 11, and 13, we present the results of the Bitbrains, Azure, and Google traces, respectively, for each combination of the oversubscription ratio and the activation (or deactivation) of the container migrations API. The purpose is to highlight the differences in scheduling performance between configurations that utilize the API and those that do not.

On the left side of the figures, we display the utilization of the physical machines within the datacenter. This metric represents the achieved resource packing efficiency for each experiment configuration. Higher utilization indicates better resource utilization and packing. On the right side, we present the empirical cumulative distribution function (ECDF) of the total time for each configuration. The total time encompasses both the waiting time and the execution time, providing an overall measure of task completion time.

Additionally, in Figures 10, 12, and 14, we provide data related to the migrations, aiming to understand their impact on the ECDF results. On the left side, we depict the cumulative number of successful migrations. This represents how many times, when facing oversubscription, the system was able to perform enough migrations to completely alleviate oversubscription. On the right side, we illustrate the cumulative penalty generated by the migrations, which reflects the additional time required for the execution of migrated tasks.

Bitbrains. Figure 9 illustrates the results for the Bitbrains trace. The packing graph demonstrates that configurations utilizing the API achieve better resource utilization, with up to a 4% higher utilization compared to configurations without the API. However, as time progresses, all configurations converge to similar utilization levels, indicating that the initial packing advantage diminishes over time.

The migrations graph in Figure 9 reveals that configurations utilizing the API experience a higher number of successful migrations, approximately 170,000 (93%) more migrations in total. However, it is important to note that these migrations come at a cost. The cumulative penalty associated with the migrations is also higher with the API, resulting in a total penalty difference of 41.6 hours (88%). The impact of these penalties may overshadow the benefits of the 3% higher resource packing achieved through container migrations, ultimately leading to limited improvements in task execution times.

Azure. Figure 10 presents the results for the Azure trace. The packing graph demonstrates that configurations utilizing the API achieve better resource packing. The difference in packing efficiency becomes more pronounced with higher oversubscription ratios. Specifically, at a ratio of 5.0, the API yields a 15% higher utilization, while at a ratio of 4.0, the difference is around 8%. However, at a ratio of 3.0, using the API results in worse packing compared to not using the API, with a difference of approximately 6%.

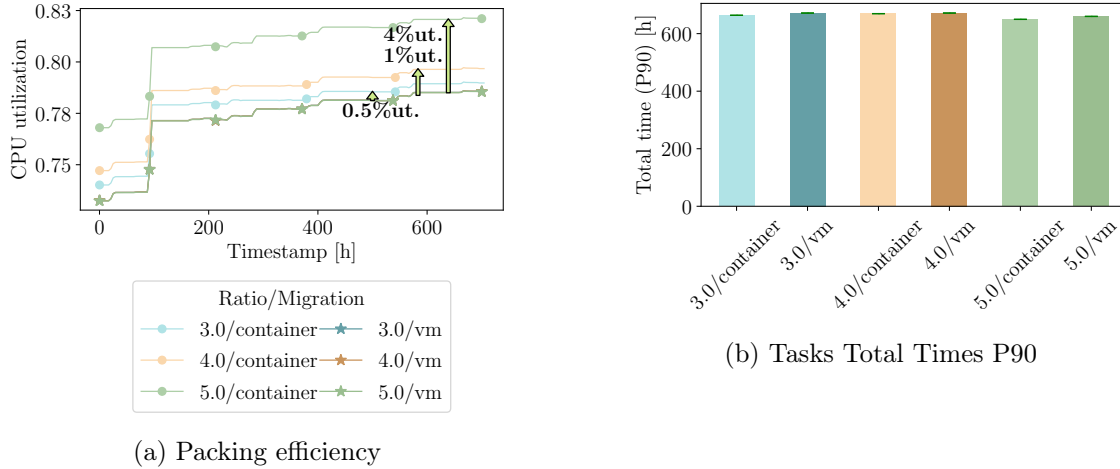


Figure 9: Packing efficiency (left) and tasks Total Times P90 (right) of Bitbrains trace. Each line and bar represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

Certainly! Here’s the modified code to align the subfigures in the second figure at the top:

Regarding task execution times, the API consistently yields shorter times, indicating higher performance. The greatest differences in performance are observed at the oversubscription ratios of 5.0 and 4.0. Using the API with a ratio of 5.0, the 90th percentile experiences a time reduction of 260 hours (81%), while at a ratio of 4.0, the reduction is 180 hours (13%). For a ratio of 3.0, the API results in a time reduction of around 45 hours (13%).

In Figure 12, we observe that using the API for migrations leads to a higher number of successful migrations. Specifically, for oversubscription ratios of 3, 4, and 5, the number of successful migrations increases by 400 (25%), 1300 (61%), and 1000 (23%) respectively. Additionally, there is a penalty reduction of 1.6 (25%) and 2.5 (25%) minutes for oversubscription ratios of 3 and 4, respectively, while an increase of 8.8 minutes is observed for an oversubscription ratio of 5.

Google. The packing graph in Figure 13 for the Google trace shows minimal differences among the different configurations, regardless of the oversubscription ratio and API usage. However, when utilizing the API, shorter task execution times and higher performance are observed. This can be attributed to the unique nature of the Google trace, where tasks are extremely small and utilize a single CPU core. Although the packing differences may not be noticeable due to the short task durations, significant improvements are seen in the 90th percentile total times. Notably, using the API with a ratio of 5.0 results in a 66% reduction (4 hours) in the 90th percentile times, while ratios of 3.0 and 4.0 achieve 21% (1 hour) and 3% (8 minutes) lower times, respectively.

