



# The Cost of Simplicity: Understanding Datacenter Scheduler Programming Abstractions

Aratz Manterola Lasa*
aratz@warpstreamlabs.com
WarpStream Labs
USA

Sacheendra Talluri
s.talluri@vu.nl
Vrije Universiteit Amsterdam
The Netherlands

Tiziano De Matteis
t.de.matteis@vu.nl
Vrije Universiteit Amsterdam
The Netherlands

Alexandru Iosup
a.iosup@vu.nl
Vrije Universiteit Amsterdam
The Netherlands

## ABSTRACT

Schedulers are a crucial component in datacenter resource management. Each scheduler offers different capabilities, and users use them through their APIs. However, there is no clear understanding of what programming abstractions they offer, nor why they offer some and not others. Consequently, it is difficult to understand their differences and the performance costs imposed by their APIs. In this work, we study the programming abstractions offered by industrial schedulers, their shortcomings, and their related performance costs. We propose a general reference architecture for scheduler programming abstractions. Specifically, we analyze the programming abstractions of five popular industrial schedulers, understand the differences in their APIs, and identify the missing abstractions. Finally, we carry out exemplary experiments using trace-driven simulation demonstrating that an API extension, such as container migration, can improve total execution time per task by 81%, highlighting how schedulers sacrifice performance by implementing simpler programming abstractions. All the relevant software and data artifacts are publicly available at https://github.com/atlarge-research/quantifying-api-design.

## CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; • **Software and its engineering** → **Cloud computing**.

## KEYWORDS

cloud, scheduler, API, design, performance

**ACM Reference Format:**
Aratz Manterola Lasa, Sacheendra Talluri, Tiziano De Matteis, and Alexandru Iosup. 2024. The Cost of Simplicity: Understanding Datacenter Scheduler Programming Abstractions. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3645038

*This work was done while the author was affiliated with Vrije Universiteit Amsterdam.




*ICPE '24, May 7–11, 2024, London, United Kingdom*

ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3645038

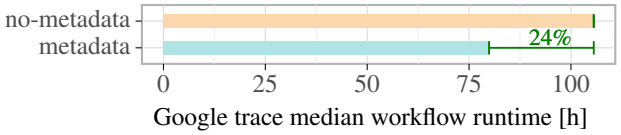


**Figure 1: Performance penalty due to a missing programming abstraction: storage metadata access.**

## 1 INTRODUCTION

Society's increasing dependence on digital technologies and infrastructure has led to the widespread use of datacenters for deploying digital services [20, 28]. Schedulers play a vital role in orchestrating datacenter resources to meet the demands of these services [24, 45]. The interfaces schedulers offer to users determine the limits of the users' ability to mold the orchestration process to support their application needs. Different schedulers offer different levels of programmability and control to users [27, 47, 51, 58]. For example, some schedulers provide restricted programming abstractions [1], minimizing user input, while others offer more flexible interfaces that empower users with greater control over resource allocation and job placement [47, 58]. This spectrum of scheduler programming abstractions raises 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 [46].

The third question is: **What is the performance cost of the sacrificed abstractions?** Despite their security and robustness

[1]We use programming abstraction and API interchangeably.

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 different experiments. Figure 1 depicts an exemplary result with the median execution time of workflows in a trace from Google [55]. 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 24% reduction in median workflow runtime when using the abstraction.

To address these questions and enhance our understanding of scheduler programming abstractions, we develop a comprehensive and structured reference architecture that provides a unified view of the programming interfaces offered by schedulers. This reference architecture compliments earlier work on scheduler internals [5, 30]. It guides developers and researchers in designing and implementing scheduling APIs, capturing the essential abstractions in task scheduling and resource management within datacenter environments.

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 [5]. Second, it serves as a knowledge base for designing better schedulers that can meet the demands of modern applications by addressing shortcomings [5, 11, 22, 36]. 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 [51] and Borg [10] when the need for a new design arises.

To understand datacenter scheduler programming abstractions and the cost of missing ones, we make a four-fold contribution:

(1) We design a reference architecture for datacenter scheduler programming abstraction (Section 3). We propose a set of design principles and, with them, design an architecture that considers different stakeholders and the programming abstractions of existing schedulers.
(2) We analyze existing industrial and academic schedulers by mapping them to the reference architecture (Section 4). This mapping allows us to compare them using a common language. The comparison reveals abstractions proposed in literature but missing from industrial schedulers.
(3) We analyze the effect of missing abstractions on the performance of modern schedulers (Section 5). To this end, we implement three missing abstractions in an event-driven simulator and conduct simulations using real-world traces collected by major datacenter operators, e.g., Google and Microsoft.
(4) We contribute to open science and reproducibility by releasing data and software artifacts. To enable the experiments in this work, we have significantly extended OpenDC [39], a state-of-the-art simulator. We release the code enabling this work's capabilities through Github: https://github.com/atlarge-research/quantifying-api-design. The repository has been archived using Zenodo at: https://zenodo.org/doi/10.5281/zenodo.10605424.

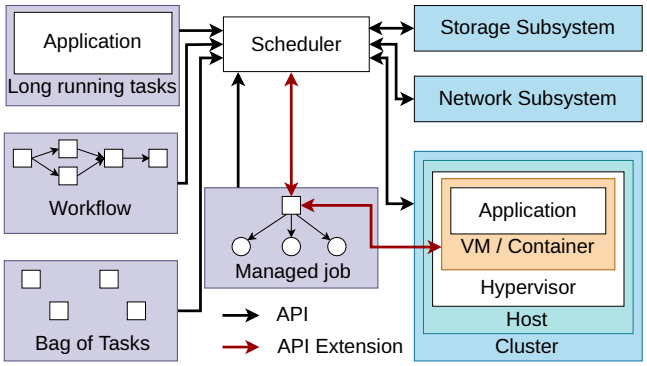


**Figure 2: Datacenter scheduling system model.**

## 2 DATACENTER SCHEDULER SYSTEM MODEL

This section contextualizes this work by describing common datacenter scheduling-related concepts depicted by Figure 2.

### 2.1 Workload

The workload is executed using the resources the scheduler assigns to the user. Following the taxonomy proposed by Andreadis et al. [5], we consider four types of workloads:

(1) **Batch workflows** are workloads comprising several tasks with dependencies between them.
(2) **Bag-of-tasks** are jobs formed by several tasks without any dependency between them.
(3) **Long running tasks** run for a very long time and are usually inside a host such as a VM.
(4) **Managed jobs** are workloads where a manager coordinates all the tasks, such as Spark.

The users specify the requirements to execute the workload. Usually, these comprise the amount of CPU and memory. However, in some cases, other requirements, 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 scheduler through its API.

### 2.2 Scheduling

A user submits a workload to use the resources through a central component, the scheduler [10, 41]. 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 resources available to the scheduler can change. Therefore, the scheduler must adapt to changing workload requirements by increasing or decreasing dynamically allocated resources. This is usually done through a specific subcomponent (e.g., the *autoscaler* in Kubernetes [2]). A

scheduler can also preempt, recover, and migrate workloads when the amount of available resources changes.

