D4.1 – Analysis of State-of-the-Art Workload Managers for System-Level Scheduling

Document Properties

Contract Number	101177590
Contractual Deadline	M6 (30.11.2025)
Dissemination Level	Public
Nature	Report
Author(s)	P.-F. Dutot (UGA)
Contributor(s)	L. Alonso (BSC), E. Arima (TUM), B. Bzeznik (UGA), A. Faure (RYAX), Y. Georgiou (RYAX), S. Happ (PARTEC), D. Huber (TUM), S. Krempel (PARTEC), S. Maloney (FZJ), S. Mimouni (RYAX), K. Missoh (LXP), A. Monterubbiano (CINECA), S. Pickartz (PARTEC), P. Pochelu (LXP), M. Rauh (PARTEC), O. Richard (UGA), N. Triantafyllis (NTUA), O. Vysocký (IT4I@VSB)
Reviewers	C. von Elm (TUD), U. Sinha (FZJ)
Date	28.11.2025
Keywords	SEANERGYS, HPC, Exascale, Software, Energy Efficiency
Status	Submitted
Release	1.0

The SEANERGYS project receives funding from the European High Performance Computing Joint Undertaking (JU) under grant agreement no 101177590. The JU receives support from the European Union's Horizon Europe research and innovation programme and Czechia, France, Germany, Greece, Italy, and Spain. Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or of the granting authority. Neither the European Union nor the granting authority can be held responsible for them.

Document Status Sheet

Release	Date	Author, Organisation	Description of Changes
1.0	28.11.2025		Final version submitted to EuroHPC JU

Table of Contents

List of Tables

Executive Summary

As one of the three main pillars of the SEANERGYS project, the Dynamic Scheduling and Resource Management system (DSRM) system is the primary contribution of Work Package 4 (WP4). Developing a DSRM system is complex, and a four-year time frame calls for building on existing foundations rather than starting from scratch. Therefore, the project will build on proven, mature system-software components that can be extended to support the SEANERGYS objectives. This Deliverable evaluates three candidate schedulers, namely Slurm, Flux, and OAR, based on their scalability, active development communities, and suitability for long-term maintenance.

This document outlines the technical requirements for the DSRM system, categorising them into critical, high importance, normal, and other categories. These requirements are aligned with the use cases described in SEANERGYS Deliverable 1.2 (D1.2) [1] and serve as a foundation for the schedulers feature analysis. While mostly focused on technical considerations, the Deliverable also presents legal and organisational aspects to ensure the long-term viability of the SEANERGYS Software (SW).

1 Introduction

As presented in the call for proposal `HORIZON-EUROHPC-JU-2023-ENERGY-04-01` [2], a Dynamic Scheduling and Resource Management system (DSRM) must enable dynamicity and malleability across all layers of the Software (SW) stack whilst accommodating different workload types simultaneously, such as rigid, moldable, malleable, and evolving workloads. Such a solution should also support constrained operation, particularly with regard to power capping at different levels, including global, job and node levels. Additionally, the solution should facilitate the co-scheduling of workloads that share heterogeneous resources on the same node in the most efficient manner. It should also incorporate effective profiling and monitoring mechanisms to enable the efficient combination of CPU-, memory-, network-, and I/O-bound workloads on the same node.

The SEANERGYS objective of building a DSRM is central to Work Package 4 (WP4). As mentioned in the Description of the Action (Annex 1 of the Grant Agreement), the introduction of WP4 states that existing proven and mature system software components that are suitable to be extended to support the SEANERGYS objectives will serve as the basis for further development. Therefore, the necessity arises to build on system SW components which are both actively being developed and mature enough to be used in production. Moreover, an assessment of these SW components should also include their extensibility to support the SEANERGYS objectives, both technically in our development platform but more importantly also as a permanent part of a long-term SW project that will be maintained for years to come.

1.1 Potential Solutions for Realising the DSRM

From all the existing solutions, the three which fit the requirement of maturity and continued development are Slurm, Flux, and OAR. A short description of these system-level schedulers is provided below. Other solutions such as Torque and OpenPBS derivatives are legacy systems seeing limited community support and are not as actively developed. Therefore, those were excluded from this state-of-the-art analysis.

1.1.1 Slurm

Slurm is the most-widely deployed open-source workload manager in High Performance Computing (HPC). It is used by over half of the world's largest supercomputers [3]. Slurm's architecture is modular and plugin-based, supporting advanced features such as allocation of Graphics Processing Units (GPUs), fault-tolerant daemons, and database-backed accounting. In practice, Slurm has demonstrated extreme scalability, and is used on five of the top ten largest machines in the last Top500 [4] ranking. This high scalability, combined with an active development community and broad adoption, make Slurm a natural contender.

1.1.2 Flux

Flux is a next-generation, hierarchical Resource and Job Management System (RJMS) developed by Lawrence Livermore National Laboratory (LLNL) and collaborators to address Exascale challenges. Flux

departs from the single-master model by allowing a tree of schedulers: each Flux job is a nested instance that can itself launch sub-jobs under its own scheduling policy. This enables new resource types and multi-level scheduling strategies. Its architecture makes Flux well-suited to complex workflows (e.g. large ensembles, co-scheduled MPI & analytics jobs) that traditional schedulers struggle to handle.

1.1.3 OAR

OAR is an open-source HPC batch scheduler originally developed for clusters in the French national grid (e.g. Grid5000). It is designed to be versatile and production-ready. OAR provides fine-grained control (e.g. Central Processing Unit (CPU)-set allocation, timesharing, container jobs) and advanced features (job suspend/resume, checkpointing, nested reservations) that support heterogeneous workloads. The OAR project remains under active development with a new version 3.0 released in July 2025.

1.2 General Requirements for Job and Resource Management

In SEANERGYS, the DSRM system needs to go beyond the functionalities provided by the three solutions as of today. The main objective of this report is therefore to look into the advanced requirements for the new functionalities of the DSRM system to be developed in SEANERGYS and the status of the three candidate solutions with respect to these requirements.

There are many other operational requirements which are essential to users and system administrators of HPC environments. Let us now have an quick overview of these common functionalities and briefly describe how they are addressed in Slurm, Flux, and OAR:

Scheduling Determine when and on which resources jobs should be executed and implement scheduling policies (e.g. First In, First Out (FIFO), backfilling, priorities, quotas).

Resource Allocation Reserve nodes, CPUs, GPUs, memory, and other resources.

Isolation Ensure isolation between users and their jobs (unless co-scheduling is possible).

Job Submission and Management Provide submission interfaces and allow users to monitor, cancel, or modify jobs.

Queue and Partition Management Separate resources based on specific criteria (groups, priorities, time limits, node types, etc.).

Monitoring and Accounting Collect statistics on resource utilisation and generate logs for auditing and accounting purposes (e.g. CPU-hour usage).

Support for Parallelism and Distributed Dispatching Integrate parallel execution systems (Open Multi-Processing (OpenMP), Message-Passing Interface (MPI), etc.) and manage the execution of distributed jobs across multiple nodes.

Security Manage user permissions and execution environments (e.g. namespaces, Control Groups (cgroups)).

Interoperability with HPC Environments Integrate with MPI, Singularity, containers, and environment modules.

From an architectural and SW design perspective, the resource managers have slightly different structures, but aim for the same overall goals, as elucidated in the following list with the different approaches (where applicable) taken by the three codes:

Command Line Interface Provide submission interfaces (e.g. `sbatch`, `flux submit`, or `oarsub`) and allow users to monitor (`squeue`, `flux jobs`, or `oarstat`), cancel (`scancel`, `flux cancel`, or `oardel`), or modify jobs.

Master/Agent (or Controller/Node) Architecture A central server (controller) manages scheduling and job queues while agents (daemons) run on compute nodes to execute jobs and report their status. Examples are

- Slurm → `slurmctld` (controller) + `slurmd` (node daemon)
- Flux → `flux-broker` (hierarchical but conceptually similar)
- OAR → `oaradmin`

Internal Communication System Use Remote Procedure Call (RPC) or message-based protocols (often via UNIX or Transmission Control Protocol (TCP) sockets) and ensure reliability and synchronisation among components.

Database or Persistence Structures Store job states, configuration data, and usage logs. This is supported by the following technologies in the three system-level schedulers:

- Slurm → MySQL/MariaDB database (`slurmdbd`)
- Flux → Hierarchical internal database (Flux Comms + Key Value Store (KVS)), backed to a local SQLite database file
- OAR → PostgreSQL database

Composability and Modularity Support plugins for scheduling, authentication, job launching, etc.

- Slurm: Numerous plugins (`sched`, `auth`, `proctrack`, `jobcomp`, etc.).
- Flux: Fully modular and recursive architecture (each Flux instance can manage its own subsystem).
- OAR: Fully modular

Beyond these technical considerations, the long-term inclusion and maintenance of SEANERGYS SW development also depend on other aspects. For this reason Chapter 2 gives an overview of the current status of the considered system-level schedulers with respect to legal, organisational, and deployment aspects. Chapters 3 to 6 summarise the technical requirements that we identified to meet the project's objectives while being in-line with the use-cases presented in [1]. These sections introduce the different requirements categorised into critical, high importance, normal, and other; and present a state-of-the-art analysis for the considered system-level schedulers respectively. This Deliverable concludes with Chapter 7 and serves as input to Task 1.4 (Tk1.4) defining the global system architecture.

2 Legal, Organisational, and Deployment Status

An important goal of the project is to provide a durable improvement for HPC environments, by providing a long lasting DSRM system. Our SW solution is designed to last longer than the latest systems on which it will be first deployed during this project's lifetime, and to be adaptable and extendable for future systems. In order to assess the potential durability of these developments, we need to have an idea of how these developments can be adopted by the parent project.

2.1 Slurm

Organisation SchedMD, Headquarters Located in Lehi, UT, USA [5].

Deployment status Over half of the large-scale supercomputers are using Slurm.

Legal Slurm is distributed under the GNU General Public License (GPL) [6].

After multiple direct discussions between the SEANERGYS project coordinator and SchedMD, it has become clear that the only viable solution for achieving long-term integration of our developments with Slurm would be to fork Slurm into a new separate project. We would then work to maintain and promote this new project independently of the Slurm versions promoted by SchedMD. More specifically, SchedMD does not provide assurances regarding changes to the Slurm SW structure or interfaces, which could lead our fork to become incompatible with their version. This incompatibility might result in our inability to receive essential security updates and would pose a significant risk to our project's long-term viability.

2.2 Flux

Organisation Mainly developed by LLNL. Currently working on joining the High Performance Software Foundation (HPSF).[1]

Deployment status System-level for at least six systems on the Top500, including El Capitan [7]. Also used at various sites in user-mode (not as the managing scheduler).

Legal Flux is distributed under the GNU Lesser General Public License (LGPL)-3.0. [8].

The main developers of the Flux project expressed a strong interest in incorporating our developments in their main repository to add new functionalities to Flux. To create a larger community of developers, Flux is taking the steps required to join the HPSF, a part of the Linux Foundation (LF). These steps include publishing a publicly available governance document, which must include the project's technical leadership and roles, and the path to contribute code to the project.

[1] According to discussions with the Flux lead Tom Scogland and Daniel Milroy.

2.3 OAR

Organisation Mainly developed at Université Grenoble Alpes (UGA).

Deployment status Various systems in France, including research clusters, e.g. Grid'5000 [9] and operational clusters, e.g. GRICAD [10].

Legal OAR is distributed under the GPL-2+ license. [11]

The OAR project started in 2003 as part of the Grid'5000 French infrastructure project. The major version 2.0.0 was first released in 2007, with multiple minor versions released every year up to 2.6.1 in March 2025, and the new major version 3.0.0-1 was released in July 2025. The project leader and the most recent developers are actively contributing to the SEANERGYS project. As in the Flux project, an effort is underway to structure and formalise the project's technical leadership and roles, with the goal of joining the HPSF. As Grid'5000 becomes integrated in the new Scientific Large Scale Infrastructure for Computing/Communication Experimental Studies - Research Infrastructure (SLICES-RI) project, OAR will continue being used in cutting edge HPC environments.

