H2020-JTI-EuroHPC-2019-1

REGALE: An open architecture to equip next generation HPC applications with exascale capabilities

Grant Agreement Number: 956560

D1.3

REGALE Final Architecture

Final

Version: 1.0

Author(s): Eishi Arima (TUM), Mohsen Seyedkazemi Ardebili (UNIBO), Andrea Bartolini (UNIBO)

Contributor(s): All REGALE partners, especially WP3 REGALE library members

Date: 05.04.2024

Project and Deliverable Information Sheet

<table>
<tr><td rowspan="7">REGALE Project</td><td colspan="2">Project Ref. №: 956560</td></tr>
<tr><td colspan="2">Project Title: REGALE</td></tr>
<tr><td colspan="2">Project Web Site: https://regale-project.eu</td></tr>
<tr><td colspan="2">Deliverable ID: D1.3</td></tr>
<tr><td colspan="2">Deliverable Nature: Report</td></tr>
<tr><td rowspan="2">Dissemination Level: PU *</td><td>Contractual Date of Delivery:
31 / 03 / 2024</td></tr>
<tr><td>Actual Date of Delivery:
05 / 04 / 2024</td></tr>
<tr><td colspan="2">EC Project Officer: Evangelos Floros</td></tr>
</table>

* - The dissemination levels are indicated as follows: PU = Public, fully open, e.g. web; CO = Confidential, restricted under conditions set out in Model Grant Agreement; CI = Classified, information as referred to in Commission Decision 2001/844/EC.

Document Control Sheet

<table>
<tr><td rowspan="6">Document</td><td colspan="2">Title: REGALE Requirements, Initial Architecture, and Evaluation Plan</td></tr>
<tr><td colspan="2">ID: D1.3</td></tr>
<tr><td>Version: 1.0</td><td>Status: Final</td></tr>
<tr><td colspan="2">Available at: https://regale-project.eu</td></tr>
<tr><td colspan="2">Software Tool: Google Docs</td></tr>
<tr><td colspan="2">File(s): REGALE_D1.3_Architecture_ver1.0</td></tr>
<tr><td rowspan="4">Authorship</td><td>Written by:</td><td>Eishi Arima (TUM), Mohsen Seyedkazemi Ardebili (UNIBO), Andrea Bartolini (UNIBO)</td></tr>
<tr><td>Contributors:</td><td>All REGALE partners, esp. WP3 REGALE library members</td></tr>
<tr><td>Reviewed by:</td><td>Pierre-François Dutot (UGA), Daniele Cesarini (CINECA), Martin Schulz (TUM)</td></tr>
<tr><td>Approved by:</td><td>Georgios Goumas (ICCS)</td></tr>
</table>

Document Status Sheet

Version	Date	Status	Comments
0.0	31.01.2024	Draft	Initial version
0.1	22.02.2024	Draft	Ready for our internal review
0.1	03.04.2024	Draft	Review completed
1.0	05.04.2024	Final	Minor updates after the internal review

Document Keywords

Keywords:	REGALE, HPC, Exascale, Software Architecture, Software Integration, Power Stack, Workflow Engines

Executive Summary

This deliverable document reports the final architecture defined in REGALE, as blueprinted in the REGALE prototypes. The ultimate goal of REGALE is to pave the way of next generation HPC applications to exascale systems, and to accomplish this we defined an open architecture (WP1), built prototypes (WP3) and incorporated in this system appropriate sophistications (WP2) in order to equip supercomputing systems with the mechanisms and policies for effective resource utilization and execution of complex applications (WP4). We conducted them in a cooperative manner, i.e. the architecture and the prototype were co-designed/conceptualized considering both state-of-the-art and next generation HPC applications, maximizing in this way its applicability. The main contents of this document are the descriptions of the architectural requirements for a variety of REGALE use cases (Section 4) and the final version of the REGALE architecture (Section 5), which functioned as a blueprint for the other work packages, e.g., the software prototyping in WP3 (Section 6). Further, we discuss our security, privacy, and reliability considerations in the REGALE architecture and software stack (Section 7). Finally, we conclude our work and describe future directions (Section 8).

Acknowledgement

This deliverable is a result of extensive discussions in our regular project meetings. All the REGALE partners participated in and/or contributed to the requirement specifications, architectural design decisions, applications to prototyping, and feedback more or less.

Further, we would like to extend our acknowledgement to the PowerStack community [1]. We inherited their insights on usecases/architecture, moved them forward to govern our prototypes, and shared/discussed our updates with them.

Table of Contents

1. Introduction

An exascale supercomputer will not be “yet another big machine”. With a cost of hundreds of million euros, power consumption in the order of tens of megawatts and a lifetime that reaches a decade at most, judicious management of those resources is of utmost importance. Turning our attention to the critical aspect of power consumption, the current leader in the TOP500 list as of Nov. 2023 [2], has an exascale computational capacity and a power consumption that exceeds 20MW. Even with the highest technological advancements, a post-exascale machine is expected to well exceed the 20-30MW threshold that is the current upper bound of power consumption for exascale computing, and Aurora is projected to consume just under 60MW of power at its full scale [3]. A machine of this size will not be able to operate at full power consumption, and energy consumption will become a primary concern to keep its environmental footprint and operational costs at acceptable levels without neglecting its ultimate purpose: to equip highly critical applications with the computational capacity to solve extremely resource hungry problems.

Focusing on the application side, achieving scalable performance and high system throughput has always been a cumbersome task. To make things even more challenging, next-generation HPC applications can no longer be considered as computation-/communication-intensive, monolithic blocks with minimal and infrequent I/O requirements. The revolution of Big Data and Machine Learning, the emerging Edge Computing and IoT, with the scale of modern HPC systems and cloud datacentres, are rapidly changing the way we solve scientific problems. Novel computational patterns are rapidly evolving, where the solution of a problem may require a workflow of diverse tasks, performing simulations, data ingestion, data analytics, machine learning, visualization, uncertainty quantification, verification, computational steering and more. Existing solutions may render the execution of such applications in a large-scale supercomputer either impossible, or extremely suboptimal in terms of time to solution and user cost, due to the absence or inefficiencies of appropriate methods to compose, deploy and execute workflows, and/or due to their extreme requirements in I/O resources, which cannot be met by the system capacity without holistic and sophisticated deployments.

The ultimate goal of REGALE is to pave the way of next generation HPC workflows to exascale systems. To accomplish this, we define an open architecture, build a prototype system and incorporate in this system appropriate sophistication in order to equip supercomputing systems with the mechanisms and policies for effective resource utilization and execution of complex applications. The REGALE architecture and prototype is co-designed considering both state-of-the-art and next generation HPC applications, maximizing in this way its applicability.

REGALE takes an approach that considers two interacting paths: The first path is largely motivated by the PowerStack initiative [1] that primarily targets multi-criteria operation of supercomputing services with a strong focus on power and energy efficiency. The second path focuses on the requirements posed by non-conventional, workflow-based applications and their integration with an appropriate workflow engine, with a goal to achieve easy and flexible use of supercomputing resources at large scales.

This final version of the WP1 deliverable reports the critical stepping stone for the implementation of REGALE: It starts from the project’s strategic objectives (Section 2), the

final version of our strawman architecture and software tools (Section 3), and analyzes a set of relevant use cases together with their requirements (Section 4). These are then used to define the REGALE architecture, components, and interfaces (Section 5) instantiated with the use of the various modules brought in REGALE and evolved throughout the project by the partners (Section 6). Further, we discuss our security, privacy, and reliability considerations in the REGALE architecture and software stack (Section 7). Finally, Section 8 concludes the final status of this work and introduces several future research directions to future studies.

2. Project Strategic Objectives

REGALE Strategic Objectives: REGALE envisions to meet the Strategic Objectives (SO) presented below.

Strategic Objective 1 (SO1): Effective utilization of resources. This strategic objective considers the huge amount of resources available in exascale class machines and the resource footprints of both traditional and emerging applications. The improvement in resource utilization will indicatively translate to a combination of:

- **SO1.1: Improved application performance.** Better allocation of resources that considers the exact application footprint, data requirements, control and data flows will drastically improve performance for critical applications. This is especially the case for the next generation, workflow-based applications where one of the major problems is the highly suboptimal use of resources, leading to disappointing performance, inability to scale, misuse of resources and consequent over charges of end users.
- **SO1.2: Increased system throughput.** By taking global and elaborate decisions considering the entire mix of workloads to be executed in the supercomputer, we will be able to significantly raise the system throughput, servicing more applications per day and ultimately increasing user satisfaction and system impact.
- **SO1.3: Minimized performance degradation under the power constraints.** Power capping is a common mechanism to align supercomputer consumption with the power availability and charges of the supplier. In REGALE we will replace the current brute-force, performance-oblivious strategies by a set of sophisticated policies for dynamic adaptation to power envelopes without compromising application performance and system throughput.
- **SO1.4: Decreased energy to solution.** REGALE supports the operation of a supercomputer with energy consumption as a first class citizen. In this case we will incorporate mechanisms and policies to minimize energy to solution if this is promoted by the operation policy.

Strategic Objective 2 (SO2): Broad applicability. This strategic objective guided our architecture design and prototyping towards maximizing openness, platform independence, scalability, modularity, extensibility and simplicity, allowing for its implementation with various software modules, on any supercomputing platform, for the realization of SO1. In particular, this will be achieved through compatibility to relevant specifications and standards.

To assess if this SO is met, we will validate the existence of the following key features:

- **Scalability:** The REGALE system should be able to operate in exascale setups and beyond. To this end, first our power management is conducted in a hierarchical manner to localize the overhead. Second, we exploit workflow-level inter-job parallelism in addition to the traditional intra-job parallelism (weak/strong scaling) for modern workloads. We further try to minimize the resource management overhead by carefully selecting the underlying middleware for the REGALE common library. To assess the scalability of the REGALE approach, we performed experimental results and simulations, and we also extrapolated our results to larger system scales for some cases (see D1.4).

- **Platform independence:** The REGALE system should be able to operate across all major architectures of large supercomputing facilities and be free of any vendor lock-in. This was validated by our integration process where we provided full integration scenarios with multiple vendor-specific solutions and provided indicative solutions for all major modules of the HPC ecosystem (see D1.4).
- **Extensibility:** The REGALE system should be extensible to any new feature or component that aligns to its open architecture. This is realized simply by becoming a publisher/subscriber of any functionalities in the newly introduced REGALE common library layer (only if approved by the administrator).

Strategic Objective 3 (SO3): Easy and flexible use of supercomputing services. Widening the use of advanced computational and data facilities beyond the highly skilled traditional HPC users requires significant efforts on the side of the centers. In REGALE we released the developers and users of complex applications that originate from new industrial use cases from the extremely cumbersome task to finetune the execution of their application on an exascale system. Moreover, we equipped them with an easy-to-use set of tools to facilitate the development and deployment of their applications to exascale systems.

To assess if this SO is met, we validated the existence of the following key features:

- **Automatic allocation of resources:** Users of complex applications should not bother with the way their application is distributed on an exascale system. We will compare the process of requesting resources between the current state-of-the-art systems and applications and the REGALE solution.
- **Programmability:** Application developers should find the REGALE architecture and system easily accessible to develop and deploy their code(s). This was qualitatively assessed by the application developers and pilot users of the consortium by comparing the features of their application before and after the optimizations within REGALE (see D1.4).
- **Flexibility:** Applications should be able to execute under lightweight virtualization within the REGALE-enabled system. This was explored such as with several Pilot applications by integrating with the RYAX tool.

3. Final Architecture and Software Tools

In this section, we first introduce the REGALE final architecture and its components/actors. We then summarize the software tools to be used in this project. We finally introduce our implementation paths that integrate the tools.

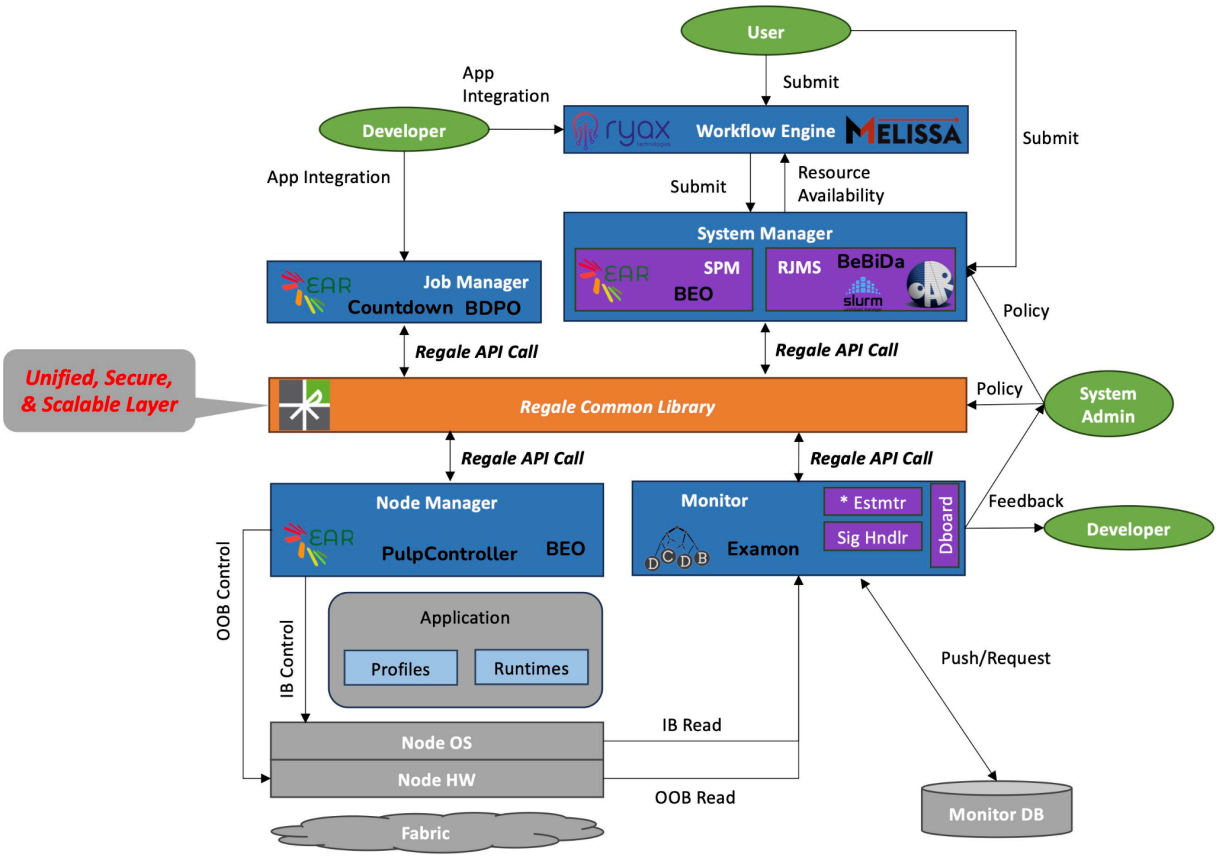

Figure 1: REGALE Final Architecture

Figure 1 illustrates the general intermediate architecture. The descriptions of key actors and software components are as follows.

Human actors:

A. **Site administrator:** Configures the site-level policy appropriately prioritizing between power/energy/performance and quantifies the relevant constraints. The policy can be changed according to the current needs with respect to objectives and/or constraints.

B. **User:** This actor submits a job for execution to the system, requests resources for her job and optionally provides information on the performance behaviour of her application.

C. **Developer:** This actor develops, optimizes and instruments her application with regard to relevant objectives to facilitate further optimization by the system and collection of profiling information.

System modules:

1. **System manager:** The system manager receives as input a set of jobs to be scheduled within the system and indicatively decides upon when to schedule each job, to which specific compute nodes to map it, and under which power budget or

setting. For this, it constantly monitors and records power and energy telemetry data, and controls power budgets/settings and/or user fairness. The system manager applies system-wide optimizations and consists of the following two sub-modules that work cooperatively.