Schedulers can be monolithic [35] and run in a single process that handles all tasks. They can be distributed where 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 [42], where a single entity implements the scheduler and dictates the policies and mechanisms, or it can be decentralized [51] 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 Scheduler resources

The workloads are executed on top of the resources that the scheduler manages. Resources typically refer to physical machines usually located within a datacenter. These datacenters consist of multiple clusters, each housing several hosts, with each host functioning as a node within a rack. It is important to note that while our discussion primarily focuses on virtualized resources such as VMs or containers running on hosts through a hypervisor, it is also possible to manage bare metal resources. However, virtualized environments are more prevalent and present a wider range of interesting phenomena for modeling and analysis.

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 number of cores. Memory and storage can have different sizes. We model resource consumption using a discrete model, where the workload reports how many resource it requires and for how long. 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. Programming abstractions are offered through a GUI, CLI, or a protocol such as HTTP.

The API includes both the interactions of the scheduler with the applications and the resources. In this work, we investigate API extensions that allow the scheduler to interact with applications and the resources allocated after the initial resource allocation.

Resource management systems, such as autoscalers, interact with schedulers and other resource managers in a completely automated manner without any user intervention. We consider the API between these different systems a part of the scheduler programming abstraction. The API constrains the actions available to these systems. Obtaining system data and performing actions not supported by the API is difficult for the systems we analyze in this work.

# 3 REFERENCE ARCHITECTURE FOR SCHEDULER PROGRAMMING ABSTRACTIONS

We propose a *reference architecture* to understand and describe standard programming abstractions available in current schedulers. With systematic categorization and organization, the reference architecture will offer a framework for analyzing and comparing existing schedulers and a comprehensive view of the range of common abstractions that a scheduler can implement. This helps us answer the question *What are the programming abstractions common in current schedulers?*.

Our process for designing the reference architecture has the following steps:

(1) Stakeholder and use case identification
(2) Requirements analysis
(3) Model industrial schedulers
(4) Model emerging concepts from academia
(5) Unify industrial schedulers with emerging concepts

We describe our requirements in Section 3.1. We identify five popular schedulers in the industry, and we analyze their APIs. Consulting experts in the field, we select the following schedulers: Kubernetes [3], SLURM [35], Spark [56], Condor [51], and Airflow [1]. We further analyze these schedulers in Section 4.

For emerging concepts from academia, we conduct a systematic literature survey [33], sort the results by citations, and pick the top 15 papers with new APIs different from what we identified in industrial schedulers. We end up analyzing the following 15 academic schedulers: [9, 12, 15, 17, 18, 25, 31, 32, 38, 44, 48, 50, 53, 54, 59].

After analyzing industrial and emerging scheduler designs from academia, we extract, filter, generalize, and unify them into a reference architecture.

## 3.1 Requirements

We identify the requirements that must be met by the reference architecture. This has to be:

**R1 Understandable.** Different stakeholders should be able to easily understand the different components that make up the reference architecture, how they relate to each other, or their high-level meaning. We enable this through the principles in Section 3.2 and the description language in Section 3.3.

**R2 Actionable.** The design must take into account whether users can use it to take concrete actions. We use the architecture in Section 4 to identify missing abstractions in industrial schedulers. We quantify the cost of missing abstractions in Section 5.

**R3 Pragmatic.** The reference architecture concepts can be implemented in code and evaluate different programming abstractions comparatively. The reference architecture has been realized in the OpenDC simulator and used for experiments in Section 5.

**R4 Comprehensive.** Can represent *all* already known concepts used in industrial schedulers and emerging concepts from academia. We map five industrial schedulers to the reference

architecture in comparison in Section 4. The reference architecture was built by analyzing 15 research prototypes from the community.

## 3.2 Design principles

For the design of the scheduling programming abstractions reference architecture, we identify the following design principles.

- **P1 Separation of objects from actions.** We distinguish between the actions that can be performed and the objects, which represent the system's state, that are used as input to the actions. This separation facilitates comprehension (**R1**).
- **P2 Grouping of related actions.** There may be several actions that are related to each other. Therefore, to facilitate comprehension, related actions are grouped.
- **P3 Avoidance of concrete technologies in objects.** We keep the objects as high-level as possible to avoid strong coupling to a specific technology.

## 3.3 Reference Architecture Design

We analyze industrial and emerging scheduler designs from academia for scheduling abstractions. Then we extract, filter, generalize, and unify them into a reference architecture. In this process, we follow the requirements and design principles we set out in the previous subsections. The reference architecture allows us to describe the different abstractions provided by the schedulers we analyzed using a common language. This common language allows enables us to compare the schedulers' APIs to each other in Section 4.

The reference architecture is depicted in Figure 3. The high-level components of the reference architecture are *actions* and *objects* that comprise the scheduler API. Object describes the current or desired state of the system. Actions describe physical events (such as leasing a VM) that are executed when certain conditions are met. The conditions use objects in their specification. Each action must have three types of conditions: WHAT, WHEN, and WHERE, and for each condition, there can be one or more objects. 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 using the reference architecture.

**Listing 1: Example scheduler action.**

```
Provision:Lease
UserResource<type:job, runtime:5 days>
IN SchedulerResource<type:vm,
                cpu:2.4Ghz, memory:16Gb>
WHEN Event<day:11, month:12, year:2023>
```

Consider the scheduler interaction in Listing 1; the action is "Provision:Lease," indicating the provisioning and leasing of resources. The objects involved are "UserResource" with specific characteristics such as job type and a runtime of 5 days, and "SchedulerResource" with attributes like VM type, CPU of 2.4GHz, and memory of 16GB. The condition "IN" specifies that the "UserResource" is allocated within the "SchedulerResource". Lastly, the "WHEN" condition indicates an event occurring on December 31, 2022.

Tables 1 and 2 define and describe the actions and objects within the reference architecture. These tables serve as a resource for understanding the specific elements of the reference architecture and their respective functionalities.

In addition to the visual representation of the reference architecture for scheduling programming abstractions shown in Figure 3, we have also defined a formally defined syntax which we use in Listing 1. The syntax is based on the Extended Backus–Naur Form (EBNF) and provides a structured and consistent way to express conditions using actions and objects in the programming abstractions. Due to space constraints, we do not present the formal syntax definition here but will add it as an appendix. The formally defined syntax enables precise communication using the reference architecture.

# 4 ANALYSIS OF INDUSTRIAL SCHEDULERS

We analyze the scheduler APIs of industrial schedulers by qualitatively mapping their features to the reference architecture: Kubernetes (v1.27) [3], SLURM (v23.02) [35], Spark (v3.4.0) [56], Condor (v10.4.3) [51], and Airflow (v3.3.0) [1]. Through the mapping, we respond to the question of *What programming abstractions are sacrificed for simplicity?* The mapping provides insights into their alignment with the idealized model. This comparison helps identify missing abstractions compared to all the ones in the reference architecture, highlighting potential shortcomings.

