Energy-aware Co-location of Scientific Workflow Tasks

Master's Thesis

Author
Niklas Fomin
464308
niklas.fomin@campus.tu-berlin.de

Advisor
Jonathan Bader

Examiners
Prof. Dr. habil. Odej Kao
Prof. Dr. Volker Markl

Technische Universität Berlin, 2025
Fakultät Elektrotechnik und Informatik
Fachgebiet Distributed and Operating Systems

Energy-aware Co-location of Scientific Workflow Tasks

Master's Thesis

Submitted by:
Niklas Fomin
464308
niklas.fomin@campus.tu-berlin.de

Technische Universität Berlin
Fakultät Elektrotechnik und Informatik
Fachgebiet Distributed and Operating Systems

2025

Hiermit versichere ich, dass ich die vorliegende Arbeit eigenständig ohne Hilfe Dritter und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Alle Stellen die den benutzten Quellen und Hilfsmitteln unverändert oder sinngemäß entnommen sind, habe ich als solche kenntlich gemacht.

Sofern generische KI-Tools verwendet wurden, habe ich Produktnamen, Hersteller, die jeweils verwendete Softwareversion und die jeweiligen Einsatzzwecke (z. B. sprachliche Überprüfung und Verbesserung der Texte, systematische Recherche) benannt. Ich verantworte die Auswahl, die Übernahme und sämtliche Ergebnisse des von mir verwendeten KI-generierten Outputs vollumfänglich selbst.

Die Satzung zur Sicherung guter wissenschaftlicher Praxis an der TU Berlin vom 8. März 2017* habe ich zur Kenntnis genommen.

Ich erkläre weiterhin, dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

(Unterschrift) Niklas Fomin, Berlin, 24. Oktober 2025

*https://www.static.tu.berlin/fileadmin/www/10000060/FSC/Promotion___Habilitation/Dokumente/Grundsaetze_gute_wissenschaftliche_Praxis_2017.pdf

Abstract

FaaS is a cutting-edge new service model that has developed with the current advancement of cloud computing. It allows software developers to deploy their applications quickly and with needed flexibility and scalability, while keeping infrastructure maintenance requirements very low. These benefits are very desirable in edge computing, where ever-changing technologies and requirements need to be implemented rapidly and the fluctuation and heterogeneity of service consumers is a considerable factor. However, edge nodes can often provide only a fraction of the performance of cloud computing infrastructure, which makes running traditional FaaS platforms infeasible. In this thesis, we present a new approach to FaaS that is designed from the ground up with edge computing and IoT requirements in mind. To keep it as lightweight as possible, we use CoAP for communication and Docker to allow for isolation between tenants while re-using containers to achieve the best performance. We also present a proof-of-concept implementation of our design, which we have benchmarked using a custom benchmarking tool, and we compare our results with benchmarks of Lean OpenWhisk, another FaaS platform for the edge. We find that our platform can outperform Lean OpenWhisk in terms of latency and throughput in all tests but that Lean OpenWhisk has higher success rates for a low number of simultaneous clients.

Kurzfassung

FaaS ist ein innovatives neues Servicemodell, das sich mit dem aktuellen Vormarsch des Cloud Computing entwickelt hat. Softwareentwicklende können ihre Anwendungen schneller und mit der erforderlichen Flexibilität und Skalierbarkeit bereitstellen und gleichzeitig den Wartungsaufwand für die Infrastruktur sehr gering halten. Diese Vorteile sind im Edge-Computing sehr wünschenswert, da sich ständig ändernde Technologien und Anforderungen schnell umgesetzt werden müssen und die Fluktuation und Heterogenität der Service-Consumer ein wichtiger Faktor ist. Edge-Nodes können jedoch häufig nur einen Bruchteil der Leistung von Cloud-Computing-Infrastruktur bereitstellen, was die Ausführung herkömmlicher FaaS-Plattformen unmöglich macht. In dieser Arbeit stellen wir einen neuen Ansatz für FaaS eine Platform vor, die von Grund auf unter Berücksichtigung von Edge-Computing- und IoT-Anforderungen entwickelt wurde. Um den Overhead so gering wie möglich zu halten, nutzen wir CoAP als Messaging-Protokoll und Docker, um Applikationen voneinander zu isolieren, während Container wiederverwendet werden, um die bestmögliche Leistung zu erzielen. Wir präsentieren auch eine Proof-of-Concept-Implementierung unseres Designs, die wir mit einem eigenen Benchmarking-Tool getestet haben, und vergleichen unsere Ergebnisse mit Benchmarks von Lean OpenWhisk, einer weiteren FaaS-Plattform für die Edge. Wir stellen fest, dass unsere Plattform Lean OpenWhisk in Bezug auf Latenz und Durchsatz in allen Tests übertreffen kann, Lean OpenWhisk jedoch höhere Erfolgsraten bei einer geringen Anzahl gleichzeitiger Clients aufweist.

Contents

1 Introduction

1.1 Problem Motivation & Description

The carbon footprint of information and communication technologies (ICT) continues to increase, despite the urgent imperative to decarbonize society and remain within planetary boundaries [citation]. A key driver of this trend is the exponential growth in data collection, storage, and processing across scientific disciplines [citation]. Scientific Workflow Management Systems (SWMSs) have become essential tools for managing this complexity, enabling researchers to exploit computing clusters for large-scale data analysis in domains such as remote sensing, astronomy, and bioinformatics [citation]. However, workflows executed through SWMSs are often long-running, resource-intensive, and computationally demanding, which translates into high energy consumption and substantial greenhouse gas emissions [citation]. Techniques to mitigate these impacts include energy-efficient code generation for workflow tasks and energy-aware scheduling strategies [citation]. Nevertheless, practitioners face challenges in assessing which approaches are most applicable, effective, and feasible to implement without excessive effort [Tha+25].

Scientific workflows have become a central paradigm for automating computational workloads on parallel and distributed platforms. With the rapid growth in data volumes and processing requirements over the last two decades, workflow applications have grown in complexity, while computing infrastructures have advanced in processing capacity and workload management capabilities. A critical component of this evolution is energy management, where scheduling and resource provisioning strategies are designed to maximize throughput while reducing or constraining energy consumption [citation]. In recent years, energy management in scientific workflows has gained particular importance, not only because of rising energy costs and sustainability concerns but also due to the pivotal role workflows play in enabling major scientific breakthroughs [Col+21].

High-performance computing (HPC) refers to the pursuit of maximizing computational capabilities through advanced technologies, methodologies, and applications, enabling the solution of complex scientific and societal problems [citation]. HPC systems consist of large numbers of interconnected compute and storage nodes, often numbering in the thousands, and rely on job schedulers to allocate resources, manage queues, and monitor execution. While these infrastructures provide the backbone for highly demanding applications, their intensive power draw—arising not only from computation but also from networking, cooling, and auxiliary equipment—makes them major consumers of electricity and significant contributors to climate change [citation]. The demand for HPC continues to rise across both public and private sectors, fueled by emerging computationally intensive domains such as artificial intelligence, Internet of Things, cryptocurrencies, and 5G networks [citation]. The transition from petascale to exascale performance has amplified sustainability concerns, as operational costs approach parity with capital investment and energy efficiency becomes a limiting factor [citation]. Recent exascale systems exemplify this trend: while achieving unprecedented performance, they consume tens of megawatts of power and highlight the urgency of energy-aware design. To increase efficiency, modern HPC architectures increasingly integrate heterogeneous hardware, combining multicore CPUs with specialized accelerators such as GPUs, enabling more operations per second but also driving higher power densities at the node and rack level. This creates additional challenges for energy management and cooling, compounded by scheduling constraints and the demand for near-continuous system availability. Addressing these issues is essential to

ensure that future HPC developments meet performance goals without exacerbating their environmental footprint [Sil+24].

As scientific workflows grow in scale and complexity, coarse models of resource usage are no longer sufficient for ensuring efficient and sustainable execution. Tasks within data-driven workflows often appear in multiple instances and may vary substantially depending on input data, parameters, or execution environments. This results in highly dynamic patterns of resource demand, energy consumption, and carbon emissions, which fluctuate during runtime rather than remaining constant. At the same time, the carbon intensity of the underlying infrastructure changes over time, reflecting variations in energy availability and network conditions across sites. To address these challenges, there is a need for fine-grained, time-dependent task models that capture detailed resource usage profiles, incorporate infrastructure energy characteristics, and adapt dynamically during execution. Such models would enable more accurate task-to-machine mappings, informed scheduling decisions across multiple sites, and strategies for co-locating complementary tasks to reduce interference, energy waste, and communication overhead. Developing these models requires continuous monitoring of tasks, predictive techniques to handle incomplete prior knowledge, and adaptive mechanisms that respond in real time to evolving infrastructure conditions. Ultimately, this approach provides the foundation for carbon-aware workflow execution, where tasks are scheduled and allocated in ways that minimize energy consumption and associated emissions while maintaining performance and scalability. Task mapping addresses the challenge of assigning tasks to machines in a way that not only satisfies resource requirements but also minimizes energy consumption and carbon emissions. Traditional approaches largely focused on matching resource demands with machine capabilities, but they overlooked the variability in energy efficiency and carbon intensity over time. By leveraging fine-grained task models and infrastructure profiles, mappings can be extended to consider spatio-temporal variations in energy supply and demand. A key component of this is task co-location: strategically placing tasks on the same machine, within the same cluster, or in proximity across sites. When executed with awareness of complementary resource usage patterns, co-location reduces interference, avoids idle resource blocking, and lowers communication overhead, thereby improving both performance and energy efficiency. Together, advanced task mapping and co-location strategies provide a pathway toward reducing the carbon footprint of computational workflows while maintaining scalability and reliability.

Reducing power consumption while maintaining acceptable levels of performance remains a central challenge in containerized High Performance Computing (HPC). Existing approaches to Dynamic Power Management (DPM) typically rely on profile-guided prediction techniques to balance energy efficiency and computational throughput. However, in multi-tenant containerized HPC environments, this balance becomes significantly more complex due to heterogeneous user demands and contention over shared resources. These factors increase the difficulty of accurately predicting and managing power-performance trade-offs. Furthermore, while containerization frameworks such as Docker are widely adopted, they were not originally designed with HPC workloads in mind, leading to limited research and insufficient software-level mechanisms for monitoring and controlling power consumption in such environments [KP23].

Task clustering is a technique aimed at improving workflow efficiency by aggregating fine-grained computational tasks into larger units, commonly referred to as jobs. This aggregation reduces the overhead associated with scheduling numerous small tasks individually, which in turn lowers energy consumption and shortens the overall makespan of the work-

flow. By combining coarse-grained and fine-grained tasks into clustered jobs, resource utilization becomes more balanced, and the performance of workflow execution can be significantly enhanced [Saa+23].

This paper focuses on effective energy and resource costs management for scientific workflows. Our work is driven by the observation that tasks of a given workflow can have substantially different resource requirements, and even the resource requirements of a particular task can vary over time. Mapping each workflow task to a different server can be energy inefficient, as servers with a very low load also consume more than 50% of the peak power [8]. Thus, server consolidation, i.e. allowing workflow tasks to be consolidated onto a smaller number of servers, can be a promising approach for reducing resource and energy costs.

We apply consolidation to tasks of a scientific workflow, with the goal of minimizing the total power consumption and resource costs, without a substantial degradation in performance. Particularly, the consolidation is performed on a single workflow with multiple tasks. Effective consolidation, however, poses several challenges. First, we must carefully decide which workloads can be combined together, as the workload resource usage, performance, and power consumption are not additive. Interference of combined workloads, particularly those hosted in virtualized machines, and the resulting power consumption and application performance need to be carefully understood. Second, due to the time-varying resource requirements, resources should be provisioned at runtime among consolidated workloads.

We have developed pSciMapper, a power-aware consolidation framework to perform consolidation to scientific workflow tasks in virtualized environments. We first study how the resource usage impacts the total power consumption, particularly taking virtualization into account. Then, we investigate the correlation between workloads with different resource usage profiles, and how power and performance are impacted by their interference. Our algorithm derives from the insights gained from these experiments. We first summarize the key temporal features of the CPU, memory, disk, and network activity associated with each workflow task. Based on this profile, consolidation is viewed as a hierarchical clustering problem, with a distance metric capturing the interference between the tasks. We also consider the possibility that the servers may be heterogeneous, and an optimization method is used to map each consolidated set (cluster) onto a server. As an enhancement to our static method, we also perform time varying resource provisioning at runtime [ZZA10].

1.2 Research Question & Core Contributions

The central questions this thesis seeks to address are:

RQ1 How can fine-grained, time-dependent models of workflow tasks be developed to capture fluctuating patterns of computational resource usage and energy consumption during execution?

RQ2 How can the co-location of workflow tasks be modeled so that their interference is minimized and the resulting shared usage of resources leads to lower overall energy consumption and carbon emissions while performance is maintained?

RQ3 How can co-location models and time-dependent task characterizations be integrated into resource management systems and workflow scheduling frameworks to enable adaptive, energy-aware execution at scale?

The resulting core contributions of this thesis are:

- An architectural mapping that defines which layers of workflow execution need to be monitored and systematically assigns suitable data exporters to them
- The implementation of a monitoring client, capable of serving the relevant monitoring layers for scientific workflow execution which collects fine-grained, time-dependent resource usage data for workflow tasks during execution that was used for data collection on running 10 nf-core pipelines.
- An analysis of co-location effects on the core and node-level that are later on used for implementing the proposed co-location approach.
- The design and implementation of a novel co-location approach that utilizes time series data to compute for any given set of tasks clusters where the resource usage patterns are complementary.
- The application of 2 multivariate, statistical learning methods on the time series data of the workflow tasks: Kernel Canonical Correlation Analysis (KCCA) and Random Forest Regressor (RFR).
- The development of an evaluation test-bed in the WRENCH simulation framework that integrates the proposed co-location approach and allows for the simulation of scientific workflow execution with and without co-location.
- The evaluation of the proposed co-location approach using 10 nf-core workflows, demonstrating its effectiveness in reducing energy consumption while maintaining performance.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows: Chapter 2 presents the fundamentals of this work, covering scientific workflow systems, the co-location problem, cluster resource management, machine learning and more. Chapter 3 introduces related work for monitoring scientific workflows, modeling the co-location of tasks and enabling energy-awareness through minimizing resource contention. In addition, the main state-of-the-art approaches for the co-location problem are presented. Furthermore, Chapter 4 depicts the approach to the problem in both a theoretical and practical manner and ultimately defines requirements for the realization of the approach. The corresponding implementation is elaborated in Chapter 5. Moreover, Chapter 6 evaluates the models compared to the state-of-the-art approaches using a simulation. Chapter 7 interprets the key findings and discusses some limitations of this work. Finally, Chapter 8 concludes this thesis and gives an outlook on the impact of this work for the future.

2 Background

This chapter provides the necessary background for the thesis. Section 2.1 introduces the domain of High-Performance Computing (HPC), with a focus on modern computational hardware (2.1.1) and virtualization technologies (2.1.2). Section 2.2 then discusses scientific workflows as a representative HPC application, beginning with a definition of scientific workflows (2.2.1), followed by their management systems (2.2.2), and concluding with workflow tasks as their smallest management unit (2.2.3). Section 2.3 introduces the domain of energy-aware computing, while Section 2.4 addresses workflow monitoring, including monitoring targets (2.4.1), resource monitoring (2.4.2), energy monitoring (2.4.3), and task characterization using monitoring data (2.4.4). Section 2.5 presents the co-location problem, which constitutes the core of this thesis. It motivates the problem through resource contention (2.5.1), discusses its relationship to workflow scheduling and task mapping (2.5.2), and provides a general overview with the guiding boundaries that shape the definition of workflow task co-location used throughout this work. Section 2.6 introduces the role of machine learning in scientific workflow processing, focusing on its application in resource management (2.6.1) and the theoretical background of the models applied in this thesis. This includes Kernel Canonical Correlation Analysis (KCCA) (2.6.2.1), Random Forest Regression (2.6.2.2), Linear Regression (2.6.2.3), and Agglomerative Clustering for task clustering (2.6.2.4). Finally, Section 2.7 outlines the evaluation methodology, with an emphasis on simulation approaches and a description of the WRENCH framework for simulating distributed computing environments (2.7.1).

2.1 High Performance Computing

High-Performance Computing (HPC) encompasses a collection of interrelated disciplines that together aim to maximize computational capability at the limits of current technology, methodology, and application. At its core, HPC relies on specialized electronic digital machines, commonly referred to as supercomputers, to execute a wide variety of computational problems or workloads at the highest possible speed. The process of running such workloads on supercomputers is often called supercomputing and is synonymous with HPC. The fundamental purpose of HPC is to address questions that cannot be adequately solved through theory, empiricism, or conventional commercial computing systems. The scope of problems tackled by supercomputers extends beyond traditional scientific and engineering applications to include challenges in socioeconomics, the natural sciences, large-scale data management, and machine learning. An HPC application refers both to the problem being solved and to the body of code, or ordered instructions, that define its computational solution.

What distinguishes HPC systems from conventional computers is their organization, interconnectivity, and scale. Here, scale refers to the degree of both physical and logical parallelism: the replication of essential physical components such as processors and memory banks, and the partitioning of tasks into units that can be executed simultaneously. While parallelism exists in consumer devices like laptops with multicore processors, HPC systems exploit it on a vastly larger scale, structured across multiple hierarchical levels. Their supporting software is designed to orchestrate and manage operations at this level of complexity, ensuring efficient execution across thousands of interconnected components.

2.1.1 Modern HPC Hardware

High performance computer architecture determines how very fast computers are formed and function. High performance computing (HPC) architecture is not specifically about the lowest-level technologies and circuit design, but is heavily influenced by them and how they can be most effectively employed in supercomputers. HPC architecture is the organization and functionality of its constituent components and the logical instruction set architecture (ISA) it presents to computer programs that run on supercomputers. HPC architecture exploits its enabling technologies to minimize time to solution, maximize throughput of operation, and serve the class of computations associated with large, usually numeric-intensive, applications. In recent years supercomputers have been applied to data-intensive problems as well, popularly referred to as big data or graph analytics. For either class of high-end applications, HPC architecture is created to overcome the principal sources of performance degradation, including starvation, latency, overheads, and delays due to contention. It must facilitate reliability and minimize energy consumption within the scope of performance and data requirements. Cost is also a factor, affecting market size and ultimate value to domain scientists and other user communities. Finally, architecture shares in combination with the many other layers of the total HPC system the need to make application programming by end users as easy as possible.

HPC system performance depends on the speed of its components, with processor clock rate being a key factor. A major challenge arises from mismatches in cycle times across technologies, such as fast processor cores versus much slower DRAM. To bridge this gap, modern architectures use memory hierarchies that combine high-capacity DRAM with faster SRAM caches. Performance is also shaped by communication speed, measured in bandwidth and latency, which vary with technology and distance. Ultimately, HPC architecture seeks to balance computation, memory, and communication speeds while optimizing cost, power, and usability to maximize application performance. Efficiency in HPC refers to how effectively system components are utilized when executing a workload. A common metric is floating-point efficiency, defined as the ratio of sustained floating-point performance to the theoretical peak, both measured in FLOPs. While once meaningful in an era when floating-point operations were costly, this measure has become less representative as data movement and memory access now dominate in terms of time, energy, and die space. Nevertheless, FLOP-based efficiency remains the most widely reported measure.

Power consumption is a critical factor in HPC, as processors, memory, interconnects, and I/O devices all require electricity, and the resulting heat must be removed to avoid failure. Processor sockets alone may consume 80200 W, and cooling adds significant overhead, sometimes exceeding 20% of total power use. Air cooling suffices for smaller systems, but high-density, high-performance systems increasingly rely on liquid cooling to achieve higher packing density and performance. Modern processors further support power management through dynamic voltage and frequency scaling, variable core activation, and thermal monitoring. These mechanisms enable a balance between power consumption and performance, guided by software that can set or adjust configurations at runtime based on workload demands.

The multiprocessor class of parallel computer represents the dominant architecture in contemporary supercomputing. Broadly defined, it consists of multiple independent processors, each with its own instruction control, interconnected through a communication network and coordinated to execute a single workload. Three main configurations of multi-

processor systems exist: symmetric multiprocessors (SMPs), massively parallel processors (MPPs), and commodity clusters. The distinction lies in memory organization. SMPs use a shared memory model accessible by all processors, MPPs assign private memory to each processor, and cluster-based designs combine both approaches by grouping processors into nodes that share memory locally while maintaining separation across nodes. Modern multicore systems often follow this hybrid structure, balancing performance, scalability, and complexity.

A shared-memory multiprocessor consists of multiple processors with direct hardware access to a common main memory. This architecture allows any processor to read or write data produced by another, requiring an interconnection network that ensures cache coherence across all processors. Cache coherence guarantees correctness by keeping local caches consistent, often implemented through protocols such as MESI, where writes are detected and other caches are updated or invalidated accordingly.

Shared-memory multiprocessors are commonly divided into two categories based on memory access times. In symmetric multiprocessors (SMPs), all processors can access any memory block in equal time, known as uniform memory access (UMA). While contention for memory banks can still cause delays, the system provides equal opportunity to all processors. SMPs are widely used in enterprise servers, workstations, and multicore laptops, and often serve as nodes within larger parallel systems.

Nonuniform memory access (NUMA) architectures extend shared-memory designs by allowing all processors to access the full memory space but with different access times depending on locality. NUMA leverages fast local memory channels alongside slower global interconnects, enabling greater scalability than SMPs. However, this places additional responsibility on software developers to optimize data placement in order to achieve high performance. NUMA systems emerged in the late 20th century and remain a key design for scaling shared-memory multiprocessors.

2.1.2 Virtualization in HPC

With the growing demand for High-Performance Computing (HPC), hardware resources have continued to expand in scale and complexity, with increasingly intricate interconnections between system components. A central challenge lies in maximizing both performance and resource utilization. Virtualization technologies offer a means to improve resource utilization in HPC environments, but often at the cost of performance overhead. Multi-user usage scenarios, combined with the stringent performance requirements of HPC workloads, place high demands on virtualization approaches. Furthermore, the highly customized nature of HPC systems complicates the integration of virtualization solutions. This work examines operating system-level and application-level virtualization within HPC, outlines the current limitations, and discusses potential directions for future developments.