3 Critical Requirements

The requirements in this section define the essential capabilities without which the DSRM system cannot operate effectively. Dynamic scheduling (Section 3.1) and energy-aware policies (Section 3.2) provide the core algorithmic foundation, while co-scheduling (Section 3.3) enables coordinated resource allocation across related jobs. The interaction mechanisms (Section 3.4) establish the communication infrastructure necessary for the scheduler to coordinate with applications and runtime systems. Together, these form the non-negotiable baseline for a functional DSRM.

3.1 Dynamic Scheduling

A DSRM system is capable of adapting scheduling decisions dynamically in response to changes in the workload, availability of resources, and the overall conditions and state of the system. External factors including—but not limited to—power and energy consumption limits must be integrable into scheduling decisions.

	Time of decision	
Decision maker	**At submission**	**During execution**
User	Rigid	Evolving
System	Moldable	Malleable

Table 1: Classification of Job Types Based on Decider and Decision Time for Resource Allocations Adapted from Table 2 in [12].

A DSRM system can adapt resource allocations of both already running jobs and jobs that are in the job queue(s) based on the (dynamic) resource demand of the jobs and the state of the system. For doing so, scheduling algorithms need to take into account scheduling objectives along with their policies and constraints. They have to deal with different types of jobs [12, 13] (see also Table 1):

Rigid jobs have a static resource allocation that does not change during their lifetime and is explicitly specified by the user at submission time. Rigid jobs are unable to change their resource allocation dynamically.

Moldable jobs can be started with different acceptable resource allocations provided by the user (e.g. by defining viable ranges). Once started with a certain resource allocation, moldable jobs cannot change their resource allocation dynamically.

Evolving jobs can change their resource allocation dynamically, based on their own decisions and requests to the scheduling system. Evolving jobs evaluate their resource usage by themselves and decide on resource allocation changes based on their own evaluation.

Malleable jobs change their resource allocation dynamically based on decisions taken by the DSRM system. Malleable jobs react to resource allocation change requests made by the DSRM system and adapt themselves accordingly to fit to the new resource allocation.

Dynamic scheduling will be the heart of the SEANERGYS DSRM system and hence it is a critical requirement for the project.

3.1.1 Slurm

These days, Slurm is the most widely used resource management system in HPC centres and with its default back-fill scheduler it is perfectly capable to handle *rigid* and *moldable* (see Section 5.2.1) jobs, homogeneous, as well as heterogeneous ones. It is a purely static scheduler and does not modify the resources assigned to jobs while they are running. Hence, Slurm does currently not support *evolving* and *malleable* jobs.

Technically, the scheduler in Slurm is implemented as a plugin and can be exchanged. But each replacement has the same constraints as the original scheduler and cannot enhance the capabilities how to handle jobs, e.g. there is no way to enlarge an allocation. Another problem is the distribution of information. In general, all job information is managed on the first node of an allocation and only a subset is shared with the Slurm controller where the scheduler resides. So the first step to enable dynamic scheduling decisions needs to extend the protocol to get all the information currently distributed over the job leaders to one place.

3.1.2 Flux

The Flux framework is developed largely as a collection of interchangeable plugins around a central daemon (known as a *Broker*), including key components such as the scheduler. This should ease development by allowing a choice between integrating project features (e.g. scheduling algorithms) directly into the existing scheduler plugin, or creating separate plugins that can be swapped in for different use cases, as the need dictates.

Other plugins can be used to create hooks at different points within a job's life cycle to affect decision-making and integrate external information, e.g. from the Artificial Intelligence Data Analytics System (AIDAS). For instance, in order to properly inform the scheduler's decision process for *moldable* and *malleable* jobs, models could be run to augment a job request at submission time, with the information then used (and perhaps updated) later during (re-)scheduling events.

As explained further in Section 3.4.2, the communication protocol of Flux should be very flexible to extend in order to provide the necessary functionality for *evolving* jobs to communicate their desired allocation changes to the scheduler, or vice-versa in the case of *malleable* jobs.

3.1.3 OAR

The OAR framework supports *moldable* jobs, where acceptable resource allocations are expressed as a list of individual *rigid* allocation requests, each associated with a wall time. This list is expected to remain short—typically fewer than ten requests—to avoid overloading the scheduling process. The selection mechanism is straightforward: each allocation request is evaluated successively, and the one with the earliest expected end time is selected. Additional constraints, such as quotas and priorities, are also applied during the scheduling process. These selections are re-evaluated at each scheduling round.

Support for *evolving* and *malleable* jobs cannot be viewed solely as a specific scheduling variant. A key reason is that dynamicity affects not only the scheduling component but also the interactions with low-level job execution mechanisms, such as manipulating the application's operating system process environment (e.g. cgroups) and interfacing with application middleware (e.g. Process Management Interface for Exascale (PMIx) for MPI applications).

The experimental approach explored in OAR introduces a level of job indirection by modelling *evolving* or *malleable* jobs as a sequence of *rigid* jobs. More precisely, when a job needs to be modified, it is replaced by a new one while preserving the application's processes on the resources that remain allocated, and by appropriately creating or terminating processes on newly assigned or released resources.

Evolving and *malleable* jobs are thus represented by a job envelope composed of successive *rigid* jobs, which together capture the history of dynamic scheduling steps. The main advantage of this approach is that it minimises the impact and required modifications within the job management and execution chain.

The upper-level scheduling components for *evolving* and *malleable* jobs remain largely unexplored and could be further developed through the implementation of dedicated scheduling and allocation functions using the existing plugin mechanisms.

3.2 Energy and Power Aware Scheduling

Energy-aware and power-aware scheduling has emerged as a key capability for modern RJMSs, enabling them to make informed scheduling and allocation decisions that balance performance objectives with energy efficiency and power control.

At its core lies the integration of hierarchical power-capping mechanisms that regulate energy usage across possibly multiple levels of granularity: system-wide, rack-level, and node-level. Such multi-layer power regulation ensures adherence to facility-wide power budgets while providing local safeguards against transient peaks. These hierarchical controls form the foundation for coordinated and adaptive scheduling strategies, where the DSRM dynamically allocates jobs and resources within predefined power caps to sustain performance under energy constraints.

Complementing these mechanisms are control interfaces along with optimisation strategies, such as energy-to-solution and performance-per-watt, which translate power policies into operational actions. Through interfaces such as Dynamic Voltage and Frequency Scaling (DVFS) and GPU power limit adjustments, an energy-aware scheduler can actively modulate hardware behaviour in response to real-time system states. Beyond heterogeneous architectures, such optimisation remains relevant even in clusters of homogeneous nodes configured with fixed but distinct CPU frequencies. By exploiting these subtle differences in energy-performance characteristics, schedulers can improve overall efficiency, minimising energy-to-solution while maintaining throughput and respecting power caps.

To enable energy-aware scheduling, the DSRM must incorporate tight integration with real-time telemetry and monitoring systems. This integration should allow the scheduler to continuously retrieve and interpret instantaneous power consumption and energy efficiency metrics, using this information as part of the decision-making process for job placement and resource allocation. In addition, accounting mechanisms are required to record per-job and per-user energy usage, supporting energy-based quota or billing models. Scalability is equally essential, ensuring that the DSRM can process high-frequency telemetry data and

enforce control actions efficiently at exascale. Furthermore, extensibility must be a core design feature, allowing the seamless development and integration of new energy and power aware policies, models, and plugins through modular interfaces.

Finally, adapted energy and power aware evaluation methodologies and tools are vital to enable reproducible scheduling experimentation, allowing new heuristics, energy models, and experimental policies to be tested and validated systematically.

3.2.1 Slurm

Currently, Slurm supports energy job accounting (the scheduler is aware of job power consumption, supporting a few common power monitoring systems) and limited power management capabilities (e.g. the selection of different CPU governors as well as frequency settings for GPUs [14]). However, a combination of these capabilities to adapt a power or energy-aware scheduling algorithm has not been implemented/integrated. There have been research studies in extending Slurm to support power-aware strategies [15, 16, 17].

3.2.2 Flux

Flux provides powerful building blocks for energy-aware and power-aware scheduling through its *Fluxion* graph-based resource model. Due to its extensible architecture and by representing power as a flow resource, *Fluxion* allows hierarchical modelling at multiple levels—power-unit, node, rack, or cluster—enabling fine-grained control and dynamic allocation adjustments. This flexibility supports dynamic scheduling policies tailored to diverse power-aware strategies.

Moreover, two power-aware tools have been introduced: the *flux-power-monitor* and the *flux-power-manager* [18]. The *flux-power-monitor* module provides vendor-neutral, low-overhead power telemetry, collecting detailed, per-job power statistics for components like CPUs and GPUs across diverse HPC systems. The real-time data from *flux-power-monitor* can feed into custom scheduler policies to optimise for objectives like energy-to-solution or performance-per-watt. The *flux-power-manager* module (not yet public) complements this by enabling hierarchical power management, handling dynamic power capping at the cluster, job, and node/GPU levels, which can be facilitated to implement policies, e.g. to reduce energy consumption compared to vendor-specific static policies.

The key SW used by these tools is *Variorum*, which is an open-source, vendor-neutral library that provides a unified way to monitor and control power and performance across diverse CPU and GPU architectures from vendors such as Intel, AMD, IBM, ARM, and NVIDIA. It enables power telemetry and capping, integrating easily with higher-level system SW such as schedulers and runtime systems. In practice, the Flux integration leverages three *Variorum* Application Programming Interfaces (APIs) to provide essential functions for vendor-neutral telemetry, node-level power capping, and GPU power capping. Whether the SEANERGYS project would use *Variorum* or a different SW is a decision beyond the scope of this Deliverable, but the integration demonstrates how energy-aware scheduling can be realised with Flux regardless of the backend ultimately used.

3.2.3 OAR

In the previous European REGALE project, energy and power-aware scheduling support was explored for OAR [19]. A simple approach was considered, where power is treated as a new type of resource. These power resources can be declared and managed in various ways. One approach involves managing power at a coarse-grained level, which can be applied at different levels of the hierarchy, e.g. node, rack, partition, or system-wide.

It is expected that OAR will interact with dedicated monitoring and fine-grained energy or power management systems, such as Energy Aware Runtime (EAR), to refine job allocations and possibly trigger energy or power-aware actions, such as waking up or suspending nodes. Energy monitoring is also delegated to external and specialised tools.

3.3 Co-scheduling

Co-scheduling is a resource management paradigm that assigns multiple jobs (or tasks in a workflow) with complementary resource requirements simultaneously on the same node, i.e. implementing a spatial sharing of the compute nodes. The objective is to improve the total system throughput or energy efficiency by mitigating waste of idle resources inside a compute node. The resource waste is becoming more and more serious, driven by recent architectural trends, i.e. compute nodes are becoming fatter and more heterogeneous. As examples, memory-, network-, or Input/Output (I/O)-bound jobs significantly waste compute resources, or GPU-bound jobs typically cannot fully utilise CPU resources or even the full GPU compute inside a node. SEANERGYS aims to enable high-throughput, fair, and flexible co-scheduling and node sharing mechanisms, targeting both homogeneous CPU-only systems as well as CPU+GPU heterogeneous nodes.

As a first step, the following functionalities are required to enable co-scheduling without any throughput, performance, and fairness optimisations:

CPU Node Sharing Functionality RJMSs must provide a functionality to co-locate multiple jobs on the CPUs of the same node in a space-sharing manner using user-specified information.

CPU+GPU Node Sharing Functionality RJMSs must provide a functionality to co-locate CPU-only and GPU jobs or multiple GPU jobs on the same node in a space-sharing manner using user-specified information.

With these basic functionalities, users can designate the necessary node resources (e.g. core count, memory capacity, GPU count) at job submission time, and the scheduler can co-schedule multiple jobs based on the user requests, while complying with node resource limits using a simple policy such as basic First Come, First Served (FCFS) scheduling. These finer-grained node resource assignment mechanisms can also be used by the workflow management tools (or they can internally partition the node resources inside a job). Overall, these functionalities are widely supported in modern job schedulers.

As this basic approach, however, does not take into account several important factors, such as the interference effect, we further evaluate the following requirements for enabling more advanced co-scheduling:

Hardware-level QoS Control Functionality Modern commodity hardware typically offers Quality of Service (QoS) knobs on node shared resources (e.g. shared cache partitioning, memory bandwidth partitioning, and GPU partitioning). These knobs can significantly assist with mitigation of co-scheduling interference, especially when a memory-intensive workload is running.