- **Resource and Job Management System (RJMS):** The RJMS manages jobs submitted by users and decides the assignments of node resources to them and their launch timing as well. The decisions are based on the job information given by the users, power/performance features characterized by the Monitor (see next), and the node/job/system power budget information managed by the SPM as well as the scheduling policy given by the site administrator. The decisions are principally made in a static and proactive manner.
- **System Power Manager (SPM):** The SPM manages the power budget allocations across nodes/jobs, including compute nodes, I/O nodes, and others. The SPM provides the functionality to set the power cap to the entire system and also can optimize the power budgeting across nodes depending on the objective/constraints given by the site administrator, while interacting with other modules such as the node manager, monitor, and others.

2. **Job manager:** The job manager performs job-centric optimizations considering the performance behaviour of each application, its fine-grained resource footprint, its phases and any interactions/dependencies dictated by the entire workflow it participates in. It manages the control knobs in all compute nodes participating in the job and optimizes them during runtime to achieve the desired power consumption (at maximum possible performance), efficiency, or other settings. Additionally, it scalably aggregates application profile/telemetry data from each node servicing the given job through the system manager.
3. **Node manager:** The node manager provides access to node-level hardware controls and monitors. Moreover, the node manager implements processor level and node level power management policies, as well as preserving the power integrity, security and safety of the node. For this reason, all the power management requests coming from the software stack are mediated by the node management. The node manager is able to access in-band[1] and/or out-of-band[2] node-granular power knobs.
4. **Workflow engine:** The workflow engine analyses the dependencies and resource requirements of each workflow and decides on how to break the workflow into specific jobs that will be fed to the system manager. Modern workflows may be composed of hybrid Big Data, Machine Learning and HPC jobs; hence a key role for the workflow engine is to provide the right interfaces and abstractions in order to enable the expression and deployment of combined Big Data, HPC jobs. The distribution of jobs can vary depending on the objective goals defined by the optimization strategy.
5. **Monitor:** The monitor is responsible for collecting in-band and out-of-band data for performance, resource utilization, status, power and energy. The monitor operates continuously without interfering with execution, with minimal footprint, and collects, aggregates, records, and analyses various metrics, and pushes necessary real-time

[1] In-band communication refers to a communication paradigm that requires a regular operating system to access the target service [4]. One prominent example is MSR-based monitoring/controlling features using rdmsr/wrmsr instructions.

[2] Out-of-band communication does not require an OS for its interaction with the target service [4]. One prominent example is measurement/control via BMC (Baseboard Management Controller) which is a special controller embedded in a node, often implemented in a form of SoC that operates independently from the node OS.

data to the system manager, the node manager and the job manager. The monitor has the following sub-modules.

- **Signature Handler:** The signature handler receives the information of job identification from other components and then generates the signature to characterize the job. The signature could be calculated with the job information given by the user, the associated job profile of previous runs, or the statistics acquired at runtime.
- **Estimator:** The estimator assesses the job properties (e.g. performance, power/energy consumption, or others) or system status (e.g. anomaly) by using such as the signature generated by the signature handler.
- **Dashboard:** The dashboard provides a set of functionalities that display the node/job status obtained at runtime (or given from a profile) to the developer.

6. **REGALE Common Library:** This is a software module newly introduced in the project. This new layer bridges and co-ordinates different components via API functions to realize our use cases. This common library stands over the DDS (Data Distribution Service) middleware that works in a publish–subscribe pattern. Our approach is agnostic to the actual DDS implementation (the same as other standards such as OpenMP, MPI, etc.) and the choice of the DDS implementation (e.g., OpenDDS, FastDDS, etc.) determines the overhead of power and resource management, and other properties.

TABLE 1 represents the coverage of the architectural components by our software tools.
Note, the details of our software tools are described in our previous deliverables. In REGALE project we introduced the following new software modules:

- Execution Profile Compute Module (EPCM) is newly introduced to help our OAR-BEO integration plugin with providing functionalities to estimate the power/performance properties based on the relevant job profile.
- REGALE Common Library: identical to the component mentioned above.

These tools are integrated based on their coverages, and we offer several different integration instances (or scenarios) that support different use cases which we present later in this deliverable. Also, we first divided our software integration tasks into *PowerStack* and *workflow engine paths*, and the latter consists of *Mellissa path* and *RYAX path*. On one hand, the PowerStack path aimed to prototype a software stack to enable full-scale production-grade solutions for a variety of power/energy management use cases. On the other hand, the workflow engine paths focused more on the application side, i.e., integrating the workflow management tools (Mellissa or RYAX) with our pilot applications as well as other components in our architecture, in order to realize next-generation application management techniques including automatic parameter sensitivity analysis, ML-based simulation surrogate and dynamic concurrency controlling. The PowerStack and workflow engine paths were first integrated individually because of their different focuses, and we combined them in some prototypes in the later stage of the project. In this work package, we mainly focused on the PowerStack path to define use cases and their requirements. We then merged them as a sophisticated use case in addition to those in WP2.

TABLE 1: Tool Coverage

	Common Layer	Monitor		Job Mngr	Node Mngr	System Manager		Workflow Engine
		Estmtr	The Others			SPM	RJMS	
Melissa								
RYAX								
OAR								
Slurm								
BeBiDa								
EAR			*					
BEO			*					
PULPcontroller								
BDPO								
COUNTDOWN								
Examon								
DCDB								
EPCM								
REGALE Lib								

* EAR and BEO have their own database as well.

4. Use Cases and Requirements

In this section, we first describe a big picture of our use cases encompassing PowerStack, workflow engines, and sophistications. We then introduce common requirements all the use cases need to follow. Next, we describe per-component requirements for each use case.

4.1 High-level Overview

Figure 2 illustrates an overview of our power and resource management, with a particular focus on extreme-scale HPC systems. In the early stage of the project, we had three different software integration paths: (1) PowerStack, (2) Melissa, and (3) RYAX paths. In the PowerStack path, we first inherited a strawman architecture as well as its use cases considered in the HPC PowerStack community [1], and we then extended them to realize our prototyping and integrations. On the other hand, the latter two paths dealt with the integrations of pilot applications with our workflow engine tools. These different paths were converged into one as shown in the figure.

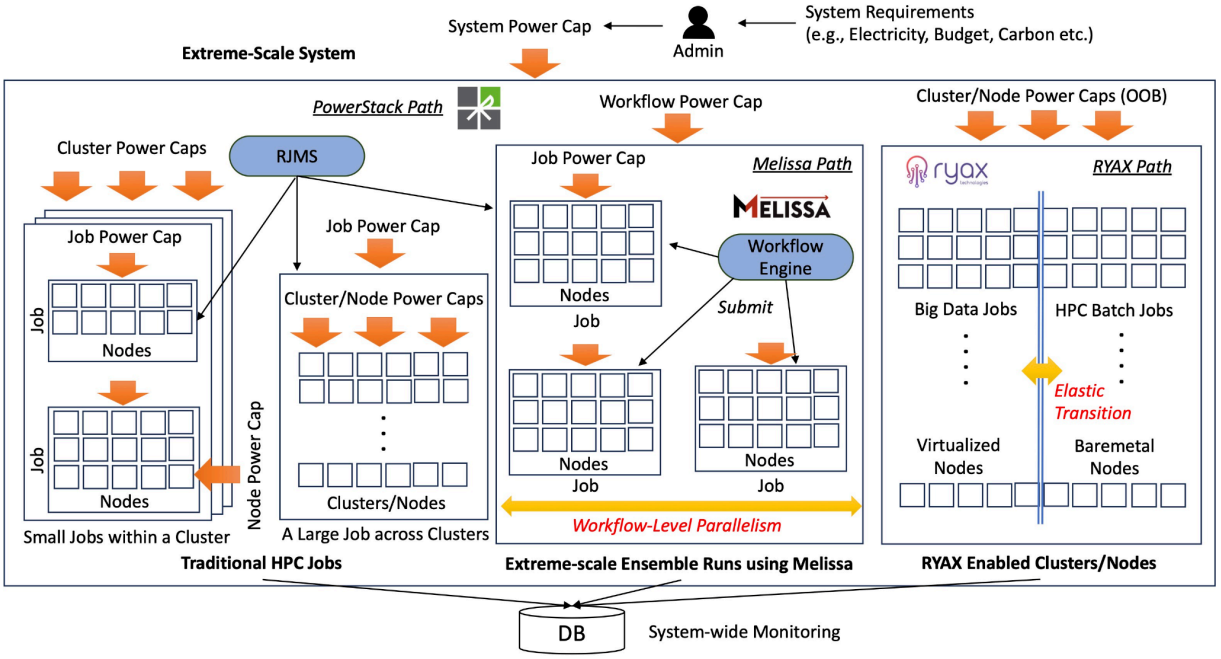

Figure 2: Holistic, Hierarchical, and Scalable Power/Resource Management

The REGALE PowerStack path considers hierarchical power management as follows. First, the system administrator determines the total system power budget and sets the system-wide power cap in accordance with electricity cost, remaining budget, carbon efficiency of the power grid, and so forth. Next, the system power budget is distributed across compute clusters depending on their needs. Then, on each cluster, the assigned power budget is split into the jobs running on the cluster. Finally, the job power budget is divided into per-node power budgets, and the power cap is set on each node. This is the case when we handle smaller jobs than a cluster (e.g., 1K nodes) associated with small-job queues. In the case of running a job larger than the above cluster size, one option is forming a cluster of clusters and determining the power cap to it first, and then we set the job power cap within the bound. In this hierarchical power management scenario, we consider different levels of sophistications from different angles (e.g., dynamic or static).

The Melissa path realizes sophisticated ensemble runs for sweeping a given parameter space, i.e., running an application continuously while changing the inputs. The Melissa workflow engine interacts with the RJMS and decides the concurrency, i.e., the number of jobs running at the same time, depending on the available compute nodes on the system, and several pilot applications were integrated with the tool in the REGALE project. The REGALE PowerStack was extended to support this use case, i.e, the concurrency should be restricted also by the remaining power budget, not limited to the remaining node resources, while taking the power hungriness of jobs into account. To this end, the workflow engine needs to interact also with the system power manager to obtain the usable power budget and with the power estimator to decide per job power budgeting.

The RYAX path enables dynamic Big Data/AI workloads (i.e. Spark) to be executed on HPC environments in an elastic manner. The technique is based on Kubernetes managing the dynamic workload on the HPC side as low-priority HPC jobs through prolog/epilog scripts on HPC scheduler (Slurm, OAR, etc). It provides a feature to dynamically change/trade the scales of virtualized/baremetal regions. The major objective is to minimize the turnaround time of BD jobs with deadline/time-critical aware optimizations. In the REGALE project, we did not explore a sophistication to coordinate this elastic resource management and power budgeting across virtualized/baremetal nodes, however the RYAX path can co-exists with the PowerStack path – one option is forcing the cluster/node power cappings via out-of-band power knobs controlled by the system/node power manager.

4.2 System Requirements

System Architecture: The REGALE project targets both CPU-only homogeneous and CPU-GPU heterogeneous compute nodes. Here, we assume the node hardware architecture is uniform across all compute nodes. Future exa-scale systems could consist of multiple different types of clusters (e.g., Jupiter's modular architecture [5]), and our approach is applicable to each of them separately and independently. The coordination between different clusters by trading the power budgets across them is not covered by the project and would be one of our major future challenges. In our target systems, we apply our power management technique particularly to compute nodes, which is because they are the major power consumers in modern HPC systems, and other system components such as I/O nodes, cooling facilities, network infrastructures are not in the loop of our power management.

Power Knobs: The overall power management is governed by the system power manager daemon launched on the scheduler (or admin) nodes. More specifically, the system power manager (SPM) distributes power budgets across clusters/nodes, which could be in a closed or open loop manner. The closed-loop control makes power budget decisions based on the actual power consumptions, while the open-loop option does not utilize them. The node manager and the monitor are distributed across the compute nodes, and they are responsible for the in-/out-of-band power setups and measurement on each node.

In order to exchange power budgets across different kinds of hardware components, we need a common currency valid among them. For this reason, we require all the target hardware to support a power capping interface to set up the upper limit wattage, regardless of the internal hardware power control features. The power capping needs to be conducted in a closed-loop manner, i.e. the power consumption needs to be measured internally to configure the low-level power control setup (e.g. clock frequency) in an adaptive manner so

as not to exceed the given power cap. Further, the interface needs to offer the measured power consumption to the overlying software stack. This is required especially for a system-wide closed-loop power management use case – for instance, the node manager detects an unused power budget and gives the extra budget back to the system manager. The left diagram of Figure 3 dipicts a software/hardware interface that meets the above requirements. Note, they are commonly supported in modern hardware components (e.g., RAPL [6] for CPUs/DRAMs, and NVML for NVIDIA GPUs [7]), while others such as cooling facilities generally do not offer them and thus require an additional software layer to set a power cap.

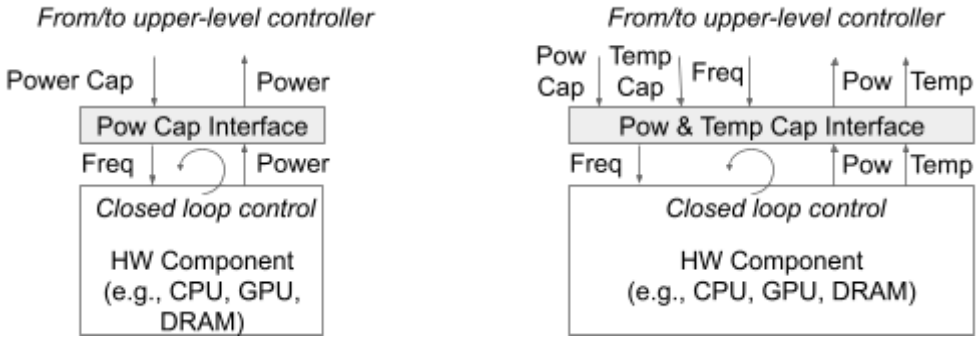

Figure 3: Hardware Interface Requirements

Further Hardware Knobs: A thermal capping capability is also required for several use cases. To this end, the hardware components need to support the following functionalities, similar to the requirements for the power capping feature. First, the thermal cap interface needs to be accessible by the overlying software. Second, the thermal cap is strictly followed by using a closed-loop hardware control. Third, the current temperature needs to be visible also for the overlying software stack so that temperature anomalies become detectable. The right image of Figure 3 illustrates these requirements. The thermal capping feature is widely supported in modern commercial processors. They usually have a temperature controlling hardware to prevent overheating, and if the temperature exceeds a predetermined threshold (usually set very high such as 80°C), the hardware module attempts to throttle the throughput by scaling down the clock frequency, thinning out the clocks, or any other throttling mechanisms. The thermal threshold is exposed to the operating system layer for various processors [8]. Also, the thermal capping can be handled by several node manager tools, such as PULP Controller, BEO, and EAR.

As mentioned above, the power and thermal capping features rely on the clock frequency (and voltage) scaling which almost all the commercial processors support. Several use cases also require direct access to the clock frequency setup in order to trade-off power and performance while keeping the power and thermal constraints (e.g., minimizing energy while keeping the power and thermal constraints). Therefore the clock frequency setup interface is also required for them.

Site Administrator's Inputs: The system administrator plays an important role in our architecture and several use cases. He/she first clarifies their requirements, sets the goal for the system optimization, determines the exact constraints to meet the requirements, and then selects a right management policy with optimal parameter setups to achieve the goal under the given set of constraints. The requirements here are broad encompassing system performance, economical budget, carbon emission requirement, infrastructure limitations,

user experiences, etc. We regard the conversion from these requirements into the policy selection and parameter setups as given by the system administrator, however we would explore this aspect as well in future work (e.g. automatically determining the total system power boundary based on the current carbon efficiency of the power grid).