Virtualization at the operating system level has been widely explored through technologies such as VMware, Xen, LXC, and KVM, each offering different trade-offs between usability, security, and performance. VMware, one of the earliest and most established solutions, allows multiple operating systems to run simultaneously on a single host, offering strong isolation, security, and ease of configuration. However, the performance overhead introduced by full virtual machines is too high for HPC workloads, which demand near-native computational speed. Xen, developed as an open-source virtual machine monitor, improved performance compared to VMware but required operating system modifications, making

it less developer-friendly and leading to reduced long-term adoption. More recent solutions such as LXC and KVM integrate virtualization more closely with the Linux kernel, reducing overhead compared to traditional VMs. While these operating systemvel containers are lighter than VMware or Xen, they still introduce performance penalties, with KVM in particular showing unacceptable overhead for CPU-intensive HPC applications. Moreover, KVM requires hardware-level virtualization support, limiting its applicability.

In multi-tenant HPC environments, the dynamic sharing of hardware resources introduces complex powerperformance relationships, as concurrently executed tasks exhibit varying levels of power consumption across different resources. Users increasingly expect the flexibility of cloud-like environments, where execution conditions resemble their native setups without significant performance loss. Container-based HPC environments address this demand by isolating groups of processes into separate applications that run densely on available hardware threads. [KP23].

Virtualization enables the provisioning of resources to multiple users on the same physical machine with the goal of maximizing utilization. Depending on the abstraction layer, different virtualization technologies provide distinct benefits. Containers, which virtualize at the operating system level, offer lightweight isolation and have become widely adopted for deploying microservices. They effectively bridge the gap between development and production by supporting continuous integration and deployment pipelines. Containers leverage the host operating system kernel and typically incur far less management and runtime overhead compared to virtual machines. Prior studies have shown that container performance often approaches native execution, particularly for CPU- and memory-intensive workloads, whereas VMs tend to suffer greater degradation for memory, disk, and network-intensive applications. In comparative analyses, Docker containers have demonstrated near-native efficiency for CPU and memory tasks but exhibit performance bottlenecks in certain networking and storage configurations. Research has also highlighted scalability challenges, including increased startup times as the number of containers grows, as well as interference effects when multiple containers share disk- or network-intensive workloads on the same host. These findings underline the fact that performance in containerized environments is strongly dependent on workload characteristics and resource contention patterns. Despite advances such as workload-aware brokering systems aimed at improving energy efficiency and utilization, limited attention has been paid to how workload nature and interference influence consolidation decisions [GL17].

Application-level virtualization operates above the kernel layer, sharing both the kernel and underlying hardware. This approach is most prominently represented by containers, with Docker introducing a transformative model of packaging and deployment. By encapsulating application dependencies into images and leveraging ecosystems such as Docker Hub, Docker enables rapid and portable application deployment. However, in HPC environments, Docker raises serious concerns. Security risks emerge from its daemon-based execution model, which runs with root privileges, and its resource allocation mechanism conflicts with HPC scheduling managers like Slurm, PBS, or SGE. These schedulers typically rely on cgroups to enforce job-level resource limits, but such constraints are ineffective when bypassed by Dockers daemon. Furthermore, Docker lacks robust support for MPI and multi-node collaboration, making it unsuitable for large-scale HPC workloads.

To address these issues, container solutions such as Singularity and Shifter were developed specifically with HPC requirements in mind. Singularity allows users to build and test applications in local environments and then deploy them seamlessly on HPC systems. Unlike

Docker, it does not rely on a persistent daemon, thus resolving security and resource management concerns. Singularity also supports multiple container image formats, including compatibility with Docker images, and integrates efficiently with HPC resource managers. Its design ensures that containers incur minimal overhead, as virtualization occurs only at the application level with a shared kernel, making it particularly well-suited for HPC contexts where performance efficiency is critical.

2.2 Scientific Workflows

2.2.1 Scientific Workflow Management Systems

Scientific Workflow Management Systems (SWMSs) enable the composition of complex workflow applications by connecting individual data processing tasks. These tasks, often treated as black boxes, can represent arbitrary programs whose internal logic is abstracted away from the workflow system. The resulting workflows are typically modeled as directed acyclic graphs (DAGs), where channels define the dependencies between tasks: the output of one task serves as the input for one or more downstream tasks. This abstraction allows users to design scalable, reproducible workflows while managing execution complexity across diverse computational environments [Tha+25].

Scientific Workflow Management Systems (SWMSs) provide a simplified interface for specifying input and output data, enabling domain scientists to integrate and reuse existing scripts and applications without rewriting code or engaging with complex big data APIs. This abstraction lowers the entry barrier for developing and executing large-scale workflows. In parallel, cloud computing has become an increasingly popular execution environment, complementing or replacing traditional HPC clusters. Clouds offer advantages such as elastic scalability, flexible pay-as-you-go pricing models, and access to a wide range of heterogeneous hardware resources, making them attractive for diverse workflow workloads [Bad+22].

2.2.2 Examples of Scientific Workflows

Scientific workflows are compositions of sequential and con- current data processing tasks, whose order is determined by data interdependencies [4]. A task is the basic data processing component of a scientific workflow, consuming data from input files or previous tasks and producing data for follow- up tasks or output files (see Figure 1). A scientific workflow is usually specified in the form of a directed, acyclic graph (DAG), in which individual tasks are represented as nodes. Scientific workflows exist at different levels of abstraction: abstract, concrete, and physical. An abstract workflow mod- els data flow as a concatenation of conceptual processing steps. Assigning actual methods to abstract tasks results in a concrete workflow. If this mapping is performed auto- matically, it is called workflow planning [5]. To execute a concrete workflow, input data and processing tasks have to be assigned to physical compute resources. In the context of scientific workflows, this assignment is called scheduling and results in a physical and executable workflow [6]. Low-level batch scripts are a typical example of physical workflows [BL13].

2.2.3 Scientific Workflow Tasks

Workflow applications are typically executed per input, or per set of related inputs, enabling coarse-grained data parallelism at the application level. Multiple tasks can run concurrently if independent inputs are processed by the same workflow, and parallelism

also arises within a single input when workflow graphs fork, allowing downstream tasks to execute simultaneously. Conversely, many workflows begin with parallel execution of different tasks whose outputs are later joined, synchronizing multiple workflow paths.

Due to their complexity and scale, workflows often consist of large numbers of tasks with multiple parallel paths and are executed on clusters—collections of interconnected compute nodes managed as a single system. Scientific Workflow Management Systems (SWMSs) submit ready-to-run tasks to cluster resource managers such as Slurm or Kubernetes, which allocate resources according to user-defined requirements like CPU cores and memory. Task communication is typically implemented via persistent cluster storage systems (e.g., Ceph, HDFS, or NFS), where intermediate data is written and read between tasks. This approach ensures flexible scheduling, fault tolerance through restartable intermediate states, and simplified execution management.

While SWMSs usually treat tasks as black boxes, opening these tasks for optimization can significantly improve performance. In particular, adapting code to the available hardware, such as optimizing for specific CPUs or accelerators, can reduce runtime. Since certain code segments disproportionately affect overall task execution time, focusing on these hotspots offers an effective strategy for workflow-level performance improvement [Tha+25].

2.3 Monitoring Scientific Workflows

Performance monitoring is a critical stage in application development, extending beyond functional correctness and validation of results. Even after thorough testing with diverse datasets and computational modes, hidden inefficiencies may remain that limit the applications ability to fully exploit the underlying hardware. This is especially significant in parallel computing, where inefficiencies are amplified across many processor cores, leading not only to longer runtimes but also to higher computational costs, as users are often charged proportionally to aggregate machine time. Monitoring helps ensure that performance is not degraded by preventable factors, for example by checking whether computation times match processor capabilities or whether communication delays align with message sizes and network bandwidth. Instrumentation of code segments can provide such insights, though even basic measurements introduce latency and overhead that may distort results or even alter execution flow. To reduce this, statistical sampling is often employed, recording snapshots of program state at intervals rather than logging every event. Sampling periods can be tuned to balance accuracy against intrusiveness. Hardware-based approaches offer another alternative: modern CPUs provide dedicated performance counters for events such as instruction retirement, cache misses, or branches. These counters operate transparently, incurring virtually no overhead, though they are limited to a predefined set of measurable events. Together, these techniques form the basis of effective performance monitoring, allowing developers to identify and address bottlenecks without unduly interfering with execution.

2.3.1 Monitoring Layers

Highl-level Monitoring While low-level tools can provide detailed insights into system resource usage, linking these measurements to higher-level abstractions such as workflow tasks remains challenging. In large-scale scientific workflows executed on distributed environments like clusters or clouds, tasks may be scheduled on arbitrary nodes and may even share resources with other tasks. As a result, locally observed traces of CPU, memory, or I/O usage cannot be directly attributed to specific tasks. To bridge this gap, resource

usage profiles from compute nodes must be correlated with metadata such as workflow logs or job orchestrator information (e.g., container management data). Establishing this information chain enables the classification of observed behavior at the task level, highlighting inefficient or resource-intensive components of the workflow that offer the greatest potential for optimization.

Existing monitoring frameworks focus primarily on processes or systems and generally lack the ability to map resource usage back to workflow tasks, especially in multi-tenant or distributed environments. This limits the ability to isolate the contribution of individual tasks to overall resource consumption. Moreover, metrics natively provided by workflow management systems are often coarse-grained, capturing only summary statistics for task lifetimes. To obtain fine-grained insights into task-level behavior, additional instrumentation and mapping strategies are required to connect low-level monitoring data with workflow abstractions [Wit+24].

The proposed architectural blueprint for workflow monitoring is structured into four layers: the resource manager, the workflow, the machine, and the task layer. These layers represent logical distinctions within a scientific workflow execution environment, each focusing on a different monitoring subject and retrieving metrics from lower layers as needed. Unlike user-centric designs, this approach emphasizes system components rather than interaction flows. Higher layers provide increasingly abstract views, relying only on selected metrics from underlying layers while, in theory, being able to access all. For instance, at the resource manager level, systems such as Slurm, Kubernetes, or HTCondor only require aggregate metrics like task resource consumption or available machine resources, without depending on fine-grained traces such as syscalls. The workflow layer captures metrics tied to the workflow specification, while the machine layer delivers detailed reports of node-level resource usage. At the lowest level, the task layer focuses on fine-grained task execution metrics. Together, this hierarchy balances abstraction and granularity, ensuring that relevant monitoring information is exposed at the appropriate level to support both workflow execution and performance optimization [Bad+22].

The monitoring architecture is structured into four interdependent layers that capture different levels of abstraction within a workflow execution environment. At the top, the resource manager layer supervises the cluster, orchestrates task assignments, and provides coarse-grained monitoring. It contributes aggregated information such as active workflows, running and queued tasks, node health status, and distributed file system states. To enable correct task placement, it draws on workflow-level identifiers and DAG structures, machine-level metrics like available cores or memory, and task-level resource usage and execution status. The workflow layer refines this view by capturing execution semantics, such as dependencies between tasks expressed as DAGs, workflow progress, runtime statistics, makespan, and error reports. The machine layer focuses on per-node monitoring, reporting both general metrics (e.g., CPU and memory utilization) and fine-grained hardware characteristics such as architecture, clock rates, disk partitions, and virtualization context. At the lowest level, the task layer provides the most detailed insights, including logs, execution times, time-series resource usage, and kernel-level traces such as system calls or I/O activity. These fine-grained metrics are essential for diagnosing failures and identifying performance bottlenecks. Together, the four layers form a hierarchical structure where higher layers abstract and summarize information from lower ones, enabling a comprehensive and scalable approach to workflow monitoring [Bad+22].

Scientific workflow management systems (SWMSs) provide varying degrees of built-in monitoring support, which is often complemented by external tools. Pegasus integrates tightly with HTCondor as its resource manager and submits monitoring data directly into a relational database, enabling real-time querying and visualization with external tools such as ElasticSearch or Grafana. Nextflow, originally developed for bioinformatics workflows, has since expanded into other domains and produces monitoring reports once workflow executions are complete. Airflow, with its strong integration into Python, exports selected metrics to StatsD, from where they can be forwarded to systems such as Prometheus for monitoring. Snakemake, also Python-based, supports report generation after execution and offers live monitoring through its Panoptes service, which streams data to an external server accessible via API. Argo, designed natively for Kubernetes, provides a web-based interface where users can access reports and logs of previous executions alongside live monitoring features. Despite these capabilities, most resource managers do not natively handle workflow-level submissions. Consequently, SWMSs typically manage task dependencies and submit tasks sequentially as their requirements are satisfied, with notable exceptions such as HTCondors DAGMan meta-scheduler and Slurms built-in dependency mechanisms [Bad+22].

Low-level Resource Monitoring Monitoring scientific workflows requires tools that can capture and relate information across different components of a computing system, since no single perspective provides sufficient detail for effective optimization. Low-level system monitors can reveal CPU, memory, or I/O usage but lack awareness of which workflow tasks generate that load, while workflow management systems expose task-level metrics that are often too coarse to identify inefficiencies. Similarly, resource managers like Slurm or Kubernetes aggregate machine usage but obscure fine-grained behavior. To close these gaps, specialized tools must be employed at each system layer—resource manager, workflow, machine, and task—and their outputs correlated. This layered approach ensures that high-level abstractions such as workflows can be linked to detailed system traces, allowing bottlenecks to be identified, failures diagnosed, and task placements optimized with respect to both performance and energy consumption. Without combining tools across layers, monitoring remains fragmented, leaving critical inefficiencies hidden in complex, distributed execution environments. In the following, we briefly discuss tools and approaches for monitoring each of these layers in detail while section will go into much more detail.

Building on the need for layered monitoring, it is essential to introduce some fundamental terms and techniques that describe how monitoring data is collected and analyzed across system components. Central among these is tracing, which captures fine-grained event-based records such as system calls, I/O operations, or network packets. Tracing provides detailed raw data, often with high volume, that can either be post-processed into summaries or analyzed on the fly using programmatic tracers such as those enabled by the Berkeley Packet Filter (BPF). BPF allows small programs to run directly in the kernel, making it possible to process events in real time and reduce the overhead of storing and analyzing massive trace logs. In contrast, sampling tools collect subsets of data at regular intervals to create coarse-grained performance profiles. While sampling introduces less overhead than tracing, it provides only partial insights and can miss important events. Together with fixed hardware or software counters, tracing and sampling form the backbone of observability—the practice of understanding system behavior through passive observation rather than active benchmarking. Observability thus provides the conceptual umbrella under which monitoring tools operate, ranging from low-level system probes

to workflow-aware abstractions. The BPF ecosystem provides several user-friendly front ends for tracing, most notably BCC (BPF Compiler Collection) and bpftrace. BCC was the first higher-level framework, offering C-based kernel programming support alongside Python, Lua, and C++ interfaces, and it introduced the libbcc and libbpf libraries that remain central to BPF instrumentation. It also provides a large collection of ready-to-use tools for performance analysis and troubleshooting. bpftrace, in contrast, offers a concise, domain-specific language that makes it well suited for one-liners and short custom scripts, while BCC is better for complex programs and daemons. A lighter-weight option, ply, is also being developed for embedded Linux environments. These frameworks, maintained under the Linux Foundations IO Visor project, collectively form what is often referred to as the BPF tracing ecosystem.These distinctions are crucial, as they shape how different tools are applied at the task, machine, workflow, and resource manager layers, and they form the basis for the more concrete monitoring techniques discussed in the following sections.

Building on this, monitoring CPU usage fundamentally relies on understanding how time is distributed across these execution modes and how the scheduler allocates processor resources among competing tasks. Metrics such as user time, system time, and idle time provide the basis for identifying imbalances, while additional indicators like context switches, interrupt handling, and run queue lengths reveal how efficiently the scheduler manages concurrency. Since background kernel activities and hardware interrupts can consume significant CPU cycles outside of explicit user processes, distinguishing their contribution is essential for accurate analysis. Together, these metrics allow researchers and practitioners to detect inefficiencies such as excessive kernel overhead, overloaded cores, or unfair task distribution—insights that are critical when analyzing the performance of scientific workflows running on shared or large-scale computing infrastructures. A practical strategy for CPU performance analysis begins with verifying that a workload is running and truly CPU-bound, which can be confirmed through utilization metrics and run queue latency. Once established, usage should be quantified across processes, modes, and CPUs to identify hotspots such as excessive system time or uneven load distribution. Additional steps include measuring time spent in interrupts, exploring hardware performance counters like instructions per cycle (IPC), and using specialized BPF tools for deeper insights into stalls, cache behavior, or kernel overhead. This structured approach helps progressively narrow down the root causes of performance issues. In close relation to CPU monitoring, effective memory performance analysis requires tracking how memory behavior influences compute efficiency. Since CPUs frequently stall waiting for data, metrics such as page faults, cache misses, and swap activity can directly translate into wasted cycles. A systematic strategy begins with checking whether the out-of-memory (OOM) killer has been invoked, as this signals critical memory pressure. From there, swap usage and I/O activity should be examined, since heavy swapping almost always leads to severe slowdowns. System-wide free memory and cache usage provide a high-level view of available resources, while per-process metrics help identify applications with excessive resident set sizes (RSS). Monitoring page fault rates and correlating stack traces can reveal which tasks or files drive memory pressure, while tracing allocation calls such as brk() and mmap() offers a complementary perspective on memory growth. At the hardware level, performance counters measuring cache misses and memory accesses give insights into where CPUs are stalling on memory I/O. Together, these monitoring steps build a detailed picture of how memory usage interacts with CPU performance, helping to uncover bottlenecks and inefficiencies that limit overall throughput.

Following CPU and memory, file system monitoring should focus on workload behavior at the logical I/O layer and its interaction with caches and the underlying devices. Key signals include operation mix and rates (reads, writes, opens/closes, metadata ops such as rename/unlink), latency distributions for I/O (including tails), and the balance of synchronous versus asynchronous writes. Cache effectiveness is central: track page-cache hit ratios over time, read-ahead usefulness (sequential vs random access), volumes of dirty pages and write-back activity, and directory/inode cache hit/miss rates. Attribute I/O to files and processes to spot hot files and short-lived file churn, and examine I/O size distributions to detect pathologically small requests. Relate logical I/O to physical I/O to assess whether caching is working or the workload is spilling to disk; include error rates and filesystem-specific events (e.g., journaling) for completeness. Finally, watch capacity and fragmentation risk (very high fill levels), mount options that affect performance semantics (e.g., atime, sync modes), and any lock contention visible in the file system path. A compact table or time-series dashboard for these metrics is helpful for diagnosis and comparison across workloads [cite].

In storage subsystems, background monitoring should characterize I/O where it matters for delivered performance and capacity planning rather than enumerate specific tools. At minimum, track request latency as a distribution (including tails) and decompose it into time queued in the operating system versus service time at the device; sustained high queue time indicates saturation regardless of nominal utilization. Measure throughput and IOPS per device together with queue depth to relate load to response, and record I/O size histograms and access locality (sequential vs. random) to explain latency modes. Distinguish operation classes—reads vs. writes, synchronous vs. asynchronous, metadata, readahead, flush/discard—since policies and devices handle them differently and they can interfere (e.g., reads stalling behind large write bursts). Attribute I/O to processes/containers and, where applicable, to workflow tasks so that noisy neighbors and hot spots can be isolated. Track error and timeout rates at the device interface to separate performance issues from reliability faults. Contextual signals such as filesystem cache hit ratio (logical vs. physical I/O), write-back pressure, and device fill level complement block-layer metrics and help explain shifts in latency or throughput. Segment all measurements per device and scheduling policy, and analyze them over time to identify persistent contention, bimodal behavior, and regressions that warrant tuning, task remapping, or rescheduling.

Containers are a lightweight virtualization technology that isolate applications while sharing the same host operating system kernel. Their implementation in Linux builds on two core mechanisms: namespaces, which provide isolation by restricting the view of system resources such as processes, file systems, and networks; and control groups (cgroups), which regulate resource usage, including CPU, memory, and I/O. Container runtimes such as Docker or orchestration frameworks like Kubernetes configure and combine these mechanisms to provide isolated execution environments. Two versions of cgroups exist in the kernel. Version 1 is still widely used in production systems, while version 2 addresses several shortcomings, including inconsistent hierarchies and limited composability, and is expected to become the default for containerized workloads in the near future.

From a performance monitoring perspective, containers introduce challenges that extend beyond those encountered in traditional multi-application systems. First, cgroups may impose software limits on CPU, memory, or disk usage that can constrain workloads before physical hardware limits are reached. Detecting such throttling requires monitoring metrics that are not visible through standard process- or system-wide tools. Second, containers can suffer from resource contention in multi-tenant environments, where noisy

neighborsconsume disproportionate shares of the available resources, leading to unpredictable performance degradation for co-located containers. This is particularly critical in Kubernetes deployments, where hundreds of containers may share a host and rely on fair scheduling enforced by the kernel and the container runtime.

A further complication lies in attribution. The Linux kernel itself does not assign a global container identifier. Instead, containers are represented as a combination of namespaces and cgroups, which complicates mapping low-level kernel events back to the higher-level abstraction of a specific container. While some workarounds exist—such as deriving identifiers from PID or network namespaces, or from cgroup paths—this mapping is non-trivial and runtime-specific. Consequently, many traditional monitoring tools remain container-unaware, often reporting host-level metrics even when executed inside containers. This mismatch can obscure important performance characteristics, such as CPU scheduling delays or memory pressure within a specific container, and may hinder the diagnosis of resource bottlenecks.

To address these issues, container monitoring must operate at multiple levels of abstraction. Metrics must capture both coarse-grained resource usage across cgroups and fine-grained kernel events such as scheduling latencies, memory allocation faults, and block I/O delays. At the same time, the collected data needs to be attributed correctly to the corresponding container environment in order to identify interference, diagnose bottlenecks, and evaluate orchestration policies. These requirements have motivated the use of extended monitoring techniques such as Berkeley Packet Filter (BPF) tracing, which can instrument kernel paths at runtime and associate low-level events with container-related constructs such as namespaces and cgroups.