Job Mix Optimisation In addition to the node sharing and partitioning functionalities, the DSRMs should be able to optimise the job mix, while taking into account the interference effects and job affinities. For instance, the interference can be smaller when mixing CPU- and memory-intensive workloads. Also, for a multi-node job, a significant load imbalance across nodes can happen when different sets of jobs are mixed among them.

Node Partitioning Optimisation For a given job mix co-located on the same node, the DSRMs should be able to optimise the partitioning setup and node resource assignment. This decision-making should be based on the model(s) developed in WP3, and this partitioning decision can be made at runtime in a reactive manner.

Flexibility and Extensibility In addition to these advanced optimisations, the interoperability with other key use cases is also important and our ultimate goal is to enable co-scheduling under moldable job scheduling, dynamic resource management, and power capping.

3.3.1 Slurm

CPU Node Sharing Functionality is supported by Slurm's consumable resource allocation plugin (`select/cons_res`). This plugin allows fine-grained compute and/or memory resource assignments, which enables multiple jobs to share the same nodes. As an example, by using the `CR_CPU_Memory` option, users are asked to request the compute resources at the granularity of logical core (or thread) as well as the memory capacity, in addition to the node count. The scheduling plugin can then launch a job on the nodes where one or more jobs are already running if the space permits.

GPU Node Sharing Functionality is supported by the `select/cons_tres` and Generic Resource (GRES) plugins of Slurm. As an example, the former sets up the number of CPUs per GPU in the configuration file (users can request more CPUs than that), and it accepts co-scheduling if sufficient numbers of CPU and GPU resources are available for a job.

Hardware-level QoS Control Functionality is currently not supported by Slurm, e.g. there is no possibility to control hardware-level QoS knobs for CPUs such as Intel Cache Allocation Technology (CAT) [20] and Intel Memory Bandwidth Allocation Architecture (MBA) [21], which requires updating the plugin and/or epilogue/prologue scripts to:

1. let the user request specific cache slices and bandwidth (or have them determined by the scheduler and/or node manager using an interference model), and
2. to communicate with the node Model-Specific Register (MSR) control mechanism or a management library, e.g. as part of Intel Resource Director Technology (RDT) [22]).

As for GPUs, Slurm supports NVIDIA Multi-Process Service (MPS) [23] and Multi-Instance GPU (MIG) features; however, these are not interoperable, and changing the MIG partitioning setup requires relaunching the node daemon.

Job Mix Optimisation Currently, the plugins mentioned above do not co-schedule jobs based on the prediction of the interference effects. This requires an extended plugin to characterise a given set of jobs, to interact with the model repository developed in WP3, and then to select a best set of nodes to minimise the interference effects introduced by co-scheduling.

Node Partitioning Optimisation The plugins mentioned above also do not optimise the partitioning setup but rely fully on the user-specified information. To enable this feature, it is required for the node daemon to obtain the runtime job characteristics, input them to a model provided by the model repository, and then determine the partitioning setup.

Flexibility and Extensibility Slurm does not support any functionalities to co-schedule moldable or malleable jobs by default. Power capping under co-scheduling is functional but not performant as well.

3.3.2 Flux

CPU Node Sharing Functionality Flux does not offer explicit structured CPU node sharing, such as half-socket allocation, in its vanilla configuration. While fine-grained resource requests in the Job Specification (jobspec) YAML format allow users to specify numbers of cores and memory for basic space-sharing on partially utilised nodes when resources are available, and task pinning to specific cores is supported via options (e.g. `flux-shell --cpu-affinity`), advanced socket-bound constraints require custom development. However, Flux provides hooks for this, including configuration of the system's full resource topology in the `flux-config-resource` file (using properties to label and partition resources) and the Flux scheduler (i.e. Fluxion) to express and match such requests via the jobspec.

CPU+GPU Node Sharing Functionality Flux supports fine-grained resource specification through its jobspec model, including CPU and GPU affinities. When GPU devices are described in the jobspec and associated resource labels are available, Flux sets task-to-device mappings (e.g. via Compute Unified Device Architecture (CUDA) visibility and GPU affinity controls) and enforces CPU affinity accordingly. As a result, CPU-only and GPU jobs, as well as multi-GPU jobs, can be co-located on the same node as long as users (or higher-level tools) specify the corresponding resource requirements. In practice, co-scheduling on heterogeneous nodes is therefore supported, but depends on explicit jobspec annotation.

Hardware-level QoS Control Functionality Currently, Flux does not expose hardware QoS mechanisms (e.g. Intel CAT/MBA or GPU partitioning primitives) as natively schedulable resources. Enabling cache or memory bandwidth partitioning requires integrating node-side controls (e.g. Intel RDT) through Flux job prologue/epilogue scripts or node-resident agents. GPU partitioning can be supported when the underlying platform exposes capabilities (e.g. NVIDIA MIG), though reconfiguration typically requires privileged node management outside the scheduling path. In summary, Flux can coordinate hardware QoS actions, but doing so requires external hooks and coordination logic rather than relying on built-in scheduler capabilities.

Job Mix Optimisation Flux's scheduling architecture separates the resource description from the scheduling policy, allowing operators to introduce customised logic to influence job placement decisions. While Flux does not natively account for interference effects or workload affinities when co-scheduling jobs, these behaviours can be implemented by extending the scheduler component (e.g. via queue policies in Fluxion) to classify jobs, consult external performance and interference models, and prioritise complementary workloads (e.g. pairing CPU-bound with memory-bound applications). Thus, interference-aware job mix optimisation is feasible, but requires custom policy development and integration with external modelling components.

Node Partitioning Optimisation Flux allocates node-level resources according to job the specification and resource graph matching, without automatically determining optimal intra-node partitioning for co-located workloads. To support dynamic optimisation—such as adjusting core, memory, or GPU partitioning based on runtime behaviour—Flux must be extended with mechanisms to collect performance metrics, query decision models, and apply enhanced resource assignments. The underlying architecture (jobspec graph, resource model, and extensible scheduler) supports such adaptive behaviour, but it is not available by default and requires additional decision and enforcement components.

Flexibility and Extensibility Flux's graph-based Fluxion resource model supports moldable jobs (with user-defined min/max resources) and malleable jobs (allowing dynamic resource scaling during runtime). While moldability is fully supported, full malleability across diverse workloads is still under development. Its elastic and hierarchical design decouples resource representation from scheduling policies, enabling advanced scheduling extensions such as co-scheduling, adaptive resource management, and performance-class-aware scheduling (e.g. variation-aware policies). Additionally, Flux's modular architecture, with Python APIs, configurable queues, and hierarchical scheduling, ensures high extensibility—facilitating custom co-scheduling policies.

3.3.3 OAR

CPU Node Sharing Functionality OAR enables multiple jobs to share the same node's CPUs through fine-grained allocation, using CPU sets to isolate and assign specific CPU cores and memory amounts. Users request these resources directly with the `oarsub` command, and the scheduler assigns portions of the node's resources, allowing co-location on partially used nodes as long as space remains available. For example, a "half socket allocation approach", where resources are assigned in the granularity of half the number of cores of a CPU, can be easily applied by adding a custom `halfcpu` property to the OAR database, which logically splits each CPU into two halves. This component can then function as an integration point for the evolution of the co-scheduler's logic.

CPU+GPU Node Sharing Functionality OAR implements GPU support via hierarchical resource management, with GPU devices represented in the resources table using attributes such as `gpu` and `gpudevice`. This mechanism allows simultaneous execution of CPU-only and GPU workloads, including multiple GPU jobs per node. Scheduling decisions jointly consider CPU, memory, and GPU availability, with device cgroups ensuring isolation and facilitating node-level co-scheduling.

Hardware-level QoS Control Functionality OAR does not provide built-in functionality to control hardware level QoS knobs such as Intel CAT/MBA for cache and memory bandwidth partitioning. Implementing this would require custom extensions to interact with hardware management libraries (e.g. Intel RDT tools) and the introduction of new resource management hooks in OAR in order to be able to set partitions based on job requests or scheduler decisions. For GPUs, basic isolation through cgroups is currently supported. In addition, OAR provides resource-management hooks that could be leveraged to support NVIDIA MIG-based partitioning.

Job Mix Optimisation OAR does not include mechanisms to co-schedule jobs based on predictions of interference effects or job affinities. This would require developing custom scheduler plugins or admission rules that characterise jobs, integrate with external interference models (e.g. from WP3), and select node allocations to minimise race conditions, such as pairing CPU-bound with memory-bound workloads.

Node Partitioning Optimisation OAR allocates resources strictly according to user-defined requests, using mechanisms such as cpusets and property filters, without automatically optimising node partitioning based on application runtime behaviour. Supporting this would require augmenting the co-scheduler to gather job-related metrics, interface with WP3 models, and adjust allocations on the fly. Additionally, co-scheduling development could be improved by enforcing structured resource layouts, such as spreading jobs across half the CPU sockets, enabling predictable and reliable node sharing.

Flexibility and Extensibility OAR does not provide native support for co-scheduling moldable or malleable jobs. Although OAR supports moldable jobs, allowing users to provide multiple resource configurations (e.g. via the `-l` option to `oarsub`), co-scheduling requires dedicated scheduling policies or tag-based job mechanisms (such as the spread tag) to be explicitly configured.

3.4 DSRM, Runtime Environment, and Application Interfaces

SEANERGYS is dedicated to optimising resource utilisation and reducing energy consumption for diverse, real-world workload mixes on both current and next-generation supercomputing systems. Achieving these objectives necessitates the deployment of advanced resource management techniques, including dynamic scheduling, power- and energy-aware scheduling, and co-scheduling. Such techniques rely on a close integration between resource management SW and jobs or runtime systems to enable the timely and effective execution of resource management actions. Consequently, it is imperative to define standardised, efficient, and scalable communication protocols between these components.

While the *SEANERGYS Data Plane* could potentially cover parts of the required data access, many existing components already offer communication mechanisms that should be favoured to facilitate upstream integration. These include:

Component-specific protocols and command-line interfaces Many resource managers and runtime systems already provide their own protocols or Command Line Interface (CLI) commands for job submission, monitoring, and control. Leveraging these existing interfaces can enable efficient integration without requiring deep changes to either the job runtime or the resource manager.

The PMIx Standard The PMIx defines a standardised API for exchanging information between resource managers and runtime systems. It supports advanced features such as dynamic resource allocation, process coordination, and fault tolerance, making it well-suited for dynamic, energy- and performance-aware co-scheduling in SEANERGYS.

Restful APIs Some components expose REpresentational State Transfer (REST)ful APIs that allow external systems to query resource usage, submit jobs, or trigger scheduling actions over Hypertext Transfer Protocol (HTTP). These interfaces can simplify integration with web-based or distributed orchestration systems and support flexible, scalable interactions across heterogeneous compute environments.

3.4.1 Slurm

Slurm distribution provides a set of command-line utilities that allow receiving metadata about a job and the scheduler status. It includes `sinfo` (show status of nodes, partitions, and availability), `sstat` (display real-time status of running jobs/steps), or `sacct` (report job and step accounting information). For automatised interaction with these utilities JSON format is supported. Slurm has internal C libraries that expose much of the same functionality as its command-line utilities. However, these are intended for Slurm's internal use, and its APIs may change at any time.

`slurmrestd` is a daemon that exposes Slurm functionality over HTTP/Hypertext Transfer Protocol Secure (HTTPS) via a REST API. It communicates directly with the Slurm controller (`slurmctld`) or the accounting database daemon (`slurmdbd`), translating standard Slurm RPCs into JavaScript Object Notation (JSON) REST calls. In contrast to the internal libraries, the `slurmrestd` provide stable JSON API for job submission, monitoring, configuration, and accounting.

Slurm implements PMIx through a plugin architecture under `src/plugins/pmix/`. It requires Slurm to be compiled with the PMIx headers. Supported plugin types include:

- `pmix/pmix_v1` – early support (PMIx v1.x API)
- `pmix/pmix_v2` – a newer generation
- `pmix/pmix_v3`, `pmix/pmix_v4` – supported in releases from Slurm 23.x
- `pmix/pmix_none` – fallback plugin when PMIx is not used

3.4.2 Flux

Flux provides the *Flux Command Line Interface* to interact with different components of the Flux framework. Additionally, the *Flux communication protocol* provides four message types: *request* and *response* messages to enable the RPC idiom, *event* messages to enable the Publish/Subscribe (Pub/Sub) idiom, and *control* messages for internal needs. In conjunction with the use of JSON to carry message payload data, which is opaque to the communication layer itself, this makes the protocol very flexible to adapt to all the needs of the resource manager and job runtime interaction in SEANERGYS.