In Figure 14, the use of the API leads to a significant increase in successful migrations, with approximately 35,000 more migrations compared to when the API is not used. Addi-

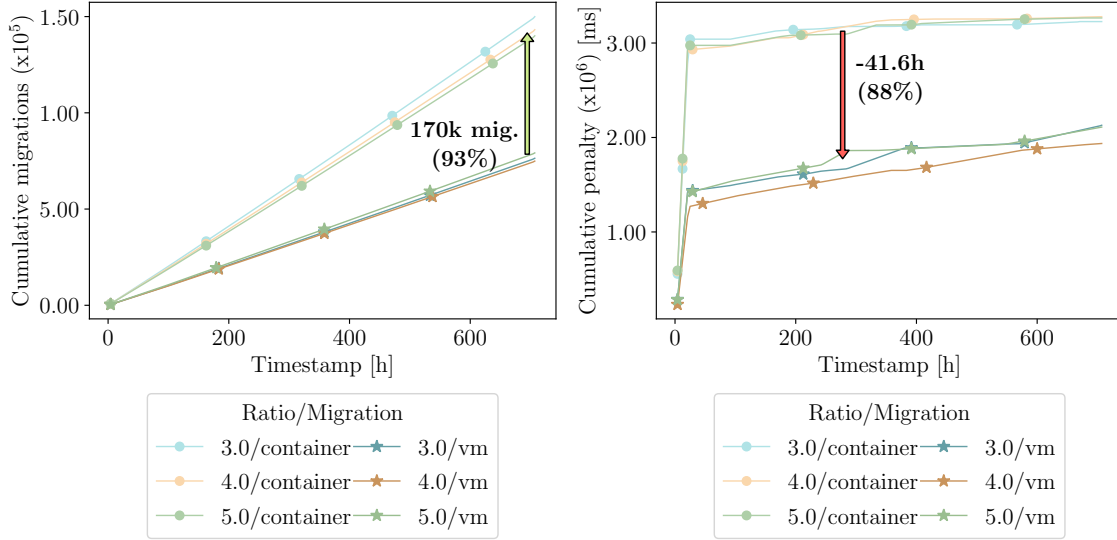


Figure 10: Migrations cumulative amount (left) and migrations cumulative penalty (right) of Bitbrains trace. Each line, bar, or boxplot represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

tionally, in this trace, no penalties are observed as the requested memory of the tasks are small enough to avoid any penalties.

5.6.10 Discussion

Our main findings from this experiment are:

- MF2.1** In all traces except for Bitbrains, the performance is improved by using the extended API for container migrations.
- MF2.2** The highest oversubscription ratio of 5.0 obtains the highest performance improvement using container migrations.
- MF2.3** The main benefit of migrations is greater packing, i.e., greater utilization of resources. However, when the tasks are very small, the benefits of migrations cannot be appreciated through packing.
- MF2.4** It is complex to explain performance improvements through migration metrics, and it is necessary to generate more metrics and deeper analysis to have a complete picture.

This experiment highlights the trade-off between simplicity and performance in schedulers that do not offer callbacks to users. The results demonstrate that providing user-level migrations as a programming abstraction significantly improves performance in terms of total times, except for the Bitbrains trace. The highest oversubscription ratio of 5.0 achieves the greatest performance improvement, up to 260 hours lower total times for the 90th percentile. However, there are cases where using the lowest oversubscription ratio of 3.0 with user-level migrations results in worse performance, indicating the need for further research

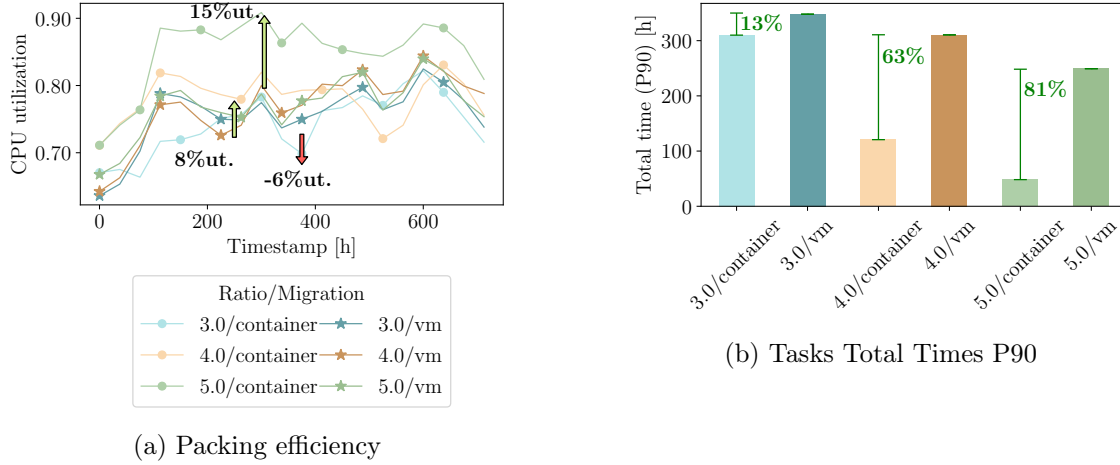


Figure 11: Packing efficiency (left) and tasks Total Times P90 (right) of Azure trace. Each line and bar represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

to understand the underlying causes.

Furthermore, in this experiment, we demonstrate that container migrations improve performance and that offering it as a programming abstraction is necessary. The user does not always have a second scheduling layer like a Kubernetes cluster. Moreover, the datacenter schedulers do not have the business logic knowledge to decide which tasks to migrate and where. Therefore, the scheduler cannot internally implement the container migration logic without exposing programmability to the user. Our experiments simulate these neutral cases when we deactivate the container migrations.

The use of parameter sweeps to evaluate the impact of different costs for migration could have provided additional insights into the performance behavior of the system. By systematically varying the migration costs, it would be possible to analyze how different cost settings affect the scheduler’s decision-making process and overall performance. This approach would have allowed for a more comprehensive understanding of the trade-offs involved and potentially revealed alternative configurations that optimize performance under specific cost scenarios. While not explored in this study, incorporating parameter sweeps for migration costs could be a valuable direction for future research.

Lastly, to understand how migrations affect performance, we also show graphs on the number of migrations and their penalties. This helps to understand how the Bitbrains trace, despite having a lot of migrations, does not show almost any improvement in performance because it also presents the highest penalties. However, it is not straightforward to understand how the penalties and migrations interplay. This creates an opportunity for future research to understand better how migrations affect task scheduling performance.