Hypervisors enable hardware virtualization by abstracting physical resources and presenting them as fully isolated virtual machines, each running its own kernel. Two common configurations exist: bare-metal hypervisors, such as Xen, which schedule guest virtual CPUs directly on physical processors, and host-based hypervisors, such as KVM, which rely on a host kernel for scheduling. Both models often involve additional I/O handling layers, for example QEMU, which introduce latency but can be optimized through techniques such as shared memory transports and paravirtualized device drivers. Over time, virtualization efficiency has improved significantly with processor extensions like Intel VT-x and AMD-V, paravirtualization interfaces for hypercalls, and device-level virtualization (e.g., SR-IOV). Modern platforms, such as AWS Nitro, minimize software overhead by offloading core hypervisor functionality to dedicated hardware components.

Monitoring hypervisors poses challenges similar to containers but with a different focus. Since each VM runs its own kernel, guest-level monitoring tools can operate normally, but they cannot always observe the virtualization overheads introduced by the hypervisor. Key concerns include the frequency and latency of hypercalls in paravirtualized environments, the impact of hypervisor callbacks on application scheduling, and the amount of stolen CPU time—periods when a guests vCPU is preempted by the hypervisor. From the host perspective, additional events such as VM exits provide insight into how often guests trigger traps to the hypervisor, for example on I/O instructions or privileged operations, and how long these exits last. Attribution is further complicated because exit reasons vary widely across workloads and hypervisor implementations.

Effective analysis therefore requires combining guest-side monitoring of virtualized resources with host-side tracing of hypervisor interactions. Guest instrumentation can measure hypercall behavior, detect high callback overheads, or quantify CPU steal time, while

host tracing can expose exit patterns, QEMU-induced I/O delays, and hypervisor scheduling effects. As with containers, BPF-based tracing has become particularly valuable in this domain, since it can capture low-level kernel events with high resolution, linking them to hypervisor operations. With the shift toward hardware-assisted virtualization, fewer hypervisor-specific events remain visible to the guest, making host-level tracing and cross-layer correlation increasingly central to performance monitoring of virtualized environments.

2.3.2 Monitoring Energy Consumption

Energy monitoring of servers combines hardware-aware measurement with system-level attribution to quantify operational emissions and guide optimizations. At its core is a decomposition of power into static (idle) and dynamic (utilization-dependent) components; modern platforms aim for energy-proportional behavior via DVFS and workload consolidation, yet static draw can still be a large fraction of peak power. Component contributions vary—CPUs frequently dominate (tens to 150 W per socket depending on model and load), memory adds a steadier baseline (few watts per 8 GB, modest dynamic range), storage ranges from low-swing HDDs to more energy-proportional SSDs, and network switches exhibit very low dynamic variability—so monitoring must separate baseline from incremental use. Measurements should be translated into environmental impact using facility Power Usage Effectiveness (PUE) and grid carbon intensity (CI), acknowledging that CI varies by region and time. Practically, on-node electrical telemetry is obtained from Intel RAPL via MSRs and OS interfaces (powercap, perf events), with user-space polling or eBPF-assisted collection; higher-level tools (e.g., CodeCarbon, PowerAPI, Scaphandre) build on these to attribute energy to processes. For multi-tenant hosts, thread/container-level attribution benefits from correlating RAPL energy with performance counters across context switches (e.g., BPF-based approaches such as DEEP-mon), enabling per-thread, per-container, and per-host aggregation with negligible overhead. A robust methodology therefore (i) establishes the static baseline and a CPU model (e.g., ACPS/frequency-aware) to avoid double counting, (ii) profiles disks and NICs under controlled I/O sizes/rates while flushing caches and subtracting baseline+CPU, and (iii) scales results through PUE and CI for carbon accounting, yielding repeatable, architecture-agnostic estimates suitable for power-aware scheduling, capping, and capacity planning. Improving the energy efficiency of high-performance computing (HPC) and data center systems requires accurate and fine-grained monitoring of power consumption at the level of individual servers and their components. Traditional approaches based on external wattmeters or hardware sensors are costly, intrusive, and often lack the accuracy and granularity required for detailed analysis, particularly in heterogeneous and highly dynamic workloads. This has motivated the adoption of hardware-assisted mechanisms such as Intel Running Average Power Limit (RAPL), which since the Sandy Bridge architecture has provided direct, low-overhead measurements of energy consumption across CPU cores, the processor package, DRAM, and sometimes integrated GPUs. RAPL enables real-time power monitoring that can be integrated into scheduling, optimization, and power modeling frameworks, offering a practical alternative to coarse external meters or performance-counter-based estimation models. While RAPL has been widely used due to its accuracy, availability, and negligible overhead, its limitations in granularity and the interpretation of certain domains remain open research challenges. In HPC environments, where maximizing performance per watt is critical, RAPL has become a central tool for attributing energy costs to workloads, profiling applications, and guiding energy-aware scheduling, but ongoing work is needed to refine its role in comprehensive system-level energy modeling and management.

Beyond general breakdowns of server energy consumption, the focus in high-performance computing has shifted towards how to obtain accurate and fine-grained measurements without relying on costly or intrusive external hardware. Traditional approaches such as wattmeters or chassis-level sensors remain valuable for coarse measurements, but they lack the granularity to attribute consumption to individual subsystems or workloads. Performance monitoring counters and operating system statistics have therefore been increasingly combined with modeling techniques to approximate component-level power use, though such models often struggle with accuracy under fluctuating workloads. This has motivated the adoption of hardware-integrated interfaces, most notably Intels Running Average Power Limit (RAPL), which provides direct access to energy consumption data for CPU packages, cores, and memory domains. In contrast to external metering, RAPL enables low-overhead, programmatic, and high-resolution measurements, making it a widely used foundation for power modeling and energy-aware scheduling in HPC systems.

The Running Average Power Limit (RAPL) interface, first introduced with Intels Sandy Bridge architecture, provides a hardware-based mechanism to both measure and limit energy consumption across different CPU domains. Its primary purpose is twofold: offering fine-grained, high-frequency energy measurements and enabling power capping to manage thermal output. RAPL exposes several power domains depending on the processor generation, such as the processor package (PKG), which accounts for the entire socket including cores and uncore components, PP0 for cores, PP1 for integrated GPUs, DRAM for attached memory, and PSys in newer architectures like Skylake for system-level monitoring of the SoC. Among these, the PKG domain is universally available, while support for others varies across server and desktop models. Each domain provides cumulative energy consumption values via model-specific registers (MSRs), updated at millisecond resolution and expressed in architecture-specific energy units. These values can be accessed directly through the MSR driver in Linux or via higher-level interfaces such as sysfs, perf, or the PAPI library, offering flexibility in integrating RAPL into monitoring and profiling workflows. With its combination of accuracy, low overhead, and broad accessibility, RAPL has become a central mechanism for energy measurement and modeling in high-performance and data center computing.

With the increasing adoption of containerization in both cloud and HPC environments, fine-grained energy attribution has become a crucial requirement for efficient workload management. While Intels RAPL interface provides accurate measurements of CPU and memory energy consumption at high granularity, its integration into container-level monitoring fills an important gap left by earlier VM- and node-centric approaches. Scientific workflows, which often run as complex pipelines of heterogeneous tasks within containers, particularly benefit from this capability, as it enables precise accounting of energy usage per workflow component. This allows power-aware schedulers and orchestrators to balance performance and energy efficiency, identify hotspots, and optimize resource allocation across distributed systems. Tools such as DEEP-mon build on RAPL by combining kernel-level event tracing with container-level aggregation, offering low-overhead monitoring that can attribute energy costs down to threads and containers. Such capabilities are essential to advance sustainable HPC by enabling detailed profiling of workflow execution and supporting energy-aware scheduling decisions in containerized infrastructures.

2.4 The Co-location Problem

The problem of scientific workflow task co-location can be traced back to classical operating system scheduling, where the core challenge lies in allocating activities to functional units

in both time and space. Traditionally, scheduling in operating systems refers to assigning processes or threads to processors, but this problem appears at multiple levels of granularity, from whole programs to fine-grained instruction streams executed within superscalar architectures. On multiprocessor and multicore systems, the scheduler must not only decide when to run a task but also where, since shared caches, memory bandwidth, and execution units create interdependencies between colocated workloads. Early work in operating systems introduced coscheduling or gang scheduling, where groups of related tasks are executed simultaneously to reduce synchronization delays, exploit data locality, and minimize contention for shared resources. These concepts are directly relevant to modern scientific workflows, where multiple interdependent tasks are often colocated on the same nodes or cores in high-performance computing environments. The performance and energy efficiency of workflows are therefore closely tied to scheduling decisions, as co-located tasks may either benefit from shared resource usage or suffer from interference, making scheduling a fundamental problem for efficient workflow execution. In high-performance computing (HPC), task co-location poses unique challenges due to the complex interplay of shared resources in modern multi-core architectures. Processors share on-chip structures such as last-level caches, memory controllers, and interconnects, as well as off-chip memory bandwidth, creating significant opportunities for contention when multiple applications run concurrently. This contention can result in severe performance degradation, making it critical to understand and predict the impact of co-location on application efficiency. A naïve approach of exhaustively profiling all potential co-locations is infeasible in practice, given the enormous number of possible workload combinations. Instead, research has focused on predictive methodologies that use application-level indicators, such as cache usage or memory access patterns, to estimate interference effects. This has led to classification schemes that characterize applications both by their sensitivity to co-located workloads and by their capacity to disrupt others. Building on the challenges of co-location in multi-core systems, the problem becomes even more pronounced when considering scientific workflows, which consist of heterogeneous tasks with highly variable and dynamic resource requirements. Assigning each task to a separate server leads to poor energy efficiency, as servers continue to draw a substantial fraction of their peak power even under low utilization. Server consolidation thus emerges as a promising strategy, where multiple workflow tasks are mapped onto fewer servers to reduce total power consumption and resource costs. In this context, the terms consolidation and co-location can be used interchangeably, as both refer to the placement of multiple tasks onto the same physical resources with the aim of improving efficiency. However, consolidation is far from trivial: the resource usage of colocated tasks is not additive, and interference effects can significantly impact both power consumption and application performance. Furthermore, the temporal variation in workflow task demands requires runtime-aware provisioning strategies to avoid resource contention and performance degradation. These complexities underline the need for methodologies that can accurately predict and manage co-location trade-offs, paving the way for addressing the specific challenges of scientific workflow task consolidation in HPC and cloud environments [ZZA10]

2.4.1 Impacts of Co-locaiton on Computational Resource Contention and Interference

The fundamental motivation for addressing co-location in HPC and cloud environments lies in the problem of resource contention. When processes execute on different cores of the same server, they inevitably share hardware resources such as caches, buses, memory, and storage. This sharing often leads to interference, slowing down execution compared

to scenarios where tasks have exclusive access to these resources. In extreme cases, memory traffic contention has been shown to cause super-linear slowdowns, where execution times more than double, making sequential execution more efficient than poorly chosen co-schedules. For large-scale systems hosting thousands of tasks, this contention complicates job scheduling, as schedulers must avoid placing workloads that compete heavily for the same resources. Without accurate estimates of resource usage or slowdown potential, scheduling decisions risk becoming guesswork, reducing overall efficiency. Importantly, poor scheduling is not limited to high-resource applications: even pairing computationally bound tasks that do not interfere can be suboptimal if it prevents beneficial co-scheduling with memory-bound applications. This highlights that effective co-location must avoid both high-contention pairings and missed opportunities for complementary workload placement. Addressing these challenges requires systematic strategies that recognize and mitigate interference, providing the key motivation for studying co-location in the context of scientific workflow tasks [BL16].

In HPC environments, where users submit batch jobs to multi-core compute nodes, efficient resource utilization is critical for balancing throughput, makespan, and job duration. However, parallel applications often fail to fully exploit all allocated cores due to bottlenecks in shared resources such as memory or I/O bandwidth, leading to inefficiencies and longer runtimes. Co-allocating multiple applications on the same nodes has been explored as a strategy to reduce makespan and improve overall system throughput, yet it remains uncommon in production systems because contention for shared resources like last-level caches or memory controllers can increase individual job durations. One approach to mitigate these drawbacks is process mapping, where processes of parallel applications are carefully assigned to specific cores to minimize communication costs and interference. With the growing complexity of HPC architectures, featuring deep memory hierarchies and high core densities, process mapping has become increasingly important. Still, existing solutions often focus solely on single-application performance and rely on costly profiling runs, limiting their applicability in co-located, real-world production workloads. This creates a clear need for methodologies that can address co-location and mapping challenges jointly, enabling more efficient use of HPC resources without sacrificing fairness or performance [Var+24].

2.4.2 Shared aspects and distinctions from Scientific Workflow Scheduling & Task Mapping

In this section, we describe two widely-used energy-aware workflow scheduling algorithms that leverage the traditional power consumption model for making scheduling decisions described in the previous section. We then evaluate the energy consumption for schedules computed by these two algorithms using both the traditional and the realistic models. We do so by using a simulator that can simulate the power consumption of a workflow execution on a compute platform for either model. We perform these simulations based on real-world execution traces of three I/O-intensive workflow applications. The specific scheduling problem that these algorithms aim to solve is as follows. Scheduling Problem Statement. Consider a workflow that consist of single-threaded tasks. This workflow must be executed on a cloud platform that comprises homogeneous, multi-core compute nodes. Initially, all compute nodes are powered off. A compute node can be powered on at any time. Virtual machine (VM) instances can be created at any time on a node that is powered on. Each VM instance is started for an integral number of hours. After this time expires, the VM is shutdown. A node is automatically powered off if it is not running any

VM instance. The cores on a node are never oversubscribed (i.e., a node runs at most as many VM instances as it has cores). A VM runs a single workflow task at a time, which runs uninterrupted from its start until its completion. The metrics to minimize are the workflow execution time, or makespan, and the total energy consumption of the workflow execution [HS24].

n this section, we select several representative scheduling algorithms from each category per their performance and impact. Based on the scheduling approaches, we introduce four types of scheduling algorithms: Task-based (scheduling task-by-task): In the literature, they are also called list-based scheduling. In these algorithms, tasks are ordered based on some priority ranking and then a scheduling decision is made for each task in that order. Path-based (scheduling path-by-path): In these algorithms, a workflow is partitioned into paths based on some criteria and then a scheduling decision is made for each path. BoT-based (scheduling BoT-by-BoT): In these algorithms, a workflow is partitioned into BoTs (Bag of Tasks) such that each BoT is a set of tasks that have no data dependencies among them, and a scheduling decision is made for each BoT.

Node-level and core-level co-location represent two distinct layers of contention within HPC systems. At the node level, tasks share off-chip resources such as memory bandwidth, network interfaces, and I/O subsystems, where heavy communication or I/O-intensive jobs can interfere with one another and degrade performance. At the core level, co-located tasks contend for on-chip resources like private and shared caches, pipelines, and execution units, which can lead to latency, cache thrashing, or reduced instruction throughput. These co-location challenges differ fundamentally from scheduling and task mapping: scheduling determines when tasks execute and mapping decides on which resource they run, while co-location focuses on how tasks interact when placed together on the same hardware. Even though all three may share the objective of improving throughput and minimizing energy consumption, co-location directly addresses the resource interference between tasks and therefore requires strategies that go beyond scheduling order or resource assignment. The challenge lies in predicting and managing these interference effects so that consolidation for energy savings does not undermine overall performance, which makes co-location a critical but separate consideration within workflow execution strategies. The co-location problem in scientific workflow execution arises directly from the heterogeneous ways tasks can currently be placed on HPC and cloud infrastructures. Tasks may be co-located on the same physical node, executed exclusively on a node, distributed across multiple nodes inside containers, or deployed in virtual machines where either one task runs per VM or multiple VMs are consolidated onto the same host. Each of these deployment choices introduces different levels of contention: cores may compete for shared last-level caches, memory bandwidth, or interconnect capacity, while nodes may compete for I/O channels, network interfaces, or access to distributed storage. Because workflows are executed in these diverse environments, the challenge of co-location becomes embedded into scheduling and task mapping decisions—not as a separate process, but as a cross-cutting concern that determines how efficiently tasks can share resources without incurring performance degradation or excessive energy costs. In the context of scientific workflows, the relation between resource contention and energy efficiency becomes particularly critical, as under-utilized or poorly consolidated resources can lead to significant power waste. Since servers consume a large fraction of their peak power even at low utilization, mapping each workflow task to a separate node often results in inefficiency. Consolidating tasks onto fewer servers can mitigate this by increasing utilization and reducing overall energy consumption. However, consolidation also raises the challenge of interference, as colocated

tasks may contend over CPU, memory, disk, or network resources, potentially degrading performance and offsetting the energy savings. This trade-off highlights why the task co-location problem cannot be ignored: energy-efficient workflow execution requires balancing resource consolidation to reduce idle power draw with careful awareness of contention patterns to avoid excessive slowdowns. In this sense, energy efficiency and performance are inherently tied to how tasks are colocated, making it essential to embed power-awareness into workflow scheduling and mapping strategies [ZZA10] [LZ12].

2.5 Applied Machine Learning Techniques in Scientific Workflow Computing

2.5.1 Intelligent Resource Management

While traditional HPC systems often avoid colocating applications on the same node due to unpredictable interference effects, the potential benefits of improved throughput and energy efficiency make the problem worth revisiting. When colocated tasks are bottlenecked by different resources, resource utilization can be increased without altering application code, opening opportunities for more efficient execution. Machine learning offers a promising avenue to address the complexity of this problem, as it can capture non-trivial relationships between hardware performance monitoring counters and the resulting performance degradation under colocation. Unlike rule-based heuristics, machine learning models can generalize across diverse applications and workloads, enabling predictive insights into slowdown and contention effects. Although training and inference may be computationally demanding, practical optimizations have shown that machine learning can be applied with manageable overhead, making it a viable tool to guide scheduling and task placement decisions for scientific workflows in HPC environments. Resource colocation in multi-core HPC systems inherently introduces contention for shared on- and off-chip resources such as caches, memory controllers, and interconnects, which can significantly affect application performance. Traditional heuristic-based scheduling approaches, for example those relying only on LLC miss rates, have shown limited success as they often fail to capture the complex and non-linear slowdown effects caused by colocated applications. To overcome these limitations, recent research has turned to machine learning techniques that exploit performance monitoring counters (PMCs) to predict degradation effects more accurately. By training models offline on representative workloads and deploying them in scheduling decisions at runtime, such approaches enable degradation-aware colocation strategies that minimize makespan and improve resource utilization. This integration of predictive modeling into workload management represents a significant step towards intelligent, energy-efficient execution of HPC workloads, where scheduling decisions are not only resource-aware but also contention-sensitive, directly addressing the challenges posed by colocating diverse scientific applications [Zac+21].

When executing scientific workflows on large-scale computing infrastructures, researchers are required to define task-level resource limits, such as execution time or memory usage, to ensure that tasks complete successfully under the control of cluster resource managers. However, these estimates are often inaccurate, as resource demands can vary significantly across workflow tasks and between different input datasets, leading either to task failures when limits are underestimated or to inefficient overprovisioning when limits are set too conservatively. Overprovisioning, while preventing failures, reduces cluster parallelism and throughput, as excess resources are reserved but left unused, while incorrect runtime estimates can distort scheduling decisions and degrade overall system efficiency. To address this, recent research has explored workflow task performance prediction as a means to

automate the estimation of runtime, memory, and other resource needs. Machine learning plays a central role in these efforts, with approaches ranging from offline models trained on historical execution data, to online models that adapt dynamically during workflow execution, to profiling-based methods applied before execution. Performance prediction models can integrate into both workflow management systems and resource managers, enabling more informed scheduling, efficient resource utilization, and improved energy- and cost-aware computing. This establishes task-level performance prediction as a crucial foundation for advancing scientific workflow execution towards higher reliability, efficiency, and sustainability

2.5.2 Utilized Machine Learning Algorithms in this Thesis

Linear Regression Linear regression is a fundamental statistical method used to model the relationship between one or more explanatory variables and a continuous response variable. It estimates coefficients by minimizing the residual sum of squares, providing an optimal linear fit between predictors and outcomes. The model assumes linearity, independence of errors, homoscedasticity, and normally distributed residuals. Ordinary Least Squares (OLS) is the most common approach, where the coefficients are computed analytically from the design matrix. However, when predictors are highly correlated, multicollinearity can occur, making the coefficient estimates unstable and sensitive to noise in the data. Despite this limitation, linear regression remains widely used due to its interpretability, computational efficiency, and ability to serve as a baseline for more complex regression techniques.

Kernel Canonical Correlation Analysis Canonical Correlation Analysis (CCA) is a statistical technique designed to identify and maximize correlations between two sets of variables. Unlike Principal Component Analysis, which focuses on variance within a single dataset, CCA aims to find linear projections of two datasets such that their correlation is maximized. The result is a set of canonical variables that capture the strongest relationships between the two domains. Kernel Canonical Correlation Analysis (KCCA) extends this idea by applying kernel methods, allowing the detection of nonlinear relationships. In KCCA, the data are implicitly mapped into high-dimensional feature spaces through kernel functions, and correlations are then maximized in that transformed space. This enables KCCA to capture more complex dependencies than linear CCA, making it particularly powerful in settings where relationships between datasets are not strictly linear [BJ03]. Kernel Canonical Correlation Analysis (KCCA) works by taking kernel matrices of two datasets and solving a generalized eigenvalue problem to identify projections that are maximally correlated. Intuitively, this means that KCCA maps both datasets into high-dimensional feature spaces defined by kernel functions and then finds directions in these spaces where the correlation between the two datasets is strongest. In practice, one dataset can represent resource usage metrics while the other represents performance or power measurements. KCCA then identifies correlated structures—such as clusters of similar usage patterns and their corresponding performance or energy behaviors—revealing how variations in resource usage align with variations in system-level outcomes. This makes KCCA particularly suitable for analyzing complex, nonlinear relationships in workflow execution, where resource usage and energy efficiency are intertwined [ZZA10]. Ensemble methods improve predictive performance by combining the outputs of multiple base estimators, thereby reducing variance and increasing robustness compared to using a single model. Among the most widely used ensemble approaches are gradient-boosted trees and random forests, both of which rely on decision trees as their fundamental building blocks. Ran-

dom forests in particular construct a large number of decision trees, each trained on a bootstrap sample of the data and using random subsets of features at split points. This injection of randomness ensures that the trees are diverse, and their errors are less correlated. Predictions are then aggregated, typically by averaging in regression or majority voting in classification, which cancels out some of the individual errors and leads to more stable and accurate results. Intuitively, while a single decision tree may overfit or be highly sensitive to small changes in the data, combining many such trees smooths out these instabilities. The difference between classification and regression in random forests lies in the aggregation step: for classification, the predicted class is determined by majority vote (or probability averaging across trees), while in regression the final prediction is the mean of all tree outputs, yielding a continuous value. This simple but powerful approach has made random forests one of the most effective and robust methods for both supervised learning tasks.