PMIx support in Flux is currently limited to a *PMIx shell plugin* which enables PMIx-based bootstrapping. By extending the list of implemented PMIx server callbacks, additional functionality could be made

accessible via PMIx. Flux also provides the *Flux RESTful API*, which currently supports a limited set of functionalities (for example to get information on jobs or to cancel a job).

3.4.3 OAR

The administration of OAR is primarily conducted through CLI tools and the editing of configuration files, which occasionally incorporate scriptable elements, such as admission rules. Several of the CLI utilities, including `oarstat`, `oarnodes`, and `oaraccounting`, offer an option to output data in the JSON format. This facilitates integration with users scripts and external workflow management systems.

Furthermore, OAR provides a robust RESTful API which leverages the same underlying libraries as the CLI tools. Nearly all functionalities pertinent to job submission, monitoring, and accounting are accessible via this API. Its efficacy has been well-demonstrated across several significant integration projects, notably including Grid'5000 SW ecosystem [24], CiGri [25], and Galaxy [26].

4 High Importance Requirements

These requirements significantly extend the scheduler's effectiveness and operational capabilities. Dynamic resource resizing (Section 4.1) provides the system manager with a mechanism to implement scheduling component decisions, while multi-objective optimisation (Section 4.2) enables sophisticated trade-offs between competing goals. Job event notifications (Section 4.3) establish the observability foundation required for system monitoring and external integrations. While a system could theoretically operate without these features, their absence would severely constrain practical deployment scenarios.

4.1 Dynamic Resource Allocation Resizing

While Section 3.1 focuses on dynamic scheduling policies for the different job types presented in Table 1 and dynamic resource availability, these policies rely on the dynamic resource allocation resizing capability, a distinct but complementary requirement that allows for resource adjustments of running applications, as detailed in this section. In other words, the DSRM has to be capable of shrinking and expanding the size of a resource allocation dynamically. Additionally, an exchange of resources might be necessary which corresponds to the execution of both operations (i.e. shrinking one job and expanding another) at the same time. Of course, an expansion must only take place if sufficient additional resources are available and this must be ensured by the DSRM system prior to an expansion of a resource allocation.

It is important to note that the term *resource* is not limited to the most common case where typically a *computing node* is meant. A finer-grained specification of resource requirements, i.e. with respect to CPUs, GPUs, memory, and other (hardware) requirements should be possible to consider such requirements in the DSRM system.

For the realisation of a DSRM system, an interface is required to enable dynamic resize operations for resource allocations in a reliable way without impacting the ongoing execution of the job:

Shrink operation Resources must only be de-allocated, i.e. removed from the resource allocation, once the job has ceased using the resources to be returned.

Expansion operation Additional resources must be truly available before the job is notified accordingly and will attempt to include the resources in its execution.

Resizing operations require information about available resources (for evolving jobs that want to expand) and information about requested shrink or expansion operations by the DSRM system (for malleable jobs). Such information must be made available via a DSRM interface so that the jobs can adapt and reconfigure themselves accordingly.

4.1.1 Slurm

Slurm is only prepared to shrink existing allocations and cannot let them grow. Shrinking is done using the `scontrol` command and can be done by the user. In that sense, Slurm provides partial support for evolving jobs. This is currently not used by the resource management or scheduler, but the same

mechanism should be usable from there as well, meaning that malleable jobs, that are only able to shrink could be implemented without requiring fundamental changes to the core scheduler. For supporting growing jobs, bigger efforts are required and for sure changes will be necessary that are not covered by any plugin interface provided.

4.1.2 Flux

Flux has a partial release RPC with which a subset of the allocated resources can be released back to the scheduler and made available to another job before the initial job has completed. At present this feature is used in the *housekeeping* phase at the end of a job's life cycle to avoid having nodes which get stuck during cleanup preventing the entire allocation from being freed [27]. However, it provides an existing mechanism in the communication protocol that could be used for shrink operations, where an evolving job could signal that it is finished with certain resources and the scheduler can again consider them for allocation, and this scenario is envisioned in the Flux design discussions [28].

There also exists an experimental branch[1] with a prototype implementation of an expansion operation, where a job is started on a set of nodes, and then additional nodes can be bootstrapped in at a later time during the execution. Current caveats include that the maximal number of brokers must be preconfigured at initial instance launch time, although only a single broker need actually be allocated and started, with all remaining brokers able to join later. To give the extreme example, all jobs could be launched with the full number of nodes in the system/partition set in their configuration, but only their requested subset of nodes actually allocated and launched to begin with; additional brokers could then be added later because they are guaranteed to be within the maximum number configured. This does not necessarily impact performance, but does imply some administrative/configuration overhead to ensure the correct system/partition node counts are kept updated and configured for each job. Work is also needed to properly update the resource set if the brokers do not all correspond to the same local resources, i.e. if the nodes are heterogeneous or only partially allocated (perhaps also using co-scheduling). Obviously a cleaner communication protocol for growing a resource set would be desirable, and design discussions towards this end have been mooted, as this is also a planned feature to be supported by Flux.

4.1.3 OAR

As described in Section 3.1.3, the dynamic resizing of a job is managed by creating a new job which replaces the previous one with an updated set of allocated resources. A low-level mechanism ensures that the processes of the running application are properly handled by this update. This mechanism utilises standard prologue and epilogue steps to enable customised interactions with the application, in particular for data redistribution. For processes management, prologues and epilogues manage the creation or termination of processes on newly assigned or released resources. This implementation is currently available in a specific development branch and should be included as a feature in the main branch in the next subversion release.

[1] `https://github.com/flux-framework/flux-core/pull/5184`

4.2 Multi-Objective Scheduling

Multi-Objective Scheduling (MOS) has emerged as a key capability for RJMSs, enabling coordinated decisions that jointly optimise multiple goals: energy efficiency, system productivity (utilisation/throughput), user-centric performance (average and tail slowdown), and fairness, while enforcing hard site constraints (power and memory caps, QoS/partition policies, and Service Level Objectives (SLOs)) [29, 30]. At its core is a policy and control plane that operates hierarchically (at system, workflow, job, and node levels) and integrates two pillars: monitoring/telemetry (from the CMI) and predictive analytics (from the AIDAS). This plane evaluates feasible actions under constraints and applies them in a coordinated way:

- Co-scheduling of complementary workloads (typically compute-bound with memory-bound) using explicit co-location and isolation controls.
- Dynamic resizing for moldable/malleable applications where supported, to shrink or expand allocations in response to load, queue pressure, or power corridor changes.
- Active power/frequency management under site caps (via DVFS and GPU power limits) so the system adheres to facility-wide limits while sustaining throughput and reducing energy-to-solution [30].

Complementing these mechanisms are optimisation strategies and interfaces that translate policy into operations. Priority/admission controls (like fair share, site factors) steer which jobs run first under current constraints, co-scheduling and isolation use CPU/GPU affinity and cgroup bindings to limit interference, resizing applies moldable/malleable controls to adapt parallelism, and power management uses per-node/per-device controls to cap or shape consumption. Together, these allow the DSRMto improve overall efficiency (energy-per-result or performance-per-watt) while respecting fairness and tail-latency targets.

Based on MOS, advanced algorithms and policies optimise job allocations across heterogeneous resources. These algorithms balance multiple, often conflicting goals simultaneously. Key objectives include maximising throughput, minimising energy consumption, and reducing turnaround time (total completion time of a job). They also improve resource utilisation while considering factors like workload complementarity and fairness. Unlike traditional single-objective schedulers that prioritise only one metric (e.g. shortest-job-first for speed), multi-objective approaches generate a Pareto-optimal set of solutions, allowing administrators or systems driven by Artificial Intelligence (AI) to select trade-offs based on real-time data, such as job fingerprints, power caps, or environmental constraints.

To enable MOS, the RJMS integrates tightly with real-time telemetry. This includes power, utilisation, memory, and I/O metrics. It also uses predictive models for future execution time, utilisation, phases, co-scheduling compatibility, and scalability of moldable jobs. Accounting records per-job and per-user outcomes (e.g. energy, slowdown, average power consumption) for tracking of Key Performance Indicators (KPIs), energy-aware quotas, or billing. The system must scale to high-frequency signals and enforce actions at facility scale, and remain extensible so that new multi-objective policies, models, and plugins can be added through modular interfaces. Adapted evaluation methodologies, using simulation frameworks (such as Batsim and SimGrid, and real-scale testbeds such as Grid'5000) are useful for testing and validating candidate policies and energy models before deployment [31, 32].

Multi-Objective Scheduling Decision Process

MOS employs a structured scheduling workflow that balances competing objectives while enforcing hard constraints [33]. The goal of the decision process is to pick a solution within the acceptable trade-off envelope that is as close as possible to the efficient frontier.

Therefore, MOS adds a decision layer between predictive models and final scheduling decisions. Traditional schedulers use WP3 models to optimise a single objective; MOS queries these models across multiple objectives simultaneously, obtaining predictions for each candidate placement. This enables the scheduler to balance energy, performance, fairness, and other goals while respecting hard constraints (power caps, memory limits, QoS rules).

The workflow normalises predictions to comparable scales and filters candidates by feasibility constraints. MOS constructs the Pareto frontier—the set of feasible solutions where no objective can improve without degrading another—in line with multi-criteria optimisation theory [33]. User and site preferences (objective weights or priority orderings) determine which solution on the frontier is selected as optimal. The selected solution is translated into a scheduling plan and applied through workload manager hooks for placement, power control, and preemption. Outcomes feed back to improve future predictions and parameters.

MOS could be implemented using standard multi-criteria optimisation methods—weighted sums, epsilon-constraint, or Techniques for Order Preference by Similarity to Ideal Solution (TOPSISs)—so every scheduling decision is Pareto efficient and auditable [34, 35].

4.2.1 Slurm

To implement MOS in Slurm, the scheduling and allocation processes must be extended to address multiple, potentially conflicting objectives (e.g. performance, energy efficiency, topology awareness). This can be achieved by modifying two key plugin layers within Slurm's modular architecture. At the job selection level, the priority plugin (e.g. replacing or adapting priority/multifactor) can be extended to compute composite or Pareto-based priority scores that balance these objectives, influencing which jobs are dispatched next [36]. At the resource allocation level, a custom select plugin (e.g. extending `select/cons_res`) can implement intelligent task-to-node placement that optimises node choice according to energy consumption, hardware topology, or performance characteristics [37]. Furthermore, the sched/backfill scheduler plugin can be also adapted to integrate these new priority and allocation decisions, ensuring global coordination between job ordering and resource mapping. Together, these plugin modifications can enable a unified, multi-objective scheduling framework within Slurm that dynamically balances objectives such as performance and energy efficiency.

In addition, Slurm exposes practical extension points to realise MOS:

- Job submission plugins (`job_submit` in C/Lua) allow parameter tuning and admission control.
- Priority can incorporate external scores (e.g. site factors) to reflect objective weights.
- Scheduling/placement can combine backfilling with resource-aware selection (`select/cons_tres`, GRES) and task affinity.

- Enforcement relies on cgroups and affinity bindings to manage CPU/memory/GPU isolation, while `Suspend/Resume` reduces idle power at cluster scale.
- Power/frequency shaping typically integrates external runtimes (e.g. EAR) for per-node or per-job caps.
- Runtime resizing is natively limited to shrinking/releasing resources; full elastic grow/shrink requires malleable MPI and research extensions.

These hooks enable co-scheduling, power-aware operation, and fairness/performance control consistent with MOS objectives, within a constraint-first control plane.

4.2.2 Flux

Flux, as a next-generation, distributed resource manager and scheduler is able to support MOS through its extensible plugin architecture and hierarchical design, which inherently facilitates custom policies for balancing objectives like energy efficiency and performance [38]. Native Flux components, such as the *Fluxion* scheduler module and resource set management, allow integration of multi-objective optimisations via user-defined plugins that hook into job submission, allocation, and execution phases—enabling, for instance, real-time power-aware decisions or malleable job reshaping. Flux's ongoing developments include built-in support for weighted scoring in scheduling policies, making it suitable for MOS. To fully enable this feature in Flux, developers have to implement targeted hooks into key APIs: the job-ingest plugin for initial multi-criteria evaluation during submission (e.g. parsing objectives like energy budgets), the scheduler's match-policy hook for Pareto frontier computation using external solvers, and the resource-update callback for dynamic reallocation during runtime (e.g. elastic shrinking based on live metrics).