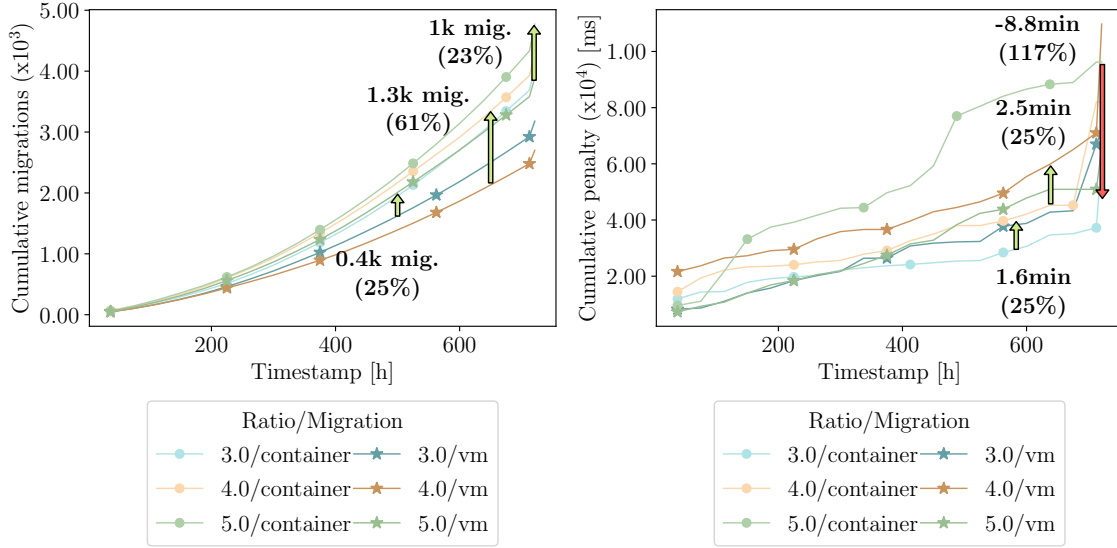


Figure 12: Migrations cumulative amount (left) and migrations cumulative penalty (right) of Azure trace. Each line, bar, or boxplot represents a different $\langle \text{Oversubscription ratio} \rangle / \langle \text{Migrations API} \rangle$ configuration.

5.7 Extension 3: Reducing Data Workflows execution times using meta-data access

In the last experiment, we will extend and evaluate a scheduler to access user metadata. Users store data in the datacenters, and later, their jobs use it. The devices where the data is stored may suffer from network congestion or high request load, so the response time varies over time, and there are moments when they are very high. In addition, the data placement is also different and can be more or less far from the virtualized VMs where the user executes the tasks and requests the data. There are scenarios where the user has to process various data items, but they do not require any specific order. In these cases, the user obtains data items from the scheduler in arbitrary order or solely with insights from the data itself, such as data size. Thus, it is likely that the data retrieval order is not optimal. If the data is spread across different storage devices, the user can request the data from a congested storage device while other data is not congested. Therefore, if the user has access to the metadata, she could obtain insights about where the data is stored and, consequently, optimize the order in which the data is obtained and processed.

However, the scheduler, through its APIs, hides all these insights from the user. None of the industrial schedulers we discussed in the previous sections offer the programming abstraction for `[access metadata]`. The only programming abstraction that all schedulers provide is the API to handle the data is access to the input data. Therefore, the user can request data stored in the datacenter, but she does not have access through metadata to the insights about congestion or placement that allow her to optimize her jobs. Therefore, in this experiment, we implemented an extension to schedulers that offers users metadata access, by which they can obtain the necessary insights to optimize the data access or-

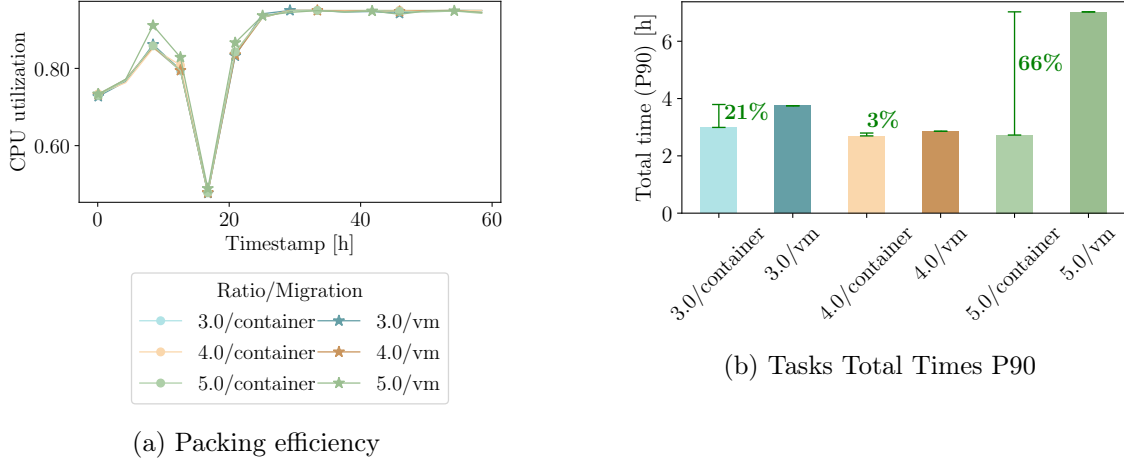


Figure 13: Packing efficiency (left) and tasks Total Times P90 (right) of Google trace. Each line and bar represents a different $\langle \text{Oversubscription ratio} \rangle / \langle \text{Migrations API} \rangle$ configuration.

der. In this way, we demonstrate that schedulers sacrifice significant performance for user applications in some scenarios by not exposing the abstraction of `access metadata`.

5.7.1 Requirements

Before designing the experiment and the API extension, we identify specific functional requirements for the experiment, and we present them below:

FR1 Enable object retrieval optimization from object storage

The main requirement of this experiment is to offer abstractions to the user to optimize object retrieval and obtain higher performance. In this way, when the object storage is congested, the order of the object retrieval is recalculated to avoid congestion.

FR Provide security and privacy.

By providing metadata about the data storage in the datacenter, the provider offers information about the underlying resources and possible aggregated data about other tenants. This has the potential to cause security breaches over. Therefore, the API must be able to enable metadata access while not offering compromised information about the underlying resources.