Random Forest Regression A Random Forest Regressor is an ensemble learning method that builds multiple decision tree regressors on random subsets of the training data and combines their predictions through averaging. This approach reduces variance compared to a single decision tree, improving predictive accuracy and robustness against overfitting. Each tree is constructed using the best possible splits of the features, while the randomness introduced through bootstrap sampling and feature selection ensures diversity among the trees. An additional advantage is the native handling of missing values: during training, the algorithm learns how to direct samples with missing entries at each split, and during prediction, such samples are consistently routed based on the learned strategy. This makes Random Forest a flexible and powerful method for regression tasks with heterogeneous and potentially incomplete data.

Ensemble methods improve predictive performance by combining the outputs of multiple base estimators, thereby reducing variance and increasing robustness compared to using a single model. Among the most widely used ensemble approaches are gradient-boosted trees and random forests, both of which rely on decision trees as their fundamental building blocks. Random forests in particular construct a large number of decision trees, each trained on a bootstrap sample of the data and using random subsets of features at split points. This injection of randomness ensures that the trees are diverse, and their errors are less correlated. Predictions are then aggregated, typically by averaging in regression or majority voting in classification, which cancels out some of the individual errors and leads to more stable and accurate results. Intuitively, while a single decision tree may overfit or be highly sensitive to small changes in the data, combining many such trees smooths out these instabilities. The difference between classification and regression in random forests lies in the aggregation step: for classification, the predicted class is determined by majority vote (or probability averaging across trees), while in regression the final prediction is the mean of all tree outputs, yielding a continuous value. This simple but powerful approach has made random forests one of the most effective and robust methods for both supervised learning tasks [Bre01].

Ensemble methods improve predictive performance by combining the outputs of multiple base estimators, thereby reducing variance and increasing robustness compared to using a single model. Among the most widely used ensemble approaches are gradient-boosted trees and random forests, both of which rely on decision trees as their fundamental building blocks. Random forests in particular construct a large number of decision trees, each trained on a bootstrap sample of the data and using random subsets of features at split

points. This injection of randomness ensures that the trees are diverse, and their errors are less correlated. Predictions are then aggregated, typically by averaging in regression or majority voting in classification, which cancels out some of the individual errors and leads to more stable and accurate results. Intuitively, while a single decision tree may overfit or be highly sensitive to small changes in the data, combining many such trees smooths out these instabilities. The difference between classification and regression in random forests lies in the aggregation step: for classification, the predicted class is determined by majority vote (or probability averaging across trees), while in regression the final prediction is the mean of all tree outputs, yielding a continuous value. This simple but powerful approach has made random forests one of the most effective and robust methods for both supervised learning tasks.

Agglomerative Clustering Hierarchical clustering is a family of algorithms that group data into nested clusters, represented as a tree-like structure called a dendrogram. In this hierarchy, each data point starts as its own cluster, and clusters are successively merged until all points form a single cluster. The commonly used agglomerative approach follows this bottom-up process, with the merging strategy determined by a chosen linkage criterion. Ward linkage minimizes the total variance within clusters, producing compact and homogeneous groups, while complete linkage minimizes the maximum distance between points in different clusters, emphasizing tight cluster boundaries. Average linkage instead considers the mean distance across all points between clusters, and single linkage focuses on the minimum distance, often resulting in elongated or chain-like clusters. Although flexible, agglomerative clustering can be computationally expensive without additional constraints, as it evaluates all possible merges at each step.

2.5.3 Research-centric Simulation of Distributed Computing

Scientific workflows have become essential in modern research across many scientific domains, supported by Workflow Management Systems (WMSs) that automate resource selection, data management, and task scheduling to optimize performance metrics such as latency, throughput, reliability, or energy consumption. Despite significant engineering progress in WMS design, many fundamental challenges remain unresolved, as theoretical approaches often rely on assumptions that fail to hold in production infrastructures. As a result, most improvements in WMS algorithms and architectures are evaluated experimentally on specific platforms, workflows, and implementations, which makes systematic evaluation and fair comparisons difficult. Addressing this limitation, WRENCH was introduced as a simulation framework that provides accurate, scalable, and expressive experimentation capabilities. Built on SimGrid, WRENCH abstracts away the complexity of simulating distributed infrastructures while preserving realistic models of computation, communication, storage, and failures. It provides a Developer API for implementing simulated WMSs and a User API for creating simulators that run workflows on simulated platforms with minimal code. Through these abstractions, WRENCH allows researchers to test scheduling, resource allocation, and fault-tolerance strategies at scale without the prohibitive cost of real-world deployments. Importantly, WRENCH supports a wide range of distributed computing scenarios, including cloud, cluster, and HPC environments, enabling reproducible and comparative studies of WMS design choices. By lowering the barrier to simulating complex workflows and providing visualization and debugging tools, WRENCH facilitates the systematic exploration of workflow scheduling, performance prediction, and energy-aware computing strategies in controlled yet realistic settings [Cas+20].

3 Related Work

The following section introduces related research that has informed and inspired the development of this thesis. It reviews selected works addressing key themes such as workflow monitoring, performance modeling, and scheduling in distributed environments. Much prior research has focused on co-scheduling in high-performance computing (HPC) at the operating system level, on contention management between batch workloads and long-running latency-critical services in cloud data centers, or on multi-objective optimization and metaheuristic approaches for job placement. In contrast, this thesis takes a novel perspective by combining machine learningbased scheduling with online consolidation techniques, explicitly targeting scientific workflows in HPC cluster environments. By embedding this adaptive scheduling logic within a simulation framework that captures resource contention and task-level dynamics, the work extends beyond existing literature and, to the best of our knowledge, represents the first integrated exploration of learning-driven, contention-aware scheduling for workflow execution in simulated HPC systems.

3.1 Monitoring of Scientific Workflows

Bader et al. present a conceptual framework for advanced monitoring in scientific workflow environments, emphasizing the heterogeneity of metrics and abstraction layers involved in large-scale workflow execution. Their work identifies four distinct monitoring layers—spanning from infrastructure-level telemetry to workflow-level behavior—and positions these as an architectural blueprint for integrating and correlating performance observations across distributed systems. The authors analyze several state-of-the-art workflow management systems to assess their monitoring capabilities and to highlight existing fragmentation in metric collection and interpretation. Their approach is primarily descriptive and infrastructural, focusing on how to organize and harmonize monitoring responsibilities across system components. In contrast, the work presented in this thesis goes beyond the architectural structuring of monitoring layers and instead operationalizes monitoring data for performance modeling and scheduling decisions. While Bader et al. focus on multi-layer metric aggregation, the present work uses collected measurements—such as runtime and power consumption under co-location—to infer quantitative models of interference and task affinity. Thus, where their contribution establishes the foundation for integrated observability in workflow systems, this thesis builds upon that idea by applying the monitored data directly to predictive, learning-driven scheduling mechanisms that adapt resource allocation based on empirically derived contention profiles [Bad+22].

Witzke et al. address the persistent challenge of connecting low-level monitoring data with high-level workflow semantics in distributed scientific computing environments. Their work focuses on tracing I/O behavior across heterogeneous compute nodes, where workflow tasks may execute concurrently and share resources. By integrating system-level telemetry with metadata extracted from workflow logs and container orchestration frameworks such as Kubernetes, the authors propose a methodology to attribute observed resource consumption—particularly I/O activity—to specific workflow tasks. This enables a task-level analysis of performance bottlenecks and inefficiencies, offering valuable insights for optimization. In contrast to their work, which centers on establishing a traceability chain between metrics and logical workflow components, the approach presented in this thesis operates at a higher abstraction level by using modeled and predicted resource interactions to inform scheduling decisions dynamically. Rather than correlating observed traces post-execution, it leverages learned contention patterns and predictive models to influ-

ence online task placement, thus turning monitoring insights into proactive scheduling intelligence [Wit+24].

3.2 Characterization of Scientific Workflow Tasks

Characterizing scientific workflow applications has been a central topic in understanding performance variability and optimization opportunities in large-scale computing environments. Early foundational work by Juve et al. systematically profiled diverse scientific workflows from domains such as astronomy, bioinformatics, and seismic modeling, revealing that despite differences in scientific goals, workflows often exhibit recurring structural and computational patterns. Their study demonstrated that a few dominant job types typically account for most of the total runtime and I/O activity, and that inefficiencies often arise from repeated data access or imbalanced task configurations. This characterization work established the importance of profiling-based insights as a prerequisite for improving workflow scheduling and system utilization.

Subsequent research extended this line of inquiry toward predictive and analytical modeling. Bader et al. conducted a broad survey of workflow task runtime prediction methods, categorizing models by their statistical or machine learning approach, training mode (offline, online, or pre-execution), and level of infrastructure awareness. Their synthesis demonstrated that accurate task-level performance prediction requires both application features and system state information, including heterogeneity and GPU support. In the context of this thesis, such predictive modeling is regarded as a necessary step toward understanding not just isolated task behavior but also inter-task interference when workflows are executed under shared resource conditions. A related direction emerged from Zhu et al., who developed power- and performance-aware workflow consolidation models based on kernel canonical correlation analysis. Their work introduced temporal signatures to represent CPU, memory, and I/O behavior as time-series features and correlated them with task runtime and power consumption. While their focus remained on offline analysis and consolidation, this approach inspired the feature representation adopted in this thesis, where resource usage patterns are used to model contention and energy impact during online workflow scheduling. Complementing these modeling efforts, Brondolin et al. introduced DEEP-Mon, a monitoring framework capable of attributing fine-grained power consumption to containerized workloads with negligible system overhead. This kind of container-level observability provides the instrumentation backbone necessary for empirically characterizing workflow behavior, as it enables associating power and performance traces with specific workflow tasks and execution contexts. Collectively, these works advanced the field from static profiling toward analytical and predictive characterization of workflow behavior. However, they generally focused on offline measurement, individual task modeling, or coarse-grained consolidation in data center contexts.

3.3 Task Co-location in Scientific Workflows

Recent research on task co-location has increasingly focused on balancing performance isolation with resource efficiency, particularly in heterogeneous or multi-tenant computing environments. While the motivation—to improve utilization through concurrent execution—is common across contexts, the techniques differ widely depending on whether the target system is a cloud, containerized service infrastructure, or HPC cluster.

Task co-location has become a central strategy for improving resource utilization and energy efficiency in large-scale distributed systems. Early work by Zhu et al. introduced

pSciMapper, a consolidation framework designed for scientific workflows in virtualized environments. Their approach treated consolidation as a hierarchical clustering problem, using kernel canonical correlation analysis to model interference between CPU, memory, disk, and network resource profiles. By correlating resource requirements with task runtime and power consumption, pSciMapper achieved significant power savings with minimal slowdown. However, it operated primarily in an offline mode—optimizing workflow placement decisions before execution—and focused on static virtualized environments rather than dynamic HPC workflows.

More recent research has expanded co-location techniques toward data- and container-level optimization. WOW (Workflow-Aware Data Movement and Task Scheduling) by Lehmann et al. proposed a coupled scheduling mechanism that simultaneously steers data movement and task placement to minimize I/O latency and network congestion during workflow execution. Implemented in Nextflow and deployed on Kubernetes, WOW demonstrated how co-location of data and tasks can drastically improve makespan for dynamic scientific workflows. This approach, however, primarily addresses data movement and distributed file system bottlenecks, rather than the interplay of compute, memory, and I/O contention among co-located workflow tasks.

Complementary to WOW, CoLoc by Renner et al. explored distributed data and container co-location in analytic frameworks such as Spark and Flink. By pre-aligning file placement and container scheduling on Hadoop YARN and HDFS, CoLoc improved data locality and reduced network overhead, resulting in execution time reductions of up to 35%. This work demonstrated the benefits of cross-layer scheduling coordination but focused on recurring, data-intensive jobs rather than the complex, dependency-driven task graphs typical of scientific workflows.

In cloud and service-oriented systems, studies such as OLPart (Chen et al., 2023) and the Interference-Aware Container Orchestration framework (Li et al., 2023) address the challenge of resource partitioning under performance interference. OLPart employs online learning via contextual multi-armed bandits to dynamically partition CPU and memory resources between latency-critical and best-effort jobs, using performance counters to infer sensitivity to contention. Its strength lies in operating without prior workload knowledge, adapting to runtime behavior. Similarly, Li et al. propose a machine learningdriven orchestration approach that introduces scheduling latency as a more accurate interference metric for mixed workloads. Their system predicts host-level interference and adjusts resource allocation in real time to preserve QoS while maximizing utilization. Both works demonstrate how adaptive learning mechanisms can manage contention in containerized or service-oriented environments but are inherently focused on online services rather than scientific workflows with inter-task dependencies.

In the HPC domain, co-location has traditionally been explored through hardware-level modeling and scheduling heuristics. Zacarias et al. present an intelligent resource manager that predicts performance degradation between co-located HPC workloads using performance monitoring counters (PMCs) as model features. The scheduler then selects application mixes that minimize degradation while improving node utilization. This approach achieves measurable improvements over conventional batch scheduling, but it remains focused on pairwise job co-location at the job-manager level and lacks integration with workflow-level task orchestration. Complementary to this, Álvarez et al. introduce nOS-V, a system-wide scheduler enabling fine-grained co-execution of HPC applications at the task level. By dynamically managing shared node resources and bypassing traditional over-

subscription mechanisms, nOS-V demonstrates notable gains in throughput and resource efficiency, though it does not incorporate predictive or learning-based adaptation.

A broad body of research has addressed the challenge of performance interference in multi-tenant and consolidated environments, where multiple tasks, processes, or virtual machines share physical resources. Much of this work stems from efforts to optimize system throughput, energy efficiency, and fairness in cloud and HPC contexts, yet the focus typically lies at the level of system services or job managers rather than workflow-specific task scheduling.

Early work by Dwyer et al. pioneered the use of machine learning for online interference estimation on multi-core processors. Their model predicts performance degradation in real time without instrumenting workloads, helping operators in data centers and HPC clusters make informed consolidation decisions. This approach was notable for applying learning-based methods to shared-resource contention but focused primarily on thread- and process-level interference rather than workflow-level orchestration. Complementary to this, Breitbart et al. analyzed co-scheduling of memory-bound and compute-bound applications in supercomputing environments. Their AutoPin+ tool automatically determines suitable combinations of applications to maximize throughput and energy efficiency, demonstrating up to 28% runtime reduction. However, their strategy remains reactive and application-type specific, without predictive or adaptive capabilities. Further refinement came from the work of Alves and Drummond, who developed a multivariate model for predicting cross-application interference in virtualized HPC environments. By considering simultaneous access patterns to shared caches, DRAM, and network interfaces, they achieved high accuracy in predicting interference effects across workloads. This work, along with their later Interference-aware Virtual Machine Placement Problem (IVMPP) formulation, demonstrated that combining interference modeling with optimization can effectively reduce degradation while minimizing physical machine usage. Nonetheless, these methods were still anchored in offline modeling and VM placement optimization rather than dynamic or online scheduling. In data center research, adaptive resource management has evolved toward online learning and multi-objective optimization. The Orchid framework (Chen et al.) and OLPart system introduced contextual multi-armed bandit algorithms for real-time resource partitioning among co-located jobs. These approaches enable systems to autonomously explore and learn optimal resource splits between latency-critical and best-effort workloads, achieving improved throughput and fairness without prior workload knowledge. Similar efforts, such as ScalCCon by Li et al., addressed scalability challenges in correlation-aware VM consolidation through hierarchical clustering, improving both consolidation speed and performance predictability for large-scale infrastructures. Finally, Sampaio and Barbosa proposed an interference- and power-aware scheduling mechanism incorporating a slowdown estimator that predicts task completion under noisy runtime conditions. Their simulations based on Google Cloud traces demonstrate effective SLA compliance with energy cost reductions, bridging the gap between consolidation efficiency and QoS guarantees.

3.4 Energy Awareness in Scientific Workflow Execution

Related Work: Energy Awareness

Energy measurement at fine granularity has been a recurring challenge across HPC and cloud environments. Raffin and Trystram dissect software-based CPU energy metering with a critical analysis of RAPL mechanisms and an exploration of eBPF as a low-overhead

path for accurate, resilient measurements. Their Rust implementation emphasizes pitfalls in timing and counter handling, providing a blueprint for building correct energy profilers on modern x86 platforms. Complementing measurement, Arjona Aroca et al. conduct a component-level characterization of server energy, showing super-linear CPU power behavior and offering validated models that keep estimation error below 5% for real analytics workloads, while also identifying efficient operating points for NICs and disks. Warade et al. bring these concerns to containers, empirically breaking down the energy footprint of common Dockerized workloads to motivate energy-aware container development and deployment practices.

A parallel line of work focuses on forecasting and control. Algarni et al. evaluate classical time-series models—AR, ARIMA, and ETS—for predicting per-container power, demonstrating that method choice should be tuned to workload/container characteristics, with ETS frequently achieving the lowest MAPE. At the orchestration level, Kuity and Peddoju propose pHPCe, a containerized HPC framework that couples an online LSTM with rolling updates to predict power/performance and a contention-aware power-cap selection mechanism, yielding double-digit power savings at low overhead. In cloud consolidation, Abdessamia et al. apply Binary Gravitational Search for VM placement in heterogeneous data centers, reporting substantial energy savings over first/Best/Worst-fit and particle-swarm baselines. Kumar et al. similarly address green consolidation with task mapping and sleep-state strategies, arguing for policies that explicitly consider both active and idle energy draw to reduce total consumption in multi-tenant clouds.

Energy-aware scheduling for scientific workflows has been studied from both modeling and algorithmic angles. Silva et al. analyze two production I/O-intensive workflows on power-instrumented platforms and show that power is not linearly tied to CPU load; I/O—and even waiting for I/O—materially impacts energy. They introduce an I/O-aware power model that, when integrated into simulation, improves accuracy by orders of magnitude relative to traditional CPU-only models. Building on the importance of accurate models for policy quality, Coleman et al. evaluate popular energy-aware workflow schedulers under this improved model and find that CPU-linearity assumptions can underestimate power by up to 360% on I/O-heavy workloads; a simple I/O-balancing scheduler guided by the accurate model produces more attractive energymakespan tradeoffs. From an optimization perspective, Durillo, Nae, and Prodan extend HEFT into MOHEFT, a Pareto-based multi-objective scheduler that captures empirical energy behavior on heterogeneous systems, achieving up to 34.5% energy reductions with marginal makespan impact. Mohammadzadeh et al. pursue metaheuristics, hybridizing Ant Lion Optimizer with Sineosine and chaos-enhanced exploration in WorkflowSim to jointly minimize makespan, cost, and energy for scientific workflows. Choudhary et al. propose a framework that combines task clustering, partial critical path sub-deadlines, and DVFS on compute nodes to reduce transmission and execution energy across several benchmark workflows. Saadi et al. revisit the interplay between clustering and scheduling choices, showing that vertical clustering paired with MaxMin scheduling tends to save energy by lowering makespan, though sensitivity to workflow structure remains.

Energy awareness often intersects with interference management and QoS. Blagodurov and Fedorova advocate contention-aware HPC scheduling, arguing for cluster schedulers that reason about shared resource bottlenecks (e.g., caches, memory controllers) to avoid pathological slowdowns that also waste energy. Sampaio and Barbosa explicitly tie consolidation, slowdown, and power via a slowdown estimator feeding an interference- and power-aware scheduler; simulations based on cloud traces suggest SLA compliance with 12% cost

reductions. Finally, several lines of work underscore that accurate performance/energy prediction is a precondition for effective energy-aware orchestration: from measurement robustness (Raffin and Trystram) and component characterization (Arjona Aroca et al.), to container-level prediction (Algarni et al., Warade et al.), to model-informed workflow scheduling (Silva et al., Coleman et al., Durillo et al.). Together, these studies advance the state of practice in measuring, modeling, and optimizing energy across the stack—from microarchitectural counters and container runtimes up to workflow-level schedulers and consolidation controllers.

4 Approach

The following section systematically outlines the methodological approach of this thesis. While concrete implementation details and technology-specific decisions are discussed in the subsequent chapter, this section focuses on establishing the theoretical foundation that builds upon the concepts introduced in the background. The proposed research problem is addressed through a threefold decomposition. First, a data collection phase captures detailed execution metrics from scientific workflow runs. Second, this collected data undergoes in-depth analysis to identify relevant performance characteristics and relationships. Finally, the insights derived from this analysis are employed in the simulation and algorithmic modeling phase, where various co-location strategies are evaluated to study their effects on performance and energy efficiency.

The structure of this chapter is organized as follows. After defining the problem statement, a general overview of the methodological approach adopted in this work is presented. Subsequently, a set of assumptions is introduced to clearly delimit the scope, applicability, and limitations of the proposed approach. The chapter then begins with the discussion of online scientific workflow task monitoring, detailing how execution data is collected and structured for further analysis. This is followed by an in-depth explanation of the data analysis phase, focusing on matching task entities from different monitoring sources and leveraging this unified data for statistical exploration. The analysis section begins with embedding methodologies and supervised learning, encompassing data preprocessing and predictive modeling techniques. Thereafter, the focus shifts to unsupervised learning, specifically addressing task clustering as applied in this work. The chapter concludes with a detailed presentation of the theoretical approach to the simulation environment, outlining the main conceptual framework, heuristic design principles for scientific workflow scheduling, the definition of baseline algorithms, and the algorithms devised to implement the novel online co-location strategies developed in this thesis.