4.2.3 OAR

OAR is able to support MOS by leveraging the following pragmatic mechanisms:

- The plugin architecture allows to straightforwardly add a new job priority component, which will then be used by the `kamelot` scheduler. Similarly, an external scheduler can be plugged in.
- Property-based resource selection and queue policies enable collocation of complementary jobs and admission control according to site priorities. Moreover, the `oarsub` command accepts user tags which can then be used as additional inputs to the scheduling algorithms when optimising for multiple objectives. For example, with compute-intensive tags and I/O intensive tags for jobs, the scheduler can improve throughput by allocating jobs who are not competing for the same resources.
- Dynamic node power management (Intelligent Platform Management Interface (IPMI)/Baseboard Management Controller (BMC), on/off) reduces idle energy and can be aligned with scheduling decisions to honour facility power caps.
- Fine-grained per-job DVFS/power caps and elastic resizing are typically provided via external tools or application/runtime cooperation; workflow-level adaptation is recommended when malleability is not available.

In combination, these controls allow OAR deployments to enforce constraints first and then optimise weighted objectives, consistent with MOS requirements.

4.3 Job Event Notifications

Job event notifications are a method to notify when the scheduler makes decisions about a certain job state. Since the scheduler is the principal component responsible for jobs starting, as well as tracking their runtime state, there needs to be a method for communicating to the other components of the SW stack whenever there is a change in state.

To be able to offer an in-depth accounting of job characteristics beyond what a scheduler traditionally offers, the monitoring components of the SW stack will need to know when a job is running as well as when each task starts, to determine which metric readings correspond to which job. Similarly, the events for a job's execution ending (whether finishing normally, being cancelled, or any other reason) is equally important to precisely measure the job's characteristics.

In the same vein, for application-aware hardware optimisations to be possible, it is necessary to know when a job event happens, so that the optimisation component can properly characterise each application as finely as possible and thus determine the best hardware settings.

In order keep the high level of accuracy required for both job accounting and application-aware optimisations, a mechanism for the events to be notified immediately has to be provided both at a job level and a task level.

4.3.1 Slurm

Slurm plugins can implement certain functions that automatically get called whenever a new event happens. This allows registered plugins to execute external code which can either forward the event to the components that need it or act by itself.

4.3.2 Flux

Flux jobs pass through a state machine with transitions between states driven by events. Plugins can then register themselves to receive callbacks when events are posted, affording great flexibility to execute external code at many points throughout a job's life cycle when specific conditions are met [39].

4.3.3 OAR

OAR stores job life cycle information in a PostgreSQL accounting database and provides hooks to emit job-event notifications. In production environments, sites typically deploy prologue and epilogue scripts, along with submission wrappers, to trigger low-latency callbacks when jobs start or end. For example, an epilogue script can post the job ID (`OAR_JOB_ID`), allocated nodes, and exit status to a DSRM webhook. Longer-running consumers can poll the database or use `oarstat` to track job state

transitions, such as Waiting → Running → Terminated or Error. For OAR3 deployments, a token-based REST API allows external controllers to submit jobs programmatically, inspect their status, and monitor state changes. Together, these mechanisms-immediate job-level signals combined with a reliable system-of-record—enable sites to correlate task-level telemetry with resource allocations.

5 Normal Requirements

This section covers requirements that provide enhanced control and optimisation opportunities. Energy monitoring (Section 5.1) enables job-level power accounting, resource range specification (Section 5.2) allows fine-grained control over moldable and malleable job parameters, and energy-based fair-share adaptation (Section 5.3) ensures equitable resource distribution while promoting energy efficiency. These features build upon the critical requirements to enable more sophisticated scheduling policies and user-level control.

5.1 Energy and Power Consumption Monitoring

Power consumption information is necessary for the job scheduler for power-aware scheduling and energy & Carbon Dioxide Equivalent (CO_2e) accounting. In a data centre, power consumption information may come from several power domains and from several sources: from individual on node components, through full-node, chassis, Power Distribution Unit (PDU), Power Distribution Board (PDB), rack, partition, and up to the overall site power consumption. Each of these power monitoring systems has a vendor-specific interface, and maintaining an implementation which supports them all is not possible. On the other hand, the power-aware scheduler should have access to all power domains for which it is supposed to optimise, or for which it must constrain power.

Energy & CO_2e accounting is a combination of job owner (user and project), set of allocated nodes, set of allocated computing units (in case of partial node allocation), job start and stop timestamp, power consumption data (provided in the SEANERGYS project by WP2), and carbon intensity monitoring (provided by an external service). If a job is halted, expanded, shrunk, or rescheduled to a different location, then a record must be kept for each period. If a scheduler exposes this job information, the energy & CO_2e accounting does not have to be an integrated feature of the scheduler. An example of such is the MERIC job budgeting service.[1]

To report job energy consumption, it is beneficial for users to receive information for multiple power domains. The reported energy consumption must identify the power domain from which the data originates. Moreover, neither the node components nor the node-level power domain is sufficient, as a significant amount of power is consumed by the necessary data centre infrastructure.

Configuration and monitoring of the power-aware scheduling must be possible using both a CLI and RESTful API. The energy & CO_2e budgeting must provide a CLI for users and a RESTful API for administrators.

5.1.1 Slurm

Energy accounting in Slurm facilitates the tracking of step-level energy consumption, seamlessly integrated into the broader accounting system through Trackable RESource (TRES). TRES represents a flexible concept that can be easily extended. Using the `slurm.conf` file, additional fields can be added and

[1] `https://code.it4i.cz/energy-efficiency/meric-suite/job-budgeting-service`

populated by developing custom Slurm plugins. These fields can be transferred and stored in the Slurm database. The `sacct` command supports querying via generic TRES fields, which represent a list of all configured TRES values. This capability is realised through plugins that collect data from hardware sensors on compute nodes, transfer it via internal messaging and RPC mechanisms to the central control daemon (`slurmctld`), and store it in a database for subsequent querying.

Slurm version 25.05 incorporates seven energy gathering plugins (`gpu`, `ibmaem`, `ipmi`, `pm_counters`, `rapl`, `rsmi`, and `xcc`), each designed to interface with distinct hardware sensors. Only one gathering plugin can be configured at a time. However, a given plugin (e.g. `ipmi`) may query multiple hardware sensors to compute energy consumption on a per-node basis. More flexible solutions are available with ParaStation Management [40], which allow integrating new hardware via a script interface. The polling frequency can be configured either globally or on a plugin-specific basis. Depending on the systems configuration the user may change the polling interval for each job at submission time. Data collection occurs periodically on the nodes by the `slurmd` daemon, where it is aggregated and transmitted to `slurmctld` via RPC-like messages, before being forwarded to `slurmdbd` for database storage. Energy data is also transferred at the completion of each job and step via their corresponding RPCs.

A usual setup employs `slurmdbd` with a MySQL/MariaDB backend. Database queries can be performed using `sacct` or directly via MySQL statements. Additionally, the `sstat` command can be utilised to obtain an updated snapshot of the current accounting data. Node power consumption is gathered through periodic node registration status updates. This information is appended to the node status, which can be inspected using `scontrol show node` and includes metrics such as current and average Watts. Notably, energy data is valid only for node-exclusive jobs and is not accurately computed in node-sharing scenarios.

5.1.2 Flux

As mentioned in Section 3.2.2, there is a *flux-power-monitor* repository in the Flux framework GitHub[2] that demonstrates integration of Flux with the *Variorum* SW [18]. *Variorum* is another SW project with a large part of its development history in the Exascale Computing Project (ECP) and provides functionality for monitoring and control of power metrics across a variety of hardware. The integration essentially uses *Variorum* to query power measurements at regular intervals and then aggregates them for each individual job. Regardless of whether SEANERGYS ultimately uses *Variorum* or a different tool for power readings, this integration demonstrates *how* to use Flux to link such measurements with the particular job that was using given hardware at a given time.

5.1.3 OAR

OAR integrates effectively with node-level power and energy sensors. Readings are linked to jobs by correlating telemetry data with OAR allocations (job ID, nodes, and time window). In practice, CPU and package energy is measured using Running Average Power Limit (RAPL) or BMC methods (IPMI or Redfish), while GPU power is obtained via Data Center GPU Manager (DCGM). Per-node agents (parts of the Comprehensive Monitoring Infrastructure (CMI)) collect these metrics and send them to a central collector, which correlates them with the job ID (`OAR_JOB_ID`) and allocated nodes. On

[2] `https://github.com/flux-framework/flux-power-monitor`

OAR3, cgroup v2 via `systemd` provides improved CPU and memory isolation and enables per-job usage collection. Low-latency sampling can be triggered during job prologue and epilogue phases (for example, by taking snapshots of counters at start and end), while periodic polling captures data during longer runs. The PostgreSQL accounting database and `oaraccounting` serve as the authoritative record for job durations and node lists. Energy totals are calculated by integrating sensor data streams across each resource allocation.

5.2 Moldable and Malleable Resource Range Specification

In order for the scheduler to make sensible decisions regarding moldable and malleable jobs, as defined in Section 3.1, it must know within what constraints the resource set of a given job may be set for the job to run successfully. These constraints may be provided directly by the user at submission time, but may also be determined and/or later modified by the DSRM system itself according to input from the AIDAS. In either case, the allowed range of resources must be clearly specified at scheduling time in order for moldable scheduling to occur, and throughout the runtime of the job for malleability decisions to be made.

The range must minimally specify the minimum and maximum amount of a given resource that the job can use, where either value could be unconstrained as long as that is clearly defined. Moreover, within these outer bounds it is also helpful to allow further constraints on what specific intermediate values a job can properly use, e.g. a job employing a Cartesian domain decomposition may only run on a square or cube number of nodes, and the submitting user should be able to specify this during submission.

To provide maximum flexibility for the scheduler, it should be possible to provide ranges for any resource managed by the DSRM system. For pragmatic reasons the scheduler may choose to only exploit moldability/malleability over a subset of resources for any given job, but it should be possible to specify constraints for any and all of them.

5.2.1 Slurm

For moldable jobs, Slurm's scheduler (e.g. backfill plugin) uses range-based requests to allocate within bounds based on availability. This is limited to node counts (via `--nodes=min[-max[:step]]` or `--nodes=csv_string` in `sbatch/srun/salloc`), where the scheduler selects a value in the range (constrained by the step size) or in the comma list of values to optimise utilisation. User-provided constraints are supported at job submission. System-determined modifications are supported via job submit plugins (e.g. Lua scripts), which can alter job descriptors based on external inputs during submission. Runtime malleability is very limited. The scheduler does not automatically make decisions or resize running jobs. Manual shrinking is possible via `scontrol update` to reduce nodes, but expansion is not supported for running jobs.

5.2.2 Flux

The Flux design allows for the count of almost any resource to be given as a range according to a *min*, *max*, *operand*, and *operator* combination [41]. The *max* is optional and can be omitted to indicate a request

for as much of a resource as is available. The *operand/operator* are also optional and can be omitted if any value in the range is allowed. When given, they can be used to specify regular constraints such as fixed increments. At present, support for such ranges using addition, multiplication, and exponentiation operators is already mostly implemented in Flux (where implemented in this section means that such jobs are allocated and run successfully, but intelligent moldable/malleable scheduling algorithms are considered separate). The primary *Fluxion* scheduler will maximise the allocated resources available within the given range. Some supporting services such as listing of job information still need additional development to properly handle job requests with non-integer resource counts.

Counts may also be given as comma-separated lists of allowed values, enabling entirely arbitrary sets of values to be specified, even if they do not fit a regular pattern amenable to the structured range syntax above. Support for this feature is only partially implemented, however, and in particular the *Fluxion* scheduler is not yet capable of ingesting such inputs.

The only resource type that cannot currently take a non-integer count is the virtual *slot* resource, which essentially determines to the number of tasks (or tasks per node) for the job. Future planned improvements to the specification of allocated resource sets should alleviate even this restriction.