5.7.2 System model

For this experiment, we model users running jobs in a datacenter that processes data stored in object storage within the datacenter. Each tenant runs a multi-core application(`A1` in Figure 15). The tenants have pre-stored data items in object storage within the datacenter (`a1-o1`), and the application processes those data items. It dynamically assigns a data item to each available CPU core. When a core finishes processing the data item, the application

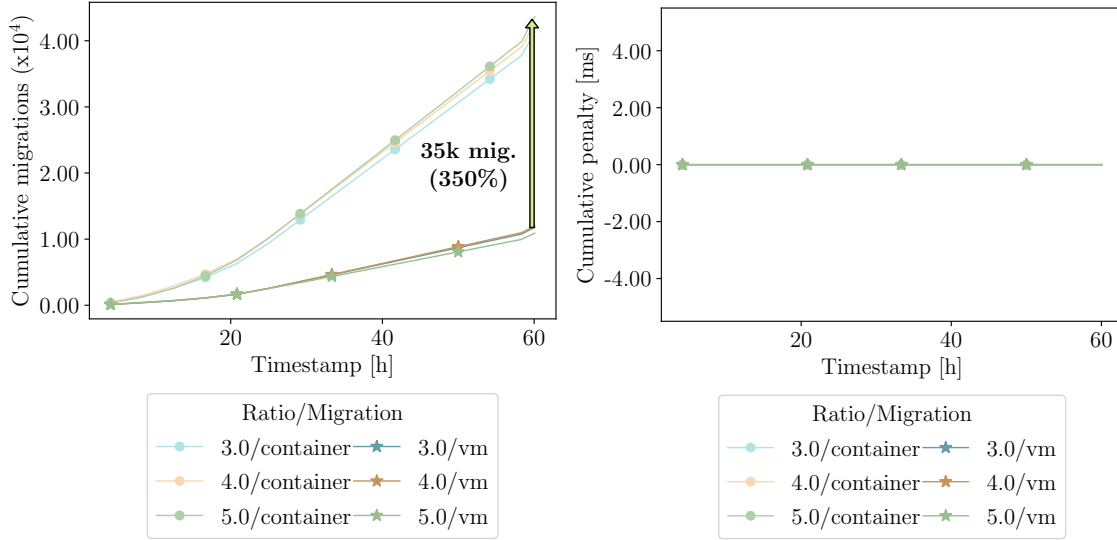


Figure 14: Migrations cumulative amount (left) and migrations cumulative penalty (right) of Google trace. Each line, bar, or boxplot represents a different $\langle \text{Oversubscription ratio} \rangle / \langle \text{Migrations API} \rangle$ configuration.

looks for a data item that has not been processed yet and assigns it to it. When the application assigns a data item to the CPU core, the core takes care of downloading the item from object storage and processing it (① and ②). The datacenter has a scheduler, the front end for object storage. So the user sends a request to the scheduler specifying an id that represents the data item he wants to obtain, and the scheduler is in charge of directing the request to the storage server that contains the object. The data items that the application processes do not present precedence between the different data items. Therefore, it can process them in any order without affecting the final result.

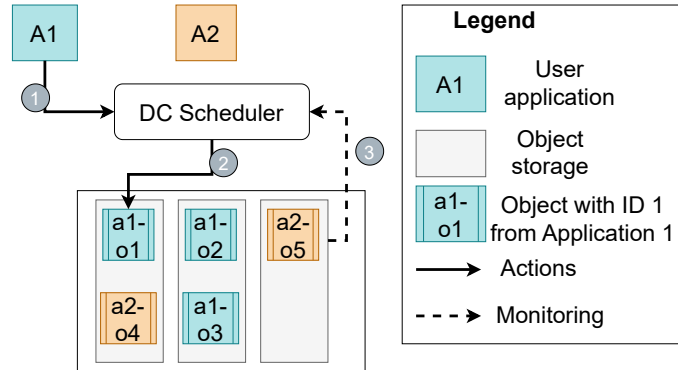


Figure 15: Optimized object retrieval experiment system model.

The object storage comprises several storage servers, and each server has several CPU cores that allow requests to be processed concurrently. Each server has several queues in case there are more concurrent requests than available cores. The server processes the

queues in FIFO order. When all CPU cores are busy, user requests go to queues, and users must wait for a CPU core to become free, and there are no requests from other tenants ahead of it in the queue. However, the user does not have access to the metadata that informs in which server the data items that need to be processed are stored, nor the load of the servers.

The API offered by the scheduler to access the object storage to users can be simplified as follows:

- `get(object-id)`: `object`: the user specifies the ID of the object he wants to download, and the scheduler sends the object back.

5.7.3 Model extension

In this experiment, we extend the model system presented so that the user application queries the metadata of the objects to decide which objects to retrieve next. In this way, if an object has a high estimated retrieval time, its processing is delayed and thus increases the chances of reducing congestion and, consequently, the retrieval time. The scheduler retrieves this information by monitoring the storage servers (③). In the initial model, the user does not know if the requested object is stored on an overloaded storage server, while other objects that it will process later may not be stored on congested servers. Compared to the initial model, the user does not retrieve objects in arbitrary order since, through the extension, she can obtain insights that help her get rid of the inefficiencies in object access, which lead to increased latency, reduced throughput, and decreased overall system performance [31, 47].

The extension is based on the designed reference architecture. The scheduler offers a new `accessMetadata` action that receives a single parameter that is the object id, which is a `userResource` object and allows to specify the object from which the metadata is obtained using the ID. In Listing 5.7.3, we provide an example of the extension, showcasing the syntax for metadata access:

```
ManageData: AccessMetadata
UserResource<type:object , id:2>
IN SchedulerResource<type:
    object-storage>
WHEN Event<datetime: now>
```

The extended programming model offered by the scheduler is the following:

- `accessMetadata(objectId)`: `[]metadata`. The user specifies an `objectId` which is a `userResource`. The scheduler internally calculates the metadata of the specified object and returns a list of metadata. The scheduler can also have the metadata computed beforehand and update it in the background.

5.7.4 Alternatives

Prior to finalizing the implementation of the metadata access API, we explored various alternatives to fulfill the system extension requirements. Next, we will briefly explain the

alternatives and argue our chosen design.

Sorter: The simplest way to optimize object retrieval is to provide a `sort([]object-ids): []object-ids` programming abstraction, which sorts a list of object ids according to their estimated retrieval time. In this way, each time a CPU core is released, the user application calls the `sort` method passing the remaining objects and assigning the objects with the lowest retrieval times to the free CPU cores.

Iterator: A second alternative is to expose a `iter([]object-id) objectIterator` action. The `iter` action receives a list of object ids. It returns an iterator over the list of the corresponding objects, sorted by the expected retrieval time and in increasing order. The iterator exposes an action `next(n)` that receives an integer `n`, and the scheduler sends back the next `n` object id.

Metadata access: The third alternative is an action that allows the user to get metadata related to a data item, which contains the estimated retrieval time or congestion level for a specific object id.