4.3 Format definition to specify use cases and requirements

In this section, we define the description format of our use cases and their requirements. In general, power and resource management can be described as an optimization problem, i.e., maximizing or minimizing one or more objective function(s) under one or more constraint(s). To this end, power and resources are managed in a hierarchical manner, and they are controlled in each layer at a certain temporal/spatial granularity. To define our use cases, we consider the following factors:

1. **Optimization objectives**
2. **Constraints**
3. **Finest temporal granularity of decisions**
4. **Power management decision levels**
5. **Resource management sophistications**

The first two are to describe the management as an optimization problem, and the rest of them characterize the knobs used to solve the problem. Each use case has a certain level of sophistication for each of the above factors. TABLE 2 lists the level definitions. In the REGALE project, we considered three levels, but we would extend these levels to four or more in future work.

TABLE 2: Different sophistication levels

	Level 1	Level 2	Level 3
1. Optimization Objectives	None	One (system wide or app focused)	Multiple objectives (e.g., both system wide & app focused)
2. Constraints	One (power constraint only)	Two (e.g., power + temperature)	More (e.g., power + temp + anomaly detectable)
3. Finest Temporal Granularity of Decisions	Static (as per human's request)	Static (when job launch)	Dynamic at runtime
4. Power Management Decision Levels	None or human only	One autonomous decision maker	Multi-layered autonomous decision makers
5. Resource Management Sophistications	None (power mgmt is orthogonal to resource mgmt)	One (e.g., energy-aware moldable scheduling)	Multiple (e.g., moldability + co-scheduling)

1. Optimization objectives: One or more objective functions are usually set to determine the goal of optimization in a power/resource management. An objective can be system wide (e.g., maximizing total system throughput or energy-efficiency) or an application-level one (e.g., minimizing application runtime or energy consumption). As we also assume a multi-level resource/power management, there can be multiple objective functions. As an extreme case, the most naive one does not have any objective functions, but just sets power management knobs to functionally enforce constraints without any optimizations.

2. Constraints: We consider one (or more) constraint(s) are set when controlling power management knobs. By default, we assume a system-wide power constraint is set (Level 1) due to an increasing demand for operating supercomputers under a certain power boundary. We then regard temperature as the next-level constraint and is manageable by several REGALE tools (Level 2). Note, in case we have no objective function, we only enforce the limits but do not optimize anything. We further extend the concept of constraints to set the anomaly tolerability/detectability as a requirement (Level 3).

3. Finest temporal granularity of decisions: The temporal granularity of an optimization also reflects how well it is sophisticated. The most naive optimization (Level 1) is that the decision happens only when it is requested by the human actor (e.g., site admin). The next step is that the optimization is applied per job launch (Level 2), however the control state is constant until a new job is launched. The most sophisticated one (Level 3) is optimizing power and resources dynamically at runtime. As the power/resource management can be hierarchical, we consider the finest granular one here.

4. Power Management Decision Levels: Here, we define the sophistication levels of power management. The most naive one is that all power control knobs are manually set by a human actor (Level 1). We also select the Level 1 for a use case if power management is not relevant and is completely orthogonal to it. Then, the next level is that the power control decision is made automatically by only one decision maker in the power control loop (Level 2). The most sophisticated one is that the power management is formed in a hierarchical manner, and multi-layered power controllers interact with each other to optimize the power knob setups (Level 3).

5. Resource management sophistications: Several use cases deal with resource assignment decisions (compute nodes, in-node components, etc.), but others do not. If a use case is completely orthogonal to resource assignment decisions, we choose Level 1. As a next step, if a use case manages resource assignments (e.g., scheduling decisions, node resource partitions, concurrency scaling in ensemble runs, etc.), we select Level 2. If a use case manages multiple factors of resource assignments (e.g., co-scheduling moldable jobs), then it is considered Level 3.

We then specify the requirements for all the components/actors except for the Regale common library layer to realize each use case:

1. **Site Admin**
2. **Users/Developers**
3. **Workflow Engine**
4. **System Manager (RJMS/SPM)**
5. **Job Manager**

6. **Node Manager**
7. **Monitor (Dashboard/Signature Handler/Estimator)**
8. **HW Knobs**

Site Admin can have one or more roles in several use cases. One example is interacting with the system to set up the total system power constraint.

Users/Developers also can have some roles for optimization as well. As an example, for a user-level power or energy optimizations, they need to link some relevant libraries to their codes. Another example is that setting up some environmental variables in their job scripts could be required to enable some features.

Workflow Engine: our workflow engine tools are involved in several use cases, and their requirements are specified for them.

System Manager (including RJMS and SPM), Job Manager, Node Manager, **Monitor** are the software components included in the REGALE architecture (see Figure 1). Some use cases need to coordinate all of them while others may need only some of them. The requirements highly depend on all the aspects that determine the use cases (objectives, constraints, etc).

HW Knobs are essential to realize our PowerStack use cases because they rely on the functionalities the HW knobs offer. We specify the requirements per use case.

Note, we do not list the REGALE common library layer here because it functions rather as a communication abstraction layer across modules, but it does not offer its own functions to realize a use case. The above modules interact with each other directly or via the common library layer depending on if the function is implemented in the layer or not. Note, the REGALE project envisions porting all the necessary API functions to this layer. The details of the interfaces for the common abstraction layer will be described in Section 5.

4.4 Requirement Specifications per Use Case

In this section, we introduce our target use cases and list the requirements for each of them. We first start from basic use cases to realize the PowerStack path, then move on to standard/advanced ones, and finally introduce leading use cases encompassing resource management sophistications via elastic resource management using workflow engines or co-scheduling, i.e., co-locating multiple jobs on the same nodes at the same time.

4.3.1 Basic PowerStack Use Cases

The most basic one is that the site administrator designates the total system power boundary, and then the node power capping is enforced on all compute nodes uniformly to keep the boundary without any optimizations. This power control just follows the human instruction, and the job scheduling and resource assignment optimizations are not coordinated with this power capping. Therefore, all the factors listed in TABLE 2 are Level1 for this use case. This is summarized in TABLE 3.

TABLE 3: Requirement Specifications for Basic Power Capping (Basic)

Use case	Sophistication Levels	Requirements
[**Basic**] Keep my system under power cap **Note:** Providing system-level power capping functionality w/o any optimizations; power budget is distributed evenly across nodes; Open-loop control w/o using measured power at runtime	**Objectives:** None (Level1) **Constraints:** Power (Level1) **Temporal Granularity:** Updated only when the site admin changes the setup (Level1) **Power Management Decision Levels:** Determined only by human (Level1) **Resource Management Sophistications:** None (Level1)	**Site Admin:** Set power cap to System Manager (e.g., 1MW) **Users/Developers:** None **Workflow Engine:** None **System Manager:** Capability/interface to talk to each node manager to set power cap to them; HW profiling functionality to obtain the range of power consumption when scaling the target knob; Report if the power budget setup is outside of the range or if a significant power budget violation happens **Job Manager:** None **Node Manager:** Communicate with HW and set up the power cap based on the instruction by the system manager; report if an error/anomaly happens to the system manager **Monitor:** None **HW Knobs:** Node-level power capping capability (in-band or out-of-band)

We then move this use case one step further with respect to the constraints. More specifically, we augment the temperature capping functionality to Basic – here, we call this use case Basic+. Note that to apply this thermal capping along with the power capping, we need a proper interface as described in Section 4.1. TABLE 4 summarizes the requirements to support this use case. Aside from the necessity for the temperature capping interface, the requirements are almost the same as those of Basic.

TABLE 4: Requirement Specifications for Basic Power and Thermal Capping (Basic+)

Use case	Sophistication Levels	Requirements

[**Basic+**] Keep my system under power and thermal caps **Note:** Providing system-level power and thermal capping functionality w/o any optimizations; power budget is distributed evenly; the same temperature setup for every node; Open-loop control w/o using measured power nor temperature at runtime	**Objectives:** None (Level1) **Constraints:** Power and temperature (Level2) **Knobs:** CPU power & thermal caps (Level1) **Temporal Granularity:** Updated only when the site admin changes the setup (Level1) **Power Management Decision Levels:** Determined only by human (Level1) **Resource Management Sophistications:** None (Level1)	**Site Admin:** Set power and thermal caps to System Manager (e.g., 1MW & 50°C) **Users/Developers:** None **Workflow Engine:** None **System Manager:** Capability/interface to talk to each node manager to set power and temperature caps to them; HW profiling functionality to obtain the range of power consumption when scaling the target knob; Report if the power budget setup is outside of the range or if a significant power budget violation happens **Job Manager:** None **Node Manager:** Talk to HW and set up power and thermal caps based on the instruction by the system manager; report if an error/anomaly happens to the system manager **Monitor:** None **HW Knobs:** Node-level power/thermal capping capability (in-band or out-of-band)

In TABLE 5, we extend Basic+ by adding the anomaly detectability requirement, which we call Basic++ here. If a target hardware region violates the power or thermal limits more than a certain threshold longer than a predetermined duration, this should be reported. Here, we just consider the detection and report functions, but in the future deliverables, we will cover more sophisticated options such as an anomaly tolerance option with automatic anomaly handling methodologies. An anomaly can happen at a variety of granularity levels, and thus anomalies should be detectable at all the software components. In the future work, the anomaly detection, correction, and mitigation should be realized at different levels in a hierarchical manner: (1) application; (2) subsystem; (3) node; and (4) room level. We can consider a variety of use cases even only on anomaly handling methodologies for different scenarios or target hardware.

TABLE 5: Requirement Specifications for Basic Power and Thermal Capping with Anomaly Detectability (Basic++)

Use case	Sophistication Levels	Requirements

[**Basic++**] Keep my system under power and thermal caps with anomaly detectability **Note:** Providing system-level power and thermal capping functionality w/o any optimizations; power budget is distributed evenly; the same temperature setup for every node; Open-loop control w/o using measured power nor temperature at runtime; Anomaly is detectable at any components	**Objectives:** None (Level1) **Constraints:** Power and temperature constraints + anomaly detectable (Level3) **Knobs:** CPU power & thermal caps (Level1) **Temporal Granularity:** Updated only when the site admin changes the setup (Level1) **Power Management Decision Levels:** Determined only by human (Level1) **Resource Management Sophistications:** None (Level1)	**Site Admin:** Set power and thermal caps to System Manager (e.g., 1MW & 50°C); Handle anomaly node reported by System Manager **Users/Developers:** None **Workflow Engine:** None **System Manager:** Capability/interface to talk to each node manager to set power and temperature caps to them; HW profiling functionality to obtain the range of power consumption when scaling the target knob; Report if the power budget setup is outside of the range or if a significant power budget violation happens; Anomaly detection/report function in terms of power and temperature (reported by other components); Report when anomaly is detected to site admin **Job Manager:** Report if an error/anomaly happens to the system manager **Node Manager:** Interact with HW and set up power and thermal caps based on the instruction by the system manager; report if an error/anomaly happens to the system manager **Monitor:** Provides information on facility and nodes anomalies **HW Knobs:** Node-level power/thermal capping capability (in-band or out-of-band)

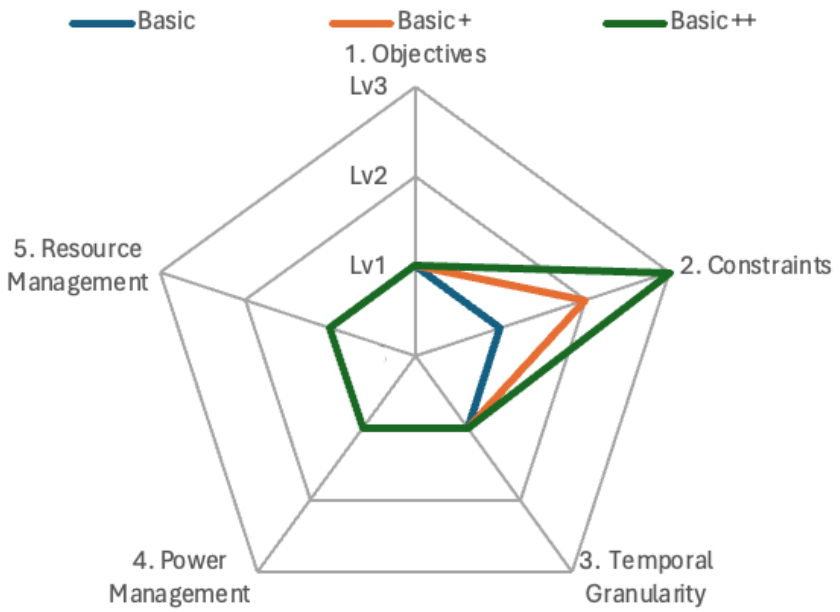

Figure 4: Sophistication Level Coverage by Basic PowerStack Use Cases

Figure 4 illustrates the sophistication levels covered by the above basic PowerStack use cases. As they are all naive use cases with offering functions to enforce power/thermal capping or to detect anomalies, these covered areas are small.

4.3.2 Standard PowerStack Use Cases

Next, we extend the Basic use case by optimizing the hardware power management knob setup while following a given objective function. Here, we cover the following objectives: maximizing total system throughput (SysThru); minimizing total system energy (SysEne); maximizing application performance (AppPerf); and minimizing application energy-to-solution (AppEtS). For all of these use cases, we assume the site administrator sets the power constraint to the entire cluster/system, and then we optimize the power budgeting across these jobs or nodes while keeping the constraint to achieve a given objective. In this sense, we handle multi-level constraints (thus constraint level is at least 2). There are two options in the decision timing: static or dynamic. For the static option, all the decisions are statically given in a proactive way, while for the dynamic option, they are reactive using runtime information. TABLE 6 describes the definition/requirements for each of these options. Here, we consider power is only the constraint, however this can be extended to cover more constraints by adding requirements listed in Basic+ or Basic++. Another option for the constraints is considering average or maximum application performance degradation. For SysThru, we consider closed-loop power controls in these use cases, i.e., we dynamically adjust the power management knob in accordance with the measured power consumption and resource utilizations at runtime. These measurements are used for estimating the power demand of a node by the Node Manager, which is then sent to the System Manager to redistribute the power budgets across nodes. SysEne is almost the same as SysThru except for the objective function and an option to scale down the total power budget allocated to the entire set of compute nodes, which could improve energy efficiency but wouldn't improve throughput. On the other hand, AppPerf and AppEtS are application level (or user level) optimizations, while the system manager does not autonomously optimize the power budget assignments across jobs/nodes. In these use cases, we utilize application profiles of

previous/test runs (static) or runtime information (dynamic), which are provided by Monitor. By analyzing the profiles, Job Manager decides the setups of the target power management knob (CPU power cap, CPU clock frequency scaling or any others) as well as performs code tuning as an option. One needs to link the specific libraries to the code to realize these application level options.