5.2.3 OAR

OAR allows users to request partial node resources and control job placement using resource properties. This enables flexible specification of acceptable resource shapes and ranges. At job submission, the `oarsub -l` command specifies resource counts (for example, CPU cores, GPUs, and walltime), while the `-p` option filters eligible nodes using boolean expressions based on their resource properties. Timesharing mode (`-t timesharing`) and best-effort mode (`-t besteffort`) support dynamic resource sharing and pre-emptible fill-in work. The scheduler uses these modes to co-place complementary jobs on shared resources. Runtime changes to job resource allocation (live resize) are limited in OAR. However, the admissible resource range can be defined through property constraints and queue policies. The DSRM selects an initial resource shape from the AIDAS scalability predictions and can adjust it at requeue points as needed.

5.3 Energy Efficiency Based Fair-share Adaptation

Classical RJMSs such as Slurm and Flux map users' jobs onto HPC resources using FCFS with backfilling queue. To preserve fairness under heavy demand, HPC centres generally enable fair-share mechanisms [42, 43]. The priority of jobs waiting in the queue is influenced by a fair-share term that compares a user's historical consumption to the user's fair share score, together with other common factors such as job age, QoS, and partition.

Some HPC sites configure their fair share according to... this is the case of University of Luxembourg [44] aiming financial cost and environmental considerations, and Fugaku [45, 46] where a bonus or malus is accumulated for a user or project based on the cumulative energy behaviour of recent jobs.

5.3.1 Slurm

The Slurm scheduler can be configured to incorporate energy related metrics into its fair-share job scheduling policy. By extending the accounting and priority mechanisms, energy consumption data—such as node power usage or job energy profiles—can be integrated into the priority calculation. This enables the scheduler to favour users and jobs that make efficient use of energy resources while maintaining fairness across users. For instance, jobs running on energy-optimised nodes or using lower power per computation unit can receive higher priority, promoting sustainable resource utilisation without compromising equitable access to computing resources.

Through the plugin API and pre/post execution scripts, Slurm can rely on external pieces of SW in order to automatically optimise the energy consumption of running jobs, possibly using configuration information provided by the user at submission time.

5.3.2 Flux

As part of its plugin system, Flux provides *jobtap* hooks (such as `job.state.priority` [47]) that enable plugins to implement multi-factor priority calculations [48]. One of the inputs to the existing multi-factor priority plugin is a fair share value based on usage [42], while a specialised SEANERGYS plugin could also augment the calculation with energy-efficiency metrics (for example, energy consumed per job in Joules, or Energy-Delay Product). These priority decisions and the data used for making fair share and priority calculations are stored using the Flux Framework's *flux-accounting* project.

5.3.3 OAR

OAR provides policy controls through admission rules, queues, and user or account limits. An external policy engine can classify jobs by energy efficiency and map these classifications to queue assignments and priority weights. Alternatively, classifications can be adjusted periodically based on measured energy usage. Together, these mechanisms allow sites to adjust scheduling fairness to reflect energy efficiency goals while maintaining auditability against Hosting Site (HS)/Hosting Site Advisory Group (HSAG) performance indicators.

6 Other Requirements

These requirements address ecosystem integration, operational efficiency, and large scale considerations. Kubernetes compatibility (Section 6.1) ensures the DSRM can operate with modern container orchestration environments, node shutdown policies (Section 6.2) enable power savings during low utilisation periods without increasing the node components ageing, and hierarchical scheduling (Section 6.3) provides architectural flexibility for extremely large-scale deployments.

6.1 Kubernetes Compatibility

While HPC systems deliver exceptional computational throughput to scientists, they are not, by themselves, optimised for end-user usability, self-service access, or public-facing online platforms. Modern workload includes AI studios with human-in-the-loop, compute-intensive workflows [49, 50], as well as emerging 3D immersive applications, Extended Reality and digital twins [51]. In parallel, several initiatives are formalising "AI Factory" infrastructure that complement scalable compute, data, and Machine Learning Operations (MLOps) and Infrastructure-as-a-service capabilities on top of HPC infrastructure [52, 53, 54]. In this landscape, deployment of containerised through middleware (e.g. Kubernetes) serves often as the bridges between HPC back with user-facing services.

HPC and cloud/Kubernetes integration is typically realised in two patterns:

Cloud in HPC (co-existence) The HPC system retains its site scheduler (e.g. Slurm, Flux, OAR) and control of compute nodes, while Kubernetes hosts long-lived, cloud-style services for users and operators (portals, APIs, registries, inference). End users interact with these services; behind the scenes, batch jobs are still dispatched by the HPC scheduler.

HPC in the Cloud (or on Kubernetes) An RJMS is deployed on top of a Kubernetes control plane ("scheduler-on-K8s") or via managed cloud HPC. Users see familiar HPC and submit in queues and performance semantics, but execution is provisioned and managed through containerised infrastructure.

This section provides more details on the technical possibilities of integrating the three RJMSs with Kubernetes.

6.1.1 Slurm

The typical production approach is co-existence: Slurm manages batch jobs on compute nodes while Kubernetes runs containerised services, and communication occurs through the `slurmrestd` for programmatic job submission and monitoring. Containers are deployed using Apptainer, Enroot+Pyxis, or rootless Docker/Podman (via `scrun`). Resource isolation is enforced using cgroups, configured via `task/proctrack=cgroup` in `cgroup.conf`. For Slurm-on-Kubernetes deployments are achieved via a set of integration tools known as Slinky, also developed by SchedMD. It uses CPUManager set to static mode, disables Slurm cgroups within pods, and configures `OverSubscribe` to Exclusive. GPUs are managed through Kubernetes device plugins and operators.

6.1.2 Flux

The *Flux Operator* uses a Kubernetes Custom Resource Definition (CRD) *MiniCluster* to run Flux as Kubernetes Jobs. This approach leverages standard container runtimes (*containerd* or CRI-O) and Kubernetes device plugins for hardware access. Accounting data spans both Flux jobs and Kubernetes pods. Organisations must decide which system serves as the primary record and establish procedures to export unified metrics across both layers.

6.1.3 OAR

The typical co-existence pattern mirrors Slurm: OAR controls compute resource allocations, while Kubernetes runs containerised services. For research experiments requiring tight integration, sites can reserve nodes using OAR and then deploy temporary Kubernetes clusters (using `kubeadm` or `k3s`) on those nodes. Alternatively, Kubernetes can interact with OAR through the token-based REST API for job submission and monitoring.

6.2 Unused Nodes Shutdown Policy

Idle compute nodes still draw significant power (often hundreds of watts). A straightforward way to save energy is to power off nodes that remain idle for a while and power them back on when demand returns. However, when many nodes boot simultaneously, the burst of service requests can overload shared infrastructure (controllers, network), causing timeouts and instability. This phenomenon is known as the "herd effect".

In a typical RJMS, an energy-saving (suspend/resume) plugin is available and the administrator configures several parameters to control its behaviour:

Suspend time The duration a node remains idle before being powered off.

Resume rate The maximum number of nodes powered on per minute. If it is too high there is a risk of the "herd" effect; if it is too low, the pending jobs may wait for booting/initialising nodes.

Suspend rate The maximum number of nodes powered off per minute. If it is too high, the pending demand may have to wait for nodes to boot or initialise during a sudden demand peak; if it is too low, energy-saving opportunities are missed.

Suspend timeout The time after which a node that fails to resume is considered permanently disconnected.

Node Draining The ability to remove nodes from scheduling while allowing currently running jobs to complete, enabling staged maintenance or shutdown operations rather than abrupt mass reboots.

While effective, automated "turn off" requires careful tuning and is sometimes disabled at sites due to known risks and operational constraints:

Infrastructure overload (herd effect) Boot storms overwhelm shared services such as Dynamic Host Configuration Protocol (DHCP)/Trivial File Transfer Protocol (TFTP), directory/name services, and parallel file systems (e.g. Lustre thundering herd) [55]. It can also trigger InfiniBand local identifier reassignment storms if large sub-trees restart simultaneously.

Service variability and race conditions Distributed services may respond unpredictably under load, leading to timeouts, retries, and node "flapping".

Manual/interactive boot paths Certain components can require operator intervention at boot (e.g. file system checks [56], GPU initialisation/attestation paths [57]), which breaks full automation.

6.2.1 Slurm

The Slurm Power Saving framework provides a built-in suspend/resume mechanism for powering down idle nodes and bringing them back online when new jobs are submitted [58, 59]. It includes rate-limiting parameters (`SuspendRate`, `ResumeRate`) and timeouts (`SuspendTime`, `ResumeTimeout`) specifically designed to prevent "herd" effects and overloading of provisioning services during large-scale transitions.

Beyond simple rate limiting, Slurm leverages a hierarchical communication topology to mitigate the impact of mass node re-registration. By configuring the `TreeWidth` parameter, the main controller (`slurmctld`) offloads communication to intermediate nodes, reducing the bottleneck when thousands of nodes report their "UP" state simultaneously after a resume operation. Nodes in power save mode are flagged with a specific state (e.g. `IDLE`), allowing the scheduler to batch resume requests efficiently via the configured `ResumeProgram`, effectively serialising the boot storm at the source.

6.2.2 Flux

Flux's hierarchical broker architecture provides some inherent mitigation against the mass reboot problem, though it does not eliminate all associated risks. The key architectural features relevant to this requirement include:

Hierarchical Broker Topology Flux uses a Tree-Based Overlay Network (TBON) where a lead broker (rank 0) coordinates follower brokers on compute nodes [60]. This hierarchy means that broker failures and reconnections (apart from the lead broker) are handled in a structured, non-simultaneous manner rather than as a flat mass reconnection event.

Ordered Startup and Reconnection When deploying Flux instances the system enforces ordered broker initialisation where the lead broker must be operational before follower brokers attempt connection [60]. This prevents simultaneous reconnection attempts, though it introduces startup latency due to ZeroMQ's exponential back off retry mechanism when follower brokers wait for the lead broker.

Graceful Node Drain Flux provides administrative mechanisms to drain nodes before shutdown via `flux resource drain` [61]. Drained node states persist in the KVS resource event log across Flux restarts, ensuring consistent resource management without requiring mass reboots to clear state.

Torpid Node Detection Flux automatically detects unresponsive ("torpid") nodes through its heartbeat mechanism and prevents new work from being scheduled on them without requiring administrator intervention [62].

Current Status Flux has implemented graceful broker shutdown mechanisms [63], and orderly instance termination using the `flux-shutdown` command [61]. The system's design philosophy emphasises resilience through hierarchical containment of failures rather than preventing reboots entirely.

6.2.3 OAR

OAR provides a configurable energy-saving module, historically known as the `hulot` meta-scheduler, that allows idle nodes to be powered off and later resumed when new jobs arrive [64, 65]. The configuration exposes parameters analogous to Slurm's suspend/resume settings (idle delay, suspend/resume scripts, and exclusion lists).

Technical implementation relies on the `scheduler_node_manager_sleep_cmd` and corresponding `scheduler_node_manager_wake_up_cmd` parameters within the main configuration. To mitigate the herd effect, OAR employs a checking window mechanism (defined by `window_start` and `window_end`) and configurable check intervals (`SCHEDULER_NODE_MANAGER_IDLE_TIME`). Operational practices show that while these parameters allow for batched wake-ups, careful tuning of the `wake_up_cmd` (often wrapping generic IPMI calls with staggered delays) is required to avoid network or storage overload during node restarts.

6.3 Hierarchical Scheduling

Hierarchical scheduling organises policies across multiple control levels. These levels include: site/system (managing global resources and limits on power and memory), account/QoS/queue (handling fair-share allocation, job admission, and preemption rules), workflow/job (controlling job ordering, packing, and ensemble management), and task/node (managing placement, cgroup isolation, and resource caps). In SEANERGYS, this structure aligns with the DSRM's control architecture and allows MOS to optimise key performance indicators while respecting hard constraints [15, 29].

Complex workloads, such as ensembles, nested schedulers, and service pods, require both global fairness and energy compliance without reducing per-workflow flexibility. A single flat queue cannot satisfy both requirements. Hierarchical scheduling separates concerns clearly: the outer level enforces quotas, energy limits, and global policy, while the inner level arranges jobs and tasks for throughput and co-scheduling efficiency [38, 29].

DSRM should sequence actions to prevent contention, for example, performing outer-level drain and preemption before triggering inner-level rescheduling. It also unifies accounting across layers to prevent double-counting when systems nest (Flux-in-Slurm or the Flux Operator in Kubernetes). All decisions should be auditable at each level and traceable to HS/HSAG performance indicators [66, 67].