4.1 Central Objective

The central objective of this work is inspired by the design proposed in [ZZA10], where task co-location is formulated as a consolidation and clustering problem within a virtualized computing environment. The goal is to consolidate workflow tasks—subject to the structural constraints imposed by the workflows Directed Acyclic Graph (DAG)—onto virtualized machines in such a way that their resource usage profiles complement each other, thereby achieving energy-aware execution. While [ZZA10] approaches this problem statically, determining task clusters and node assignments prior to workflow execution, this thesis extends the problem to a dynamic, online setting. In contrast to the static mapping-based approach, this work integrates co-location directly into the task mapping and scheduling process, arguing that co-location and scheduling are inherently interdependent and should be addressed in a unified way rather than separate optimization problems. Consequently, the formulated problem becomes an online co-location problem, where workflow tasks must be characterized before execution in order to enable contention-aware co-location decisions at runtime. The co-location in this context operates at the virtual container level, specifically focusing on virtual machines hosted on physical servers, while contention effects between virtual machines themselves are considered beyond the scope of this thesis.

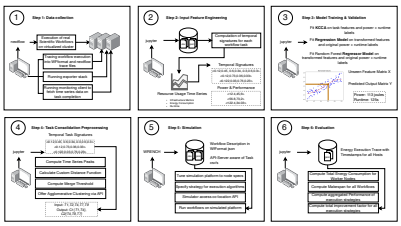

Figure 4.1: Overview of the approach consisting of 6 steps.

4.1.1 Assumptions

To clearly outline the scope, boundaries, and methodological constraints of this work, the following guiding assumptions were defined:

1. **Monitoring Configuration Limits**: A maximum of 80 monitoring features per workflow task is imposed to ensure manageable execution times and allow for statistical evaluation across varying monitoring configurations. This restriction underscores the need for future work to investigate the influence of monitoring data quality and dimensionality on predictive performance.
2. **Monitoring Data Coverage**: Short-lived tasks (typically under one second) are only partially captured or occasionally missed due to system load and sampling intervals exceeding one second.
3. **Offline Data Analysis**: All data preprocessing, model training, hyperparameter tuning, and fitting are performed offline after workflow execution. The resulting trained models are then transformed into a suitable format for integration into the simulation environment. Nevertheless remains the co-locaiton approach online during simulation accessing the data gathered during prior execution.
4. Simulation Environment and Platform Equivalence: The simulated platform is assumed to approximate the physical execution environment. It is expected that the overall behavior observed in simulation aligns with real-world execution trends.
5. Simulation Capabilities and Contention Modeling: The WRENCH framework currently supports simulation of memory contention by limiting per-Host memory consumption, where exceeding the limit results in extended task execution times. Similarly, CPU contention is modeled through proportional increases in task runtime. Other low-level contention effects like cache interference, interconnect bottlenecks, or I/O contention are not modeled in this iteration.
6. Energy Model Assumptions: The energy model provided by SimGrid is assumed to realistically approximate energy consumption variations when tasks with differing resource usage profiles are colocated on the same virtual machine. The impact on energy efficiency is attributed to CPU utilization behavior and derived from the platform description where different load levels map to consumed energy amount.
7. Evaluation Focus: Co-location efficiency is evaluated primarily through per-host energy consumption over time, total workflow energy usage, and overall makespan, which serve as the main indicators of effective virtual machine co-location.

4.2 Online Task Monitoring

The online task monitoring approach builds on a hierarchical monitoring architecture that tries to captures a wide spectrum of metrics relevant to the execution of scientific workflows. The design is inspired by [Bad+22] targeting a four-layered structure consisting of the Resource Manager, Workflow, Machine, and Task layers. Each layer represents a distinct abstraction level in the workflow execution environment and provides valuable insights into different aspects of performance and resource utilization.

Starting with the Resource Manager layer a coarse-grained view related to cluster or partition status, resource allocation, and job management is provided. The Workflow layer operates on the logical workflow representation, maintaining execution progress,

task dependencies, and overall runtime statistics. Below, the Machine layer captures system-level performance data such as CPU, memory, and storage utilization, as well as hardware-specific configurations. At the lowest level, the Task layer delivers fine-grained, time-resolved monitoring data, including per-task resource consumption, low-level kernel metrics, and execution traces. [Bad+22].

The following section provides a brief overview of the data sources and monitoring technologies that were ultimately assigned to the four layers of interest and thus selected for the approach. While the accompanying table presents a broader view of the different tools and data collection options that were tested the text below focuses only on the specific data sources that proved to be most reliable and effective within the implemented monitoring framework.

Monitoring Features	**Data Sources**						
	cAdvisor	**ebpf-energy-exporter**	**docker-activity**	**scaphandre**	**slurm-exporter**	**process-exporter**	**cgroups-exporter**
Resource Manager							
Infrastructure status	x						
File system status	x						
Running workflows	x						
Workflow							
Status	x	x					
Workflow specification		x					
Graphical representation		x					
Workflow ID	x	x					
Execution report		x					
Previous executions		x					
Machine							
Status			x	x			
Machine type			x	x			
Hardware specification			x	x			
Available resources			x	x			
Used resources			x	x			
Task							
Task status				x			
Requested resources				x			
Consumed resources				x			
Resource consumption for code parts				x			
Task ID				x			
Application logs				x			
Task duration				x			
Low-level task metrics				x			
Fault diagnosis				x			

Table 4.1: Technologies and their capabilities evaluated against a 4-layered Monitoring Framework for Scientific Workflows.

Although monitoring plays a central role in enabling the presented methodology, the main objective of this thesis is not to evaluate the performance of monitoring tools or to develop new low-level monitoring frameworks for scientific workflows. Therefore, a detailed technical assessment of these tools is not provided here and will not be revisited in Chapter 6. Instead, a concise justification for the distinction between the tools that were experimentally explored and are shown in 4.1 and those that were finally chosen and used for actual data collection within the evaluation setup.

As shown in the 4.1, the monitoring approach in this work was built around Prometheus as the chosen time-series database. Consequently, all monitoring tools were evaluated based on their interoperability with Prometheus, specifically their ability to function as exporters. The table lists the technologies that were examined for this purpose.

Since the main objective was to gain insight into workflow tasks executed as containers, the focus was on capturing detailed information at the container level and above. However, container-level data can sometimes result in non-identifiable entities, as container names

may represent metadata that cannot be directly linked to specific workflow tasks or workflow instances. Therefore, the SMWS's built-in tracing capabilities remained valuable, as they provide a reliable mapping between workflow logic and system-level execution data.

Before describing the specific tools that were ultimately used for monitoring, it is important to briefly outline why certain exporters were excluded from further use. The Slurm exporter, a custom implementation, was originally intended to retrieve job-to-process mappings from the resource manager. However, equivalent information could also be obtained directly from the workflow engines tracing mechanism, and because Slurm's daemons introduced considerable overhead, the exporter was not used in later stages. The process exporter was also omitted due to difficulties in collecting and aggregating all process IDs associated with a single container. In many cases, only the initial shell process of a workflow task was captured, while its spawned subprocesses remained untracked.

Scaphandre, despite appearing promising at first, showed limited compatibility with AMD hardware, which was used in this work. It also struggled to capture short-lived tasks from the workflow management system, leading to incomplete datasets. Docker Activity, another RAPL-based exporter, performed reasonably well but was still constrained by its dependence on Intel power counters, which are only partially supported on AMD CPUs.

In the end, the most consistent and comprehensive results were achieved using cAdvisor and a custom eBPF-based exporter named ebpf-monitor, a modified fork of DEEP-mon [BSS18]. These two tools demonstrated the best stability and coverage across different task durations and resource types and were therefore selected for the final monitoring setup.

The following table presents the final selection of data sources that were retained for the monitoring setup, along with the specific metrics enabled for each source.

cAdvisor is an open-source daemon for monitoring resource usage and performance of containers. It continuously discovers containers via Linux cgroups under the path /sys/fs/cgroup. Once started, cAdvisor subscribes to create/delete events in the cgroup filesystem, converts them to internal add/remove events, and configures per-container handlers. Metrics originate from machine-level facts parsed from /proc and /sys directories and most prominently container and process usage collected at cgroup boundaries. In practice, cAdvisor provides low-overhead, per-container telemetry.

The ebpf-monitor is based on the DEEP-Mon system, which is per-thread power attribution method to translate coarse-grained hardware power measurements into fine-grained, thread-level energy estimates by exploiting hardware performance counters. The Intel RAPL interface provides power readings per processor package or core, but it cannot distinguish how much of that energy was consumed by each thread. DEEP-Mon bridges this gap by observing how actively each thread uses the processor during each sampling interval. It does so by monitoring the number of unhalted core cycles—a counter that measures how long a core spends executing instructions rather than idling. Since power consumption correlates almost linearly with unhalted core cycles, the fraction of total cycles attributed to each thread provides a reasonable proxy for its share of energy usage. In essence, DEEP-mon first computes the weighted cycles for each thread—combining its active cycles when alone with its proportionally reduced cycles when co-running. These weighted cycles determine how much of the total core-level RAPL energy should be assigned to that thread. The final per-thread power estimate is then derived by distributing the total measured power of each socket proportionally to the weighted cycle counts of

Table 4.2: Preliminary simulation results for a subset of workflows showing the overall improvement for both runtime and energy consumption compared to the average of 2 naive baselines without co-location.

Software Tool	Primary Focus	Used Metrics	Comment
nextflow	Scientific Workflow Engine	trace file summary	Used for WfFormat generation and matching containers to nextflow processes
cAdvisor	Container Performance Monitor	container_memory_working_set_bytes container_memory_usage_bytes container_memory_rss container_fs_reads_bytes_total container_fs_writes_bytes_total container_fs_io_current	11191
ebpf-monitor	Container Energy & Performance Monitor	container_memory_working_set_bytes container_memory_usage_bytes container_memory_rss container_fs_reads_bytes_total container_fs_writes_bytes_total container_fs_io_current container_mem_rss container_mem_pss container_mem_uss container_kb_r container_kb_w container_num_reads container_disk_avg_lat container_num_writes container_cycles container_cpu_usage container_cache_misses container_cache_refs container_weighted_cycles container_power container_instruction_retired	39216.4

all threads running on that socket. This approach allows DEEP-mon to infer realistic thread-level power usage even in systems with simultaneous multithreading and time-shared workloads, without modifying the scheduler or requiring any application-specific instrumentation. The DEEP-mon tool was modified in this work to export container-level metrics directly to Prometheus.

Based on this identification of relevant metrics and the according technologies, we now introduce the monitoring algorithm built on top of the selected exporter stack and the resulting overview of the needed components for implementing a suitable monitoring client that can efficiently behave like specified in 5.6. It outlines how these components were combined to realize the monitoring logic in this work, while the technical implementation details are described in 5.

First, we outline the core components that form the monitoring client and describe the strategy used to retrieve raw data from the monitored system.

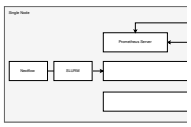

Figure 4.2: The monitoring client alongside the needed system components.

4.3 and 1 show the principle behavior of the monitoring client. The event listener continuously observes container lifecycle events—such as start and termination—emitted by the runtime environment. Once an event is detected, the data query interface dynamically formulates PromQL queries based on the configuration and metadata of the affected container. These queries are sent to Prometheus to retrieve fine-grained time-series data.

The retrieved raw metrics are then passed to the aggregation module, which aligns and consolidates them into a unified record per container.

Algorithm 1: Event-Driven Monitoring and Metric Aggregation Framework

Input: Configuration C defining monitoring targets
Output: Aggregated container-level time-series metrics for each Nextflow process

```
initialize_monitoring(C)
init_event_listener()
while true do
    event ← wait_for_container_event()
    meta ← extract_metadata(event)
    if event.type = START then
        register_process(meta.id, meta.workflow_label)
    if event.type = TERMINATE then
        metrics ← {}
        foreach target ∈ C.targets do
            query ← build_promql(target, meta.id, meta.time_window)
            metrics[target] ← execute_prometheus_query(query)
        data ← aggregate(metrics, meta.workflow_label)
        store(data)
```

The design introduced in 1 reduces monitoring overhead and ensures that only relevant data are collected when workflow tasks actually execute. When a container start event is received, the client extracts metadata such as container ID, associated workflow label, and start timestamp, then registers the process in an internal mapping to maintain correspondence between container identifiers and workflow tasks. Upon receiving a termination event, the client triggers a targeted data collection phase. The monitoring client allows for dynamic configuration of the metrics mentioned in 4.1, enabling data analysis specifically scoped to the user's needs.

4.3 Modelling Co-location of Workflow Task Behavior using Raw Time-Series Data

The next section describes the approach used to analyze the data collected through the monitoring client. The focus lies on how the raw metrics retrieved from Prometheus and other exporters were preprocessed, structured, and aligned to support both supervised and unsupervised learning tasks.

1. Parsing of raw time-series CSV files.
2. Alignment of timestamps across different sources.
3. Merging of per-task data into unified records.
4. Matching Entities between execution layer and workflow layer

The first phase of the data analysis focuses on entity matching of heterogeneous monitoring data sources into a unified representation of each executed workflow task. During workflow execution, diverse monitoring tools and system components produce data at different abstraction levels.

The matching process begins with container lifecycle records, which provide information on task identifiers, container names, process hashes, and working directories. These files serve as the primary link between workflow-level entities and low-level monitoring data. Each monitoring source is traversed recursively to locate and load the associated time-series data. These data files are organized hierarchically under source-specific directories and often contain multiple interleaved measurements from different tasks. To facilitate task-level analysis, these aggregated time-series datasets need to be split into per-task CSV files. Each per-task dataset is enriched with contextual metadata. The first enrichment phase attaches the correct working directory to every container trace, allowing file system references. The next stage introduces workflow-level semantics by matching the per-task monitoring data with the workflows own metadata. Using the exported trace file by the SWMS, the analysis extracts task names and their corresponding working directories. These entries are matched against container-level records from the monitoring data to identify which monitored container corresponds to which logical task in the workflow. The resulting mapping is used to update all per-task monitoring files by appending the correct SWMS process name. This concludes the step of entity matching and preparing the collected data for further statistical analysis.

4.3.1 Preprocessing Raw Time-Series Data for Task-Clustering

1. Peak-pattern construction. For every task and monitored workload type, we first derive a peak time series: the raw per-second resource signal is resampled into three-second buckets and the maximum per bucket is retained. When two peak series must be compared, we truncate both to the shorter length so that correlation is computed on aligned vectors without padding artifacts.

2. Workload-type affinity. Different resource domains interfere to different degrees such as CPU vs CPU peaks are typically more contentious than CPU vs file I/O. We encode this with an empirical affinity score defined in the prior section between workload types. High affinity means higher potential interference when peaks align; low affinity reflects benign coexistence.

4.3.2 Task Clustering for Contention-Aware Consolidation

Using the temporal signature feature representation of the monitored tasks, we establish the basis for energy-aware task co-location. Co-location is treated as a consolidation problem, formulated through a clustering approach. In this context, clustering groups tasks based on a defined distance measure that captures their similarity. To incorporate energy awareness into this objective we extend the distance definition to account for interference between workloads. This is achieved by conducting separate workload experiments designed to measure resource contention and interaction effects. From these experiments, affinity scores are derived to quantify how different workloads influence each other when co-located. These affinity scores are then integrated into the clustering distance function, ensuring that both performance and energy interactions guide the co-location decisions.

Consolidation is formulated as a clustering problem with an important modification: rather than grouping tasks that are similar, the goal is to cluster tasks with complementary resource usage patterns to minimize contention during co-location. Building on the previously established notion of affinity—which quantifies how strongly workloads interfere when sharing resources—the clustering process now incorporates this measure directly into its distance metric. The task task distance increases when two tasks exert pressure on the

same resources simultaneously, indicating potential contention, and decreases when their resource usage peaks complement one another. Algorithm 2 summarizes the main stages of the task consolidation procedure used in this work. The approach begins by computing pairwise similarities between task signatures, where each signature represents a multi-dimensional profile of resource usage over time. The similarity computation incorporates resource affinity weights derived from the contention experiments, ensuring that tasks are compared not only by magnitude but also by their impact on shared resources such as CPU, memory, and I/O. This yields a resource-aware similarity matrix that reflects how well tasks can coexist on the same node without interference. Next, a percentile-based threshold is applied to the similarity matrix. This adaptive rule determines when clustering should stop by selecting only the most suitable task pairs according to the defined percentile. In this way, the algorithm avoids arbitrary distance cutoffs and instead adapts to the underlying distribution of similarity values. Finally, agglomerative clustering is performed using a specified linkage criterion, such as average or complete linkage, to iteratively merge tasks into clusters. Each resulting cluster represents a potential consolidation group, meaning a set of tasks that can be executed together within a single virtual machine due to complementary or non-overlapping resource demands. The output clusters thus provide a structured view of optimal co-location candidates for subsequent scheduling and energy-aware execution strategies.

Algorithm 2: ShaReComp - Task Consolidation Algorithm

Input: task signatures sig, affinity weights w, percentile τ, linkage
Output: clusters $\mathcal{C}$
$S \leftarrow$ `compute_similarity` (sig, w) ; `// resource-aware similarity`
$\theta \leftarrow$ `percentile_threshold` (S, τ) ; `// percentile-based stopping rule`
$\mathcal{C} \leftarrow$ `agglomerative_merge` (S, θ, linkage) ; `// agglomerative clustering`
return $\mathcal{C}$

Next, we introduce the subsequent steps necessary to fullfill 2, focusing on how the task data needs to be refined.

Measuring Resource Interference of Co-located Benchmarks The measurement of resource contention follows a two-stage protocol that first establishes isolated baselines for each workload class and then repeats the same workloads under controlled co-location. In the baseline stage, CPU-bound, memory-bound, and file-I/O-bound containers are executed one at a time on pinned logical CPUs. Pinning fixes placement and removes scheduler noise; for I/O experiments the file set is prepared once and cleaned afterward to avoid warm-cache artifacts. Each run records a start and finish timestamp at microsecond resolution and derives the wall-clock duration. In parallel, the monitoring pipeline supplies per-container power time series. After a run finishes, only the power streams belonging to the participating container are retained, aligned to the containers lifetime, and summarized to a representative mean value. The co-location stage replays the same workloads in pairs to expose interference effects. Pairs are chosen to cover both homogeneous combinations, where both containers stress the same resource class, and heterogeneous combinations, where their dominant pressure differs. Placement again uses CPU pinning. Some experiments bind pairs to siblings on the same physical core to amplify shared-core effects; others place them on distinct cores to isolate memory bandwidth or storage contention. Each co-located container is measured in exactly the same way as in isolation, producing a matched set of durations and power summaries.

Table 4.3: Summary of Synthetic Benchmarks Used in Evaluation

Benchmark Label	Image / Version	
CPU	`stress-ng-cpu:latest`	`stress-ng -cpu 1 -`
Memory (VM)	`stress-ng-mem-vm:latest`	`stress-ng -v`
File I/O	`fio:latest`	`fio -name seqread -rw read -bs 1M -size 18G -`

Isolated and Co-located Metrics.

For each workload $i \in \{1, 2\}$, let

$t_i^{(iso)},\ t_i^{(coloc)}$ denote the isolated and co-located runtimes,

$P_i^{(iso)},\ P_i^{(coloc)}$ denote the average isolated and co-located power consumption.

Per-workload Slowdown Factors.

For each workload $i \in \{1, 2\}$, let

$t_i^{(iso)},\ t_i^{(coloc)}$ denote isolated and co-located runtimes,

$P_i^{(iso)},\ P_i^{(coloc)}$ denote isolated and co-located average power consumptions.

The per-workload slowdowns are defined as

$$S_i^{(t)} = \frac{t_i^{(coloc)}}{t_i^{(iso)}}, \qquad S_i^{(P)} = 1 + \log\left(\max\left(\frac{P_i^{(coloc)}}{P_i^{(iso)}}, 1\right)\right),$$

ensuring that both runtime and power slowdowns are non-negative and at least one in value.

The mean slowdowns across the workload pair are

$$\bar{S}^{(t)} = \frac{S_1^{(t)} + S_2^{(t)}}{2}, \qquad \bar{S}^{(P)} = \frac{S_1^{(P)} + S_2^{(P)}}{2}.$$

A weighted average combines both effects:

$$\bar{S} = \alpha\, \bar{S}^{(t)} + (1 - \alpha)\, \bar{S}^{(P)}, \quad \text{with } \alpha \in [0, 1],$$

where higher α emphasizes runtime effects, and lower α gives more weight to power efficiency.

The final combined slowdown factor is

$$\bar{S}_{\text{final}} = \max(1, \bar{S}),$$

guaranteeing that co-location never yields an apparent speedup (values ≥ 1 indicate slowdown).

Affinity Score.

The affinity score A quantifies the degree of interference between two co-located workloads.

First, compute pairwise affinity ratios:

$$A^{(t)} = \frac{t_1^{(coloc)} + t_2^{(coloc)}}{t_1^{(iso)} + t_2^{(iso)}}, \qquad A^{(P)} = 1 + \log\left(\max\left(\frac{P_1^{(coloc)} + P_2^{(coloc)}}{P_1^{(iso)} + P_2^{(iso)}}, 1\right)\right).$$

A weighted average combines both effects:

$$A_{\text{raw}} = \alpha\, A^{(t)} + (1 - \alpha)\, A^{(P)}, \qquad A_{\text{raw}} \geq 1.$$

The normalized affinity score is then:

$$A = \frac{1 - \frac{1}{A_{\text{raw}}}}{\beta}, \qquad A \in [0, 1],$$

where $\beta > 0$ controls scaling sensitivity. Values of $A \approx 0$ indicate minimal interference, while $A \to 1$ signifies strong co-location interference.

Table 4.4: Affinity scores between workload types indicating co-location compatibility.