## 4.1 The mapping process

For each considered scheduler, we consult its official documentation, source code, and articles we find online. Then, using these resources, for each component of the reference architecture, we identify if there is a *complete*, *partial*, 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 restricts the object parameters, and the user can flexibly specify whatever parameters they need. For example, the user can add any metadata information. A partial match means the scheduler allows the user to specify only a limited set of object parameters. For example, the user can only specify CPU constraints, not any other resource type. A no-match means that the scheduler does not allow that object type.

## 4.2 Mapping results

Using the reference architecture, we analyze the shortcomings of the selected group of five industrial schedulers. Currently, it is not known when nor why you should use some schedulers and not others. It is also unclear if any scheduler has a clear missing gap or how to fill it. For that, it is necessary to analyze the scheduling APIs. We map their APIs into the reference architecture and aggregate the results in two tables. In Table 3, we map the actions, and in Table 4, the objects. We specify whether each action and object is a full, partial, or no match.

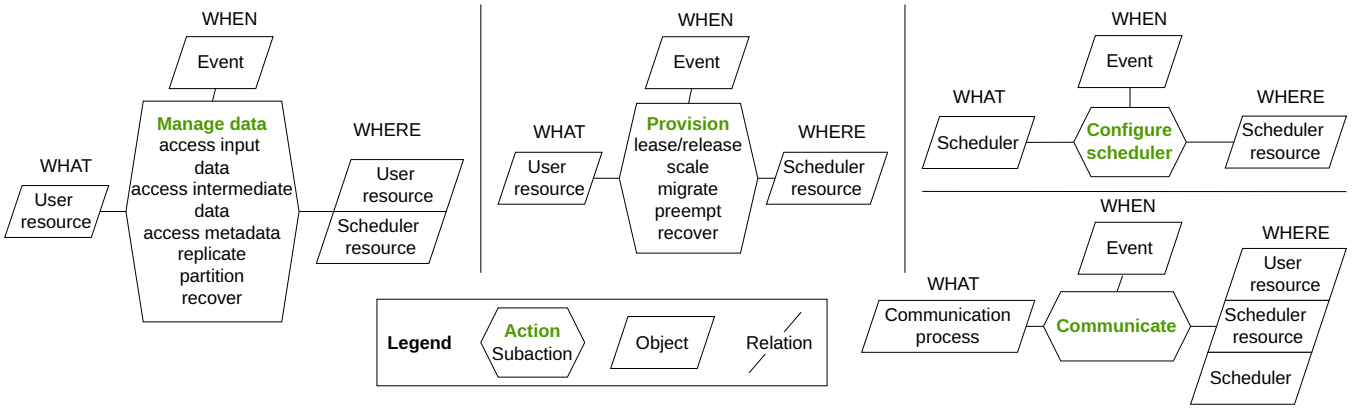


**Figure 3: Reference architecture for scheduling programming abstractions.**

**Table 1: Description of the actions that compose the reference architecture.**

| Action | Sub-action | Description |
|---|---|---|
| Provision | Lease/release | Activation and assignment of a user resource to a scheduler resource. |
| | Scale | Addition or reduction of already provisioned user resources. |
| | Migrate | Migration of a user resource to a different scheduler resource. |
| | Preempt | Abortion of execution or assignment of a user resource, putting it back in the scheduler queue. |
| | Recover | Recover a task after failure, restart execution, or put it back into the scheduler queue. |
| Configure scheduler | | Configuration of the behavior of the scheduler. |
| Replicate | Access input data | Access to data that user jobs take as input. |
| | Access intermed. | Access to data that user jobs generate during their runtime. |
| | Access metadata | Access to the information about the user data. |
| | Replicate | Replication of the user data. |
| | Partition | Partitioning of the user data so that a subset of the data is placed in different scheduler resources. |
| | Recover | Recovery of the user data after the failure of execution or the storage system. |
| Communicate | | Communication with the user resources, scheduler resources, or even the scheduler, such as setting a callback for getting notified about scheduling events. |

**Table 2: Objects in the reference architecture.**

| Object | Description |
|---|---|
| Event | Representation of objects in time or instantiations of properties in objects. Such as concrete date-times (00:00 of 31st of December 2022) or an instantiation of a property like a metric reaching a numeric value (CPU utilization is greater than 80%). |
| User resource | Representation of any kind of input from the user. This includes execution units like a job, task, etc., but also data as a file, environment variable, etc. |
| Scheduler resource | Representation of resources owned and managed by the scheduler. Resources can be virtual machines, containers, storage systems, databases, etc. |
| Communication process | Representation of the process of communication, such as a signal, message, callback, etc. |

**Table 3: Full overview of programming abstraction actions of schedulers mapped to the reference architecture. Legend: ●/◗/❍ = full/partial/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow**

| Action | Sub-Action | Schedulers | | | | |
|---|---|---|---|---|---|---|
| | | Ku | Sl | Sp | Co | Ai |
| Provision | Lease / release | ● | ● | ● | ● | ● |
| | Scale | ● | ❍ | ◗ | ❍ | ❍ |
| | Migrate | ❍ | ❍ | ❍ | ❍ | ❍ |
| | Preempt | ◗ | ● | ❍ | ● | ❍ |
| | Recover | ◗ | ◗ | ● | ◗ | ◗ |
| Configure scheduler | | ● | ◗ | ● | ● | ● |
| Manage data | Access input | ● | ◗ | ● | ● | ● |
| | Access interm. | ❍ | ❍ | ◗ | ❍ | ❍ |
| | Access metadata | ❍ | ❍ | ❍ | ❍ | ❍ |
| | Replicate | ❍ | ❍ | ● | ❍ | ❍ |
| | Partition | ❍ | ❍ | ● | ❍ | ❍ |
| | Recover | ◗ | ❍ | ● | ● | ❍ |
| Communicate | | ◗ | ● | ◗ | ◗ | ◗ |

The results indicate that industrial schedulers have several shortcomings. Several actions are under-implemented. There is a very clear pattern, where most schedulers implement three actions: `lease / release`, `configure scheduler`, `access input data`. In most cases, all others are either partially or not implemented. The biggest shortcoming is in `manage data` action and its objects, where most sub-actions and objects are not implemented. Overall, the industrial schedulers examined in our study do not provide data management abstractions to the user. This means that users have less control over the data and, consequently, less chance to optimize performance. For example, if the user has several unordered data

**Table 4: Full overview of programming abstraction objects of schedulers mapped to the reference architecture. Legend: ●/◗/❍ = full/partial/no match; Ku = Kubernetes; Sl = SLURM; Sp = Spark; Co = Condor; Ai = Airflow.**

| Action | Object | Schedulers | | | | |
|---|---|---|---|---|---|---|
| | | Ku | Sl | Sp | Co | Ai |
| Provision | user resource | ● | ◗ | ◗ | ◗ | ◗ |
| | event | ◗ | ◗ | ◗ | ◗ | ◗ |
| | sched. resource | ● | ● | ◗ | ● | ● |
| Configure scheduler | scheduler | ◗ | ◗ | ● | ● | ● |
| | event | ◗ | ◗ | ❍ | ❍ | ❍ |
| | sched. resource | ◗ | ❍ | ❍ | ❍ | ❍ |
| Manage data | user resource | ◗ | ◗ | ● | ◗ | ● |
| | event | ❍ | ❍ | ❍ | ❍ | ❍ |
| | sched. resource | ❍ | ◗ | ❍ | ◗ | ❍ |
| Communicate | comm. process | ◗ | ◗ | ◗ | ◗ | ● |
| | event | ◗ | ◗ | ◗ | ❍ | ◗ |
| | user resource | ◗ | ◗ | ● | ◗ | ● |
| | sched. resource | ❍ | ◗ | ● | ❍ | ❍ |
| | scheduler | ❍ | ◗ | ❍ | ❍ | ❍ |

items to process, consulting the metadata and obtaining information about the placement and requests load of the storage systems where the data is stored, could optimize how and when the data is processed.