We decide to implement the third alternative for two main reasons. On the one hand, it does not need to maintain a different internal state, as with the iterator, where the scheduler needs to hold the list of remaining objects. On the other hand, the first sorter alternative abstracts the user from how the sorting order is calculated. At the same time, the API for exposing the metadata gives the user control over how the data item is sorted. It is important to note that the third alternative is the most insecure since it can inadvertently leak internal implementation details. However, we argue that the scheduler must be responsible and capable of presenting the metadata in a way that is not subject to a security breach. For example, to expose congestion, the scheduler could create a way to rank different congestion levels, like a five star-system.

5.7.5 Industrial schedulers

In this section, we explain how the industrial schedulers that we identified in Section 3 and 4 would implement the abstraction that we evaluated in this experiment and if they do not have it in their API, how they would implement it.

Kubernetes does not offer an API for the object storage service or for accessing the metadata. However, Kubernetes is developing an API for object storage management ¹⁶. In this API, they provide an abstraction named *BucketInfo* to query the metadata. Below we show the API that Kubernetes offers its users and how they would receive the metadata about the expected retrieval time.

`DriverGetInfoRequest(bucketID)`

In this example, Kubernetes exposes a function that users can call via HTTP request specifying the object ID (bucket ID), and Kubernetes will respond with metadata about it. The metadata will include the expected retrieval time, calculated based on internal congestion metrics.

¹⁶<https://github.com/kubernetes/enhancements/tree/master/keps/sig-storage/1979-object-storage-support>

SLURM does not offer an API for the object storage service or an API for accessing object storage service metadata. Therefore, just like in Kubernetes, SLURM should create a complete API to offer object storage management. In this API, SLURM should offer a command to query the metadata. Next, we show how SLURM could offer metadata access.

```
smetadata -id 1
```

In this example, we create a new command *smetadata* for accessing metadata, and the user specifies the object id through the *-id* parameter. SLURM responds to this command with the object id metadata, including the expected retrieval time, based on internal congestion metrics.

Spark does not offer an API to query the metadata. To do this, Spark should create a new method or function on the objects it offers programmatically. Below we show how Spark could implement the metadata access API to optimize object access.

```
1 from pyspark.sql import SparkSession
2 spark = SparkSession.builder.getOrCreate()
3
4 meta = spark.read.metadata('object://1')
5 retrieval_time = meta["expected_retrieval_time"]
```

In this example, we create a new function *spark.read.metadata* that is used for accessing metadata. In this case, we read an object with id 1 by specifying *object://1*. Spark returns a dictionary containing metadata, including the expected retrieval time, accessed through *expected_retrieval_time*.

Condor does not offer an API for the object storage service or an API for accessing object storage service metadata. Therefore, like Kubernetes, Condor should create a complete API for object storage management. In this API, Condor should offer a command to query the metadata. Next, we show how Condor could offer metadata access.

```
condor_metadata -id 1
```

In this example, we create a new command *condor_metadata* for accessing metadata, and the user specifies the object id through the *-id* parameter. Based on internal congestion metrics, Condor responds to this command with the object id metadata, including the expected retrieval time.

Airflow Airflow offers APIs to access different object storage services such as AWS S3. It offers several functions on the object storage service, such as downloading, deleting, or creating. However, it does not offer functions to query the metadata of the objects. Therefore, Airflow must extend the API to offer metadata access API. Below we present how Airflow could implement this extension.

```
1 from airflow.operators.amazon import S3MetadataObjectsOperator
2
```

Traces	Metadata access	Utilization
Google + IBM objects	true false	80%

Table 13: An overview of all the metadata access extension experiment configurations.

```

3 metadata_objects = S3MetadataObjectsOperator(
4     task_id="get_metadata",
5     keys=[1, 2, 3],
6 )
7 retrieval_time_1 = metadata_objects[1]["expected_retrieval_time"]
8
9
10 metadata_objects

```

In this example, a new Airflow operator (task) is created named *S3MetadataObjectsOperator*, which is responsible for accessing the metadata of a set of objects. The user specifies the group of objects by passing a list of ids to the *keys* parameter. Airflow returns a dictionary containing the metadata for each of the objects. Later the user can access the expected retrieval time by accessing the corresponding object through *metadata_objects[1]["expected_retrieval_time"]*.

5.7.6 Configuration and design of the experiment

We aim to conduct experiments that prove the limitations of a scheduler that does not provide metadata access programming abstractions. Table 13 summarizes this extension’s configurations. In this experiment, the configurations are composed of a combination of a single dimension: metadata access activated/deactivated. All the other dimensions are fixed: the traces and the resource utilization of the object storage service. Below, each dimension and the different choices are explained, in addition to the metrics collected in the experiments.

Object storage traces and utilization The traces in this experiment combine Google traces and IBM’s object storage requests [15]. In our model system, user applications are data workflows; therefore, Azure and Bitbrains VM traces are not realistic or applicable. The Google trace comprises single-core tasks, which are part of workflows, making it suitable for this experiment. However, in this experiment, we evaluate data workflows that download objects from an object storage service, combining the Google object trace with an IBM object requests trace. The method we combine is extracting the GET requests from the IBM trace and assigning each request to a Google task so that the IBM trace object distribution is respected and implemented on top of the Google trace.

Regarding the utilization of resources, we set it to 80%. However, this figure is based on utilizing object storage service resources, not user tasks. The performance we expect to get is based on more optimal use of the object storage service. We set the utilization at 80%. However, despite having set it at 80%, due to the traces’ nature, it is impossible to maintain it constantly because it fluctuates from a maximum of 90% to a minimum of

50%.

The IBM object requests trace can be found at <http://iotta.snia.org/traces/key-value>.

Metrics In this experiment, we seek to improve performance through metadata access since we expect metadata insights about objects to allow us to optimize object retrievals. Therefore, it is necessary to obtain two types of metrics: workload time improvement metrics and object storage internal metrics. The time metrics allow us to analyze the performance improvement regarding the execution time of the data workflows. The object storage service internal metrics allow us to understand if the improvements come from optimized object requests, such as having less congestion, due to a higher balanced load between the different object storage servers. The metrics we collect for the analysis of the experiment are presented in Table 14.