Workload 1	**Workload 2**	**Affinity Score**	**Comment**
mem	cpu	0.543	Moderate compatibility; memory and CPU workloads can share resources with limited contention.
mem	fileio	0.148	Very low compatibility; strong interference due to I/O and memory bandwidth pressure.
fileio	cpu	0.223	Low compatibility; CPU workloads cause contention for shared I/O buffers.
cpu	cpu	0.444	Medium self-affinity; CPU-bound tasks compete for cores but remain schedulable.
mem	mem	0.514	Moderate self-affinity; memory contention manageable under shared caching.
fileio	fileio	0.346	Limited self-affinity; file I/O contention degrades throughput under co-location.

From these measurements, contention is characterized by comparing the co-located outcomes against the isolated baselines for the same workloads. For each pair, the procedure derives how much slower the workloads ran together relative to alone and how their average power changed. To avoid overfitting to any single signal, runtime and power effects are aggregated into a single scalar that captures the overall quality of the pairing. Values

above a neutral threshold indicate that the pair plays well together, while values below it signal destructive interference. This single number is what the co-location policies use downstream: it serves both as the supervision signal for learning-based components and as the ground truth to validate scheduling heuristics in simulation.

1. Anti-similarity distance. For any pair of tasks i,j, we iterate over their workload types and compute two ingredients: (i) the affinity between the two types; (ii) the correlation between their corresponding peak series computed twice, once per type, to capture both sides of the pairing. We then aggregate these terms so that highly correlated peaks in high-affinity domains increase the distance, whereas low or negative correlations in low-affinity pairs decrease it. The result is a symmetric task distance matrix whose off-diagonal entries quantify how bad it would be to co-locate the two tasks, and whose diagonals are zero by definition.

Inter-Task Distance and Resource Correlation Model

To quantify the similarity and potential contention between two workloads i and j, we define a composite distance measure that integrates both resource affinity and correlation of peak usage. Each workload utilizes a set of resources $R = \{\text{CPU}, \text{Memory}, \text{Disk}\}$, yielding in total ten pairwise combinations of resource types across two tasks. The distance term combines the precomputed affinity score with the correlation of peak resource intensities.

$$D_{i,j} = \sum_{R_1,R_2} \Big((\text{aff-score}(R_1^i, R_2^j)) \cdot \text{Corr}(\text{peak } R_1^i, \text{peak } R_1^j) \cdot \text{Corr}(\text{peak } R_2^i, \text{peak } R_2^j) \Big) \tag{1}$$

where:

$$R_1^i, R_2^j \text{ denote resource types of workloads } i \text{ and } j, \tag{1}$$

$\text{Corr}(\text{peak } R_1^i, \text{peak } R_1^j)$ is the Pearson correlation between the peak usages of resource R_1, (2)

$$\text{aff-score}(R_1^i, R_2^j) \in [0, 1] \text{ measures the degree of interference between } R_1^i \text{ and } R_2^j. \tag{3}$$

Interpretation

The intuition behind this distance metric is to *promote dissimilar task pairings* for co-location. If two workloads exhibit highly correlated peak usage on the same resources, their corresponding correlation terms will be large, thus increasing $D_{i,j}$ and discouraging their co-location. Conversely, tasks with uncorrelated or complementary resource peaks yield smaller distance values and are therefore more suitable to merge.

The affinity score modulates this behavior: smaller values of aff-score indicate lower interference between resource pairs, which can offset strong peak correlations.

Finally, clustering proceeds by iteratively merging task clusters whose inter-cluster distance satisfies:

$$D_{i,j} < \text{merge\_threshold}.$$

This ensures that only compatible workloads, in terms of both resource affinity and temporal peak correlation, are grouped together.

2. **Threshold selection**. Because the distance matrix is data-dependent, we estimate a merge threshold directly from its empirical distribution. A quantile for e.g., the 20th percentile on the raw distances acts as an automatic cut-level: any pair below this threshold is safe enough to consider for co-location, while pairs above it are kept apart.

3. **Agglomerative clustering with precomputed distances**. We run average-linkage agglomerative clustering on the precomputed distance matrix with the chosen distance threshold and no preset cluster count. This yields variable-sized clusters whose members are mutually non-contentious under our metric. Because we use a threshold rather than a fixed k, the method adapts to each workload mix, producing more or fewer groups as warranted by the observed interference structure.

4. **From clusters to co-location candidates**. Each cluster defines a candidate co-location set. To make these cluster-level entities usable by predictive models such as introduced in **??**, we construct cluster feature vectors by flattening and concatenating the per-task temporal signatures of all members and summing them dimension-wise. This potentially approximates the combined load shape we would see if the clusters tasks were executed in a co-located manner.

4.3.3 Preprocessing Raw Time-Series Data for Predictive Modelling

We transform the heterogeneous, time-stamped monitoring traces into consistent task-level feature/label matrices suitable for statistical analysis. We therefore define the following preprocessing steps:

1. Temporal signature construction (feature selection).

   Sampling and smoothing.

   Equal-length normalization.

   Container-wise collation.

2. Model Input Construction.

   Normalization and Scaling.

   Extraction of Input Features and Output Labels.

Temporal Signature and Model Input Construction.

We denote by $\mathcal{T} = \{T_1, T_2, \ldots, T_N\}$ the set of temporal signatures extracted from the monitored resource-usage profiles of N workflow tasks. Each task i is characterized by time-varying utilization traces for the monitored resource dimensions

$$R = \{\text{CPU}, \text{Memory}, \text{Disk}, \text{Network}\}.$$

For each resource $r \in R$ and task i, the temporal signature $T_i^{(r)}$ is defined as a coarse-grained summary of the normalized resource usage signal $x_i^{(r)}(t)$:

$$T_i^{(r)} = \langle p_{1,i}^{(r)},\ p_{2,i}^{(r)},\ \ldots,\ p_{10,i}^{(r)} \rangle, \tag{1}$$

where each component $p_{k,i}^{(r)}$ represents the mean usage value within segment k of the time-normalized execution window ($k = 1, \ldots, 10$). This yields a ten-dimensional vector describing the temporal pattern of resource consumption.

The feature vector for task i is obtained by concatenating the resource-specific signatures:

$$x_i = \left[T_i^{(\text{CPU})},\ T_i^{(\text{Memory})},\ T_i^{(\text{Disk})},\ T_i^{(\text{Network})} \right] \in \mathbb{R}^{d_x},$$

where $d_x = 4 \times 10 = 40$ in this example configuration.

$$X = [x_1^\top, x_2^\top, \ldots, x_N^\top]^\top \in \mathbb{R}^{N \times d_x}$$

denotes the complete input matrix used for model training.

Similarly, for each task i, the execution-level targets (time and mean power consumption) are given by

$$y_i = [\, t_i,\ P_i \,] \in \mathbb{R}^2, \quad Y = [y_1^\top, y_2^\top, \ldots, y_N^\top]^\top \in \mathbb{R}^{N \times 2}.$$

Example.

Consider $N = 3$ workflow tasks with simplified CPU and memory usage signatures (each consisting of 3 representative pattern points for brevity):

Task	$p_1^{(\text{CPU})}$	$p_2^{(\text{CPU})}$	$p_3^{(\text{CPU})}$	$p_1^{(\text{Mem})}$	$p_2^{(\text{Mem})}$	$p_3^{(\text{Mem})}$
1	0.40	0.75	0.90	0.35	0.55	0.60
2	0.20	0.50	0.70	0.25	0.40	0.50
3	0.30	0.65	0.85	0.30	0.45	0.55

Concatenating these signatures yields the input matrix

$$X = \begin{bmatrix} 0.40 & 0.75 & 0.90 & 0.35 & 0.55 & 0.60 \\ 0.20 & 0.50 & 0.70 & 0.25 & 0.40 & 0.50 \\ 0.30 & 0.65 & 0.85 & 0.30 & 0.45 & 0.55 \end{bmatrix}, \quad Y = \begin{bmatrix} 12.4 & 65 \\ 14.1 & 72 \\ 10.8 & 58 \end{bmatrix}.$$

Here, each row of X encodes the temporal resource-usage pattern of a task, while Y provides the corresponding runtime and mean power consumption used for model learning or correlation analysis.

This preprocessing yields: (i) a standardized and fixed-length feature matrix X that preserves per-metric usage distributions and (ii) a label matrix Y capturing runtime and energy

4.3.4 Modelling and Predicting the Impact of Task Behavior on Runtime and Energy Consumption with KCCA and Random Forest Regression

Algorithm 3: ShaReComp — Prediction of Energy and Performance Behavior of Consolidated Task Clusters

Input: task clusters $\mathcal{C}$, per-task signatures sig, trained model $\mathcal{M}$ (KCCA or Random Forest)
Output: predicted runtime energy pairs $\hat{Y} = \{(\hat{t}_k, \hat{E}_k)\}$

foreach *cluster* $C_k \in \mathcal{C}$ **do**
  $F_k \leftarrow$ `sum_cluster_features` $(\{\, \mathrm{sig}[t_i] \mid t_i \in C_k \,\})$; `// sum task signatures to form cluster feature`

$X \leftarrow$ `build_feature_matrix` $(\{F_k\})$; `// construct consolidated feature matrix`

foreach *cluster feature* $X_k \in X$ **do**
  $(\hat{t}_k, \hat{E}_k) \leftarrow$ `predict_runtime_and_energy` $(X_k, \mathcal{M})$

return $\hat{Y} = \{(\hat{t}_k, \hat{E}_k)\}_{k=1}^{|\mathcal{C}|}$

Kernel Canonical Correlation Analysis In the next step of the analysis, the collected and preprocessed task data are prepared for statistical learning. The objective is to identify relationships between task-specific features and their corresponding performance and energy characteristics. To achieve this, the dataset is divided into two parts: approximately 70% of the tasks are used for training the models, while the remaining 30% are reserved for testing and validation.

Before training, both the feature data (X) and the target data (Y)—representing task runtime and energy consumption—are standardized separately. Each is transformed to have a mean of zero and a standard deviation of one. This step is essential, as the original values cover different numerical scales, which could negatively affect kernel-based learning methods. Standardization ensures that all features contribute equally to the learning process and prevents those with larger magnitudes from biasing the results. After normalization, a Kernel Canonical Correlation Analysis (KCCA) model is trained to identify shared patterns between the feature data and the target data. KCCA projects both datasets into a common latent space, where it can detect nonlinear relationships between resource usage patterns and their corresponding runtime and energy behavior.

Once trained, the KCCA model is extended into a predictive framework. The latent features learned from the input data are used to fit a regression model that links these representations to runtime and energy targets. This allows the system not only to capture statistical relationships but also to predict performance and energy consumption for unseen tasks.

Kernel Canonical Correlation Analysis (KCCA).

To capture nonlinear dependencies between the resource signatures and performance power outcomes, we apply Gaussian kernels to both input and output feature spaces.

KCCA seeks directions A and B in the reproducing kernel spaces of K_x and K_y that maximize the correlation between $K_x A$ and $K_y B$. This is expressed as the generalized

eigenvalue problem:

$$\begin{bmatrix} 0 & K_y K_x \\ K_x K_y & 0 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = \lambda \begin{bmatrix} K_x^2 & 0 \\ 0 & K_y^2 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}. \tag{2}$$

Solving (2) yields paired canonical directions (A, B) that define latent projections

$$X' = K_x A, \qquad Y' = K_y B,$$

maximally correlated across the two feature spaces. These projections represent a shared latent space relating resource utilization dynamics to task performance and energy.

Illustrative Example.

Consider three workflow tasks $i = 1, 2, 3$ with aggregated temporal signatures over CPU and memory:

$x_1 = [0.40, 0.75, 0.90, 0.35, 0.55, 0.60], \quad x_2 = [0.20, 0.50, 0.70, 0.25, 0.40, 0.50], \quad x_3 = [0.30, 0.65, 0.85, 0.$

Their corresponding runtime power outcomes are:

$$y_1 = [12.4, 65], \quad y_2 = [14.1, 72], \quad y_3 = [10.8, 58].$$

KCCA maps both the temporal patterns x_i and the performance power pairs y_i into high-dimensional kernel spaces and finds projections that maximize their mutual correlation. In this example, the first canonical mode reveals that tasks with higher sustained CPU activity (x_1, x_3) correspond to lower execution time and reduced power consumption, while the less efficient task (x_2) shows a distinct temporal signature characterized by fluctuating utilization and higher runtime.

Random Forest Regression To complement the KCCA model, we trained two non-parametric regressors based on Random Forests—one to predict mean task power and one to predict task runtime from the same preprocessed feature matrix. The power model is trained on mean per-task energy-rate labels, while the runtime model uses task durations as targets. As a sanity check, we established simple baselines by predicting the training-set mean once for power and once for runtime on the test split. These baselines quantify the minimum improvement any learned model must exceed.

Example. Assume two clusters:

$$C_1 = \{t_1, t_2\}, \quad C_2 = \{t_3\},$$

and each task has a CPU Memory signature with three pattern points:

$t_1 = [0.4, 0.7, 0.9, 0.5, 0.6, 0.7], \quad t_2 = [0.3, 0.5, 0.8, 0.4, 0.5, 0.6], \quad t_3 = [0.2, 0.4, 0.6, 0.3, 0.4, 0.5].$

Cluster features are summed:

$$F_1 = t_1 + t_2 = [0.7, 1.2, 1.7, 0.9, 1.1, 1.3], \quad F_2 = t_3.$$

Model predictions yield

$$\hat{Y} = \begin{bmatrix} 10.8 & 62.5 \\ 14.2 & 75.1 \end{bmatrix},$$

representing predicted runtime (s) and energy (W) per cluster.

4.4 Simulation of Task Co-location during Workflow Execution

4.4.1 Outline of Simulation Components

The simulator for co-location strategies builds upon three fundamental design pillars that reflect the main optimization opportunities identified in the co-location problem: resource allocation, queue ordering, and job placement.

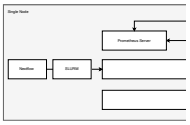

Figure 4.3: The monitoring client alongside the needed system components.

The design of distributed HPC systems centers on three core aspects: resource allocation, queue ordering and dispatching, and job placement. To apply the concepts introduced in **??** and **??**, the simulation of scientific workflow task co-location adheres to these principles, providing a framework that supports multiple strategies for realistic exploration of this objective. As seen in 4.3, we structure the concept of the simulator around these three pillars. The first pillar concerns resource allocation, which determines whether jobs are executed in exclusive or shared mode. Traditional HPC schedulers allocate full nodes to single jobs, task co-location naturally assumes that multiple tasks can coexist efficiently if they do not saturate the same resources simultaneously. The simulator therefore needs to model node-sharing policies where jobs may share cores or memory bandwidth. Since the effectiveness of co-location depends on the characteristics of jobs and directly influence the throughput of the processing engine the simulator needs to offer an interface for alternative queue management strategies, ranging from conventional FIFO ordering to heuristic reordering that prioritizes beneficial pairings. The third pillar, job placement, determines how tasks are assigned to available resources. The simulator needs to support random placement and heuristic strategies that intentionally group complementary tasks. This

setup aims at making it possible to study how the integration of task co-location choices during schedulung and task mapping influence overall system performance and resource efficiency, providing a controlled environment for algorithmic investigations.

4.4.2 Embedding Task Co-location into Scheduling Heuristics

Our algorithmic approach to define the behavior of the components resource manager, allocator and scheduler introduced in 4.3 is grounded in heuristic principles. We choose to not formulate the embedding of co-location into the scheduling process as optimization problem but through decision-making within simple, interpretable scheduling and task mapping rules. Heuristic algorithms are well suited for such settings because they rely on deterministic, rule-based traversal of the search space rather than exhaustive or stochastic exploration. They exploit domain-specific knowledge, in our case energy-awareness and structured criteria—such as task priorities, resource affinities, or workload complementarities—to produce acceptable solutions efficiently without guaranteeing global optimality. We divide the task-to-node mapping into two steps. First, a queue of ready tasks is generated according to a defined orderign strategy before the mapping to resources of these ordered tasks is performed. For the scheduling we choose to adpot list-scheduling algorithms, which form one of the most established heuristic approaches for workflow scheduling. They assign a priority or ranking to each task based on topological and performance factors (e.g., critical path length, execution cost, or communication overhead), producing a priority list; second, they iteratively select the highest-priority unscheduled task and map it to a processor. As both phases can be modeled decoupled, there opens a chance to incorporate intelligent decision-making. Machine learning list-scheduling approaches build on data-driven models that learn decision policies from historical workflow executions or directly during runtime. To complete the foundation of our heuristic design we formally introduce the simulation environment, consisting of the scientific workflow as the application that executes tasks and a system model that we use to represent our algorithmic approach to the co-location problem.

Simulation System Model Workflow Properties

Let the workflow be represented as a directed acyclic graph (DAG)

$$G = (T, E)$$

where

- $T = \{t_1, t_2, \ldots, t_n\}$ denotes the set of **tasks**, and
- $E \subseteq T \times T$ denotes **data or control dependencies** between tasks.

A directed edge $(t_i, t_j) \in E$ indicates that task t_j can start only after t_i has completed.

Task Properties

Each task $t_i \in T$ is associated with the following attributes:

$$\begin{aligned} \text{req\_cores}(t_i) &\in \mathbb{N}_{\geq 1} \quad \text{number of CPU cores required,} \\ \text{req\_mem}(t_i) &\in \mathbb{R}_{>0} \quad \text{memory requirement in bytes.} \end{aligned}$$

Infrastructure Model

Let $\mathcal{H} = \{h_1, h_2, \ldots, h_m\}$ denote the set of available **hosts**, where each host h_j is characterized by

$$C_j \in \mathbb{N}_{>0} \text{ (total number of cores)}, \qquad c(h_j) \in \mathbb{N}_{\geq 0} \text{ (current number of idle cores)}.$$

A **virtual machine (VM)** is represented as

$$v = (C_v, M_v, h_j),$$

where C_v is the number of virtual cores, M_v the assigned memory, and h_j the physical host on which it is instantiated.

The set of currently **ready tasks** (those whose dependencies are satisfied) is denoted by

$$\mathcal{Q} = \{t_1, t_2, \ldots, t_k\}.$$

Resource Assignment

A mapping of tasks to hosts and VMs is represented as

$$M = (h, \mathcal{T}_h, \text{colocMap}_h, \Phi_h),$$

where

- $h \in \mathcal{H}$ is the assigned host,
- $\mathcal{T}_h \subseteq T$ is the set of tasks mapped to h,
- colocMap_h describes task clusters on h, and
- Φ_h represents associated file or data locations.

The set of mappings for an allocation interval is written as

$$\mathcal{M} = \{M_1, M_2, \ldots, M_p\}.$$

Task Co-location

A **co-location mapping**, produced by a scheduler is defined as

$$\text{colocMap} = \{(C_i, \mathcal{C}_i) \mid \mathcal{C}_i \subseteq T\},$$

where $\mathcal{C}_i$ is a **cluster of tasks** to be executed together within a single virtual machine, ideally selected based on their resource affinity or complementary utilization patterns.

Oversubscription

An **oversubscription factor** $\alpha \in [0, 1]$ allows up to

$$N_{\max}(h_j) = \lceil C_j(1 + \alpha) \rceil$$

tasks to be scheduled concurrently on a VM on host h_j.

Execution Dynamics

At runtime:

- **Node Assigners** determine host placement.

- **Schedulers** generate task queues and co-location groupings.
- **Allocators** instantiate and start VMs according to host-task mappings.
- The **Job Manager** executes tasks, monitors VM lifecycles, and updates resource states.

To evaluate the effectiveness of the proposed approach, we compare it against a series of progressively refined baseline algorithms that address the co-location problem with increasing complexity. These baselines range from simple scheduling heuristics to more advanced strategies that gradually incorporate awareness of co-location effects. The following table summarizes all baselines considered in this study. It is important to note that while some of them already involve concurrent execution of tasks, this represents uncontrolled co-location rather than informed or optimized placement. For clarity and conciseness, detailed algorithmic designs of these baselines are provided in the appendix, while this section focuses on describing their conceptual behavior.

Baseline Algorithms

Algorithm 4: ShaReComp Simulation - WRENCH Framework

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$, scheduling policy π, oversubscription factor α, optional co-location API $\mathcal{S}$

Output: workflow executed with policy-driven scheduling, node assignment, a adaptive resource management

Initialize system state: host capacities, ready queue $\mathcal{Q}$, and monitoring layer
while *workflow G not completed* **do**
  Update $\mathcal{Q}$ with all newly ready tasks
  Perform **task scheduling**: prioritize tasks in $\mathcal{Q}$ according to policy π
  Perform **node assignment**: select suitable host(s) $h \in \mathcal{H}$ using π
  Perform **resource allocation**: determine allowed capacity $n_{\max} = f(c(h),$ and reserve resources
  if *policy π supports co-location* **then**
    Optionally query $\mathcal{S}$ to group tasks by affinity and launch them on share VMs
  else
    Launch one task per VM on assigned host
  Monitor execution and task completions
  Release resources and enqueue successors of completed tasks
return *workflow complete*

Table 4.5: Overview of ShaRiff and ShaReComp Scheduling Algorithms.

Algorithm	**Type**	**Description**
ShaRiff 1	Contention-Aware Co-location	Uses the ShaReComp API to determine affinity-based co-location groups, minimizing interference between tasks.
ShaRiff 2	Adaptive Co-location + Over-Subscription	Extends ShaRiff 1 by allowing safe CPU over-subscription based on affinity predictions.
ShaRiff 3	Round-Robin Node Assignment + Co-location	Schedules tasks round-robin across hosts while applying affinity-based co-location through the ShaReComp API.
ShaReComp	Adaptive Max-Parallel Co-location + Over-Subscription	Integrates parallel scheduling, affinity-based co-location, and controlled over-subscription for optimized throughput and energy efficiency.