In all other cases, the `communicate` action is partially implemented except in SLURM. Similarly, most communication objects are partial matches. This might imply a lower performance since it does not allow the user to inform the scheduler during runtime about application-level insights, nor vice versa, the scheduler to inform the user about scheduling-level insights. Moreover, partial matches imply that actions and objects are limited to a particular subset and do not allow the user to specify arbitrary inputs. For example, the Condor API only provides communication actions with user jobs, not the scheduler. Therefore, the user can dynamically inform about application-level insights to their jobs but not to the scheduler, reducing the scope of potential performance improvements.

**Key Takeaway:** Many actions and objects have partial or no matches, meaning their APIs are under-implemented. Consequently, they reduce users' ability and scope to optimize their applications' performance. The main shortcomings are found in `manage data` action and its objects but also in `communicate` actions and their objects to a lesser extent. Sub-actions related to provisioning other than `lease`, such as `scale`, `migrate`, and `recover`, are also not well supported by schedulers.

# 5 EVALUATING THE PERFORMANCE COST OF SIMPLE SCHEDULING ABSTRACTIONS

In this study, we address the limited programmability of industrial schedulers and highlight the need for greater user programmability to improve user-application performance. We identify under-implemented programming abstractions in scheduler APIs in Section 4. In this section, we design experiments to quantify the performance cost of these missing abstractions. The experiments focus on three specific use cases: *1) reservations*, *2) migration requests*, and *3) metadata access*. We analyze the shortcomings of various industrial schedulers in implementing these abstractions and propose extensions to address them. This answers the question *What is the performance cost of the sacrificed abstractions?* raised in Section 1. A comprehensive overview of these experiments can be found in Table 5, which outlines the API extensions, parameters, traces, and metrics for each use-case.

## 5.1 Implementation, Input Setup, and Open-Sourcing

*Software:* The reproducibility of the experiments is ensured through the use of the OpenDC data center discrete event simulator [39], which is deterministic. We performed multiple runs with different seeds of randomness to capture variations in the results. For each experiment run, we calculated the empirical cumulative distribution function (ECDF) to analyze the distribution of the measured metrics. This approach allowed us to assess the behavior and performance of the proposed extensions across different scenarios and obtain comprehensive insights.

*Input data:* Traces from private and public cloud environments, Azure [13], Google [55], and Bitbrains [49] — a Dutch ICT provider — were selected to provide realistic and diverse workload data for evaluating the proposed extensions. By leveraging real-world traces, our research captures the variability and complexity of cloud workloads, ensuring the relevance and validity of our findings. These traces are open source, and the simulator has parsers for the respective formats. The Azure and Bitbrains traces were used as they were provided, while the first 2.5 days were used from a 30-day Google trace. The characteristics of the different traces are outlined in Table 6.

*Simulated environment:* The number of machines in the simulated environment are different for different traces and utilization levels. The environments have 35 machines for the Google trace, 102 machines fore the Azure trace, and 1039 machines for the Bitbrains trace when simulating the workloads at 75% utilization. The machines are heterogeneous having 4 to 32 cores depending on the configuration. The precise environment specifications for each experiment are described in topology files located in the experiment's folder in the applications git repository.

## 5.2 Reservation

**Goal:** Schedulers utilize resources better if they know when tasks arrive and their resource requirements. We investigate if a scheduler with an API that accepts this additional information performs better for three different traces and by how much.

In the context of scheduling and resource allocation in datacenters, there is a specific category of jobs that are long-running and periodically submitted, which are provisioned into VMs ( ❶ and ❷ in Figure 4). These jobs exhibit predictable patterns, as they recur regularly and have well-defined resource requirements. Examples of such jobs include data processing pipelines, scientific simulations, and batch processing tasks.

Since their resource requirements and execution patterns are known in advance, schedulers could use this knowledge to allocate resources more efficiently and reduce waiting times. However, in

**Table 5: Summary of evaluation experiments.**

| Name | API extension | Parameters | Fixed parameters | Traces | Metrics |
|---|---|---|---|---|---|
| Reservation Section 5.2 | User provided start time and resource estimates | Reservation ratio, resource utilization | Scheduling policy (EFT) | Azure, Bitbrains, Google | Waiting time, slowdown |
| Migration Section 5.3 | Container migration via orchestrator callbacks | Migration type, oversubscription | Resource utilization (85%), FIFO policy | Azure, Bitbrains, Google | Execution time, packing efficieny |
| Metadata access Section 5.4 | Use storage subsystem busyness to order tasks | Metadata-aware task reorder policy | Resource utilization (80%) | Google and IBM combined | Buffer size, total time |

**Table 6: Characteristics of the traces used in the experiments**

| Workload | VMs/Tasks | Duration [days] | VM duration [days] | | CPU cores | | CPU capacity [GHz] | | Memory [GBs] | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Mean | $\sigma$ | Mean | $\sigma$ | Mean | $\sigma$ | Mean | $\sigma$ |
| 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 | 1000000 | 2.5 | 0.0375 | 0.083 | 1.0 | 0.0 | 1.68 | 2.08 | 0.17 | 0.2 |

practice, existing schedulers often do not effectively utilize the predictability of these long-running and predictable jobs [54]. As a result, these jobs may be subject to sub-optimal resource allocation and longer waiting times than necessary.

We propose an extension to datacenter schedulers that enhances scheduling long-running and predictable jobs by incorporating reservation programmability. This extension enables schedulers to be aware of these jobs' recurring nature and resource requirements, allowing for more optimized resource allocation and scheduling.

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. When a user submits a reservation request, instead of immediately provisioning it, the scheduler adds the request to a reservation queue ❸ alongside other pending reservations. During this time, the scheduler applies algorithmic optimizations to improve future provisioning ❹. In our experiment, we employ a simple Earliest Finish Time (EFT) scheduling policy [52] 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 ❺, fulfilling the user's reservation request. In Listing 2, we provide an example of the extension, showcasing the syntax for reservations.