Name	Unit	Description
vm.id	-	Unique identifier of the VM
vm.provision time	Epoch (ms)	The instant at which the server was enqueued for the scheduler
vm.boot time	Epoch (ms)	The instant at which the server booted
vm.stop time	Epoch (ms)	The instant at which the server stopped
vm.timestamp	Epoch (ms)	The timestamp of the current VM metric entry
machine.id	-	Unique identifier of the physical machine of the datacenter
machine.cpu utilization	-	The CPU utilization of the machine
machine.cpu count	-	The number of logical processor cores available for this machine
machine.timestamp	Epoch (ms)	The timestamp of the current physical machine metric entry
storage.server_id	-	Unique identifier of the server machine of the object storage service
storage.cpu_count	-	The number of logical processor cores available for this object storage server
storage.buffer_size	Bytes	The number of bytes that are still to be sent/-downloaded, it is equivalent to the pending requests of object retrievals from the users
storage.idle_time	Long (ms)	The time that the servers have been without receiving or processing user requests, that is, the time that they have been without using CPU cycles

Table 14: The metrics that are recorded for the metadata access extension evaluation.

We implement all the metrics related to the object storage service since OpenDC does not implement the object storage service. Therefore, we implement the flow to collect and calculate the metrics and the logic to export them to parquet files.

5.7.7 Implementation of a Software Prototype

In this section, we explain the prototype we develop to experiment. To experiment, we use OpenDC, an open-source datacenter discrete event simulator developed by AtLarge, with multiple years of development and operation. In the experiment, we extend OpenDC to include the new capabilities to run an object storage service that exposes a metadata access API and a data workflow trace. Below we explain the changes we make. The original code of the extension is available in <https://github.com/aratz-lasa/opendc/tree/master/opendc-experiments/metadata>.

Object storage service To perform this experiment, the most important component we implement is the object storage service. This service is the one that provides users with external data storage, through which they can obtain the data during their workload runtime. The simulation of this component is not so trivial. It is necessary to simulate the internal servers and CPUs it uses since otherwise; it is impossible to perceive the congestion or the different performance that the users obtain consequently. Therefore, we simulate the object storage service and its different subcomponents in charge of processing and limiting user requests in parallel. In addition, together with the logical implementation of the object storage service, we also designed how the necessary metrics are generated to analyze the performance. We generate two main metrics: the time each object storage service server is idle and the size of the buffers. The idle time allows for analyzing the resource utilization of the object storage service, while the size of the buffers indicates how efficient the processing of user requests is.

Metadata access API After implementing the object storage server, we implement the API through which the user can query metadata about the objects he has stored. In this specific case, the user gets a numerical representation of the congestion of the server where the object is stored. To do this, each storage server keeps an internal account of the size of its buffers, that is, of the waiting line. And the metadata access API generates a numeric representation depending on the size of the waiting line. The user specifies the object ID, and the scheduler returns the associated metadata.

Data workflow trace Finally, to carry out the experiment, we design and implement the execution of the data workflow. To simulate the data workflow, it is necessary to build different blocks. First, we implement the code that loads the data workflow and transforms it to its OpenDC representation, but also the logic to assign each object a different server of the object storage service. Second, we implement the most complex part of executing the data workload. The execution comprises three parts: the parallel execution of the workflow tasks that represent the provisioned CPU cores, the metadata request and the consequent download of the object from the storage service, and finally, the processing or execution of the object. Apart from loading the workload and its execution, we also implemented the integration with the existing OpenDC system so that the execution of the workflows is done on top of the scheduling system and generates the metrics of its execution.

5.7.8 Assumptions

When performing the experiments, it is necessary to make several assumptions. Otherwise, the complexity of the experiment becomes unmanageable. Next, we list the main assumptions of this experiment.

- The datacenter has enough resources to provide all data workflows without interferences or waiting times.
- Users only make GET requests; consequently, no concurrent PUT requests can modify the data size or block its download.

5.7.9 Results

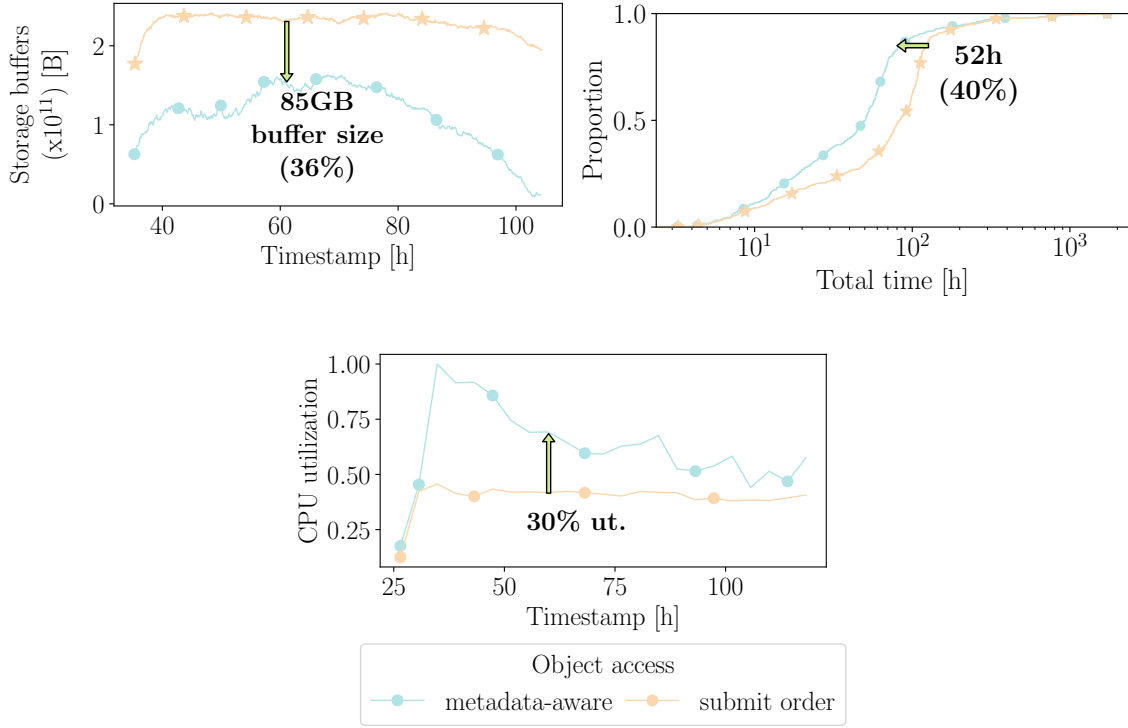


Figure 16: Comparison of buffer sizes in the object storage service (upper left), ECDF analysis of total execution times per workflow (upper right), and normalized resource utilization (lower middle) between the configuration with and without the metadata access API. This uses the Google Compute trace combined with the IBM object storage trace.