Algorithmic Examples of Task Mapping with guided Co-location

Algorithm 5: ShaRiff 1 — Scheduling

Input: workflow $G = (T, E)$ ShaReComp co-locat

Output: all tasks $t_i \in T$ exe improved efficiency

Initialize idle cores $c(h_j) \leftarrow$ source tasks of G (FIFO or

while *not all tasks* $t_i \in T$ *co*

  if $\mathcal{Q}$ *is empty* **then**

    Wait until any task t_r

    $c(h(t_r)) \leftarrow c(h(t_r)) +$

    into $\mathcal{Q}$ if all predeces

  Build available-host list L

  Initialize empty host tas

  foreach *host* $h \in L$ *and*

    Select up to $c(h)$ read

    $\Phi(\mathcal{T}_h)$; Query ShaRe

    `co-location group`

    Add mapping $(h, \mathcal{T}_h, \Phi$

  foreach *mapping* $(h, \mathcal{T}_h,$

    foreach *group* $\mathcal{C}_k \in c$

      $C_{\text{req}} \leftarrow \text{sum}(\text{req\_c}$

      $v_h = (C_{\text{req}}, M_{\text{req}},$

      $c(h) \leftarrow \max(0, c($

  Wait until any task t_r co

  $c(h(t_r)) \leftarrow c(h(t_r)) + \text{req}$

  then

    Destroy VM

  For each successor t_s of t_r

return *workflow complete*

This variant implements ShaRiff (share resources if feasible), which augments the FIFO pipeline with an external co-location adviser and a cluster-aware allocator. Tasks are still dequeued in strict arrival order. Before placement, the scheduler invokes ShaRiff with the current set of ready tasks and receives clusters of jobs that are predicted to co-locate well (i.e., complementary resource profiles / low expected interference). The node-assignment stage then ranks hosts by descending idle-core capacity and fills the largest host first: it forms a batch of up to that hosts idle cores and attaches the ShaRiff cluster map to the batch; if tasks remain, it proceeds to the next host in the ranked list. A small-queue fast path ensures dispatch even when only a few tasks are available. The allocator realizes the advisers plan one VM per recommended cluster on the chosen host. For each multi-task cluster, it provisions a VM whose vCPU count and memory equal the sum of the clustered tasks declared requirements, starts the VM, and submits the tasks to that same virtual compute service. Singleton clusters are grouped into a shared VM on the host (current variant) to avoid VM fragmentation; each submitted task keeps its own job identity, and the allocator tracks VM lifecycle across all tasks assigned to it, tearing the VM down only

after the last co-located task completes. Conceptually, ShaRiff preserves FIFO ordering and capacity-ranked host filling, but replaces random batching with adviser-driven clustering. The effect is to co-locate tasks that are likely complementary (e.g., CPU-bound with I/O-bound), thereby reducing contention and improving per-host utilization without oversubscription. When the adviser yields singletons, the system still shares if feasible by pooling them into a shared VM, maintaining the same deterministic provisioning and lifecycle rules.

Algorithm 6: ShaRiff 2 — Adaptive Max-Parallel VM Co-Location with Controlled Over-Subscription

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$, idle cores $c(h_j)$, oversubscription factor α, ShaReComp co-location API $\mathcal{S}$

Output: workflow executed with affinity-based co-location, maximal host parallelism, and safe oversubscription

Initialize idle cores $c(h_j) \leftarrow C_j$ for all $h_j \in \mathcal{H}$ Initialize ready queue $\mathcal{Q}$ with all source tasks of G (FIFO order)

while *not all tasks* $t_i \in T$ *completed* **do**

  if $\mathcal{Q}$ *is empty* **then**

    Wait until any task t_r completes Release its cores: $c(h(t_r)) \leftarrow c(h(t_r)) + \text{req\_cores}(t_r)$ For each successor t_s of t_r, enqueue t_s into $\mathcal{Q}$ if all predecessors are completed

    continue

  Build available-host list $L = \{\, h \in \mathcal{H} \mid c(h) > 0 \,\}$, sorted by $c(h)$ descending Initialize empty host task mapping list $\mathcal{M}$

  foreach *host* $h \in L$ *and while* $\mathcal{Q}$ *not empty* **do**

    Compute oversubscription limit $n_{\max} = \lceil c(h) \times (1 + \alpha) \rceil$ Select up to $n_{\max}$ ready tasks from $\mathcal{Q}$ into $\mathcal{T}_h$ Compute file-location map $\Phi(\mathcal{T}_h)$ Query ShaReComp API: colocMap $\leftarrow \mathcal{S}(\mathcal{T}_h)$; `// returns co-location groups`

    Add mapping $(h, \mathcal{T}_h, \Phi(\mathcal{T}_h), \text{colocMap})$ to $\mathcal{M}$

  foreach *mapping* $(h, \mathcal{T}_h, \Phi, \mathit{colocMap}) \in \mathcal{M}$ **do**

    foreach *group* $\mathcal{C}_k \in \mathit{colocMap}$ **do**

      $C_{\text{req}} \leftarrow \text{sum}(\text{req\_cores}(\mathcal{C}_k))$ $M_{\text{req}} \leftarrow \text{sum}(\text{req\_mem}(\mathcal{C}_k))$ **if** $C_{req} > c(h)$ **then**

        Allocate $v_h = (c(h), M_{\text{req}}, h)$; ; `// oversubscription active`

      else

        Allocate $v_h = (C_{\text{req}}, M_{\text{req}}, h)$

      Launch all $t \in \mathcal{C}_k$ on v_h $c(h) \leftarrow \max(0, c(h) - C_{\text{req}})$ Remove $\mathcal{C}_k$ from $\mathcal{Q}$

  Wait until any task t_r completes Release its cores: $c(h(t_r)) \leftarrow c(h(t_r)) + \text{req\_cores}(t_r)$ **if** *no active tasks remain on its VM* **then**

    Destroy VM

  For each successor t_s of t_r: if all predecessors are completed, enqueue t_s into $\mathcal{Q}$

return *workflow complete*

This variant retains the ShaRiff-augmented FIFO pipeline but enables controlled oversubscription during placement and VM sizing. Tasks are dequeued in arrival order. Before dispatch, the scheduler queries ShaRiff with the current ready set and receives clusters of tasks predicted to co-locate well (complementary resource use / low interference). Hosts are ranked by descending idle-core capacity; the assigner then fills the largest host first

with a batch whose size may exceed the hosts free cores by a fixed factor (e.g., +25%). If tasks remain, it proceeds to the next host and repeats. The allocator implements the advisers plan one VM per cluster on the chosen host, but with oversubscription semantics. For multi-task clusters, it provisions a VM whose vCPU and memory equal the sum of the clusters requests—even if that exceeds the hosts currently free cores (hard oversub). For singleton clusters collected on the same host, it provisions a shared VM and caps vCPUs at the hosts free cores when necessary (soft cap). In both cases the VM is started once, all tasks in the cluster are submitted to the same virtual compute service, and the VM is torn down only after the last co-located task finishes. Conceptually, this policy combines adviser-driven co-location with capacity-aware overbooking: FIFO ordering and capacity-ranked host filling are preserved, but batches may intentionally outsize instantaneous capacity to exploit latency hiding and temporal slack (e.g., I/O wait, imbalanced phases). Because ShaRiff groups complementary tasks, oversubscription tends to translate into higher throughput and energy efficiency than naive overbooking; however, when clustered tasks are less complementary, contention can surface, making this variant an explicit trade-off between utilization and interference.

Algorithm 7: ShaRiff 3 — Round-Robin Node Assignment with contention-aware VM Co-Location

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$, idle cores $c(h_j)$, ShaReComp co-location API $\mathcal{S}$

Output: tasks executed using round-robin host selection with affinity-based VM co-location

Initialize idle cores $c(h_j) \leftarrow C_j$ for all $h_j \in \mathcal{H}$ Initialize ready queue $\mathcal{Q}$ with all source tasks of G (FIFO order) Initialize round-robin index $r \leftarrow 0$

while *not all tasks* $t_i \in T$ *completed* **do**

  if $\mathcal{Q}$ *is empty* **then**

    Wait until any task t_r completes Release its cores: $c(h(t_r)) \leftarrow c(h(t_r)) + \text{req\_cores}(t_r)$ For each successor t_s of t_r, enqueue t_s into $\mathcal{Q}$ if all predecessors are completed

    continue

  Select next host $h = \mathcal{H}[r \bmod |\mathcal{H}|]$ Update round-robin index: $r \leftarrow (r + 1) \bmod |\mathcal{H}|$ Retrieve available cores $C = c(h)$ Select up to C ready tasks from $\mathcal{Q}$ into $\mathcal{T}_h$ Query ShaReComp API: colocMap $\leftarrow \mathcal{S}(\mathcal{T}_h)$;

  `// returns co-location groups`

  foreach *group* $\mathcal{C}_k \in$ *colocMap* **do**

    $C_{\text{req}} \leftarrow \text{sum}(\text{req\_cores}(\mathcal{C}_k))$ $M_{\text{req}} \leftarrow \text{sum}(\text{req\_mem}(\mathcal{C}_k))$ Allocate $v_h = (C_{\text{req}}, M_{\text{req}}, h)$ Launch all $t \in \mathcal{C}_k$ on v_h $c(h) \leftarrow \max(0, c(h) - C_{\text{req}})$ Remove $\mathcal{C}_k$ from $\mathcal{Q}$

  Wait until any task t_r completes Release its cores: $c(h(t_r)) \leftarrow c(h(t_r)) + \text{req\_cores}(t_r)$ **if** *no active tasks remain on its VM* **then** Destroy VM

  For each successor t_s of t_r: if all predecessors are completed, enqueue t_s into $\mathcal{Q}$

return *workflow complete*

This variant preserves FIFO dequeuing but combines round-robin first-fit placement with ShaRiff-guided intra-VM co-location. The scheduler releases tasks strictly in arrival order. The node-assignment component scans hosts in round-robin/first-fit fashion and picks the first host reporting at least one idle core. It then pulls up to that hosts idle-core capacity

worth of ready tasks and queries ShaRiff for a co-location plan over this batch. The allocator realizes ShaRiffs plan one VM per suggested cluster on the chosen host. For each multi-task cluster, it sizes the VM by summing vCPU and memory requirements of the clusters tasks, starts the VM, and submits all cluster tasks to the same virtual compute service. Tasks that ShaRiff leaves as singletons are grouped onto an additional shared VM on that host; its size is the aggregate of the singletons requests. VM lifecycle is managed per cluster: each VM stays up until all of its assigned tasks complete, then is shut down and destroyed. Conceptually, the policy is first-fit host, adviser-driven packing. It preserves FIFO ordering and simple first-fit host selection while letting ShaRiff decide which tasks should share a VM to reduce interference (by favoring complementary profiles). Unlike the max-parallel variants, this strategy does not oversubscribe cores; it fills only the currently free capacity of the first eligible host and relies on ShaRiffs clustering to raise utilization and efficiency through informed co-location.

This scheduler extension augments the ShaRiff-based variants with a MinMin ordering layer, a classical heuristic from list scheduling. In standard list scheduling, tasks are iteratively selected based on their earliest estimated completion time; Min Min specifically chooses, at each step, the task (or cluster) with the minimum predicted runtime among all ready candidates and schedules it first. Here, this principle is applied not to individual tasks but to task clusters produced by ShaRiffs co-location analysis. For each scheduling interval, the scheduler requests from ShaRiff a co-location partition of the ready tasks, grouping them into clusters that are predicted to interact efficiently when sharing a VM. It then queries a prediction service for each cluster, obtaining runtime estimates through the chosen model (e.g., KCCA). The clusters are ordered by ascending predicted runtime, and this order defines the execution sequence. Node selection and VM provisioning are delegated to the ShaRiff node assignment and allocator components, which handle placement and resource sizing as usual. Conceptually, this forms a Min Min list scheduler over co-located clusters: it maintains ShaRiffs intelligent co-location strategy while globally minimizing queueing delay and improving average completion time by prioritizing faster clusters. This layer is independent of the underlying allocation or node assignment logic and purely refines execution ordering to exploit performance prediction while preserving all structural and capacity constraints of the ShaRiff framework.

5 Implementation

This chapter details the technical implementation of the concepts introduced in the previous section. While the Approach chapter established the conceptual and algorithmic foundations of the proposed scheduling framework, the following sections focus on how these ideas were realized in practice. The implementation emphasizes architectural modularity, clear component interfaces, and the integration of machine learning-based decision layers within a simulation-driven environment. Each subsystem—from the monitoring client and statistical modeling backend to the simulation environment—was designed to remain functionally independent while communicating through well-defined data and control flows. This decoupled design not only facilitates reproducibility and maintainability but also enables future extensions, such as the replacement of predictive models or the addition of new scheduling policies, without major structural changes. The remainder of this chapter outlines the overall system architecture, justifies the chosen technologies and design principles, and describes the concrete implementation of the monitoring client, the statistical learning components, and the simulator setup that collectively form the experimental framework.

5.1 System Architecture

5.2 Technology and Design Choices

Table 5.6: Monitoring features and associated technical components.

Component Category	Software Used	Comments / Functionality
Workflow Execution		
Operating System	Ubuntu 22.04.5 LTS	Base system environment for all components.
Kernel	Linux 5.15.0-143-generic	Provides low-level system resource management.
Workflow Management Engine	Nextflow 25.06.0	Controls workflow DAG execution and task dependency resolution.
Virtualization Layer	Docker Engine 28.3.1 (Community)	Provides isolated VM environments for task execution.
Resource Manager	Slurm 24.05.2	Allocates CPU and memory resources dynamically across hosts.
Monitoring System		
Time-Series Database	Prometheus Server 3.3.1	Central metric collector and time-series database.
Monitoring Client	Go 1.24.1	Collects per-node and per-container resource metrics.
Workload Experiments		
Benchmark Execution	stress-ng 0.13.12	Generates controlled CPU, memory, and I/O workloads.
Monitoring	Deep-Mon (custom fork) with Python 3.10.12 & BCC 0.35.0	Captures resource traces during workload execution.
Data Analysis		
Feature Engineering	Jupyter Kernel 6.29.5, Python 3.10.12, Poetry 2.1.2	Derives and normalizes temporal task signatures.
Clustering	scikit-learn 1.5.2	Groups similar task behaviors into consolidated classes.
KCCA	CCA-Zoo 2.5.0	Learns correlations between performance and energy domains.
Random Forest Regressor	scikit-learn 1.5.2	Predicts runtime and power consumption from task signatures.
Kernel Ridge Regression	scikit-learn 1.5.2	Baseline model for non-linear regression comparison.
Simulation Framework		
Workflow Tracing	Nextflow Tracer (custom fork)	Records execution-level metadata for simulation replay.
Simulation Engine	WRENCH 2.7 & C++17	Evaluates scheduling strategies under controlled conditions.
Clustering API	FastAPI 0.1.0, Python 3.10.12	Provides task grouping via ShaReComp integration.
Prediction API	FastAPI 0.1.0, Python 3.10.12	Interfaces learned models for runtime and energy estimation.

5.3 Decoupled Design for

The extensibility of the overall system is achieved through a strictly decoupled design that separates monitoring, modeling, and simulation components while maintaining clear communication interfaces between them. Each part of the system can evolve independently without impacting others, enabling modular experimentation with new data sources, predictive models, or scheduling strategies. This modularity supports reproducibility and future extensibility, as new data collection layers or simulation backends can be added by extending well-defined interfaces rather than rewriting existing code.

5.3.1 Configurable Monitoring Client

The monitoring client exemplifies this philosophy by relying on a flexible configuration-driven architecture. Using a YAML-based configuration file, it dynamically defines which metrics to collect from heterogeneous data sources such as Prometheus exporters, cAdvisor, eBPF probes, or SNMP-based sensors. The configuration specifies not only the metric names and queries but also the identifiers and units, allowing seamless adaptation to different environments or workflow engines. This separation of logic and configuration enables the same client to operate across diverse infrastructures without recompilation. The clients implementation, built on the Prometheus API, abstracts away the complexity of time-range queries and concurrent metric fetching through lightweight threading and synchronization mechanisms. As a result, developers can extend the monitoring framework simply by adding new data sources or metrics to the configuration file, without altering the underlying collection logic.

Table 5.7: Adaptable Monitoring Configuration Overview.

Monitoring Target	**Enabled**	**Supported Data Sources**	**Collected Metric Types / Adaptability Notes**
Task Metadata		slurm-job-exporter	Collects job metadata (state, runtime, working directory). Can adapt to other job schedulers.
CPU		cAdvisor, ebpf-mon, docker-activity	Captures CPU time and cycles from both container and kernel levels. Supports switching sources for varying granularity.
Memory		cAdvisor, docker-activity	Tracks memory utilization at container or process level. Configurable for byte- or percentage-based metrics.
Disk		cAdvisor	Monitors block I/O and filesystem throughput. Supports extension with storage exporters.
Network		cAdvisor (optional)	Disabled by default due to noise. Can be enabled for network-intensive workflows.
Energy		docker-activity, ebpf-mon, ipmi-exporter, snmp-exporter	Multi-layer energy monitoring from container to node level. Adaptable to hardware sensors and external power meters.
Prometheus Configuration			
The Prometheus backend collects all metrics via configurable scrape intervals and targets. Controller and worker nodes can be flexibly defined, enabling distributed monitoring setups.			

5.3.2 Enabling Access to Co-location Hints in the Simulator

The statistical modeling component, implemented as a standalone FastAPI service, follows a similar modular design. It exposes a clean, language-agnostic HTTP API that separates the inference logic from data ingestion and model management. The service maintains state for clustering and prediction requests, delegating core computational tasks to dedicated helper functions. This decoupling makes it straightforward to replace or add new predictive models, such as neural architectures or alternative regression approaches, without modifying the API contract. The clustering and prediction endpoints can interact with any external workflow manager or simulator via standardized JSON payloads, ensuring flexibility in integrating new pipelines or retraining procedures. This independence between model serving and data processing pipelines also simplifies scalability, allowing the modeling service to be containerized and deployed independently for distributed or cloud-based setups.

Table 5.8: Overview of REST API Endpoints Exposed by the ShaReComp Service.

#	**Endpoint**	**Method**	**Description / Response Schema**
1	`/clusterize_jobs`	POST	Clusters nf-core jobs based on historical data. *Request:* `ClusterizeJobsRequest` (list of job names). *Response:* `ClusterizeJobsResponse` (cluster mapping with run ID).
2	`/predict`	POST	Predicts runtime and power consumption for consolidated job clusters. *Request:* `PredictRequest` (cluster IDs, model types). *Response:* `PredictionResponse` (predicted values for each model).
Component Schemas			
–	`ClusterizeJobsRequest`	Object	**Fields:** `job_names` (array of strings). Required.
–	`ClusterizeJobsResponse`	Object	**Fields:** `run_id` (string), `clusters` (map of arrays). Required.
–	`PredictRequest`	Object	**Fields:** `cluster_ids` (array), `prediction_models` (array of `PredictionModel`). Required.
–	`PredictionModel`	Object	**Fields:** `model_type` (array of strings). Required.
–	`PredictionResponse`	Object	**Fields:** `run_id` (string), `predictions` (nested map of numeric values). Required.

5.3.3 Used Simulator Components

The simulator setup further demonstrates the benefits of this decoupled design. The resource management layer of the simulator exposes generic interfaces for allocators, schedulers, and node assigners, allowing any of them to be replaced or combined dynamically at runtime. This separation allows the same simulator to execute both baseline and experimental resource allocation strategies—including oversubscription, co-location, or ShaRiff-based scheduling—without modifying the controller logic. Through this design, the simulator can execute diverse workflow types, including various nf-core pipelines, by merely switching configuration parameters or class bindings. Moreover, the integration of energy and performance tracing through independent services ensures that extending the simulator with new measurement capabilities does not interfere with the scheduling or execution logic.

6 Evaluation

6.1 Evaluation Setup

6.1.1 Infrastructure

The experiments were conducted on a single-node system equipped with an AMD EPYC 8224P 24-core, 48-thread processor and 188 GB of RAM. The processor supported simultaneous multithreading and frequency boosting up to 2.55 GHz, with 64 MB of shared L3 cache and a single NUMA domain ensuring uniform memory access across all cores. Storage was provided by a 3.5 TB NVMe SSD, and the system ran a 64-bit Linux environment configured for stable and reproducible execution. To collect accurate power usage data, the node was connected to a 12-way switched and outlet-metered PDU (Expert Power Control 8045 by GUDE), which provided per-outlet power measurements via a REST API.

6.1.2 Workflows

Table 6.9: Overview of evaluated nf-core workflows and their inputoutput characteristics.

Workflow	Number of Tasks	Input Files	Output Files	Data Profile
`atacseq`	72	24	185	Bulk chromatin accessibility sequencing (ATAC-seq)
`chipseq`	68	22	172	Bulk chromatin immunoprecipitation sequencing (ChIP-seq)
`rnaseq`	54	18	160	Bulk RNA-seq expression quantification
`scnanoseq`	83	25	210	Single-cell nanopore RNA-seq
`smrnaseq`	59	20	142	Small RNA sequencing (miRNA/siRNA profiling)
`pixelator`	44	16	125	Spatial transcriptomics pixel-based expression mapping
`methylseq`	65	21	170	Whole-genome or targeted DNA methylation sequencing
`viralrecon`	51	19	150	Viral genome assembly and variant analysis
`oncoanalyser`	97	28	260	Comprehensive somatic cancer genome analysis

6.1.3 Monitoring Configuration

Table 6.10: Adaptable Monitoring Configuration Overview.

Monitoring Target	Enabled	Supported Data Sources	Collected Metric Types / Adaptability Notes
Task Metadata		slurm-job-exporter	Collects job metadata (state, runtime, working directory). Can adapt to other job schedulers.
CPU		cAdvisor, ebpf-mon, docker-activity	Captures CPU time and cycles from both container and kernel levels. Supports switching sources for varying granularity.
Memory		cAdvisor, docker-activity	Tracks memory utilization at container or process level. Configurable for byte- or percentage-based metrics.
Disk		cAdvisor	Monitors block I/O and filesystem throughput. Supports extension with storage exporters.
Network		cAdvisor (optional)	Disabled by default due to noise. Can be enabled for network-intensive workflows.
Energy		docker-activity, ebpf-mon, ipmi-exporter, snmp-exporter	Multi-layer energy monitoring from container to node level. Adaptable to hardware sensors and external power meters.
Prometheus Configuration			
The Prometheus backend collects all metrics via configurable scrape intervals and targets. Controller and worker nodes can be flexibly defined, enabling distributed monitoring setups.			