**Listing 2: API for reservations using syntax from Section 3.3, with the extension highlighted in green.**

```
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>
```

We take a scheduler that does not implement reservations as our baseline and investigate the effects of incorporating reservation capabilities into this scheduler. We utilize real-world workload traces from Google, Azure, and Bitbrains to evaluate the performance. We sample a fraction (reservation ratio) of the trace to reserve and

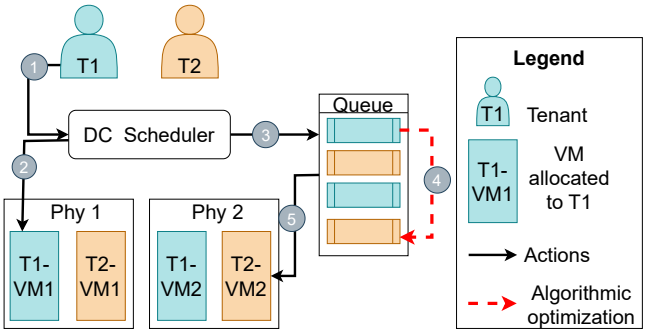


**Figure 4: Reservations experiment system model.**

assume we know the arrival time (from the original trace) of those tasks in advance.

The experiment configurations involve resource utilization and reservation ratio variations (the proportion of reserved resources compared to the total available resources). The resource utilization levels are set at 75%, 80%, and 85%, and the reservation ratios at 0, 0.5, and 1.0 to observe the impact of reservation programmability. These resource utilizations are common in datacenters with high resource utilization [6]. Metrics collected in the experiment include waiting time (the duration tasks spend in the queue before execution) and slowdown (the decrease in task execution speed).

Figure 5 depicts the Azure trace's waiting time and slowdown under nine different configurations. Slowdown, calculated as the ratio of execution time plus waiting time to execution time, represents the overall task performance. In the Azure trace data, we observe a clear relationship between reservation ratios, waiting times, and slowdowns. Specifically, when the system utilization reaches 85%, the system with reservations has a 43% (35-hour) shorter 50th percentile waiting time than the system without reservations (ratio=0.0 means no reservations). In the same scenario, reservations reduce slowdown by 70% (68 units) compared to not using reservations. However, at a lower utilization of 80%, there is an increase in waiting time of 2.5 hours (50th percentile) and a 12-unit (60th percentile) increase in slowdowns. In the other traces examined, there is no significant impact on the waiting times and

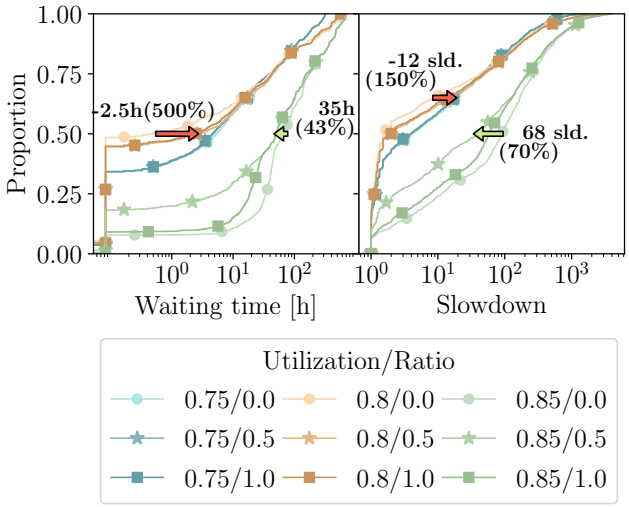


**Figure 5: ECDFs of waiting time and slowdown per task of the Azure trace using the reservation extension. We evaluate the system at different utilization levels and with a different fraction of the trace being reserved in advance (ratio). Ratio 0.0 implies no reservations.**

slowdowns with varying reservation ratios. This could be due to workload characteristics, resource utilization levels, or the configuration of the scheduling system. Further investigation is needed to determine the underlying reasons for the lack of impact.

The results are not as promising for the Google and Bitbrains traces. The Azure trace differs from the other traces as it has a multi-hour task duration. The Google trace has short tasks lasting seconds, and the Bitbrains trace has long jobs lasting weeks. The full analysis for the other traces is available in the technical report. **Key Takeaway:** Reservations reduce slowdown by as much as 70% for the Azure trace, but not as much for the other traces. The results are dependent on the durations of the tasks in the trace.

## 5.3 Migration

**Goal:** We investigate if offloading migration, to mitigate interference, to container orchestrators running on top of VMs leased from a datacenter scheduler is better than the datacenter scheduler itself performing VM migration. We investigate this for three traces.

Datacenter operators oversubscribe their machines as tenants often do not utilize all the allocated resources. Oversubscription means allocating more resources to tenants than there are physically available. Oversubscription leads to interference between tenants if tenants allocated to the same physical machine fully utilize their allocated resources. In such cases, the datacenter operator can migrate one or more tenants to less utilized physical machines to reduce interference.

Migration has a cost proportional to the size of the VM migrated [16, 37]. Therefore it is efficient to migrate only part of a VM if possible. Nowadays, tenants use container orchestrators (K1 in Figure 6), such as Kubernetes, making partial migration possible. The orchestrator requests resources from the datacenter

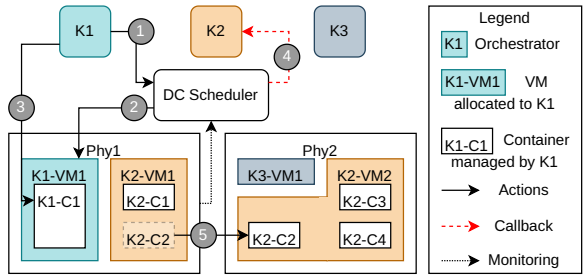


**Figure 6: Migrations experiment system model.**

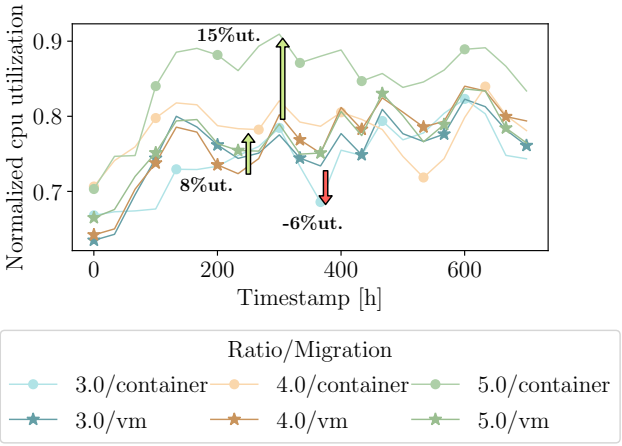


**Figure 7: Task packing efficiency of the Azure trace using different migration techniques. Each line represents a different `<Oversubscription ratio>/<Migrations API>` configuration.**

scheduler ❶. The DC scheduler allocates resources in the form of VMs ❷. The orchestrator then starts application containers inside the VM ❸.