TABLE 6: Requirement Specifications for Standard PowerStack Use Cases

Use case	Sophistication Levels	Requirements
[**SysThru**] Maximizing total system throughput under power cap **Note:** Optimize power budget allocations across nodes under the total system power cap so that the total system throughput can be maximized; Closed-loop power management at runtime or profile-driven proactive approach; Assuming over provisioned situation; Adding more constraints is an option (e,g., thermal cap, application speed-down limit)	**Objectives:** Max system throughput (Level2) **Constraints:** Two-level power capping: system + job/node (Level2) – or extensible to include more (Level3) **Temporal Granularity:** Statistically set at runtime (Level2) or dynamically adjusted at runtime (Level3) **Power Management Decision Levels:** System manager optimize the power budget distributions based on the job characteristics (Level2) **Resource Management Sophistications:** None (Level1)	**Site Admin:** Set power cap to the entire system via System Manager (e.g., 1MW) **Users/Developers:** None **Workflow Engine:** None **System Manager:** All the functionalities supported in Basic; A profile-based power budget decision making (static); Periodical power budget redistribution function based on the reported unused power and power budget request by Node Manager (dynamic); Power budget distribution algorithm to maximize throughput **Job Manager:** None **Node Manager:** All the functionalities supported in Basic; Policy to detect whether the node needs less/more power budget and report it to System Manager (dynamic) **Monitor:** Providing monitoring data to SPM and Node Manager when dynamically adjust the power cap; Providing power/performance estimation to other actors **HW Knobs:** Node-level power (& thermal if needed) capping capability (in-band or out-of-band)
[**SysEne**] Minimizing	**Objectives:** Min system	**Site Admin:** Same as SysThru

total system energy consumption under power cap **Note:** The requirements are almost the same as those for SysThru; Need to update the power distribution policy/algorithm from SysThru, in particular Node Manager level; Adding more constraints is an option (e,g., thermal cap, application speed-down limit)	energy consumption (Level2) **Constraints:** Two-level power capping: system + job/node (Level2) – or extensible to include more (Level3) **Temporal Granularity:** Statistically set at runtime (Level2) or dynamically adjusted at runtime (Level3) **Power Management Decision Levels:** Job manager optimize the power knob setup based on the job characteristics (Level2) **Resource Management Sophistications:** None (Level1)	**Users/Developers:** None **Workflow Engine:** None **System Manager:** Same as SysThru; Scaling down total system cap adaptively is an option **Job Manager:** None **Node Manager:** Same as SysThru but need updates in the power budget request policy, i.e., detecting the optimal power mgmt knob setup to minimize energy (or maximize energy efficiency) **Monitor:** Providing monitoring data to Node Manager when dynamically adjust the power cap; Providing power/performance/energy estimation to other actors **HW Knobs:** Same as SysThru
[**AppPerf**] Maximizing application performance under job power cap **Note:** System manager allows privileged users to control power knobs, but does not necessarily apply any optimization (same as basic); Job manager handles the power knob setups while keeping the given job power budget; Adding more constraints is an option (e,g., thermal cap)	**Objectives:** Max application performance (Level2) **Constraints:** Two-level power capping: system + job (Level2) – or extensible to include more (Level3) **Temporal Granularity:** Statically set when job launch (Level2) or dynamically adjusted at runtime (Level3) **Power Management Decision Levels:** Job manager optimize the power knob setup based on the job characteristics (Level2) **Resource Management Sophistications:** None (Level1)	**Site Admin:** Same as SysThru; Allow user level (or Job Manager level) power management **Users/Developers:** Link the relevant library (provided by Job Manager) to their code **Workflow Engine:** None **System Manager:** Same as Basic **Job Manager:** Power-aware code tuning functionality using code analysis or profile-based optimization; Accessible power management knobs (power cap, clock freq, etc.) **Node Manager:** Same as Basic except that it needs to provide an interface to let Job Manager know the current power knobs

		and allow it to further optimize them **Monitor:** Providing monitored stats to Job Manager **HW Knobs:** Same as SysThru
[**AppEtS**] Maximizing energy to solution for app under power cap **Note:** Almost same as AppPerf except that the objective is minimizing energy; Adding more constraints is an option (e,g., thermal cap, application speed-down limit)	**Objectives:** Minimize application energy to solution (Level2) **Constraints:** Two-level power capping: system + job (Level2) – or extensible to include more (Level3) **Temporal Granularity:** Statically set when job launch (Level2) or Dynamically adjusted at runtime (Level3) **Power Management Decision Levels:** Job manager optimize the power knob setup based on the job characteristics (Level2) **Resource Management Sophistications:** None (Level1)	**Site Admin:** Same as AppPerf **Users/Developers:** Same as AppPerf **Workflow Engine:** None **System Manager:** Same as AppPerf **Job Manager:** Same as AppPerf except that the optimization policy must be updated. **Node Manager:** Same as AppPerf **Monitor:** Same as AppPerf except that the optimization policy must be updated. **HW Knobs:** Same as AppPerf

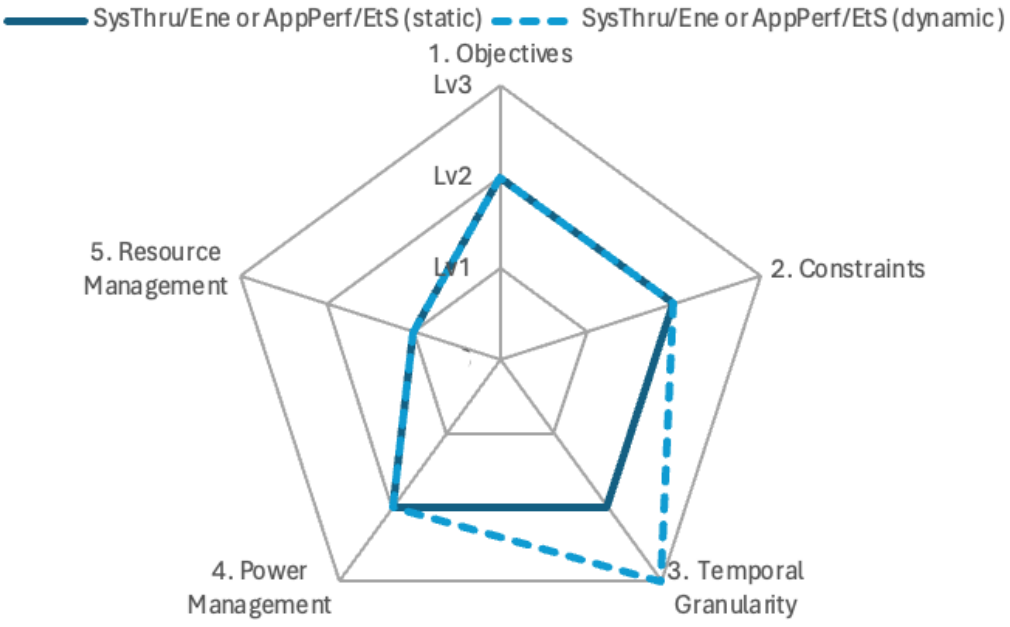

Figure 5: Sophistication Level Coverage by Standard PowerStack Use Cases

Figure 5 illustrates the sophistication levels covered by the above standard PowerStack use cases. Note, they can also cover more constraints such as thermal capping.

4.3.3 Advanced PowerStack Use Cases

We then present our advanced PowerStack use cases: SysThru&AppEtS, NodPowShft, SchedOpt, and PowAwrEns, each of which is an extension of one or more of the standard use case(s) listed in Section 4.3.2. TABLE 7 lists their definition, sophistication levels, and requirements. First, SysThru&AppEtS is a combined version of two standard use cases (SysThru and AppEtS) – we globally attempt to maximize the throughput under a system-wide power bound by optimizing the power budget distributions among jobs or nodes (SysThru), while at the same time we locally try to minimize the energy-to-solution of a job by controlling the hardware knobs while keeping the given power budget. Second, the NodPowShift use case is an extension of one of the standard use cases (or SysThru&AppEtS), i.e., we further shift power among components within a node (CPU, GPU, memory, etc.) while keeping the given node power cap. Third, SchedOpt is another type of extensions for SysThru/Ene, i.e., in addition to optimizing power budget distributions across nodes/jobs, we further optimize job scheduling decisions. One option is power-/energy-aware moldable backfilling, i.e., changing the scale of a backfilling job so that it fits for both the available number of nodes and available power budget as well. Another option is reducing the power bound of currently running jobs in order to obtain sufficient power budget to launch a target job. Finally, PowAwrEns enables power-aware ensemble runs by combining the workflow engine with the standard system-wise power management use cases (SysThru or SysEne).

TABLE 7: Requirement Specifications for Advanced Use Cases

Use case	Sophistication Levels	Requirements
[**SysThru&AppEtS**] Maximizing total system throughput under power cap while minimizing job energy-to-solution; This is a combined use case applying both SysThru and AppEtS at the same time **Note:** Optimize power budget allocations across jobs under the total system power cap so that the total system throughput is maximized; Job manager further optimizes the power knobs to minimize the application's energy-to-solution while keeping the job power boundary; Adding more	**Objectives:** Max system throughput + min job energy to solution (Level3) **Constraints:** Two-level power capping: system + job (Level2) or extensible to include more (Level3) **Temporal Granularity:** Statically set when job launch (Level2) or dynamically adjusted at runtime (Level3) **Power Management Decision Levels:** System manager optimize the power budget distributions based on the job characteristics (Level3) **Resource Management Sophistications:** None (Level1)	**Site Admin:** SysThru + AppEtS **Users/Developers:** SysThru + AppEtS **Workflow Engine:** None **System Manager:** SysThru + AppEtS **Job Manager:** SysThru + AppEtS **Node Manager:** SysThru + AppEtS **Monitor:** SysThru + AppEtS **HW Knobs:** SysThru + AppEtS

constraints is an option (e,g., thermal cap, application speed-down limit); Static or dynamic power management		
[**NodPowShft**] Optimizing node performance/power/energy by shifting power among node components **Note:** This is an extension of the standard use cases or SysThru+AppEne by shifting power budgets among different components within each node; The component- wise power capping is directed by either Job manager (user level) or Node manager (system level)	**Objectives:** Multi-objective optimization – (max/min system throughput/energy or max/min application performance/energy) and max/min node perf/energy (Level3) **Constraints:** Three-level power capping: system + job/node + component (Level3) **Temporal Granularity:** Statically set when job launch (Level2) or dynamically adjusted at runtime (Level3) **Power Management Decision Levels:** System manager optimize the power budget distributions based on the job characteristics and then Node manager optimize the power budgeting among components within the node (Level3) **Resource Management Sophistications:** None (Level1)	**Site Admin:** Same as the inherited use case **Users/Developers:** Same as the inherited use case **Workflow Engine:** None **System Manager:** Same as the inherited use case **Job Manager:** Same as the inherited use case **Node Manager:** Needs a layer to distribute a power cap to different component; Update the policy to exploit the above feature **Monitor:** Same as the inherited use case **HW Knobs:** Component-level power (& thermal if needed) capping capability (in-band or out-of-band)
[**SchedOpt**] Maximizing total system throughput (or minimizing total system energy) under power cap w/ job scheduling optimization **Note:** This is an extended version of the standard SysThru/Ene use cases; Scheduling decision is static based on job characterization using profiles provided by Monitor; Power	**Objectives:** Max system throughput or minimize energy (Level2) **Constraints:** Two-level power capping: system + job /node (Level2) – or extensible to include more (Level3) **Temporal Granularity:** Scheduling decisions are made statically at job launch (Level2) or dynamically adjusted at runtime (Level3) **Power Management Decision Levels:** System	**Site Admin:** Same as SysThru/Ene **Users/Developers:** Same as SysThru/Ene **Workflow Engine:** None **System Manager:** Power-aware scheduling policy using the historical job statistics (RJMS); Accessible to system power status such as remaining power budget on the system (RJMS) **Job Manager:** Same as

management part can be dynamic	manager optimize the power budget distributions based on the job characteristics (Level2) **Resource Management Sophistications:** RJMS is involved in the optimization, such as energy-aware back filling (Level2)	SysThru/Ene **Node Manager:** Same as SysThru/Ene **Monitor:** Providing collected job statistics to characterize jobs and assess the impact of power/node assignments **HW Knobs:** Same as SysThru/Ene
[**PowAwrEns**] Power-aware ensemble runs by combining the workflow engine with PowerStack **Note:** An extension of one of the standard or advanced use cases (SysThru, SysEne, or their extensions) while combining with the workflow engine for ensemble runs; setting the power boundary to the set of ensemble jobs from a user; optimizing the job/node power cap and the concurrency of the ensemble runs	**Objectives:** Multi-objective – PowerStack side + workflow engine side (Level3) **Constraints:** Three-level power capping: system + ensemble set + job/node (Level3) **Temporal Granularity:** Statically set per job launch (Level2) or power cap can be dynamically scaled at runtime (Level3) **Power Management Decision Levels:** Multi-layered – PowerStack side + workflow engine side (Level3) **Resource Management Sophistications:** the job concurrency decision is involved in the power management (Level2)	**Site Admin:** Same as the inherited use case **Users/Developers:** Integrate their app with the workflow engine for ensemble runs **Workflow Engine:** Receive the available power in addition to the available nodes; Decide the number of jobs to launch **System Manager:** Sending the available power to the workflow engine; Setting the power cap to each ensemble job **Job Manager:** Same as the inherited use case **Node Manager:** Same as the inherited use case **Monitor:** Same as the inherited use case **HW Knobs:** Same as the inherited use case

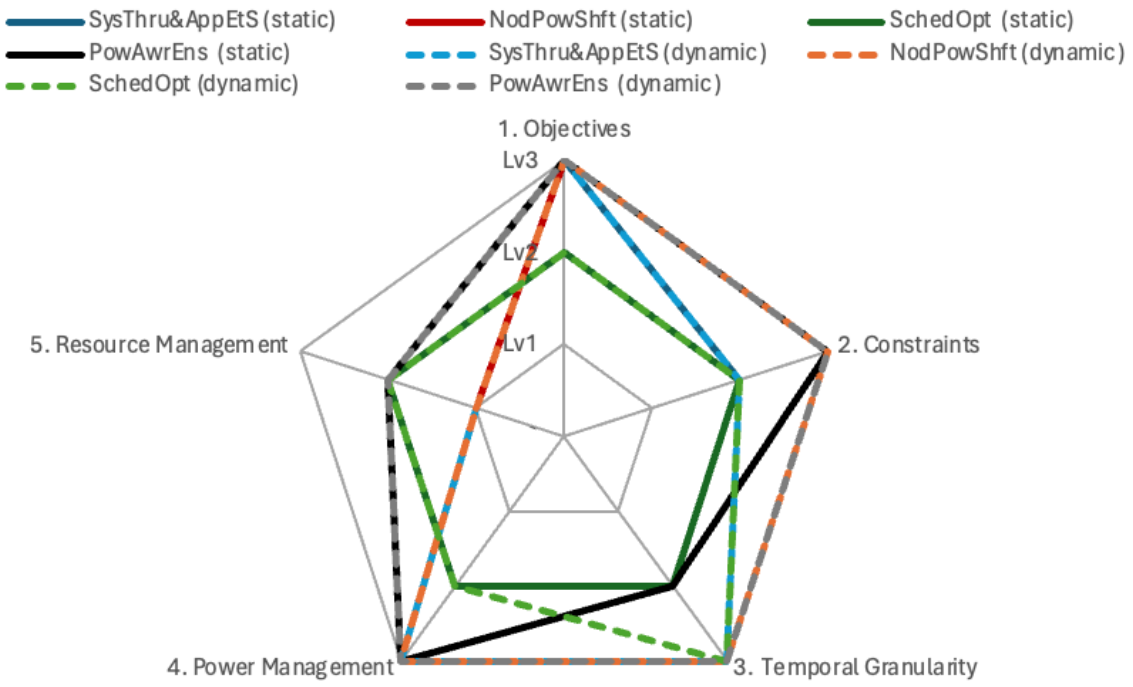

Figure 6: Sophistication Level Coverage by Advanced PowerStack Use Cases

Figure 6 illustrates the sophistication levels covered by the above advanced PowerStack use cases. The SysThru&AppEtS, NodPowShift, and PowAwrEns have hierarchical multi-layered power optimizers, while there is a single-layered power optimizer (System Manager), but the RJMS is involved in the power management in the SchedOpt. Further, PowAwrEns changes the concurrency of ensemble runs based on both the available power and resources through the interactions between the workflow engine and the SPM/RJMS. Note, they have static and dynamic options and also can cover more constraints such as thermal capping.