6.1.4 Implemented Models for Task Clustering and Prediction

As described in Chapter 5, the statistical models—including the clustering and prediction components—are provided through a FastAPI implementation, allowing external simulation engines to interact with them programmatically. At the same time, the core functionality of the coloc-app FastAPI service is based on a Jupyter Notebook that contains all implementations of the clustering and predictive models discussed in Chapter 5. Both the notebook and the coloc-app were executed on the host system described in Section

6.1.5 Simulation Setup

Simulated Platform Configuration The infrastructure described in Section 6.1.1 was replicated within the SimGrid simulation environment using its platform description tool, as shown in the following XML configuration. The hosts core performance was calibrated by executing stress-ng benchmarks to determine realistic CPU speeds, while network throughput was measured using sysbench. To determine power states (P-states), the GUDE power meter was used to record power consumption in both idle and active conditions under varying load profiles generated with stress-ng. This procedure ensured that the simulated environment accurately reflected the performance and energy characteristics of the physical test system.

```
1 <?xml version='1.0'?>
```

```
<!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
<platform version="4.1">
    <zone id="AS0" routing="Full">

        <!-- The host on which the WMS will run -->
        <host id="WMSHost" speed="10Gf" pstate="0" core="24">
            <disk id="hard_drive" read_bw="100MBps" write_bw="10000
                MBps">
                <prop id="size" value="5000GiB"/>
                <prop id="mount" value="/scratch"/>
            </disk>
            <!-- The power consumption in watt for each pstate (
                idle:all_cores) -->
            <prop id="wattage_per_state" value="115:216" />
            <prop id="wattage_off" value="0.0" />
        </host>

        <!-- The host on which the VirtualizedClusterComputeService
             will run -->
        <host id="VirtualizedClusterProviderHost" speed="32Gf"
            pstate="0" core="24">
            <prop id="wattage_per_state" value="115:216" />
            <prop id="wattage_off" value="0.0" />
       </host>

        <host id="VirtualizedClusterHost1" speed="32Gf,28Gf,20Gf,12
            Gf,8Gf" pstate="0" core="24">
            <prop id="ram" value="188GiB" />
            <!-- The power consumption in watt for each pstate (
                idle:all_cores) -->
            <prop id="wattage_per_state" value="116:133:220,
                114:129:202, 111:125:186, 108:121:170,106:118:156"/>
            <prop id="wattage_off" value="10" />
        </host>

        <host id="VirtualizedClusterHost2" speed="32Gf,28Gf,20Gf,12
            Gf,8Gf" pstate="0" core="24">
            <prop id="ram" value="188GiB" />
            <!-- The power consumption in watt for each pstate (
                idle:all_cores) -->
            <prop id="wattage_per_state" value="116:133:220,
                114:129:202, 111:125:186, 108:121:170,106:118:156"/>

            <prop id="wattage_off" value="10" />
        </host>

        <host id="VirtualizedClusterHost3" speed="32Gf,28Gf,20Gf,12
            Gf,8Gf" pstate="0" core="24">
            <prop id="ram" value="188GiB" />
            <!-- The power consumption in watt for each pstate (
                idle:all_cores) -->
            <prop id="wattage_per_state" value="116:133:220,
                114:129:202, 111:125:186, 108:121:170,106:118:156"/>

            <prop id="wattage_off" value="10" />
        </host>
```

```
45
46          <!-- A network link -->
47          <link id="network_link" bandwidth="1250MBps" latency="10us"
                />
48          <!-- WMSHost's local "loopback" link -->
49          <link id="loopback_WMSHost" bandwidth="1000EBps" latency="0
                us"/>
50          <!--VirtualizedClusterProviderHost's local "loopback" link
                -->
51          <link id="loopback_VirtualizedClusterProviderHost"
                bandwidth="1000EBps" latency="0us"/>
52          <!--VirtualizedClusterHost1's local "loopback" link -->
53          <link id="loopback_VirtualizedClusterHost1" bandwidth="1000
                EBps" latency="0us"/>
54          <!--VirtualizedClusterHost2's local "loopback" link -->
55          <link id="loopback_VirtualizedClusterHost2" bandwidth="1000
                EBps" latency="0us"/>
56          <!--VirtualizedClusterHost3's local "loopback" link -->
57          <link id="loopback_VirtualizedClusterHost3" bandwidth="1000
                EBps" latency="0us"/>
58
59          <!-- The network link connects the two hosts -->
60          <route src="WMSHost" dst="VirtualizedClusterProviderHost">
                <link_ctn id="network_link"/> </route>
61          <route src="WMSHost" dst="VirtualizedClusterHost1"> <
                link_ctn id="network_link"/> </route>
62          <route src="WMSHost" dst="VirtualizedClusterHost2"> <
                link_ctn id="network_link"/> </route>
63          <route src="WMSHost" dst="VirtualizedClusterHost3"> <
                link_ctn id="network_link"/> </route>
64          <route src="VirtualizedClusterProviderHost" dst="
                VirtualizedClusterHost1"> <link_ctn id="network_link"/>
                </route>
65          <route src="VirtualizedClusterProviderHost" dst="
                VirtualizedClusterHost2"> <link_ctn id="network_link"/>
                </route>
66          <route src="VirtualizedClusterProviderHost" dst="
                VirtualizedClusterHost3"> <link_ctn id="network_link"/>
                </route>
67          <route src="VirtualizedClusterHost1" dst="
                VirtualizedClusterHost2"> <link_ctn id="network_link"/>
                </route>
68          <route src="VirtualizedClusterHost1" dst="
                VirtualizedClusterHost3"> <link_ctn id="network_link"/>
                </route>
69          <route src="VirtualizedClusterHost2" dst="
                VirtualizedClusterHost3"> <link_ctn id="network_link"/>
                </route>
70
71          <!-- Each loopback link connects each host to itself -->
72          <route src="WMSHost" dst="WMSHost">
73              <link_ctn id="loopback_WMSHost"/>
74          </route>
75          <route src="VirtualizedClusterProviderHost" dst="
                VirtualizedClusterProviderHost">
```

```
76              <link_ctn id="loopback_VirtualizedClusterProviderHost"
                    />
77          </route>
78          <route src="VirtualizedClusterHost1" dst="
                VirtualizedClusterHost1">
79              <link_ctn id="loopback_VirtualizedClusterHost1"/>
80          </route>
81          <route src="VirtualizedClusterHost2" dst="
                VirtualizedClusterHost2">
82              <link_ctn id="loopback_VirtualizedClusterHost2"/>
83          </route>
84          <route src="VirtualizedClusterHost3" dst="
                VirtualizedClusterHost3">
85              <link_ctn id="loopback_VirtualizedClusterHost3"/>
86          </route>
87      </zone>
88  </platform>
```

Listing 1: Example XML Configuration File

Table 6.11: Overview of Baseline Scheduling Algorithms.

Algorithm	**Type**	**Description**
Baseline 1	FIFO + Round-Robin	Executes tasks in FIFO order and assigns them to hosts in a round-robin fashion without co-location or backfilling.
Baseline 2	FIFO + Backfilling	Assigns tasks in FIFO order to the first available host, allowing idle hosts to be backfilled opportunistically.
Baseline 3	FIFO + VM Co-location	Groups multiple ready tasks on the same host within a single VM if sufficient resources are available.
Baseline 3.1	Max-Core VM Co-location	Prefers the host with the largest number of idle cores for task co-location to maximize utilization.
Baseline 3.2	Max-Parallel VM Co-location	Distributes ready tasks across all available hosts in parallel, promoting high concurrency across nodes.
Baseline 4	VM Co-location + Over-Subscription	Extends co-location by allowing controlled CPU over-subscription on selected hosts using an oversubscription factor α.
Baseline 4.1	Max-Parallel Co-location + Over-Subscription	Combines parallel host utilization with co-location and controlled CPU over-subscription for improved throughput.

Baseline Scheduling Algorithms The baseline scheduling algorithm implements a simple, sequential execution model designed to simulate isolated task processing within a virtualized cluster. The scheduling process is divided into three abstract components that operate in a fixed order: task scheduling, node assignment, and resource allocation. The scheduler applies a first-in, first-out (FIFO) policy, maintaining a queue of workflow tasks sorted by their readiness. Tasks are retrieved from this queue strictly in order of

arrival, preserving dependency constraints and ensuring a fully deterministic execution sequence without reordering or prioritization. Once a task is selected for execution, the node assignment component distributes it across available compute hosts using a round-robin policy. This mechanism cycles through hosts in sequence, ensuring an even and systematic distribution of tasks across the cluster. No host is assigned more than one active task at a time, enforcing exclusive execution and preventing contention for shared resources. The next variant keeps the same FIFO scheduler and VM-based allocator as Baseline 1, but replaces exclusive node assignment with a greedy backfilling policy. Tasks are still dequeued strictly in arrival order by the FIFO scheduler. For each ready task, the node assignment component queries the cluster for the current number of idle cores per host and performs a first-fit scan: it selects the first host that reports at least one idle core, without requiring the host to be completely idle. The allocator then provisions a VM on the chosen host, binds the tasks inputs/outputs, submits the job to that VM, and on completion shuts the VM down and destroys it. Conceptually, this turns the placement step into gap filling rather than strict exclusivity. Multiple tasks can be co-located on the same host up to its core capacity, increasing instantaneous parallelism and utilization. Baseline 3 does not differ in the scheduling behavior but replaces the standard allocator and node assignment with components that allow for co-location. When the node assignment component queries the cluster for idle-core availability, it again selects the first host with available cores. However, instead of launching one VM per task, all ready tasks that fit within the hosts idle-core capacity are grouped into a single batch. These tasks are then co-located inside one shared VM instance that is dimensioned according to the aggregate resource requirements of the batch—its vCPU count and memory size are computed as the sum of the respective task demands. Conceptually, this baseline captures the behavior of intra-VM co-location, where multiple independent tasks share the same virtual machine instead of being distributed across separate ones. Baseline 3 is extended by 2 variants where the first one extends thenode assignment component to query the cluster for idle-core availability and selects the host with the maximum amount of available cores. However, instead of launching one VM per task, all ready tasks that fit within the hosts idle-core capacity are grouped into a single batch. These tasks are then co-located inside one shared VM instance that is dimensioned according to the aggregate resource requirements of the batch—its vCPU count and memory size are computed as the sum of the respective task demands. The second extension replaces the placement policy with a selection step for the host with maximum idle cores. At each dispatch, the node-assignment component queries the cluster for the current idle cores per host map and picks the host with the largest number of free cores. It then forms a batch by taking as many ready tasks from the FIFO head as the chosen host can accommodate. Compared to first-fit co-location, it tends to reduce residual fragmentation by packing work onto the most spacious node, while still honoring FIFO ordering and leaving task runtime/I/O handling unchanged. The 4th baseline retains the same FIFO scheduler but introduces a node assignment and allocation policy focused on maximizing parallel host utilization. Upon each scheduling cycle, the node assignment component queries the cluster for the current number of idle cores per host, filters out fully occupied nodes, and ranks the remaining hosts in descending order of available cores. It then assigns tasks in batches, filling the host with the highest idle capacity first and grouping as many ready tasks as the hosts idle-core count allows. Once the first host is filled, the process continues with the next host until all tasks in the ready queue are mapped. The allocator provisions one VM per host batch, sizing it to match the aggregate requirements of all tasks assigned to that host. The resulting VMs vCPU and memory configuration reflect the total core and memory

demands of the batch. Each task in the batch is submitted as an independent job to the same virtual compute service, and the VM remains active until all its co-located tasks have finished, at which point it is shut down and destroyed. The last baseline extends the previous one by allowing controlled CPU over-subscription during co-location. At each scheduling cycle, the node assignment component queries per-host idle cores, sorts hosts in descending idle capacity, and fills the largest host first. Unlike the non-oversubscribed version, the per-host batch may exceed the currently idle cores by a fixed factor the batch limit is set to. The procedure continues down the ranked host list, forming one batch per host in the same cycle. The allocator provisions one VM per host batch, but caps the VMs vCPU count to the hosts actual idle cores at allocation time not the sum of task core demands, while sizing memory to the aggregate of the batched tasks. All tasks in the batch are then submitted to that single VM and execute concurrently on a vCPU pool intentionally smaller than their combined declared cores. The VM remains active until all co-located tasks complete, then it is shut down and destroyed. Crucially, the degree of contention—and thus realized speedup or slowdown—depends on the complementarity of the co-located task profiles. When CPU-, memory-, and I/O-intensive phases overlap unfavorably oversubscription amplifies interference and queueing on scarce vCPUs. When profiles are complementary, the same oversubscription admits more useful overlap with less contention, improving per-host throughput. Conceptually, this variant implements parallel, capacity-ranked co-location with controlled oversubscription.

6.2 Experiment Results

The section on experimental results is organized as follows. We begin with a brief discussion of the monitoring results obtained using the configuration described earlier. Next, we revisit the approach introduced in Chapter 4 by examining the workload experiments and the resulting measurements that form the basis for subsequent evaluation steps. We then present the outcomes of the statistical methods applied in this work, starting with an in-depth analysis of the task consolidation approach introduced in Chapter 4. Building on these results, we continue with an interpretation of the outcomes from training two predictive models on the clustering data. Finally, we integrate all components into a unified simulation framework. Using this setup, we execute all workflows introduced at the beginning of Chapter 6 with the algorithms detailed in Chapter 4 and the appendix. Several aspects of the simulation results are discussed before Chapter 7 concludes with an overall evaluation and interpretation of the findings.

6.2.1 Measuring Interference during Benchmark Executions

6.2.2 Dissimilarity-based Task Clustering

Table 6.12: Average inter-cluster difference comparison between ShaReComp and random clustering across workflows.

Workflow	Number of Tasks	Avg. ShaReComp Cluster Difference	Avg. Random Cluster Difference
atacseq	72	0.214	0.482
chipseq	68	0.237	0.495
rnaseq	54	0.201	0.468
scnanoseq	83	0.225	0.510
smrnaseq	59	0.208	0.490
pixelator	44	0.190	0.455
methylseq	65	0.232	0.503
viralrecon	51	0.216	0.487
oncoanalyser	97	0.240	0.525

6.2.3 Predicting Runtime and Energy Consumption of Task Clusters

Table 6.13: Summary of model configurations and performance metrics for task-level prediction.

Workflow Tasks	Model Type	Hyperparameters	R^2	Cross-Validation	Comments
549	KCCA + Kernel Ridge Regression	Kernel = laplacian, latent_dim = 2	0.25	5-fold (KFold, shuffle=True, seed=42)	Used canonical correlation features; selected via GridSearchCV across 7 kernels. Combined with Kernel Ridge Regression for final prediction of runtime and power. Moderate performance due to cross-domain variance.
549	Random Forest (Runtime)	n_estimators [200, 2000], max_depth [10, 110], max_features {log2, sqrt}, bootstrap {True, False}	0.50	7-fold (RandomizedSearchCV)	Achieved best runtime prediction accuracy of 27.66%. Mean absolute error: 12.36 units. Good balance between model complexity and generalization.
549	Random Forest (Power)	n_estimators [200, 2000], same grid as runtime model	0.14	7-fold (RandomizedSearchCV)	Lower fit due to higher variance in power traces. Mean absolute error (baseline): 124.65 units. Captures coarse-grained power patterns but limited fine-grained accuracy.

6.2.4 Simulation of Scheduling Algorithms with Co-location

7 Discussion

8 Conclusion and Future Work

8.1 Final Remarks

8.2 Outlook

1. Monitoring Improvement, higher task coverage, better ebpf, lower overhead, better energy models
2. More in-depth time-series treatment for better feature-vectors
3. Dimensionality reduction, feature selection based on highest explanation
4. More sophisticated affinity score calculation based on lower-level interference measurements
5. Treatment of static time-series for distance calculation
6. Refinement of regression approach within KCCA by comparing to other means
7. Extending by more suitable or sophisticated models.
8. Integrating cost-functions into the co-location strategies so that optimization problems can be formulated and account for single or multi-obectives.
9. Extending and calibrating the simulation framework with energy-sources for more realistic simulation results
10. Reformulating the co-location problem from consolidation problem into degredataion prediction

References

[BJ03] F.R. Bach and M.I. Jordan. "Kernel independent component analysis". In: *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).* Vol. 4. 2003, pp. IV–876. DOI: 10.1109/ICASSP.2003.1202783.

[BL13] Marc Bux and Ulf Leser. "Parallelization in Scientific Workflow Management Systems". In: (2013). arXiv: 1303.7195 [astro-ph.IM].

[BL16] Andreas Blanche and Thomas Lundqvist. "Terrible Twins: A Simple Scheme to Avoid Bad Co-Schedules". In: Jan. 2016. DOI: 10.14459/2016md1286952.

[BSS18] Rolando Brondolin, Tommaso Sardelli, and Marco D. Santambrogio. "DEEPMon: Dynamic and Energy Efficient Power Monitoring for Container-Based Infrastructures". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 2018, pp. 676–684. DOI: 10.1109/IPDPSW.2018.00110.

[Bad+22] Jonathan Bader, Joel Witzke, Soeren Becker, Ansgar Loser, Fabian Lehmann, Leon Doehler, Anh Duc Vu, and Odej Kao. "Towards Advanced Monitoring for Scientific Workflows". In: *2022 IEEE International Conference on Big Data (Big Data).* IEEE, 2022, pp. 2709–2715. DOI: 10.1109/bigdata55660.2022.10020864.

[Bre01] L. Breiman. "Random Forests". In: *Machine Learning* 45 (2001), pp. 5–32.

[Cas+20] Henri Casanova, Rafael Ferreira da Silva, Ryan Tanaka, Suraj Pandey, Gautam Jethwani, William Koch, Spencer Albrecht, James Oeth, and Frédéric Suter. "Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH". In: *Future Generation Computer Systems* 112 (2020), pp. 162–175. DOI: 10.1016/j.future.2020.05.030.

[Col+21] TainÃ£ Coleman, Henri Casanova, Ty Gwartney, and Rafael Ferreira da Silva. "Evaluating Energy-Aware Scheduling Algorithms for I/O-Intensive Scientific Workflows". In: *Computational Science - ICCS 2021.* Springer International Publishing, 2021, pp. 183–197. ISBN: 9783030779610. DOI: 10.1007/978-3-030-77961-0_16.

[GL17] Surya Kant Garg and J. Lakshmi. "Workload performance and interference on containers". In: *2017 IEEE SmartWorld, Ubiquitous Intelligence and Computing, Advanced and Trusted Computed, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI).* 2017, pp. 1–6. DOI: 10.1109/UIC-ATC.2017.8397647.

[HS24] Mirsaeid Hosseini Shirvani. "A survey study on task scheduling schemes for workflow executions in cloud computing environment: classification and challenges". In: *The Journal of Supercomputing* 80.7 (2024), pp. 9384–9437. ISSN: 1573-0484. DOI: 10.1007/s11227-023-05806-y.

[KP23] Animesh Kuity and Sateesh K. Peddoju. "pHPCe: a hybrid power conservation approach for containerized HPC environment". In: *Cluster Computing* 27.3 (2023), pp. 2611–2634. ISSN: 1573-7543. DOI: 10.1007/s10586-023-04105-8.

[LZ12] Young Choon Lee and Albert Y. Zomaya. "Energy efficient utilization of resources in cloud computing systems". In: *The Journal of Supercomputing* 60.2 (2012), pp. 268–280. ISSN: 1573-0484. DOI: 10.1007/s11227-010-0421-3.

[Saa+23] Youssef Saadi, Soufiane Jounaidi, Said El Kafhali, and Hicham Zougagh. "Reducing energy footprint in cloud computing: a study on the impact of clustering

techniques and scheduling algorithms for scientific workflows". In: *Computing* 105.10 (2023), pp. 2231–2261. ISSN: 1436-5057. DOI: 10.1007/s00607-023-01182-w.

[Sil+24] C.A. Silva, R. VilaÃ§a, A. Pereira, and R.J. Bessa. "A review on the decarbonization of high-performance computing centers". In: *Renewable and Sustainable Energy Reviews* 189 (2024), p. 114019. ISSN: 1364-0321. DOI: 10.1016/j.rser.2023.114019.

[Tha+25] Lauritz Thamsen, Yehia Elkhatib, Paul Harvey, Syed Waqar Nabi, Jeremy Singer, and Wim Vanderbauwhede. *Energy-Aware Workflow Execution: An Overview of Techniques for Saving Energy and Emissions in Scientific Compute Clusters*. 2025. arXiv: 2506.04062 [cs.DC].

[Var+24] Ioannis Vardas, Sascha Hunold, Philippe Swartvagher, and Jesper Larsson Träff. "Exploring Mapping Strategies for Co-allocated HPC Applications". In: *Euro-Par 2023: Parallel Processing Workshops*. Ed. by Demetris Zeinalipour, Dora Blanco Heras, George Pallis, Herodotos Herodotou, Demetris Trihinas, Daniel Balouek, Patrick Diehl, Terry Cojean, Karl Fürlinger, Maja Hanne Kirkeby, Matteo Nardelli, and Pierangelo Di Sanzo. Cham: Springer Nature Switzerland, 2024, pp. 271–276. ISBN: 978-3-031-48803-0.

[Wit+24] Joel Witzke, Ansgar LÃ¶Ãer, Vasilis Bountris, Florian Schintke, and BjÃ¶rn Scheuermann. "Low-level I/O Monitoring for Scientific Workflows". In: (2024). arXiv: 2408.00411 [astro-ph.IM].

[ZZA10] Qian Zhu, Jiedan Zhu, and Gagan Agrawal. "Power-Aware Consolidation of Scientific Workflows in Virtualized Environments". In: *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–12. DOI: 10.1109/SC.2010.43.

[Zac+21] Felippe Vieira Zacarias, Vinicius Petrucci, Rajiv Nishtala, Paul Carpenter, and Daniel Mossé. "Intelligent colocation of HPC workloads". In: *Journal of Parallel and Distributed Computing* 151 (2021), pp. 125–137. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2021.02.010.