We propose an extension to datacenter schedulers that enables partial migration by making them aware of the tenants' orchestrators. The key to enabling partial migration is to enable bidirectional communication between the datacenter scheduler and the orchestrator. The orchestrator registers a remote callback with the datacenter scheduler before it requests any VM allocations. The datacenter scheduler uses this callback (❹ in Figure 6) to request the orchestrator to migrate ❺ some containers when its monitoring detects interference. In Listing 3, we provide an example of the extension, showcasing the syntax for migrations.

**Listing 3: API for migration using syntax from Section 3.3, with the extension highlighted in green.**

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

As a baseline, we take a scheduler implementing VM migrations and determine the impact of adding container migrations to that

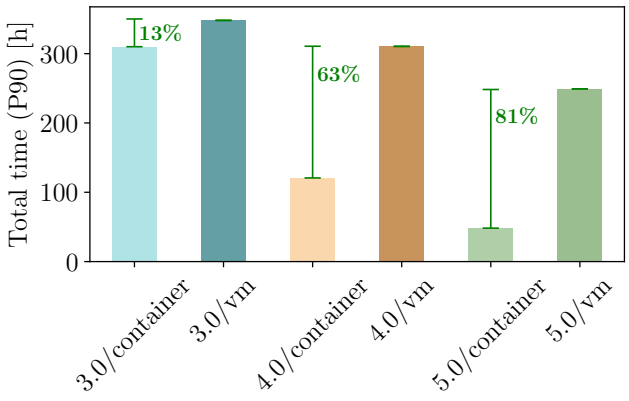


**Figure 8: 90th percentile (P90) total runtime per task of the Azure trace using different migration techniques. Each bar represents a different <Oversubscription ratio>/<Migrations API> configuration.**

scheduler. We use three real-world workload traces from Google, Azure, and Bitbrains for our evaluation.

For each trace, we evaluate the impact of migrations at three oversubscription ratios: 3, 4, and 5. An oversubscription ratio of 3 means that each physical CPU was fully available to three tenants. Oversubscription ratios ranging from 3 to 16 are common in datacenters whose users have low utilization [34, 43]. We model the cost of migration as the time it takes to migrate the RAM used by the VM/container at a conservative rate of 512Mbps. The RAM based cost model and the migration bandwidth are supported by existing literature [37]. Our hypothesis is that migrating a container takes less time than migrating a VM running multiple containers.

We simulate 5 Kubernetes clusters simultaneously using the datacenter. We configure the datacenter topology such that the traces run at 85% average utilization. The metrics we use are total workload execution time and packing efficiency. We calculate packing efficiency by summing the CPU utilization of each virtual machine (VM) and dividing it by the total number of VMs. This metric provides insights into how effectively the resources allocated to the VMs were utilized. A higher packing efficiency indicates better utilization of resources, while a lower value suggests potential inefficiencies or underutilization. By analyzing packing efficiency, we can assess the effectiveness of the scheduling mechanisms in optimizing resource allocation and maximizing overall system performance.

Figure 7 and 8 depict the packing efficiency and the total execution time (90th percentile) of the Azure trace under six different configurations, respectively. In the Azure trace, the highest oversubscription ratio of 5.0 achieved a remarkable 15% improvement in packing compared to configurations without the API extension. Additionally, using the API led to improved performance in terms of total time per task. For example, with the highest oversubscription ratio of 5.0, the 90th percentile (P90) of total time per task in the Azure trace were reduced by 81% when container-level migrations were employed.

In the remaining Google and Bitbrains traces, using the API resulted in shorter total time per task, indicating higher performance. The 99th percentile total time per task in the Google trace showed a reduction of 73% (4.4 hours) with the highest oversubscription ratio of 5.0. However, it is important to note that not all configurations yield better performance with container-level migrations. However, in the Bitbrains trace, no significant improvement in performance is observed. The results indicate the minimal impact of container-level migrations on performance in this particular trace.

**Key Takeaway:** Offloading migration to container orchestrators benefited the Azure and the Google traces, not the Bitbrains trace. The Bitbrains trace differs from other traces as it has an extremely long task duration, with tasks running for weeks.

## 5.4 Metadata access

**Goal:** We investigate if providing datacenter schedulers access to additional information about task data accesses and storage subsystem busyness has a performance impact. We analyze the impact of a trace from IBM object storage [21] combined with the compute trace from Google.

Datacenters offer object storage services that enable users to store and retrieve data efficiently. Services like AWS S3 provide a scalable and reliable solution for storing large amounts of data. In the context of data analysis workloads, users often deploy applications that require accessing multiple objects from the storage ( ❶ and ❷ in Figure 9). These workloads (e.g.: data analytics [4], ML [19]) are often "bag of tasks" where tasks are executed independently and the objects to read are known in advance. Such workloads benefit from reordering their storage access based on the prevailing resource utilization at the time of access.

Without access to fine-grained information about object placement and load levels, users cannot optimize their data retrieval process. As a result, the workload takes longer to complete. The inefficiencies in object access lead to increased latency, reduced throughput, and decreased overall system performance [40, 57].

We propose an extension that empowers users to access object metadata to address this limitation. This extension allows users to make informed decisions regarding the order in which they retrieve data items. By introducing the `accessMetadata` action in the scheduler's programming model, users can query the metadata for specific object IDs and obtain estimates of retrieval times. The scheduler retrieves this information by monitoring the storage servers (❸). This capability enables users to strategically postpone the retrieval of objects from congested storage servers, allowing them to process those objects later when congestion levels have subsided. In Listing 4, we provide an example of the extension, showcasing the syntax for metadata access.

**Listing 4: API for metadata access using syntax from Section 3.3, with the extension highlighted in green.**

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

We aim to determine how much performance existing schedulers are losing out on by not implementing metadata access. As a baseline, we take a scheduler providing an object storage service and

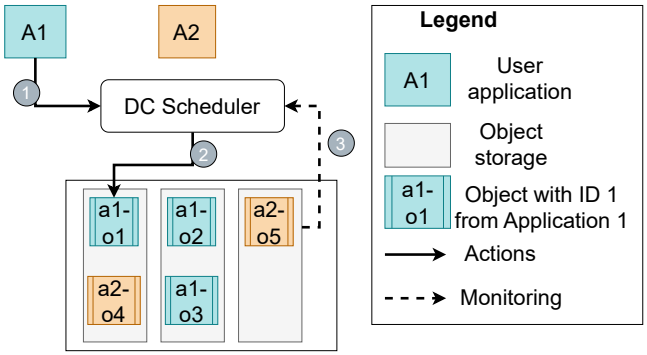


**Figure 9: Metadata access experiment system model.**

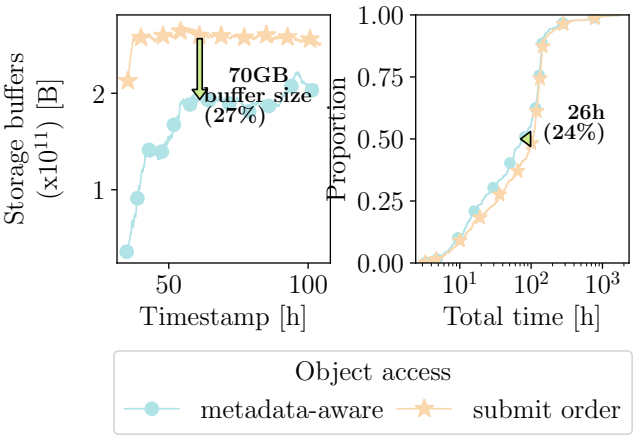


**Figure 10: Comparison of buffer sizes in the object storage service (left) and ECDF analysis of total execution times per workflow (right) between the configuration with and without the metadata access API. This uses the Google Compute trace combined with the IBM object storage trace.**

determine the impact of adding metadata access to that scheduler. Our evaluation is based on a combination of real-world workload traces, specifically a trace from Google and an IBM object storage trace [21]. We chose to focus on Google trace for this experiment due to its availability of detailed information about workflows.