6.3.1 Slurm

Hierarchical scheduling is implemented through an association tree (organising accounts, projects, and users) and QoS/partition layering. Site policies are specified in `slurmdbd` and `slurm.conf` and managed using `sacctmgr` for accounts and QoS, and `scontrol`/`sprio` for partition limits, preemption, and priority. Priorities and limits flow down through the association hierarchy. QoS settings, including preemption eligibility and rate or usage limits, enforce constraints at the outer level, while inner workflow controllers can organise job steps within allocations. Observability is provided through `slurmdbd` tools (`sacct`, `sreport`, `sshare`) and TRES/TRESBillingWeights metrics, enabling DSRM to attribute KPIs and coordinate draining and preemption with inner schedulers [68, 67].

6.3.2 Flux

Flux is natively hierarchical. Users can run nested broker instances inside outer allocations (such as those from Slurm or OAR) or under Kubernetes (via the Flux Operator). Each instance defines queue and resource policies and can be extended through jobtap plugins to implement multi-factor priority at the workflow, job, or task level. This design allows site-level policies to reside in the outer workload manager, while Flux handles job ordering, packing, and co-scheduling within allocations. Accounting is managed via `flux-accounting` to a policy database. DSRM coordinates by setting outer allocation sizes and allowing inner Flux instances to adapt, while consolidating accounting information across layers [38, 66].

6.3.3 OAR

The first mechanism for hierarchical scheduling OAR is the meta scheduler, in charge of delegating the scheduling to a set of different queues with attached priority levels. Queues impact scheduling on three aspects. Priorities can be set between queues with different configurations such as fair sharing. Furthermore, it is also possible to split the resources used by each queue.

Secondly, OAR features a mechanism of container jobs [69]. First, a job with a special type of container is submitted, creating a frame in which one can submit jobs that shall execute within the defined frame (job submission must provide the ID for the container job to be routed inside the frame). Note that job containers are recursive (i.e. a job container can be scheduled inside another job container).

Finally, the resource definition for the computing resources in OAR is defined hierarchically [70]. Job resource requests can be expressed as a hierarchy of resources, such as `/switch=1/hosts=4/cpu=1` (i.e. the job should have 4 CPUs under 4 different hosts on the same switch). Note that during the scheduling, the scheduler unrolls the resource request hierarchy to the smallest unit (to the core level, for instance) and does not stop at the upper hierarchy level. For a job requesting solely a whole switch (`switch=1`), the scheduler will transform this allocation to a set of cores, thus limiting the benefits of this approach to the request's expressivity without a notable performance gain.

7 Summary

This Deliverable presents an analysis of state-of-the-art RJMSs for system-level scheduling, based on a prioritised requirements framework for the DSRM system, establishing clear technical objectives across four priority tiers. These requirements address the fundamental challenges we will face to bring the current state-of-art up to our stated project objectives of a production-ready DSRM system.

The requirements hierarchy reflects both technical dependencies and practical constraints. Critical requirements (Chapter 3) establish the algorithmic and communication foundations essential for any DSRM system. High importance requirements (Chapter 4) provide the runtime mechanisms and observability features necessary for production deployment. Normal requirements (Chapter 5) enable sophisticated optimisation and user control, whilst additional requirements (Chapter 6) brings potential additions which have been identified as reachable during the project time frame.

Importantly, these requirements are defined in relation to the concrete use cases documented in Deliverable 1.2 (D1.2) [1] and reflect real operational needs identified by SEANERGYS project partners. Finally, by assessing Slurm, Flux, and OAR against these requirements, the project will be able to make an informed decision about which existing scheduler provides the strongest basis for DSRM development.

Acronyms and Abbreviations

A

AI *(Artificial Intelligence)* 25, 35, 42, 43
AIDAS *(Artificial Intelligence Data Analytics System)* One of the three pillars of the SEANERGYS project and the primary output of WP3 . . . . . 13, 25, 32, 33
API *(Application Programming Interface)* 15, 19, 21, 22, 27, 29, 30, 34, 36, 41, 43, 44
ARM Family of Reduced Instruction Set Computing (RISC) architectures for computer processors, configured for various environments. Formerly standing for Advanced RISC Machine, or Acorn RISC Machine 15

B

Batsim Scientific simulator to analyse batch schedulers, based on SimGrid 25
BMC *(Baseboard Management Controller)* Hardware management interface, typically on a server's motherboard, used for remote monitoring and management of the system's physical state 27, 31
BSC *(Barcelona Supercomputing Centre)* HPC Centre and HS of MareNostrum 5, located in Barcelona, Spain 42

C

CAT *(Cache Allocation Technology)* Technology introduced by Intel to reserve portions of the Last-level Cache (LLC) for individual cores 17, 18, 20
cgroup *(Control Group)* Linux kernel feature which allows processes to be organised into hierarchical groups whose usage of various types of resources can then be limited and monitored 8, 14, 19, 20, 25, 27, 32, 35, 38
CLI *(Command Line Interface)* 20, 22, 30
CMI *(Comprehensive Monitoring Infrastructure)* One of the three pillars of the SEANERGYS project and the primary output of WP2 . . . . . 25, 31
CO_2e *(Carbon Dioxide Equivalent)* Metric measure used to compare the emissions from various greenhouse gases on the basis of their Global-Warming Potential (GWP) 30
CPU *(Central Processing Unit)* 8, 14–20, 23, 25, 27, 31–33, 43, 44
CRD *(Custom Resource Definition)* Extension of the Kubernetes API 36
CRI *(Container Runtime Interface)* Kubernetes plugin interface which enables the use of a wide variety of container runtimes, without needing to recompile the cluster components 41
CRI-O Open Container Initiative (OCI)-based implementation of Kubernetes Container Runtime Interface (CRI) 36
CUDA *(Compute Unified Device Architecture)* Parallel computing platform and programming model developed by NVIDIA for general computing on GPUs 18, 43

D

D *(Deliverable)* 6, 40
DCGM *(Data Center GPU Manager)* Suite of tools for managing and monitoring NVIDIA Datacenter GPUs in cluster environments 31
DHCP *(Dynamic Host Configuration Protocol)* 37
DOE *(Department of Energy)* Executive department of the U.S. federal government that (among other things) oversees many of its national laboratories 43

DSRM *(Dynamic Scheduling and Resource Management system)* Alternatively, Dynamic Scheduler and Resource Manager; one of the three pillars of the SEANERGYS project and the primary output of WP4 . . . . . 6–8, 10, 12–14, 17, 23, 25, 28, 32, 33, 35, 38–40

DVFS *(Dynamic Voltage and Frequency Scaling)* Dynamic power management technique where the supply voltage and clock frequency are tuned according to workload requirements 14, 25, 27

E

EAR *(Energy Aware Runtime)* System software for energy management, accounting, and optimisation for supercomputers developed by Barcelona Supercomputing Centre (BSC) 16, 27

ECP *(Exascale Computing Project)* U.S. project operated between 2016 and 2024 responsible for delivering a capable exascale computing ecosystem, including software, applications, and hardware technology 31

EuroHPC JU *(European High Performance Computing Joint Undertaking)* Legal and funding entity, created in 2018 and located in Luxembourg; Group of EU-state members joining forces to foster HPC in Europe 44

F

FCFS *(First Come, First Served)* 16, 33

FIFO *(First In, First Out)* 8

Flux RJMS developed by LLNL, USA 6–11, 13, 15, 18, 19, 21, 24, 27, 28, 33–40, 42

Fluxion An advanced graph-based scheduler for HPC and part of the Flux framework 15, 18, 19, 27, 33

G

GPL *(GNU General Public License)* 10, 11

GPU *(Graphics Processing Unit)* 7, 8, 14–20, 23, 25, 27, 31, 33, 35, 41, 43

GRES *(Generic Resource)* Slurm feature to support the ability to define and schedule arbitrary Generic RESources 17, 26

Grid'5000 Large-scale and flexible testbed for experiment-driven research in all areas of computer science, with a focus on parallel and distributed computing, including Cloud, HPC, Big Data, and AI 11, 22, 25

GWP *(Global-Warming Potential)* Term used to describe the relative potency, molecule for molecule, of a greenhouse gas, taking account of how long it remains active in the atmosphere 41

H

HPC *(High Performance Computing)* 7–11, 13, 15, 33, 41–44

HPSF *(High Performance Software Foundation)* Neutral hub for open source high performance software and part of the nonprofit LF 10, 11

HS *(Hosting Site)* 34, 38, 41, 43

HSAG *(Hosting Site Advisory Group)* 34, 38

HTTP *(Hypertext Transfer Protocol)* 21

HTTPS *(Hypertext Transfer Protocol Secure)* 21

I

I/O *(Input/Output)* 16, 25

IPMI *(Intelligent Platform Management Interface)* 27, 31, 38

IT4I *(IT4Innovations)* Short form of formal acronym "IT4I@VSB" 43

J

K

L

M

N

O

OpenMP *(Open Multi-Processing)* API that supports multi-platform shared memory multiprocessing 8

P

P2P *(Peer-to-Peer)* 44
PDB *(Power Distribution Board)* 30
PDU *(Power Distribution Unit)* 30
PMIx *(Process Management Interface for Exascale)* 14, 21, 22
Pub/Sub *(Publish/Subscribe)* Messaging pattern where publishers send messages to a central topic without knowing which subscribers will receive them 21

Q

QoS *(Quality of Service)* 17, 18, 20, 25, 26, 33, 38, 39

R

RAPL *(Running Average Power Limit)* Feature in Intel processors that allows software to measure power consumption and set power limits for the CPU and other components 31
RDT *(Resource Director Technology)* Intel framework with several component features for cache and memory monitoring and allocation capabilities 17, 18, 20
REGALE European High Performance Computing Joint Undertaking (EuroHPC JU) project to develop an open architecture to equip next generation HPC applications with exascale capabilities 16
REST *(REpresentational State Transfer)* An interface for web services 21, 22, 29, 36, 44
RESTful Interface that is fully compliant with the REST principles 22, 30
RISC *(Reduced Instruction Set Computing)* Microprocessor architecture using a small, optimised set of instructions, allowing simplified instructions to execute more rapidly 41
RJMS *(Resource and Job Management System)* Software that tracks and monitors the hardware deployed in a data centre, arbitrates access as users submit work they would like to run, and manages scheduling and placement of that work onto the available hardware 7, 14, 16, 25, 33, 35, 36, 40, 42–44
RPC *(Remote Procedure Call)* Messaging pattern where a program may request a service or execute a procedure on another computer (or process) across a network as if it were a local operation 9, 21, 24, 31

S

SimGrid Framework to simulate large-scale distributed systems such as Grids, Clouds, HPC, or Peer-to-Peer (P2P) systems 25, 41
SLICES-RI *(Scientific Large Scale Infrastructure for Computing/Communication Experimental Studies - Research Infrastructure)* Flexible platform designed to support large-scale, experimental research focused on networking protocols, radio technologies, services, data collection, parallel and distributed computing and in particular cloud and edge-based computing architectures and services 11
SLO *(Service Level Objective)* Quantified performance target for a service metric (e.g. availability, response time, throughput) that defines the boundary between acceptable and unacceptable service behaviour 25
Slurm RJMS developed by SchedMD LLC 6–10, 13, 15, 17, 18, 21, 23, 26, 28, 30, 33–40, 42, 45

Bibliography

[1] P. Pochelu et al. *SEANERGYS Deliverable 1.2: Use cases and requirements*. Tech. rep. Nov. 2025

[2] European Commission, EU Funding & Tenders Portal. *Call for Proposals: Energy Efficient Technologies in HPC (HORIZON-EUROHPC-JU-2023-ENERGY-04-01)*. URL: https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/horizon-eurohpc-ju-2023-energy-04-01

[3] SchedMD. *Slurm Scheduler by SchedMD*. URL: https://www.schedmd.com/slurm/

[4] *Website of the TOP500 Project*. URL: https://top500.org

[5] SchedMD. *Slurm Support & Development | SchedMD*. URL: https://www.schedmd.com/

[6] SchedMD. *Slurm Source Code*. URL: https://github.com/SchedMD/slurm

[7] Lawrence Livermore National Laboratory: Livermore Computing. *User Guides: Using El Capitan Systems*. URL: https://hpc.llnl.gov/documentation/user-guides/using-el-capitan-systems