4.3.4 Sophisticated Resource Management beyond PowerStack

We finally present several resource management sophistications explored in the REGALE project beyond the PowerStack path. TABLE 8 lists the sophistication use cases. The first three sophistications are completely orthogonal to PowerStack use cases (e.g. Basic via out-of-band power control) and are applicable with them at the same time. These sophistications could be potentially combined and coordinated with the standard/advanced PowerStack use cases in future work, by newly introducing coordination mechanisms. In the table, the first sophistication (MoldCosh) aims to enable co-scheduling multiple jobs on the same set of nodes at the same time, in particular using a moldable job scheduling functionality offered by the RJMS. The second one (NodResPart) also targets co-scheduling but rather focuses on the optimization of resource partitioning and co-scheduled job assignments. The third one (ElastSched) offers a coordination mechanism between the workflow engine and the RJMS in order for launching rigid HPC jobs and elastic big data jobs simultaneously on HPC clusters to minimize the resource waste (and thus maximize the system throughput). The last one (MalPowStack) is explored under a collaboration with the DEEP-SEA project via simulations [9]. In this use case, we consider applying power management to malleable job scheduling where the node assignment to jobs can be dynamically scaled at runtime depending on their needs. To realize malleable job scheduling and resource management, the entire HPC software stack from MPI application to resource

manager need to be modified. We further consider applying dynamic job power budget scaling and determine per-node power budgeting as well as job scale.

TABLE 8: Requirement Specifications for Sophisticated Resource Management Use Cases

Use case	Sophistication Levels	Requirements
[**MoldCosh**] Mold and co-schedule a pair of jobs on the same set of nodes **Note:** Allow the RJMS to schedule multiple jobs on the same set of nodes; Mold a pair of complementary jobs so that they fit to the same set of nodes; Need a model-based job characterization to choose a pair; Can co-exist with a use case of PowerStack if it manages power using out-of-band knobs and is independent of the scheduling decisions	**Objectives:** Maximize system throughput + minimize per-job slowdown (Level3) **Constraints:** System-level power constraint (Level1) or more **Temporal Granularity:** Scheduling decisions are made at job launch (Level2) **Power Management Decision Levels:** Power management decision is out of scope by default (Level1), could be coordinated with a multi-layered power control **Resource Management Sophistications:** Job moldability support + node sharing support (Level3)	**Site Admin:** Setup a queue that allows co-scheduling **Users/Developers:** Specify moldable and co-scheduling options in the job script **Workflow Engine:** None **System Manager:** Supports of moldable resource assignment and node sharing (or job overcommit) functions in RJMS; Implementation of a job scheduling algorithm tailored for co-scheduling moldable jobs in RJMS **Job Manager:** None **Node Manager:** Support a functionality for binding node resources (cores, CPUs, etc.) to the co-located jobs and provide it to the system manager **Monitor:** Characterize jobs based on their profiles **HW Knobs:** None
[**NodResPart**] Node-level resource partitioning optimization for co-scheduled jobs **Note:** Optimize node resource partitioning and assignments to co-scheduled jobs; a model-based optimization using job characteristics; can operate under a given node power cap	**Objectives:** Maximize system/node throughput + minimize per-job slowdown (Level3) **Constraints:** System- & node-level power constraints (Level2) or more **Temporal Granularity:** Static decision when job launch (Level2) or dynamic at runtime (Level3) **Power Management**	**Site Admin:** Setup a queue that allows co-scheduling **Users/Developers:** Submit their jobs to the co-scheduling queue **Workflow Engine:** None **System Manager:** Support a node sharing (or job overcommit) functionality and a co-scheduling algorithm in RJMS

	Decision Levels: Power control decision is out-of-scope by default (Level1), could be extended to coordinate with a multi-layered system- + job- + component-wise control (Level3) **Resource Management Sophistications:** Job pair selections, job allocations, and node resource partitioning (Level3)	**Job Manager:** None **Node Manager:** Support a function to optimize resource partitioning knobs and job allocations to the partitioned node regions **Monitor:** Characterize jobs based on their profiles **HW Knobs:** A resource partitioning feature on the target component to control job QoS
[**ElastSched**] Convergence of elastic big data workloads and traditional HPC jobs in an HPC cluster **Note:** Users/developers submit/develop their big data or HPC workflow by using the workflow engine; submitted workflows are executed on the big data or HPC partition inside an HPC cluster; the system manager dynamically adjusts the partition to minimize the resource waste	**Objectives:** Minimize resource waste + improve usability/flexibility (Level 3) **Constraints:** System power constraint (Level 1) **Temporal Granularity:** Dynamic at runtime (Level 3) **Power Management Decision Levels:** Power management decision is out of scope by default (Level1), could be coordinated with a multi-layered power control **Resource Management Sophistications:** Elastic partitioning between the big data and HPC region by the meta scheduler + job scheduling inside of each region (Level 3)	**Site Admin:** Configure a partition to divide a cluster into big data part (virtualized) and HPC batch job part (baremetal) **Users/Developers:** Develop their code using the workflow engine **Workflow Engine:** Provide UI to describe a workflow; interact with the RJMS to submit elastic big data or rigid HPC workflows **System Manager:** Support a meta scheduling function to submit jobs accordingly to the partitioned regions (RJMS); adjust the partition based on their needs (RJMS); have a VM/batch scheduler internally in each region (RJMS) **Job Manager:** None **Node Manager:** Deploy a containerized environment to offer big data workload software stack or MPI-based HPC software stack **Monitor:** None **HW Knobs:** None
[**MalPowStack**] Power-aware malleable job scheduling	**Objectives:** Maximize system throughput + maximize app perf (Level 3)	**Site Admin:** Set power cap to the entire system via System Manager (e.g., 1MW)

Note: Target malleable MPI jobs; Dynamically scale the job power budgets to the currently running jobs; Decide the job scale and per-node power budget for each malleable job	**Constraints:** System + job + node power constraints (Level3) **Temporal Granularity:** Dynamic at runtime (Level 3) **Power Management Decision Levels:** Per-job power budgeting + per-node power budgeting (Level3) **Resource Management Sophistications:** the RJMS's job launch decisions + the job manager's resource allocation decisions (Level3)	**Users/Developers:** Develop their code using malleable MPI **System Manager:** Support dynamic resource add/remove for malleable MPI jobs (RJMS); support dynamic job power budgeting while setting total system power cap (SPM) **Job Manager:** Set num of nodes and per-node power cap based on scalability for a given job **Node Manager:** Access hardware power knobs to keep the given node power cap **Monitor:** Dynamically report job statistics; analyze job scalability in power budget and num of nodes **Workflow Engine:** None **HW Knobs:** Node-level power capping capability (in-band or out-of-band)

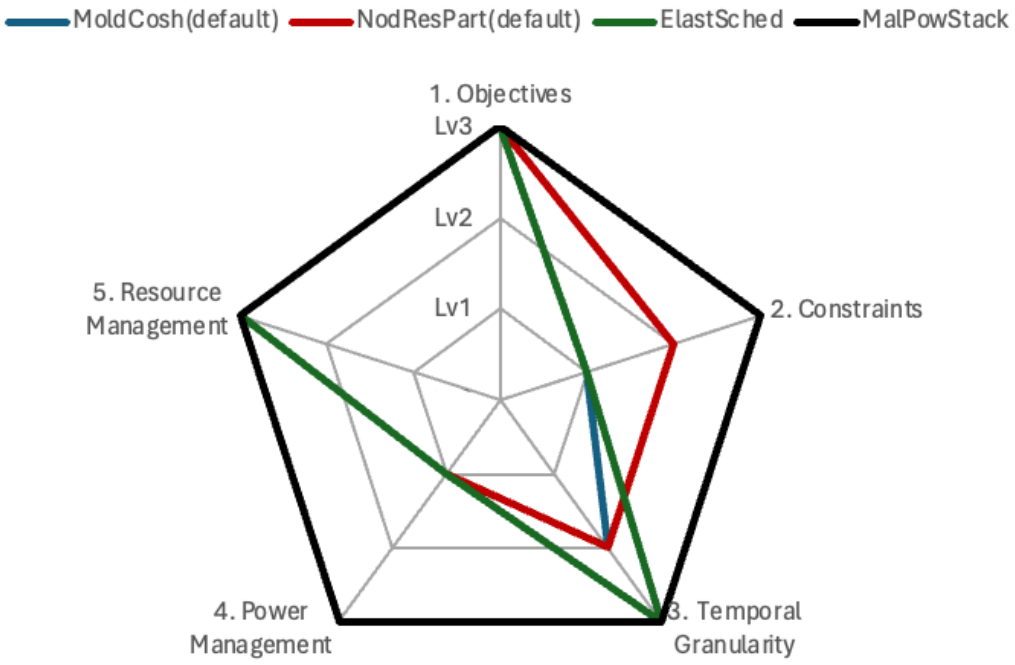

Figure 7: Sophistication Level Coverage by Sophisticated Use Cases

Figure 7 illustrates the sophistication levels covered by the above sophisticated use cases. As the first three use cases are orthogonal to power management, they all can be potentially combined with the PowerStack use cases, and thus can potentially cover a larger area. The MalPowStack is the most ambitious use case, reaching the level 3 in all aspects, as it dynamically scales resource/power in a hierarchical manner. As such, it requires significant updates in the current software stack, and thus cross-project collaborations are inevitable.

4.3.5 Use Case Coverage

TABLE 9 summarizes the use cases and classes we defined in this deliverable. Some of them were aligned with the PowerStack initiative community [1]. Our major contributions here are (1) describing them in a generalized form while defining the levels of sophistication in different aspects shown in TABLE 2, and (2) extending the use cases to reflect modern sophisticated resource/power management techniques, including ensemble runs, co-scheduling, and elastic/malleable resource management.

TABLE 9: Summary of classes and use cases

Class	Code name
Basic	Basic (WP3), Basic+ (WP2), Basic++ (pap [10])
Standard	SysThru (WP3), SysEne (WP2), AppPerf (WP2/WP3), AppEtS (WP3)
Advanced	SysThru&AppAtS (WP3), NodPowShft (WP3), SchedOpt (WP2/WP3), PowAwrEns (WP2/WP3)
Sophisticated	MoldCosh (WP2), NodResPart (WP2), ElastSched (WP2), MalPowStack (pap [9])

Most of them were included in the software integration scenarios in WP3 or sophistication implementations explored in WP2. Several use cases were not covered by these work packages but were presented in our scientific papers. For instance, an ML-based anomaly detection mechanism was proposed in [10], and the convergence of malleable MPI job scheduling and the PowerStack path was explored via our simulation in [9]. In our future work, we could extend these use cases also for the carbon reduction purpose as we presented in [11]. The carbon emission of an HPC system is denoted as the time integral of total system power consumption multiplied by carbon intensity (i.e. the ratio of power supply generated by burning fossil fuels) of the power grid where the supercomputing site is located. It is known that the carbon intensity varies across time depending on the availability of renewable power supplies. Therefore, automatically scaling the total system power bound in accordance with the carbon intensity variability is a promising solution to limit the carbon emission of the site, which could operate at the top of the power management hierarchy. We will share our extensions with the PowerStack community as a branch of their activities and will continue the extensions in the community as well.

5. Architecture and Interface Descriptions

In this section, we introduce the details of our architectural entities and their high-level interfaces to realize the use cases described in the last section. Clarifying the functionalities and interfaces for each architectural module is pivotal for software tool assessments and integrations as well as the standardization procedure. In the last deliverable (D1.2), we defined our architecture with a necessary and sufficient set of interface functions to realize the PowerStack use cases, in particular from a top-down point of view. They were revisited in WP3 based on a bottom-up approach defining the list of interfaces needed by the tools. Then, the core functions used in our major use cases and integration scenarios were implemented in the ***REGALE common library*** to enable extensible, scalable, and secure power management. Throughout this procedure, the naming, selections, definitions of these functions have been changed from the last edition. Also, note several functions introduced here are not yet available in the REGALE common library, as they are less fundamental, however including them eventually in the future version of our REGALE library would be a great addition.

5.1 REGALE Final Architecture

Figure 8 depicts the REGALE final architecture and mapping of our high-level API functions on it. As shown in the figure, the architecture and its API functions have been significantly modified since the last edition. This is due to our decision to have a common library layer to bridge software modules for extensibility, security, and scalability. The REGALE common library is implemented using a DDS middleware that works in a publisher-subscriber pattern. By doing so, a new software module can easily join in our power and resource management loop by simply registering it as a publisher or subscriber of a certain API function. As for security or scalability, see our descriptions in Section 7.1 or D1.4.

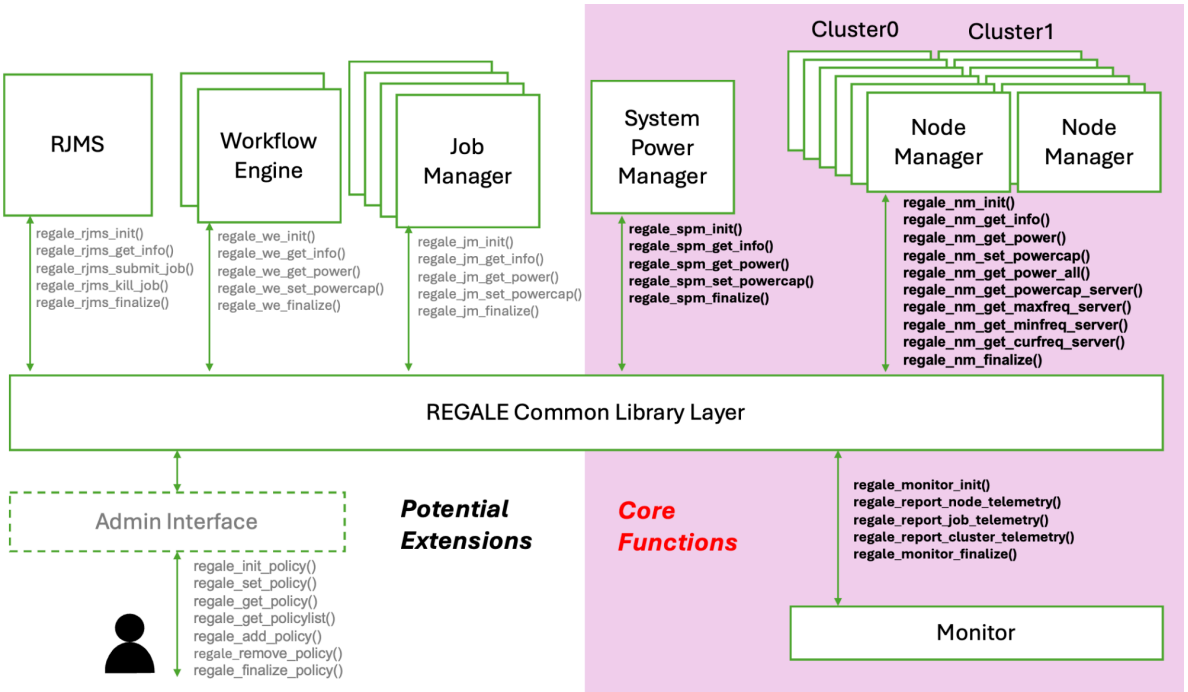

Figure 8: Overall Architecture and API Mapping

In the current version of REGALE common library, we target the combination of the system power manager, node manager, and monitoring tool. Their core interface functions are already available there. In our future extensions, we consider putting more software tools in this power/resource management loop, such as the job manager, workflow engine, and RJMS, in order to realize all the standard and advanced use cases. Although our software tools were directly integrated with each other in some integration scenarios, our ambition here is realizing all the necessary integrations and interactions via the REGALE common library layer. Therefore, we also describe API functions on each component, which would be potentially exposed to the others via the REGALE common library layer. In addition, we also consider several functions to select a power management policy, which would be exposed to the site administrator and would be useful when using our software toolchain.

5.2 REGALE API Descriptions

This section lists our API functions per software components. The core functions that were already implemented in the REGALE common library are shown with a bold font. Here, we cover functions for the Basic/Standard/Advanced use cases to focus on power management. The Sophisticated resource management use cases require more complexities inside of software modules and additional interface functions are also needed to enable flexible resource management for co-scheduling, elasticity, or malleability.