In Figure 16, we show the experiment’s results for activating and deactivating the metadata access API. The objective is to show the differences in scheduling performance between the configurations that use the API and those that do not. On the upper left, we present the internal buffer sizes of the object storage service, where the Y axis is the aggregated buffer sizes of the storage service in Bytes. This represents the object storage service’s waiting time and expected performance when a user requests an object download. On the

upper right, we show the ECDF of the execution time of the data workflows. The execution time is the time spent from the data workflow that started executing until it is finished. It excludes the waiting time for the provisioning of resources, because, as explained in the previous section, we assume the datacenter does not have enough resources not to have interferences or waiting times for provisioning. This assumption is made to reduce the complexity of the experiment and to be able to obtain clear conclusions.

The object storage service buffer sizes graph clearly shows that the API configuration obtains smaller buffer sizes, around 85 GB smaller buffers (36%). The difference in the size of the buffers remains relatively constant, and there are no significant spikes. Consequently, in the execution times ECDF, there is a parallel in the results. The API obtains shorter times, that is, higher performance. The greatest difference in ECDF performance is at the 90th percentile. Using the API, the users obtain 52h hours (40%) lower total times.

In addition, in the lower middle, we show the normalized resource utilization of the object storage service. The Y axis is obtained by calculating the percentage of time that the CPU cores of the object storage service are idle without processing user requests. The objective is to maintain the utilization at around 80%, and the graph shows how it is at a maximum of 100% and drops to 40%. Due to the nature of the traces, we cannot obtain regular use. However, when users use the metadata access API, the resource utilization is 30% higher, meaning the object storage has a higher efficient usage of resources.

5.7.10 Discussion

Our main findings from this experiment are:

- MF3.1** The performance is improved in execution time by using the extended API for metadata access.
- MF3.2** The main benefit of the metadata access API for the user is an optimized execution of the data workflow, where the processing ordering of the data objects takes into account the congestion level of the underlying infrastructure.
- MF3.3** The main benefit of the metadata access API for the datacenter provider is the higher utilization of the resources.

This experiment aims to demonstrate that schedulers may be sacrificing performance in exchange for simplicity if they do not offer metadata access to their users. The most important takeaway from the results of this experiment is that performance increases in execution times using the extended API. This improvement is achieved because the metadata provides insights about the expected retrieval times affected by the underlying congestion of resources. This way, when users have the metadata access API, they can delay the download of objects that will take longer than necessary due to congestion. Globally, when all users get congestion insights, the storage service gets better load balancing of requests among object storage servers. This load balancing is not optimal since each user optimizes for their workload, but it is more optimal than when users are oblivious to increased download times because of congestion.

Due to a higher load balance, the object storage service servers have smaller buffers, and user requests are processed faster, leading to higher user performance. This load balance

also supposes a greater utilization of the provider’s resources since when the buffers are less balanced and more full, other servers receive fewer requests and lower utilization. Therefore, with a greater load balance and the same amount of user workloads, the buffers are less full, which means that the utilization of resources is more efficient, and consequently, the utilization is higher.

Furthermore, in this experiment, we demonstrate that using metadata access improves performance and that it is necessary to offer it as a programming abstraction. This is because the provider does not have the business logic knowledge to know what objects the user needs to download the objects. Therefore, the scheduler cannot internally implement the optimization of data workflows without exposing programmability to the user.

6 Conclusion

In this work, we undertook the challenge of designing a comprehensive reference architecture for datacenter scheduler APIs, encompassing both industrial and academic schedulers. By conducting a meticulous comparison between this architecture and five representative industrial schedulers, we successfully identified crucial gaps and limitations in the existing implementations, particularly in the areas of data management, task migration, and autoscaling.

To quantify the impact of these missing abstractions, we conducted rigorous performance evaluations. The results were striking, revealing the substantial improvements that can be achieved by incorporating the identified abstractions. Notably, the implementation of metadata access led to an impressive 36% enhancement in resource utilization and a remarkable 40% reduction in total execution time per workflow. Similarly, the utilization of container migrations yielded a substantial 15% increase in utilization and a remarkable 81% improvement in total execution time per task. Additionally, the utilization of reservations resulted in a significant 43% reduction in waiting times.

These findings underscore the critical importance of addressing the identified gaps in datacenter scheduling systems. By providing concrete evidence of the performance impact of missing abstractions, our work highlights the tremendous value that can be added by incorporating these abstractions into the design and implementation of schedulers. This research not only sheds light on the potential for improved resource utilization and more efficient task execution but also serves as a catalyst for future advancements in the field.

The significant contributions of this work lie in the meticulous design of the reference architecture, which serves as a comprehensive guide for the development of more robust and feature-rich datacenter scheduler APIs. Moreover, the detailed performance evaluations conducted across multiple dimensions provide invaluable insights into the potential benefits and trade-offs associated with incorporating the identified abstractions. This work paves the way for further research and development efforts aimed at refining scheduling systems and maximizing their performance and efficiency.

The creation of the reference architecture presented one of the biggest intellectual challenges in this work. Designing a comprehensive and versatile architecture that captures the essential components of datacenter scheduler APIs, while addressing the limitations in

existing systems required a deep understanding of the underlying concepts and a meticulous analysis of various schedulers. It involved carefully considering and identifying the key abstractions necessary for efficient resource management. The process involved extensive research, critical thinking, and iterative refinement to ensure the architecture accurately represented the requirements of real-world schedulers. But also a non-methodic approach, relying on intuition and creativity to navigate the complex landscape of datacenter scheduling systems.

Another significant intellectual effort was dedicated to designing the experiments and conducting thorough evaluations. This involved carefully selecting the appropriate metrics, defining the experimental setups, and conducting performance measurements across different traces and workloads. Designing meaningful experiments required thoughtful consideration of the factors that influence scheduler performance and the potential impact of the proposed abstractions. It involved developing a systematic approach to gather accurate performance data, analyze the results, and draw meaningful conclusions. This process required a combination of theoretical knowledge, experimental skills, and careful interpretation of the results.