We use the interarrival time, duration, and resource usage of tasks from the Google trace. For each task, we associate an object identifier from the IBM trace. We read identifiers from the IBM trace sequentially. This maintains the popularity distribution of object identifiers and their temporal locality. We assume each task reads from distributed storage at 1Gbps [8]. We simulate a 10 node distributed object storage system, with objects accessed by their identifiers.

We analyze the impact by activating and deactivating metadata access while maintaining a fixed workload trace and storage service utilization. The workload trace utilization is set at 80%. We capture two key metrics to evaluate the system's performance: buffer sizes of the object storage service and total workflow times. The buffer sizes provide insights into the waiting line and load balancing across servers. Smaller buffer sizes indicate lower system load and more efficient workload distribution across servers. Additionally, we measure the total time for each workflow, which encompasses both the waiting time and the execution time.

Figure 10 displays the normalized buffer sizes and total execution times of the trace. The results demonstrate that activating the metadata access API leads to substantially reduced buffer sizes, approximately 27% (70 GB), within the object storage service, resulting in improved performance. Furthermore, metadata-aware workflow execution substantially reduces total time per workflow, with a notable 24% (26-hour) decrease in the median value. These findings emphasize the critical role of metadata access in optimizing object retrievals and enhancing overall performance.

**Key Takeaway:** The significant performance improvements observed in reduced buffer sizes and shorter execution times highlight the value of exposing storage metadata using an API.

## 6 THREATS TO VALIDITY

The reference architecture we proposed has two main limitations.

First, the reference architecture design is limited to the objects we define. In our reference architecture, we identify only five distinct objects and do not specify sub-objects for each. For example, our `Scheduler Resource` object does not differentiate between an API that offers VMs or Edge mobile devices. While this is a limitation, we have deliberately chosen to keep our objects at a high level of abstraction to future-proof our architecture. As the types of resources available for scheduling are constantly changing, we believe it is more important to differentiate objects by what they represent in the highest level of abstraction than by their specific content.

However, to fully leverage the power of our reference architecture, it will be necessary to build more specific models that differentiate between schedulers with different requirements. For example, Spark-like schedulers have different scheduling requirements than Kubernetes-like schedulers. These models must differentiate between objects based on their specific content rather than just their highest level of abstraction.

Second, the simulation scenarios we use and the simulator itself are not a replacement for real-world systems. However, the simulator we use, OpenDC, has been validated for VM and container scheduling for the Bitbrains and Azure traces [39]. The storage part of the simulator and the Google trace have not yet been validated. But we do use realistic models for migration [37] and storage accesses [8]. These models based on measurements from real systems ensure that our results are indicative of real-world performance.

## 7 RELATED WORK

Schopf's multi-stage model of the grid scheduling process [30], the Global Grid Forum [26], and the datacenter scheduler reference architecture [5] offer conceptual models of the internal workings of schedulers. Our work complements these models by specifically addressing the external-facing aspects of scheduling, the programming interface.

Conceptual models of APIs have been proposed for specific computing environments, such as grid computing and cloud computing. Foster et al. presented a reference architecture for grid computing [22], and the National Institute of Standards and Technology

(NIST) introduced models for cloud computing [36]. While these models provide valuable guidance for designing APIs in their respective domains, they do not deal with the concrete API needs of schedulers like Spark and Kubernetes, which have unique characteristics and requirements.

Efforts have been made to develop schedulers that combine multiple scheduling abstractions into a single system, such as Ghost [29] and ESCHER [7]. Ghost delegates OS kernel scheduling decisions to users, granting them greater control over the scheduling process. ESCHER allows users to express arbitrary scheduling constraints as resource requirements, enabling fine-grained control over the scheduling process. Apache Beam [23] and CWL [14] allow users to specify a workflow and run it on multiple resource managers. But they do not allow control over the scheduling mechanism apart from simple labels.

## 8 CONCLUSION

In this work, we designed a reference architecture for datacenter scheduler APIs (Section 3). Our reference architecture covers APIs implemented in 5 industrial schedulers (Kubernetes, SLURM, Spark, Condor, Airflow) and 15 academic schedulers. We use the reference architecture to identify abstraction not implemented or under-implemented in the five industrial schedulers (Section 4). We find that the industrial schedulers do not implement abstractions for data management, task migration, and autoscaling.

We evaluate the performance impact of missing abstractions related to resource reservation, container migration, and storage metadata access in Section 5. We find a 27% improvement in resource usage and a 24% reduction in median workflow runtime when implementing metadata access, a 15% increase in utilization and an 81% improvement in total execution time per task (90th percentile) for container migrations, and a 43% reduction in waiting times (50th percentile) for reservations.

For future work, we intend to provide a toolkit for users to experiment with different designs using the OpenDC simulator. We also plan to validate our simulations beyond the basic validation with VMs, including validation with containers and storage services.

All our data and software artifacts are publicly available at https://github.com/atlarge-research/quantifying-api-design. The repository has been archived using Zenodo at: https://zenodo.org/doi/10.5281/zenodo.10605424

## ACKNOWLEDGEMENTS

This work is supported by EU Horizon Graph Massivizer and EU MSCA CloudStars projects. This research is partly supported by a National Growth Fund through the Dutch 6G flagship project"Future Network Services".