[System Power Manager]

The following functions with the bold font are core functions used from the Basic to the Advanced use cases. As these API functions are essential to realize our use cases, they are actually implemented and available in the REGALE common library.

```
regale_state_t regale_spm_init()
        # Initialization of the system power manager
        OUT regale_state_t out              # Error flag

regale_state_t regale_spm_get_info(regale_info_t *info);
        # Get the SPM status/setup information
        OUT regale_info_t *info             # Obtained status/setup of SPM
        OUT regale_state_t out              # Error flag

regale_state_t regale_spm_get_power(regale_power_status_t *power)
        # Read current system power consumption
        OUT regale_power_status_t *power  # System power consumption output
        OUT regale_state_t out              # Error flag

regale_state_t regale_spm_set_powercap(uint32_t power, bool wait_for_ack)
        # Set system-wide power capping
        IN uint32_t power                   # Total system power cap
        IN bool wait_for_ack                # Flag to tell if the capping is complete or not
        OUT regale_state_t out              # Error flag

regale_state_t regale_spm_finalize()
        # Finalization of the system power manager
        OUT regale_state_t out              # Error flag
```

The following functions are for some of our use cases that control temperature (e.g., Basic+). So far, the temperature management functions are not supported in the REGALE common library as the temperature capping does not require a sophisticated approach but is typically set statically and uniformly. However, it is extensible to support them as well.

```
regale_state_t regale_spm_get_temp(regale_temp_status_t *temp)
        # Read current system temperature
        OUT regale_power_status_t *temp   # System-wide temperature output
        OUT regale_state_t out            # Error flag

regale_state_t regale_spm_set_tempcap(uint32_t temp, bool wait_for_ack)
        # Set system-wide temperature capping
        IN uint32_t temp                  # System-wide temperature cap
        IN bool wait_for_ack              # Flag to tell if the capping is complete or not
        OUT regale_state_t out            # Error flag
```

[Node Manager]

The following API functions are important to manage nodes and are already exposed via the REGALE common library.

```
regale_state_t regale_nm_init()
        # Initialization of the node manager
        OUT regale_state_t out            # Error flag

regale_state_t regale_nm_get_info(regale_info_t *info)
        # Get the node manager status/setup information
        OUT regale_info_t *info           # Obtained status/setup of node manager
        OUT regale_state_t out            # Error flag

regale_state_t regale_nm_get_power(uint32_t *power)
        # Read current node power consumption
        OUT uint32_t *power               # Node power consumption output
        OUT regale_state_t out            # Error flag

regale_state_t regale_nm_set_powercap(uint32_t power)
        # Set node-level power capping
        IN uint32_t power                 # Node power cap
        OUT regale_state_t out            # Error flag

regale_state_t regale_nm_get_power_all(uint32_t *power, int32_t expected_nodes)
        # Read current node power consumption
        OUT uint32_t *power               # Output of node power consumptions
        IN uint32_t expected_nodes        # Nodes selector
        OUT regale_state_t out            # Error flag

regale_state_t regale_nm_get_powercap_server(uint32_t *power)
        # Read the current node powercap
        OUT uint32_t *power               # Node powercap output
        OUT regale_state_t out            # Error flag
```

```
regale_state_t regale_nm_get_maxfreq_server(uint32_t *max_freq)
        # Read the current maximum frequency
        OUT uint32_t *max_freq                  # Maximum frequency output
        OUT regale_state_t out                  # Error flag

regale_state_t regale_nm_get_minfreq_server(uint32_t *min_freq)
        # Read the current minimum frequency
        OUT uint32_t *min_freq                  # Minimum frequency output
        OUT regale_state_t out                  # Error flag

regale_state_t regale_nm_get_curfreq_server(uint32_t *cur_freq)
        # Read the current clock frequency
        OUT uint32_t *cur_freq                  # Current frequency output
        OUT regale_state_t out                  # Error flag

regale_state_t regale_nm_finalize()
        # Finalization of the node manager
        OUT regale_state_t out          # Error flag
```

The following are not yet exposed via the REGALE common library, but doing so would be beneficial to support more use cases. Here, we assume component-level power control functions are not exposed to the REGALE common library layer as the intra-node power shifting is managed within the node manager.

```
regale_state_t regale_nm_get_temp_server(uint32_t *temp)
        # Read the current temperature
        OUT uint32_t *temp              # Temperature output
        OUT regale_state_t out          # Error flag

regale_state_t regale_nm_set_temp_server(uint32_t temp)
        # Set the temperature capping
        IN uint32_t temp                # Temperature cap
        OUT regale_state_t out          # Error flag

regale_state_t regale_nm_set_maxfreq_server(uint32_t max_freq)
        # Set the current node powercap
        IN uint32_t max_freq            # Set up max frequency
        OUT regale_state_t out          # Error flag

regale_state_t regale_nm_set_minfreq_server(uint32_t min_freq)
        # Read the current node powercap
        IN uint32_t min_freq            #  Set up min frequency
        OUT regale_state_t out          # Error flag

regale_state_t regale_nm_set_freq_server(uint32_t freq)
        # Read the current node powercap
        IN uint32_t freq                #  Set up frequency
        OUT regale_state_t out          # Error flag
```

[Monitor]

The following API functions already exist in the REGALE common library.

```
regale_state_t regale_monitor_init()
        # Initialization of the monitor
        OUT regale_state_t out              # Error flag

regale_state_t regale_report_node_telemetry(regale_node_id_t *node,
regale_node_data_t *data)
        # Get the node-level telemetry report
        IN regale_node_id_t *node           # Node ID input
        OUT regale_node_data_t *data        # Node-level telemetry data output
        OUT regale_state_t out              # Error flag

regale_state_t regale_report_job_telemetry(regale_job_id_t *job, regale_job_data_t
*data);
        # Get the job-level telemetry report
        IN regale_job_id_t *node            # Job ID input
        OUT regale_job_data_t *data         # Job-level telemetry data output
        OUT regale_state_t out              # Error flag

regale_state_t regale_report_cluster_telemetry(regale_cluster_id_t *cluster,
regale_cluster_data_t *data)
        # Get the cluster-level telemetry report
        IN regale_cluster_id_t *cluster     # Cluster ID input
        OUT regale_cluster_data_t *data     # Cluster-level telemetry data output
        OUT regale_state_t out              # Error flag

regale_state_t regale_monitor_finalize()
        # Finalization of the monitor
        OUT regale_state_t out              # Error flag
```

The above telemetry functions are used to obtain the statistical information of job, node, or cluster behaviors. Here, we can consider several options with respect to what exact data we send. For instance, the monitoring module could also send characterization or estimation data generated by the estimator inside of it. By doing so, we could share a common model among different components, instead of having their own different models distributed across different modules.

[RJMS]

The following functions are not yet available via the REGALE common library, and so far one needs to directly communicate with the RJMS software tool. However, these functions are important to realize advanced use cases when the RJMS is involved in the power management loop.

```
regale_state_t regale_rjms_init()
        # Initialization of the RJMS
        OUT regale_state_t out              # Error flag
```

```
regale_state_t regale_rjms_get_info(regale_info_t *info)
        # Get the RJMS status information (e.g., queueing status, job=>node mappings)
        OUT regale_info_t *info          # Obtained status/setup of RJMS
        OUT regale_state_t out           # Error flag

regale_state_t regale_rjms_submit_job(regale_job_t *job)
        # Submit a job
        IN regale_job_t *job             # Pointer to the job submission info
        OUT regale_state_t out           # Error flag

regale_state_t regale_rjms_kill_job(regale_job_id_t job)
        # Kill a job
        IN regale_job_id_t job           # Job ID to kill
        OUT regale_state_t out           # Error flag

regale_state_t regale_rjms_finalize()
        # Finalization of the RJMS
        OUT regale_state_t out           # Error flag
```

[Workflow Engine]

The following functions are not yet available via the REGALE common library, but relevant functions are needed for interactions between the Workflow Engine and the PowerStack to converge these paths.

```
regale_state_t regale_we_init()
        # Initialization of the workflow engine
        OUT regale_state_t out           # Error flag

regale_state_t regale_we_get_info(regale_info_t *info)
        # Get the workflow information
        OUT regale_info_t *info          # Workflow information
        OUT regale_state_t out           # Error flag

regale_state_t regale_we_get_power(uint32_t *power)
        #Read out workflow-level power consumption
        OUT uint32_t *power              # Workflow power consumption
        OUT regale_state_t out           # Error flag

regale_state_t regale_we_set_powercap(uint32_t power)
        # Set workflow-level power capping
        IN uint32_t power                # Workflow power cap
        OUT regale_state_t out           # Error flag

regale_state_t regale_we_finalize()
        # Finalization of the workflow engine
        OUT regale_state_t out           # Error flag
```

[Job Manager]

The following would be required to set up job level power management. The job manager tool then calls the REGALE APIs connected to the Node Manager for such as power budget distributions across nodes.

```
regale_state_t regale_jm_init()
        # Initialization of the job manager
        OUT regale_state_t out          # Error flag

regale_state_t regale_jm_get_info(regale_info_t *info)
        # Get the job information
        OUT regale_info_t *info         # Job info
        OUT regale_state_t out          # Error flag

regale_state_t regale_jm_get_power(uint32_t *power)
        #Read out job-level power consumption
        OUT uint32_t *power             # Job power consumption
        OUT regale_state_t out          # Error flag

regale_state_t regale_jm_set_powercap(uint32_t power)
        # Set job-level power capping
        IN uint32_t power               # Job power cap
        OUT regale_state_t out          # Error flag

regale_state_t regale_jm_finalize()
        # Finalization of the job manager
        OUT regale_state_t out          # Error flag
```

[Admin Interface]

The following interface functions are meant to be offered to the site administrator to select or configure a power/resource management policy for improving manageability.

```
regale_state_t regale_init_policy()
        # Initialization of the power/resource management policy setup
        OUT regale_state_t out          # Error report

regale_state_t regale_set_policy(regale_policy_t policy)
        # Set a policy as an instance of a use case
        IN regale_policy_t policy       # Policy to set
        OUT regale_state_t out          # Error report

regale_state_t regale_get_policy(regale_policy_t *policy)
        # Get the current policy
        OUT regale_policy_t policy      # Current policy
        OUT regale_state_t out          # Error report

regale_state_t regale_get_policylist(regale_info_t *policylist)
        # List of power/resource management policies registered on this system
        OUT regale_info_t *policylist   # Policy list output
```

```
        OUT regale_state_t out              # Error report

regale_state_t regale_add_policy(regale_info_t *policyconfig)
        # A new power management policy to install on this system
        IN regale_info_t *policyconfig      # Descriptor of policy configuration file
        OUT regale_state_t out              # Error report

regale_state_t regale_remove_policy(regale_policy_t policy)
        # Remove a policy from the list
        IN regale_policy_t policy           # Policy to remove
        OUT regale_state_t out              # Error report

regale_state_t regale_finalize_policy()
        # Finalization of the general policy setup
        OUT regale_state_t out              # Error report
```

These functions are to be exposed to the site administrator so as to designate a power management policy (regale_policy_t policy) from a set of policies (regale_info_t *policylist) derived from the use cases. Once the policy setup is changed, the associated configuration or plugin is selected at each component (e.g., SPM, node manager, RJMS, etc.). The output variable is the acknowledgement or error type to report if the policy setup completes properly or not. These functions are not yet supported in the REGALE common library, but they will be promising additions in order to orchestrate the module setups to enable each use case.

The site administrator should be able to add/remove/modify a policy freely. To this end, the following interface should also be provided. The function regale_gen_add_policy() is used to install a new policy with the policy setup (e.g. json, xml, etc.).

5.3 Case Studies

In this section, we make cases for various representative use cases to understand how our architectural modules interact with each other using the APIs mentioned above so as to enable the use cases. Note, in case these functions are not yet available in the REGALE common library, the layer is simply ignored and a relevant tool specific interface is used. We are planning to move them to the

[Basic]

Once the total system power cap is designated by the system administrator, the power cap setup request is sent to the REGALE common library. The system power manager reacts to the request, naively divides the budget evenly to all nodes on the system/cluster, and then it sends out the node power cap requests. Then, the node manager on each node reacts to it, sets up the power capping to the node, and returns the acknowledgement. This simple procedure is illustrated in Figure 9. In the case of Basic+, we also apply temperature capping in the same way using different functions.

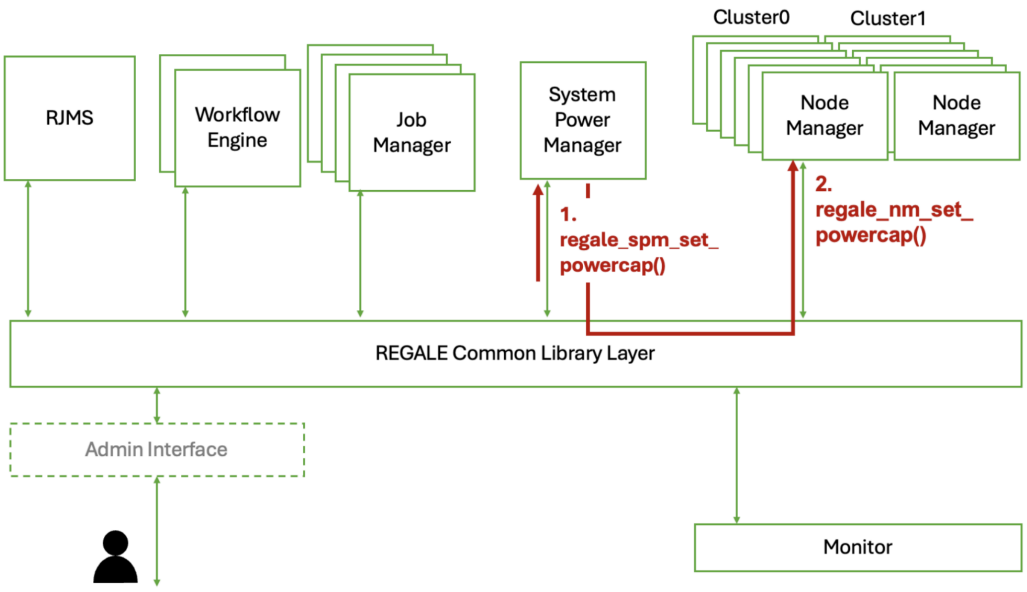

Figure 9: Component interactions in Basic

[SysThru/SysEne]

In SysThru/SysEne, system-level throughput/energy optimizations are involved, which are usually based on statistical data to characterize the running jobs. Therefore, before sending out the power capping requests to the target nodes, the system power manager accesses the job/node associated monitoring data to help with optimizing the power capping decisions. Figure 10 illustrates this additional process. If it utilizes job telemetries, it needs to know the jobs to nodes mapping, which can be obtained from the RJMS using such as regale_rjms_get_info(). The trigger of power capping update can be a job launch, and thus the RJMS should be able to send the system power manager to update the power capping setup using such as regale_spm_set_powercap().