[8] *Flux Licensing and Collaboration Guidelines*. URL: https://flux-framework.readthedocs.io/projects/flux-rfc/en/latest/spec_2.html

[9] *Grid5000 Website*. July 11, 2025. URL: https://www.grid5000.fr/

[10] *UAR GRICAD*. URL: https://gricad.univ-grenoble-alpes.fr/

[11] *OAR Copyright and License*. Apr. 23, 2020. URL: https://github.com/oar-team/oar3/blob/master/COPYRIGHT

[12] D. G. Feitelson and L. Rudolph. "Toward Convergence in Job Schedulers for Parallel Supercomputers". In: *Job Scheduling Strategies for Parallel Processing*. IPPS '96 Workshop, JSSPP 1996 (Apr. 16, 1996). Ed. by D. G. Feitelson and L. Rudolph. Red. by G. Goos, J. Hartmanis, and J. Van Leeuwen. Vol. 1162. Lecture Notes in Computer Science. Honolulu, HI, USA: Springer, Berlin, Heidelberg, Oct. 16, 1996, pp. 1–26. https://doi.org/10.1007/BFb0022284

[13] A. Tarraf et al. "Malleability in Modern HPC Systems: Current Experiences, Challenges, and Future Opportunities". In: *IEEE Transactions on Parallel and Distributed Systems* 35.9 (2024), pp. 1551–1564. https://doi.org/10.1109/TPDS.2024.3406764

[14] SchedMD. *Documentation of slurm.conf*. URL: https://slurm.schedmd.com/slurm.conf.html

[15] M. Chadha, J. John, and M. Gerndt. "Extending SLURM for Dynamic Resource-Aware Adaptive Batch Scheduling". In: *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2020, pp. 223–232. https://doi.org/10.1109/HiPC50609.2020.00036

[16] M. D'Amico and J. C. Gonzalez. "Energy hardware and workload aware job scheduling towards interconnected HPC environments". In: *IEEE Transactions on Parallel and Distributed Systems* (2021). https://doi.org/10.1109/TPDS.2021.3090334

[17] A. Aaen Springborg, M. Albano, and S. Xavier-de-Souza. "Automatic Energy-Efficient Job Scheduling in HPC: A Novel SLURM Plugin Approach". In: *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. Denver, CO, USA: Association for Computing Machinery, 2023, pp. 1831–1838. https://doi.org/10.1145/3624062.3624265

[18] N. Kulshreshtha et al. "Vendor-Neutral and Production-Grade Job Power Management in High Performance Computing". In: *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (Atlanta, GA, USA, Nov. 17–22, 2024). IEEE, Nov. 17, 2024, pp. 1845–1855. https://doi.org/10.1109/SCW63240.2024.00231

[19] P.-F. Dutot et al. *REGALE Deliverable 2.3: Final Integration of Sophisticated Policies in the REGALE Prototype*. Tech. rep. Apr. 2024. URL: https://regale-project.eu/wp-content/uploads/2025/07/REGALE_D2.3_Final_integration.pdf

[20] Intel Corporation. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. White Paper 331843-001US. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf. Intel Corporation, 2015

[21] Intel Corporation. *Introduction to Memory Bandwidth Allocation*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html

[22] Intel Corporation. *Intel Resource Director Technology (Intel RDT) Framework*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html

[23] NVIDIA Corporation. *Multi-Process Service, Release r575*. Guide. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. NVIDIA Corporation, May 2025

[24] *Grid5000: Software*. July 11, 2025. URL: https://www.grid5000.fr/w/Grid5000:Software

[25] *GRICAD user documentation: Computing on the grid with CiGri*. URL: https://gricad-doc.univ-grenoble-alpes.fr/en/hpc/grid/

[26] *Galaxy Community Hub*. URL: https://galaxyproject.org/

[27] *Flux housekeeping service*. URL: https://flux-framework.readthedocs.io/projects/flux-core/en/latest/man1/flux-housekeeping.html

[28] *Flux Job Execution Protocol Version 1*. URL: https://flux-framework.readthedocs.io/projects/flux-rfc/en/latest/spec_32.html#design-criteria

[29] B. Kocot, P. Czarnul, and J. Proficz. "Energy-Aware Scheduling for High-Performance Computing Systems: A Survey". In: *Energies* 16 (Jan. 2023), p. 890. https://doi.org/10.3390/en16020890

[30] G. A. Koenig et al. *Energy and Power Aware Job Scheduling and Resource Management: Global Survey—An In-Depth Analysis*. Tech. rep. 2018. URL: https://datacenters.lbl.gov/sites/default/files/Energy%20and%20Power%20Aware%20Job%20Scheduling%20and%20Resource%20Management_%20Global%20Survey%20%E2%80%94%20An%20In-Depth%20Analysis.pdf

[31] *Batsim Documentation*. URL: https://batsim.readthedocs.io/

[32] H. Casanova, A. Legrand, and M. Quinson. "Versatile, scalable, and accurate simulation of distributed applications and platforms". In: *Journal of Parallel and Distributed Computing* (2014). https://doi.org/10.1016/j.jpdc.2013.12.005

[33] K. Miettinen. *Nonlinear Multiobjective Optimization*. Springer, 1999. https://doi.org/10.1007/978-1-4615-5563-6

[34] C.-L. Hwang and K. Yoon. *Multiple Attribute Decision Making: Methods and Applications*. Springer, 1981. https://doi.org/10.1007/978-3-642-48318-9

[35] G. Mavrotas. "Effective implementation of the ε-constraint method in Multi-Objective Mathematical Programming problems". In: *Applied Mathematics and Computation* 213.2 (2009), pp. 455–465. https://doi.org/10.1016/j.amc.2009.03.037

[36] SchedMD. *Priority Multifactor — Slurm Workload Manager*. URL: https://slurm.schedmd.com/priority_multifactor.html

[37] SchedMD. *Consumable Resources (TRES) — Slurm Workload Manager*. URL: https://slurm.schedmd.com/cons_tres.html

[38] D. H. Ahn et al. "Flux: Overcoming scheduling challenges for exascale workflows". In: *Future Generation Computer Systems* 110 (2020), pp. 202–213. https://doi.org/https://doi.org/10.1016/j.future.2020.04.006

[39] *Flux Job States and Events Version 1*. URL: https://flux-framework.readthedocs.io/projects/flux-rfc/en/latest/spec_21.html

[40] *ParaStation Management repository*. URL: https://github.com/parastation/psmgmt

[41] *Flux Canonical Job Specification*. URL: https://flux-framework.readthedocs.io/projects/flux-rfc/en/latest/spec_14.html

[42] *Flux Fair-Share Calculation*. URL: https://flux-framework.readthedocs.io/projects/flux-accounting/en/latest/components/fair-share.html

[43] SchedMD. *Classic Fairshare Algorithm*. URL: https://slurm.schedmd.com/classic_fair_share.html

[44] S. Varrette et al. "Management of an Academic HPC & Research Computing Facility: The ULHPC Experience 2.0". In: *Proceedings of the 2022 6th High Performance Computing and Cluster Technologies Conference*. HPCCT '22. Fuzhou, China: Association for Computing Machinery, 2022, pp. 14–24. https://doi.org/10.1145/3560442.3560445

[45] A. L. V. Solorzano et al. "Operational Experience with Incentive-Based Power Efficiency Mechanisms on the Fugaku Supercomputer". In: *SC '24: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Atlanta, GA, USA: IEEE, 2024. https://doi.org/10.1109/SC41406.2024.00030

[46] RIKEN R-CCS. *Executing High Performance and Energy Efficient Jobs on Supercomputer Fugaku*. Presentation slides. Includes details of the Fugaku Point Program and priority redemption. May 2025. URL: https://www.r-ccs.riken.jp/fugaku/docs/performance-data/en/perf-power-en.pdf

[47] *Flux Jobtap Plugins*. URL: https://flux-framework.readthedocs.io/projects/flux-core/en/latest/man7/flux-jobtap-plugins.html

[48] *Flux Job Priorities*. URL: https://flux-framework.readthedocs.io/projects/flux-accounting/en/latest/components/job-priorities.html

[49] Mistral AI. *Mistral Compute — European AI Cloud*. fr. Product page. URL: https://mistral.ai/fr/products/mistral-compute#european-ai-cloud

[50] NVIDIA Corporation. *NVIDIA AI Enterprise*. en. Product page. URL: https://www.nvidia.com/en-us/data-center/products/ai-enterprise/

[51] NVIDIA Isaac Lab Team. *Deploying CloudXR Teleoperation on Kubernetes — Isaac Lab Documentation*. en. Documentation. NVIDIA. URL: https://isaac-sim.github.io/IsaacLab/main/source/deployment/cloudxr_teleoperation_cluster.html

[52] European Commission. *AI Factories*. en. Shaping Europe's digital future. Directorate–General for Communications Networks, Content and Technology (DG CONNECT). Oct. 30, 2025. URL: https://digital-strategy.ec.europa.eu/en/policies/ai-factories

[53] EuroHPC Joint Undertaking. *AI Factories*. en. Overview, access information and list of AI Factories and Antennas. URL: https://www.eurohpc-ju.europa.eu/ai-factories_en

[54] D. Harris. *AI Factories Are Redefining Data Centers and Enabling the Next Era of AI*. en. NVIDIA Blog. Mar. 18, 2025. URL: https://blogs.nvidia.com/blog/ai-factory/

[55] IT for Science group, University of Helsinki. *Lustre Troubleshooting Guide — Client Eviction*. Section "Client Eviction"; accessed 2025-11-11. URL: https://wiki.helsinki.fi/xwiki/bin/view/it4sci/IT%20for%20Science%20group/HPC%20Environment%20User%20Guide/Lustre%20User%20Guide/Lustre%20Troubleshooting%20Guide/

[56] Red Hat, Inc. *12.2. File System-Specific Information for fsck*. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/storage_administration_guide/fsck-fs-specific

[57] NVIDIA Corporation. *NVIDIA Trusted Computing Solutions: Release Notes*. Manual RN-12523-001_r580_02. NVIDIA Corporation, Aug. 2025. URL: https://docs.nvidia.com/580trd1-trusted-computing-solutions-release-notes.pdf

[58] SchedMD. *Slurm Power Saving Guide*. https://slurm.schedmd.com/power_save.html. SchedMD, Inc. Seattle, WA, USA, 2025

[59] O. H. Nielsen. "Power Saving with Slurm". In: *Proceedings of SLUG'23*. https://slurm.schedmd.com/SLUG23/DTU-SLUG23.pdf. 2023

[60] V. Sochat et al. "The Flux Operator". In: *F1000Research* 13 (203 2024). https://doi.org/10.12688/f1000research.147989.1

[61] Flux Framework Community. *flux-resource(1) – Flux core documentation*. URL: https://flux-framework.readthedocs.io/projects/flux-core/en/stable/man1/flux-resource.html

[62] Flux Framework Community. *Automatically drain unresponsive nodes*. GitHub Issue #3448. URL: https://github.com/flux-framework/flux-core/issues/3448

[63] Flux Framework Community. *Implement graceful broker shutdown*. GitHub Issue #25. URL: https://github.com/flux-framework/flux-core/issues/25

[64] OAR Team. *OAR Documentation (Energy Saving Module)*. Inria / OAR Project. 2016. URL: https://media.readthedocs.org/pdf/oar/stable/oar.pdf

[65] OAR Team. *Green computing — energy saving scripts for OAR*. https://oar.imag.fr/wiki:green_computing. 2020

[66] *Flux Operator — Kubernetes MiniCluster CRD*. URL: https://github.com/flux-framework/flux-operator

[67] SchedMD. *Accounting — Slurm Workload Manager*. URL: https://slurm.schedmd.com/accounting.html

[68] SchedMD. *QoS — Slurm Workload Manager*. URL: https://slurm.schedmd.com/qos.html

[69] *Advanced OAR Tutorial: Container jobs*. Mar. 31, 2025. URL: https://www.grid5000.fr/w/Advanced_OAR#Container_jobs

[70] *Advanced OAR Tutorial: Using the resources hierarchies*. Mar. 31, 2025. URL: https://www.grid5000.fr/w/Advanced_OAR#Using_the_resources_hierarchies