## REFERENCES

[1] 2023. Apache Airflow. https://github.com/apache/airflow.
[2] 2023. Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[3] 2023. Kubernetes. https://github.com/kubernetes/kubernetes.
[4] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 267–280. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/ananthanarayanan
[5] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. 2018. A reference architecture for datacenter scheduling: design, validation, and experiments. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 478–492.
[6] Noman Bashir, Nan Deng, Krzysztof Rzadca, David E. Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take it to the limit: peak prediction-driven resource overcommitment in datacenters. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 556–573. https://doi.org/10.1145/3447786.3456259
[7] Romil Bhardwaj, Alexey Tumanov, Stephanie Wang, Richard Liaw, Philipp Moritz, Robert Nishihara, and Ion Stoica. 2022. ESCHER: expressive scheduling with ephemeral resources. In *Proceedings of the 13th Symposium on Cloud Computing*. 47–62.
[8] Haoqiong Bian and Anastasia Ailamaki. 2022. Pixels: An Efficient Column Store for Cloud Data Lakes. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 3078–3090. https://doi.org/10.1109/ICDE53745.2022.00276
[9] Luiz F Bittencourt, Javier Diaz-Montes, Rajkumar Buyya, Omer F Rana, and Manish Parashar. 2017. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing* 4, 2 (2017), 26–35.
[10] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, omega, and kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.
[11] Wo L Chang, David Boyd, Orit Levin, et al. 2019. NIST Big Data Interoperability Framework: Volume 6, Reference Architecture. (2019).
[12] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. 2000. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of network and computer applications* 23, 3 (2000), 187–200.
[13] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 153–167. https://doi.org/10.1145/3132747.3132772
[14] Michael R Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojša Tijanić, Hervé Ménager, Stian Soiland-Reyes, Bogdan Gavrilović, Carole Goble, et al. 2022. Methods included: standardizing computational reuse and portability with the common workflow language. *Commun. ACM* 65, 6 (2022), 54–63.
[15] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based scheduling: If you're late don't blame us!. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–14.
[16] Waltenegus Dargie. 2014. Estimation of the cost of VM migration. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–8.
[17] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices* 48, 4 (2013), 77–88.
[18] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.
[19] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant prefetching for distributed machine learning I/O. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 92. https://doi.org/10.1145/3458817.3476181
[20] Tomi Dufva and Mikko Dufva. 2019. Grasping the future of the digital society. *Futures* 107 (2019), 17–28.
[21] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen I. Kat. 2020. It's Time to Revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*, Anirudh Badam and Vijay Chidambaram (Eds.). USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/eytan
[22] Ian Foster, Carl Kesselman, and Steven Tuecke. 2001. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications* 15, 3 (2001), 200–222.
[23] Apache Foundation. [n. d.]. Apache Beam. https://beam.apache.org/
[24] F Gens. 2014. Worldwide and Regional Public IT Cloud Services.
[25] Robert Grandl, Arjun Singhvi, Raajay Viswanathan, and Aditya Akella. 2021. Whiz:{Data-Driven} Analytics Execution. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 407–423.
[26] Christian Grimme, Joachim Lepping, Alexander Papaspyrou, Philipp Wieder, Ramin Yahyapour, Ariel Oleksiak, Oliver Wäldrich, and Wolfgang Ziegler. 2008. Towards a standards-based grid scheduling architecture. In *Grid Computing*. Springer, 147–158.
[27] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX*

*Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 845–861. https://www.usenix.org/conference/osdi20/presentation/hadary

[28] Michael Haenlein and Andreas Kaplan. 2019. A brief history of artificial intelligence: On the past, present, and future of artificial intelligence. *California management review* 61, 4 (2019), 5–14.

[29] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 588–604. https://doi.org/10.1145/3477132.3483542

[30] M Schopf Jennifer. 2004. 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* (2004), 15–24.

[31] Fredy Juarez, Jorge Ejarque, and Rosa M Badia. 2018. Dynamic energy-aware scheduling for parallel task-based application in cloud computing. *Future Generation Computer Systems* 78 (2018), 257–271.

[32] Nakku Kim, Jungwook Cho, and Euiseong Seo. 2014. Energy-credit scheduler: an energy-aware virtual machine scheduler for cloud systems. *Future Generation Computer Systems* 32 (2014), 128–137.

[33] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology* 51, 1 (2009), 7–15.

[34] Ken Leoni. [n. d.]. Heroix: Maximizing VMware Performance and CPU Utilization. https://www.heroix.com/blog/vmware-vcpu-over-allocation/

[35] Don Lipari. 2012. The slurm scheduler design. In *SLURM User Group Meeting, Oct*, Vol. 9. 52.

[36] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, Dawn Leaf, et al. 2011. NIST cloud computing reference architecture. *NIST special publication* 500, 2011 (2011), 1–28.

[37] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. 2011. Performance and energy modeling for live migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*. 171–182.

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

[39] Fabian Mastenbroek, Georgios Andreadis, Soufiane Jounaid, Wenchen Lai, Jacob Burley, Jaro Bosch, Erwin Van Eyk, Laurens Versluis, Vincent Van Beek, and Alexandru Iosup. 2021. OpenDC 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 455–464.

[40] Somnath Mazumdar, Daniel Seybold, Kyriakos Kritikos, and Yiannis Verginadis. 2019. A survey on data storage and placement methodologies for cloud-big data ecosystem. *Journal of Big Data* 6, 1 (2019), 1–37.

[41] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 69–84.

[42] Paul Owen. [n. d.]. Slurm Used on the Fastest of the TOP500 Supercomputers. https://www.prweb.com/releases/2012/11/prweb10149109.htm. Accessed: 2023-03-18.

[43] Platform9. [n. d.]. Resource Overcommitment. https://platform9.com/docs/openstack/infrastructure-resource-overcommitment

[44] Kavitha Ranganathan and Ian Foster. 2002. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 352–358.

[45] Tech. Rep. 2022. *2022 Leadership Vision for Infrastructure and Operations*. Gartner.

[46] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. 199–212.

[47] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek (Eds.). ACM, 351–364. https://doi.org/10.1145/2465351.2465386

[48] Siqi Shen, Alexandru Iosup, Assaf Israel, Walfredo Cirne, Danny Raz, and Dick Epema. 2015. An availability-on-demand mechanism for datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 495–504.

[49] Siqi Shen, Vincent van Beek, and Alexandru Iosup. 2015. Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*. IEEE Computer Society, 465–474. https://doi.org/10.1109/CCGrid.2015.60

[50] Karnam Sreenu and M Sreelatha. 2019. W-Scheduler: whale optimization for task scheduling in cloud computing. *Cluster Computing* 22, 1 (2019), 1087–1098.

[51] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the Condor experience. *Concurr. Pract. Exp.* 17, 2-4 (2005), 323–356. https://doi.org/10.1002/cpe.938

[52] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.

[53] Alexey Tumanov, James Cipar, Gregory R Ganger, and Michael A Kozuch. 2012. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the third ACM Symposium on Cloud Computing*. 1–7.

[54] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[55] John Wilkes. 2011. More Google cluster data. Google research blog. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html..

[56] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

[57] Yanlong Zhai, Jude Tchaye-Kondi, Kwei-Jay Lin, Liehuang Zhu, Wenjun Tao, Xiaojiang Du, and Mohsen Guizani. 2021. Hadoop perfect file: A fast and memory-efficient metadata access archive file to face small files problem in hdfs. *J. Parallel and Distrib. Comput.* 156 (2021), 119–130.

[58] Chao Zheng, Ben Tovar, and Douglas Thain. 2017. 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*. IEEE Computer Society / ACM, 130–139. https://doi.org/10.1109/CCGRID.2017.9

[59] Qiang Zheng, Kan Zheng, Haijun Zhang, and Victor CM Leung. 2016. Delay-optimal virtualized radio resource scheduling in software-defined vehicular networks via stochastic learning. *IEEE Transactions on Vehicular Technology* 65, 10 (2016), 7857–7867.