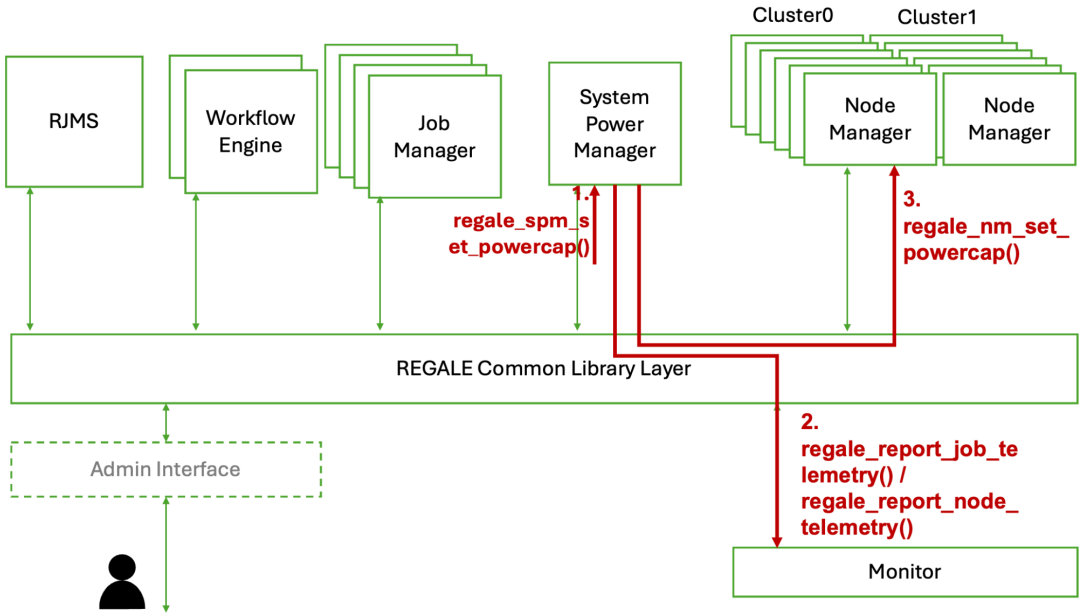

Figure 10: Component interactions in SysThru/SysEne

[SysThru&AppEtS]

In this use case, application-level energy optimizations are also involved in addition to the system-level throughput optimization mentioned above. In our integration, the system power manager decides the power capping, while the job manager optimizes the clock frequency while keeping the given job power limit, or sum of the node power limits where the job is mapped. For both decisions, the associated job telemetry is used. Figure 11 depicts how the entire procedure goes.

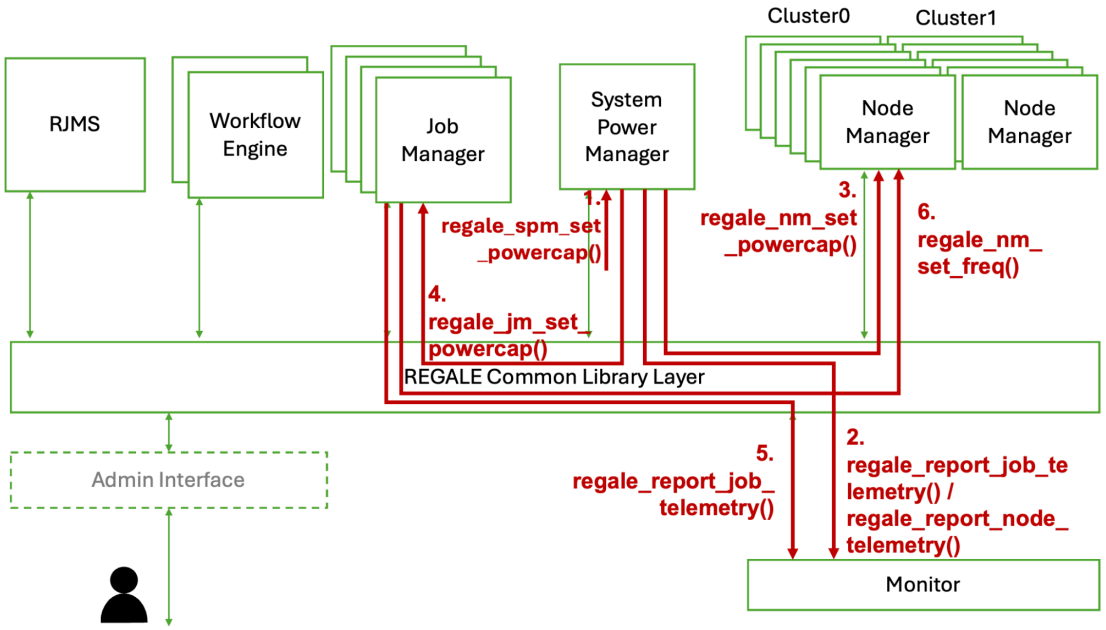

Figure 11: Component interactions in SysThru&AppEtS

Figure 12 shows an alternative approach to realize the use case. In this approach, the job manager decides both the node power capping and clock frequency setup. As users can be in the decision loop, the power capping control privilege needs to be managed carefully.

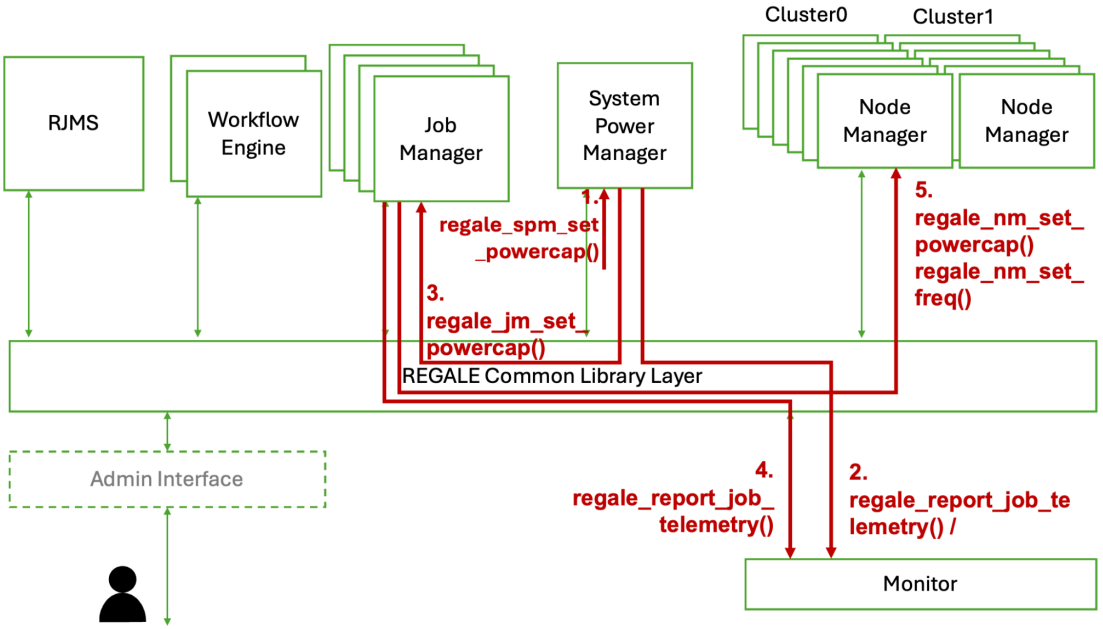

Figure 12: A Variant Implementation for SysThru&AppEtS

[SchedOpt]

In this use case, the RJMS is additionally included in the power management optimization loop. It interacts with the system power manager to obtain the remaining system power budget, i.e., system-level power boundary minus current system power usage. Then this information is used for optimizing scheduling decision making, such as energy-aware backfilling using moldable jobs. This decision should be also made based on the associated job telemetry data. Further, we can consider using the RJMS information (e.g., job queueing state) in the system power manager side as well when determining the power capping setups to nodes or jobs. Figure 13 summarizes the additions in this use case.

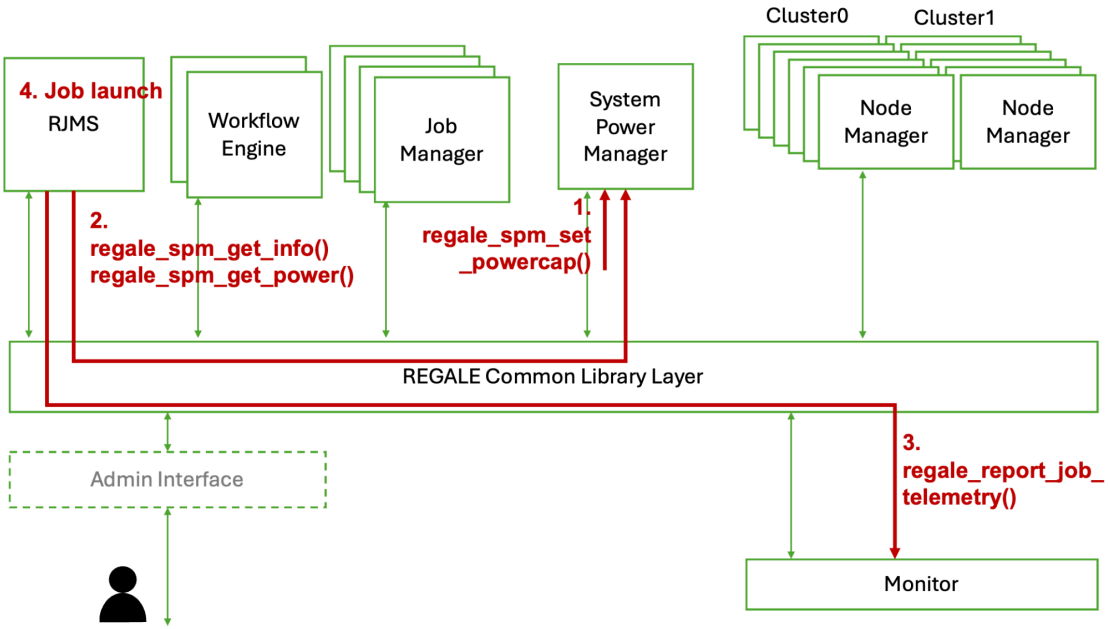

Figure 13: Additional Component Interactions for SchedOpt

[PowAwrEns]

Finally, we introduce a convergence use case between an HPC workflow management and the HPC PowerStack. Here, we assume ensemble runs of scientific applications using such as Melissa, however the procedure here in general works if a workflow engine sets up both the concurrency of jobs and per-job power boundary under a given workflow power cap. Figure 14 illustrates the overall procedure except for the node-/job-level power budgeting. The system power manager first sets the workflow power boundary. Then, within the boundary workflow engine tries to co-optimize per-job power budgeting as well as how many jobs can concurrently run by submitting/killing jobs within the workflow. The workflow engine tools are managed also by users, and thus the power control privilege needs to be managed carefully. This aspect will be discussed in Section 7.

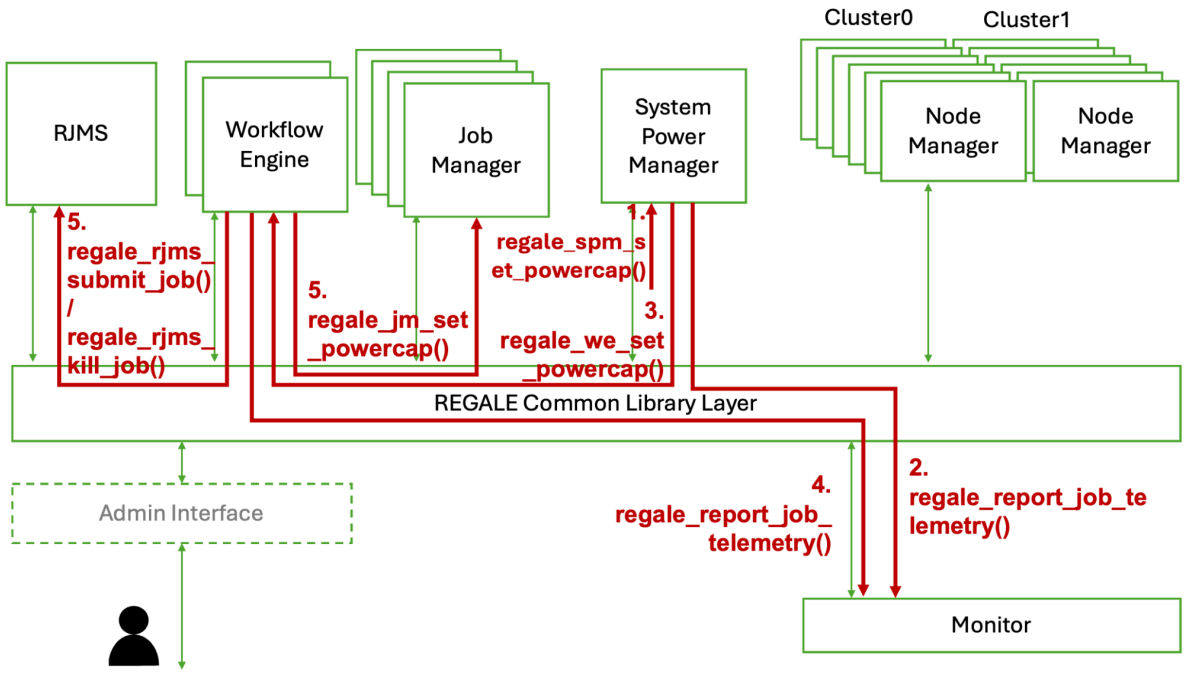

Figure 14: Additional Component Interactions for PowAwrEns

6. Integration Overview

In this section, we first introduce an assessment of our software tools we made to realize the PowerStack path, i.e., mapping them to the architectural components specified in the last section and discussing the missing pieces in order to support these use cases. We then move on to integration plans conducted based on this assessment. We then finally introduce the summary of our integration in WP3 to present how the use cases were converted into the integrations.

6.1 Tool Assessment for PowerStack

TABLE 10: Candidates for System Manager / Monitor and Functionality Support / = Fully Supported or Need Minor Updates, X = w/ Some Restrictions or Need Moderate/Major Updates, Blank = No (or Almost No) Support

	System Manager			Monitor						
	RJMS	SPM	API/Lib	Sensors		Estimator		DB	Signature	API/Lib
				In band	Out of band	Anomaly	Pow/perf			
SLURM /OAR	/	X	/	/		X		/		/
DCDB			/	/	/	X	/	/	/	/
Examon			/	/	/	X	/	/	/	/
BEO				/	X	X				/
EAR		X	/	/		X		/		/

TABLE 10 lists the relevant tools for System Manager and Monitoring to realize the PowerStack use cases. SLURM and OAR are RJMS tools, and they support key functionalities including job scheduling and token management. They also support other functionalities with respect to power management (SPM) and statistics recording (monitoring), however some other tools offer more functionalities. To implement the SchedOpt use case, a new scheduling plugin needs to be developed. As for the SPM, EAR (EARGM) is one of the best options because the interaction with SLURM to obtain job information is already supported in the tool via SLURM plugins/APIs. As for the statistics analysis functions, several monitoring tools (e.g., DCDB and Examon) already support them, such as ML-based modelings, and they are extensible and changeable by plugins. These analytics functions can be extensible and will be useful for a variety of use cases.

As for the monitoring aspect, DCDB and Examon can measure both in-band and out-of-band sensors periodically (from 0.1Hz up to 100Hz of frequency depending on what we measure) and are extensible to support any sensors by plugins, including such as those in cooling facilities. These collected information is recorded with the associated job information on their database – these tools are also able to interact with SLURM to obtain job information. The

measurement function is accessible by other tools, such as those Node Manager tools, by using their APIs. For the above strengths, DCDB and Examon are selected for the mainstream option of the Monitor module, however the others also can work as the Monitor in several implementations depending on the use cases.