We are committed to promoting transparency and open science. As part of our dedication to advancing research and fostering collaboration, we have made all our data and software artifacts publicly available. Researchers, practitioners, and interested individuals can access these valuable resources at the following link: <https://github.com/aratz-lasa/opensdc>. By sharing our data and software, we aim to facilitate further exploration, replication, and validation of our findings, as well as encourage the development of new insights and advancements in the field of datacenter scheduling.

References

- [1] Apache airflow. <https://github.com/apache/airflow>, 2023.
- [2] Horizontal pod autoscaling, Mar 2023.
- [3] Kubernetes. <https://github.com/kubernetes/kubernetes>, 2023.
- [4] G. Andreadis, L. Versluis, F. Mastenbroek, and A. Iosup. A reference architecture for datacenter scheduling: design, validation, and experiments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 37:1–37:15. IEEE / ACM, 2018.
- [5] R. Bhardwaj, A. Tumanov, S. Wang, R. Liaw, P. Moritz, R. Nishihara, and I. Stoica. ESCHER: expressive scheduling with ephemeral resources. In A. Gavrilovska, D. Altinbüken, and C. Binnig, editors, *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, pages 47–62. ACM, 2022.
- [6] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, and M. Parashar. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing*, 4(2):26–35, 2017.

- [7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.
- [8] W. L. Chang, D. Boyd, O. Levin, et al. Nist big data interoperability framework: Volume 6, reference architecture. 2019.
- [9] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of network and computer applications*, 23(3):187–200, 2000.
- [10] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you’re late don’t blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.
- [11] W. Dargie. Estimation of the cost of VM migration. In *23rd International Conference on Computer Communication and Networks, ICCCN 2014, Shanghai, China, August 4-7, 2014*, pages 1–8. IEEE, 2014.
- [12] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.
- [13] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.
- [14] T. Dufva and M. Dufva. Grasping the future of the digital society. *Futures*, 107:17–28, 2019.
- [15] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. I. Kat. It’s time to revisit LRU vs. FIFO. In A. Badam and V. Chidambaram, editors, *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*. USENIX Association, 2020.
- [16] R. Feynman. Ebnf: A notation to describe syntax. *Cited on*, page 10, 2016.
- [17] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.
- [18] F. Gens. Worldwide and regional public it cloud services, 2014.
- [19] R. Grandl, A. Singhvi, R. Viswanathan, and A. Akella. Whiz:{Data-Driven} analytics execution. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 407–423, 2021.
- [20] C. Grimme, J. Lepping, A. Papaspyrou, P. Wieder, R. Yahyapour, A. Oleksiak, O. Wäldrich, and W. Ziegler. Towards A standards-based grid scheduling architecture. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, *Grid Computing - Achievements and Prospects: CoreGRID Integration Workshop 2008, Hersonissos, Crete, Greece, April 2-4, 2008*, pages 147–158. Springer, 2008.
- [21] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, and T. Moscibroda. Protean: VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Imple-*

- mentation, *OSDI 2020, Virtual Event, November 4-6, 2020*, pages 845–861. USENIX Association, 2020.
- [22] M. Haenlein and A. Kaplan. A brief history of artificial intelligence: On the past, present, and future of artificial intelligence. *California management review*, 61(4):5–14, 2019.
 - [23] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In R. van Renesse and N. Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 588–604. ACM, 2021.
 - [24] 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 *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, pages 1–10, 2012.
 - [25] M. S. Jennifer. Ten actions when grid scheduling: The user as a grid scheduler. *Grid Resource Management: State of the Art and Future Trends, Norwell, MA, USA, Kluwer Academic Publishers*, pages 15–24, 2004.
 - [26] F. Juarez, J. Ejarque, and R. M. Badia. Dynamic energy-aware scheduling for parallel task-based application in cloud computing. *Future Generation Computer Systems*, 78:257–271, 2018.
 - [27] N. Kim, J. Cho, and E. Seo. Energy-credit scheduler: an energy-aware virtual machine scheduler for cloud systems. *Future Generation Computer Systems*, 32:128–137, 2014.
 - [28] D. Lipari. The slurm scheduler design. In *SLURM User Group Meeting, Oct*, volume 9, page 52, 2012.
 - [29] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, D. Leaf, et al. Nist cloud computing reference architecture. *NIST special publication*, 500(2011):1–28, 2011.
 - [30] 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*, 48:1–18, 2015.
 - [31] S. Mazumdar, D. Seybold, K. Kritikos, and Y. Verginadis. A survey on data storage and placement methodologies for cloud-big data ecosystem. *J. Big Data*, 6:15, 2019.
 - [32] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 69–84, 2013.
 - [33] P. Owen. Slurm used on the fastest of the top500 supercomputers. <https://www.prweb.com/releases/2012/11/prweb10149109.htm>. Accessed: 2023-03-18.
 - [34] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, pages 352–358. IEEE, 2002.

- [35] T. Rep. *2022 Leadership Vision for Infrastructure And Operations*. Gartner, 2022.
- [36] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 199–212. ACM, 2009.
- [37] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 351–364. ACM, 2013.
- [38] S. Shen, A. Iosup, A. Israel, W. Cirne, D. Raz, and D. Epema. An availability-on-demand mechanism for datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE, 2015.
- [39] K. Sreenu and M. Sreelatha. W-scheduler: whale optimization for task scheduling in cloud computing. *Cluster Computing*, 22(1):1087–1098, 2019.
- [40] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurr. Pract. Exp.*, 17(2-4):323–356, 2005.
- [41] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Syst.*, 13(3):260–274, 2002.
- [42] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the third ACM Symposium on Cloud Computing*, pages 1–7, 2012.
- [43] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.
- [44] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 301–316, 2014.
- [45] J. Wilkes. More Google cluster data. Google research blog, Nov. 2011. Posted at <http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html>.
- [46] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In E. M. Nahum and D. Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.
- [47] Y. Zhai, J. Tchaye-Kondi, K. Lin, L. Zhu, W. Tao, X. Du, and M. Guizani. Hadoop perfect file: A fast and memory-efficient metadata access archive file to face small files problem in HDFS. *J. Parallel Distributed Comput.*, 156:119–130, 2021.

- [48] C. Zheng, B. Tovar, and D. Thain. Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*, pages 130–139. IEEE Computer Society / ACM, 2017.
- [49] Q. Zheng, K. Zheng, H. Zhang, and V. C. Leung. Delay-optimal virtualized radio resource scheduling in software-defined vehicular networks via stochastic learning. *IEEE Transactions on Vehicular Technology*, 65(10):7857–7867, 2016.