TABLE 11: Candidates for Node/Job Manager and Current Functionality Support / = Fully Supported or Need Minor Updates, X = w/ Some Restrictions or Need Moderate/Major Updates, Blank = No (or Almost No) Support

	Node Manager			Job Manager			HW access	
	Closed loop ctr	Access to sys mngr	API/Lib	Profiling	Pow control	API/Lib	In-band	Out-of-band
SLURM	X	/	/				/	
PULP ctr	/						/	X
BEO	X	X	/				/	X
BDPO				/	X	/	/	
EAR		/	/	/	X	/	/	
Countdown					X	/	/	

Next, TABLE 11 summarizes the candidates for Node or Job Managers, and the current support for their key functionalities as well as their in-band or out-of-band hardware knob/sensor accessibilities. The SLURM node daemon works together with the SLURM system controller, and there are a variety of API functions to access their information. Although the default power management support in the node daemon is limited (e.g., no node-level power shifting support), the tool is useful as an interface to interact with the system manager. PULP controller is a low-level power controller, works transparently to the application, user, and system software, currently targeting EPI processors [12]. The tool can access both in-band and out-of-band sensors and automatically optimizes power management knobs using model predictive control algorithms under thermal and power limits. BEO is an out-of-band power monitoring and controlling tool. The supported hardware is a set of AMD/Intel machines in the Atos catalog because the tool is developed in Atos, with a particular focus on their products, and a plugin is needed for other systems. It monitors power consumption using out-of-band sensors and can set the power cap using the in-band RAPL interface. The tool is going to be improved by implementing the following: setting the node power/termal capping via Slurm (Basic/Basic+); and sophisticated power control and anomaly detection mechanisms. BDPO is a job-oriented profile-based power-performance optimization tool, which optimizes clock frequency to trade-off performance and energy or to minimize energy (AppEtS use case supported), and can be extensible to cover other job-oriented use cases. EAR is another job-oriented power-performance monitoring tool. It transparently optimizes the power management knobs on CPUs and GPUs using the profiles of previous runs that are automatically detected. The power management policies are implemented as plugins. The tool currently supports the minimizing energy to solution (AppEtS) use case, and is extensible to cover other use cases such as AppPerf or NodPowShft. COUNTDOWN is another tool that enables job-level power/performance optimizations based on a different focus than others. It tries to minimize the power consumption while waiting for the completion of an MPI communication, by scaling down the

clock frequency or going into one of the CPU sleep states (C-state). It targets Intel CPUs, but is going to support other hardware including GPUs. The tool will reinforce the AppEtS use case and will open up new research opportunities and use cases, which are going to be covered in the future deliverables.

6.2 Conversion into Integrations

Based on the above tool integration assessment, we determined several integration scenarios (or instances). Figure 15 depicts the general concept with respect to how we convert these use cases into the integration scenarios. Each integration scenario supports its own set of use cases depending on what functionalities their tools support. In general, the site administrator can pick one use case suitable for the objective, which is in practice realized by selecting the associated configuration and plugin. The implementation/integration is conducted in the WP3, and the more detailed descriptions are provided in the deliverable D3.3. The coverage of use cases by the WP3 integrations and the WP2 sophistications are summarized in TABLE 9 (see Section 4.3.5). In this section, we introduce what tools are involved in and which use cases are relevant for each integration.

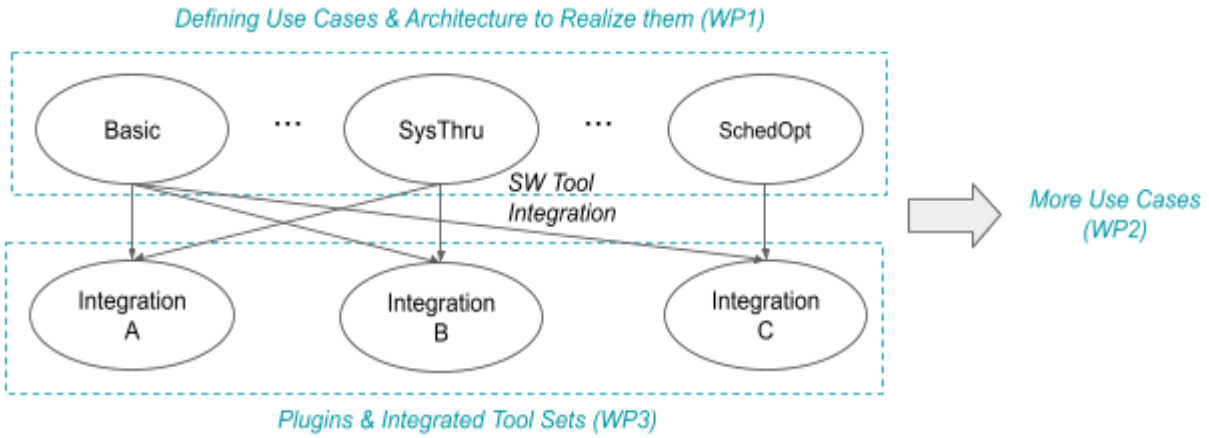

Figure 15: Mapping of Use Cases into Integrations

Integration Scenario #1: Application-aware system power capping

This integration scenario aims at maximizing total system throughput under a power cap. The system throughput is maximized thanks to the feedback that is manually provided by end-users to the RJMS about the performance behavior of the job or the one that is automatically computed by the Execution Profile Compute Module (EPCM), which works as a signature handler. The feedback is then conveyed to the SPM which generates per-job power limits for each node. In this scenario, the SPM plays the key role in terms of global power management at the system scale, and in terms of power capping strategies at the job scale.

The relevant use cases and the involved software tools are as follows. As described above, this scenario is relevant to SysThru, in particular profile-driven proactive power management. This also covers a more naive Basic use case as it simply distributes the power cap evenly across nodes.

Relevant use cases	Basic, SysThru
Involved software tools	OAR(RJMS), BEO(SPM), ECPM(Sig Handler), EXAMON/DCDB(Monitor)

To realize this integration scenario, we integrate OAR, BEO, EXAMON/DCDB, and ECPM. The ECPM is a newly developed external module to enable the signature computation functionality.

Integration Scenario #2: Application-aware energy optimization under a system power cap

This integration scenario is complementary to the previous one. As in the previous scenario, the Job scheduler and the Node manager enforce the power cap. In addition, the Job manager and the Node manager optimize the power state of the compute resources based on application demand. The Monitor provides insights to the system administrator on platform metrics and to the users on job's efficiency through dashboards.

The relevant use cases and the involved software tools are as follows. This scenario covers application-level power optimizations (AppThru, AppEtS, SysThu&AppEtS), while following the power constraint set by the system and node manager.

Relevant use cases	AppThru, AppEtS, SysThu&AppEtS
Involved software tools	BEO(SPM), EAR/BEO(Node Manager), EXAMON(Monitor), COUNTDOWN(Job Manager)

To realize this scenario, BEO, EAR, EXAMON, and COUNTDOWN are integrated. In particular, the COUNTDOWN plays an important role in this scenario by setting the compute resources into a low power mode while waiting for MPI communications.

Integration Scenario #3: Application-aware power capping with job scheduler support

This integration scenario covers a use case where the RJMS plays an active role in the power management. In this scenario, the SPM applies the cluster powercap algorithm to dynamically setting the node powercap based on application characteristics. The RJMS receives information from the SPM about the system status in terms of power consumption. Based on this information, the RJMS can take several actions/decisions: It can (1) influence the scheduling policy, the order of jobs, and job priorities in order to adapt the scheduling to the system status, and (2) it can force the System Power Manager to reduce the allocated power of running nodes in order to guarantee some power for new jobs. The following table lists the relevant use cases and the involved software tools. In addition to SysThru, NodPowShift is also applicable to this integration scenario thanks to EAR's multiple device support.

Relevant use cases	Basic, SysThru, SchedOpt, NodPowShft
Involved software tools	OAR(RJMS), EARGM(SPM), EARD(Node Manager), Examon(Monitor)

Integration Scenario #4: Application-aware power capping and frequency optimization with job scheduler support

The integration is an extension of the integration scenario #3 and attempts to further apply job-level power management. It augments an additional layer in the power management hierarchy and also applies the frequency optimization under a job/node power budget.

Relevant use cases	Basic, SysThru, SchedOpt, NodPowShft, AppThru, AppEtS, SysThu&AppEtS
Involved software tools	OAR(RJMS), EARGM(SPM), EARD(Node Manager), Examon(Monitor), EARL(Job Manager)

Integration Scenario #5:

The final integration scenario bridges between the PowerStack path and the Melissa path. The integration aims to realize power monitoring/control as well as concurrency control for ensemble runs (kill clients if the power budget is overrun). In this integration scenario, Melissa is integrated with EAR and OAR.

Relevant use cases	PowAwrEns
Involved software tools	OAR(RJMS), EARGM(SPM), EARD(Node Manager), Examon(Monitor), Melissa (Workflow Engine)

7. Security, Privacy, and Reliability Perspectives

Although security, privacy, and reliability aspects are not part of the strategic objectives of the project, we introduce how these are managed in the REGALE architecture and software stack due to an increasing demand in the HPC community. We classify our approaches into (1) privilege control for power management, (2) privilege/privacy management for monitoring, (3) anomaly detection mechanism and its extension for security, and (4) trustworthy job execution environments.

7.1 Privilege Control for Power Management

As power budget is a first-class resource on a power-constrained HPC system, the access privilege to power control functions need to be controlled in a secure way. In our project, this is realized at multiple different layers. One is at the REGALE common library level, and the privilege is controlled by registering the publishers and subscribers per API function. This registration is handled by the privileged site administrator, and software can access these functions only if they are a publisher/subscriber as long as the underlying DDS middleware is robust to vulnerability attacks. The other control layer is at the node level, i.e., access privilege control to the hardware power knobs (or MSR) to permit or deny user-level power management. This is handled at the Linux kernel level – we create a specific group to which we permit the hardware power knob access, and users can access the knobs only if they are included in the group as long as the kernel is trustworthy. Further, the node manager should have access to out-of-band power control knobs to force power capping with the highest priority in case of an emergency.

7.2 Privilege/Privacy Management for Monitoring

The collected statistics by the monitoring tools are used as fuels for system analyses and optimizations. They need to be carefully managed as they could include sensitive information that must not be shared in public. They need to be correctly classified into public or private sectors, and the access privilege control and anonymization mechanisms are key features for monitoring tools. In the REGALE tools, the monitoring system can be viewed as composed by four main parts: (i) the data bus, (ii) the database, (iii) the visualization web server, and (iv) the API and query server. The (i), (ii), (iv) have restricted accesses and are not publicly available, (iii) is configured by restricted users, but exports pre-deifined dashboards to general users. To support the research community, an anonymization procedure has been defined with the site administrators to anonymize and/or remove privacy critical collected data when datasets are extracted from the internal monitoring database to be released [13].

7.3 Anomaly Detection Mechanism and Its Extension for Security

REGALE provides an anomaly detection mechanism (e.g., node failure, temperature anomaly, etc.) using an ML-based approach with monitored data. More specifically, We developed GRAAFE: GRaph Anomaly Anticipation Framework for Exascale HPC systems, a framework for continuously predicting compute node failures in the supercomputer. The framework consists of (i) an anomaly prediction model based on graph neural networks (GNNs) that leverage nodes' physical layout in the compute room and (ii) the computationally efficient integration into the Marconi100's ExaMon holistic monitoring system with Kubeflow,

an MLOps Kubernetes framework which enables continuous deployment of AI pipelines. We have developed HazardNet, a framework for predicting thermal hazards. HazardNet utilizes a complete pipeline of deep learning models to capture the intricate spatial and temporal dependency between operational parameters of data centers and thermal hazards. Such a mechanism is extensible also for malware activity detections in HPC centers, which would be explored in future work.

7.4 Trustworthy Job Execution Environments

Program and data protection/isolation are of necessity for a certain class of modern workloads such as AI-based big data analytics using privacy sensitive data. These emerging workloads need to run in a trusted execution environment (or TEE) with a program/data encryption capability, and recent commercial processors offer hardware-level protection mechanisms such as AMD SEV, Intel SGX, and Intel TDX. While these technologies can induce a considerable performance overhead, and thus are not always preferable for traditional scientific workloads, they are required for emerging security/privacy sensitive workloads. As the RYAX path targets also such a class of workloads, it can offer the use of hardware-based TEEs. One option for this is using protected docker images and deploying them in the virtual cluster region.

8. Conclusions and Future Directions

In this final deliverable, we provided the final state of REGALE architecture, its requirements and interfaces, mapping to our prototypes, and security/privacy/reliability management. We first introduced the architecture and how our software tools are mapped to it. We then formulated all the use cases the project targeted, encompassing both power and workflow engines while stating the requirements for each software component. Next, we described the high-level interfaces between components to represent how they interact with each other to realize the mentioned use cases. We then introduced prototypes integrated in WP2/3 and presented the coverages of use cases as well as the SOs/KPIs. We finally introduced the security/privacy/reliability aspects considered in the architecture even though they were not included in our SOs/KPIs we set at the beginning due to an increasing demand for them.

The architecture is considered an European version of the PowerStack standard [1]. Before the start of the project, the PowerStack community defined a strawman architecture. We followed their activities, inherited the standard, pushed it forward, and used it as a milestone for our prototyping and software integrations. We are sharing not only our updates in the architecture but also our lessons learnt throughout the whole procedure with the community in order to drive the standardization activities.

There are a variety of opportunities to apply/extend the REGALE approach toward power-, energy-, and carbon-efficient supercomputing in the post-exascale era. Although our scope was limited to power management on compute nodes, which was because they were known as one of the most power-hungry components in almost all the supercomputers, we can extend the target, with comprehensively covering also other components such as cooling systems, I/O nodes, network infrastructures, etc. Shifting power budgets between compute nodes and them depending on the system/workload status would be a promising option to improve the efficiency one step further. Although we targeted uniform compute clusters/nodes in our prototyping, several upcoming exascale systems (e.g. Jupiter [5]) will be based on modular architectures, consisting of multiple different types of compute clusters. Our approach is extensible to such systems by supporting power budgeting across them while providing a reasonable algorithm/methodology/implementation to shift power in our future work. In this project, we did not explore the optimization of system-level power capping, but we assumed this was set accordingly by the system administrator. However, scaling the system power budget in an automatic way in accordance with the carbon efficiency of the power grid would be a promising solution to minimize the operational carbon footprint of a supercomputer, given that carbon reduction would be pivotal in future supercomputing [9]. Nevertheless, the contributions presented here in the project still apply and form the necessary foundation.

References

[1] “The HPC PowerStack.” *The HPC PowerStack | HPC PowerStack Seminar Website*, https://hpcpowerstack.github.io/. Accessed 24 January 2024.

[2] “TOP500.” *TOP500*, https://www.top500.org/lists/top500/2021/11/. Accessed 24 January 2021.

[3] Zhao, Zhengji, et al. "Power Analysis of NERSC Production Workloads." *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023.

[4] Ali, Ghazanfar, et al. "Redfish-nagios: A scalable out-of-band data center monitoring framework based on redfish telemetry model." In Fifth International Workshop on Systems and Network Telemetry and Analytics, pp. 3-11. 2022.

[5] JUPITER | The Arrival of Exascale in Europe https://www.fz-juelich.de/en/ias/jsc/jupiter

[6] Huazhe Zhang, et al. “Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques.” In *Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 545–559. 2016.

[7] “NVIDIA System Management Interface.” *NVIDIA Developer*, https://developer.nvidia.com/nvidia-system-management-interface. Accessed 17 Feb 2021.

[8] Dey, Somdip, et al. "EdgeCoolingMode: An agent based thermal management mechanism for dvfs enabled heterogeneous mpsocs." In *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*. IEEE, 2019.

[9] Comprés, Isaías, et al. "Probabilistic Job History Conversion and Performance Model Generation for Malleable Scheduling Simulations." In *International Conference on High Performance Computing*, pp. 82-94. 2023.

[10] M. S. Ardebili, et al. "Prediction of Thermal Hazards in a Real Datacenter Room Using Temporal Convolutional Networks," *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1256-1259

[11] Mohak Chadha, et al. "Sustainability in HPC: Vision and Opportunities" SC-W’23, pp.1876-1880, Nov. 2023.

[12] “EPI: European Processor Initiative” EPI, https://www.european-processor-initiative.eu/. Accessed 9 December 2021.

[13] Borghesi, Andrea, et al. "M100 ExaData: a data collection campaign on the CINECA’s Marconi100 Tier-0 supercomputer." Scientific Data 10, no. 1 (2023): 288.