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, 6. 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 workflow. 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.

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 80–200 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 multiprocessor 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.

2.1.1 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 system–level 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 power–performance 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 Docker's 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 Energy-Aware Computing

[Tha+25] Power consumption in computing systems can broadly be divided into static and dynamic components. Static power is drawn even when components are idle, often reaching up to half of peak power, while dynamic power depends on utilization and increases with workload. This inefficiency has motivated the concept of energy-proportional computing, in which energy use per operation scales directly with utilization. Hardware trends and techniques such as Dynamic Voltage and Frequency Scaling (DVFS) have improved proportionality, while workload consolidation further increases efficiency by concentrating tasks on fewer resources.

Beyond compute, data centers also consume energy for cooling, lighting, and auxiliary operations. This is measured by Power Usage Effectiveness (PUE), which expresses the ratio between total data center energy use and the share consumed by IT equipment. While leading hyperscale centers achieve PUE values close to 1.1, the global average remains higher, and the metric has known limitations, including sensitivity to workload characteristics and the inability to capture whole-system trade-offs.

The climate impact of energy use further depends on carbon intensity, i.e., the emissions associated with each unit of consumed electricity. This varies geographically and temporally with the mix of fossil, nuclear, and renewable generation sources. As such, carbon intensity can fluctuate substantially over hours and regions, meaning that both location and timing of computation influence emissions.

Breaking down component-level energy use, CPUs typically dominate server consumption, accounting for 40–66% of total power, with significant static and dynamic contributions. Memory is the second largest consumer, drawing a smaller but relatively stable share, while storage devices show limited dynamic variation and modest overall consumption. Networking equipment consumes comparatively little, with low dynamic variation, though attributing its energy use to specific workloads is non-trivial.

A geoscience workflow (FORCE) was executed on a commodity cluster using 21 nodes for 315 minutes to process 304 GB of satellite images. The hardware setup led to an estimated operational energy use of 15.8 kWh, factoring in average data center overhead (PUE 1.61). Given Germany's 2021 grid carbon intensity of 439 gCOe/kWh, this corresponds to about 6.95 kgCOe emissions per run, or 20.8 kgCOe for the three repetitions used to report median results—roughly equal to driving 53 miles in a gasoline car. Beyond operational emissions, the embodied carbon footprint of the hardware was considered. With an estimated 1200 kgCOe per node over its lifetime and 0.00599% of lifetime usage during the experiment, the workflow's share amounts to about 1.5 kgCOe per run, or 4.5 kgCOe for three executions. Thus, both operational and embodied emissions contribute to the total footprint of the workflow evaluation.

2.4 Monitoring Scientific Workflows

2.4.1 Performance Monitoring

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 application's 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.

2.4.2 Monitoring Layers

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 HTCondor's DAGMan meta-scheduler and Slurm's built-in dependency mechanisms [Bad+22].

2.4.3 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 Foundation's 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 neighbors" consume 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 guest's 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.

2.4.4 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's 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 Intel's 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 Intel's 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 Intel's 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.

2.5 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]

2.5.1 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].

2.5.2 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. Workflow-based (scheduling workflow-by-workflow): In these algorithms, all tasks in a workflow are simultaneously scheduled and then the schedule for each task is improved using some procedure to improve the quality of the global workflow schedule [**a survery**].

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].

2.6 Machine Learning in Scientific Workflow Computing

2.6.1 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 [**bader2025predictingperformancescientificworkflows**].

2.6.2 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. 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.

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.

2.6.3 Simulating 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

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, as described in Chapter 2. 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.

Based on this threefold design, the structure of this chapter is organized as follows. After formally 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 Problem Statement

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 workflow's 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. Here, co-location decisions are made during workflow execution, allowing the system to adapt to evolving runtime conditions.

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 within a unified framework rather than as 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.

4.2 Overview

4.2.1 Assumptions

To clearly outline the scope, boundaries, and methodological constraints of this work, the following guiding assumptions were defined to facilitate this first iteration of research on dynamic scientific workflow co-location:

1. Monitoring Configuration Limits: A maximum of 50 monitoring features per workflow task is imposed to ensure manageable execution times and allow for statistical evaluation across varying monitoring configurations. This restriction also underscores the need for future work to investigate the influence of monitoring data quality and dimensionality on predictive performance.
2. Monitoring Data Quality and Coverage: Not all low-level monitoring capabilities are fully exploited. Instead, existing monitoring data sources are leveraged with minimal modifications to improve compatibility with the monitoring client. 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.
4. Simulation Environment and Platform Equivalence: The simulated platform is assumed to approximate the physical execution environment despite potentially having more nodes of identical configuration. 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, built on SimGrid, currently supports simulation of memory contention by limiting per-VM 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 (e.g., cache, interconnect, 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 strongest impact on energy efficiency is attributed to CPU utilization behavior.
7. Evaluation Focus: Co-location efficiency is evaluated primarily through per-host energy consumption over time, total workflow energy usage, and overall makespan reduction, which serve as the main indicators of effective virtual machine co-location.

4.3 Online Task Monitoring

The online task monitoring approach developed in this work builds on a hierarchical monitoring architecture that captures the full spectrum of metrics relevant to the execution of scientific workflows. The design follows 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 focuses on monitoring its respective component while retrieving necessary metrics from the layers beneath it. This hierarchical organization ensures that monitoring is both scalable and context-aware, enabling adaptive scheduling and subsequent co-location decisions.

The approach deliberately avoids a user-centric design, instead aligning the monitoring structure with the operational hierarchy of distributed workflow execution systems. At the top, the Resource Manager layer provides coarse-grained monitoring information related to cluster status, resource allocation, and job management. 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.

In this layered structure, each upper layer can access data from its subordinate layers but depends only on a subset of their metrics to operate. For example, the Resource Manager relies on summarized data from the Workflow and Machine layers—such as node availability, workflow DAG progress, and per-task resource utilization—without needing low-level system calls or detailed task traces. The hierarchical data flow thus facilitates aggregation and abstraction of monitoring information, ensuring that higher-level components can make informed decisions without being burdened by excessive data granularity [Bad+22].

Scaphandre is a power monitoring tool designed to measure and attribute energy consumption across computing systems down to the process level. It integrates hardware-based sensors—most notably Intel's RAPL technology—with system-level resource monitoring to provide fine-grained insights into power usage. By combining energy counters from CPUs and GPUs with resource utilization metrics such as CPU load and memory usage, Scaphandre correlates hardware activity with power consumption in real time. Its operation relies on process-level tracking over short scheduling intervals (jiffies), allowing it to estimate the energy consumed by individual processes or containers. Through its exporter modules, Scaphandre exposes standardized metrics for external monitoring systems such as Prometheus. Key metrics include total host power and energy consumption (scaph$_hos t_p ower_m icrowatts, scaph_h ost_e nergy_m icrojoules), per-processpowerusage(scaph_p rocess_p ower_c on$
$awareworkloadmanagementandsustainabilityanalysis.Scaphandre'smeasurementapproachleveragesLin$
$basedreadingsandsupportsvirtualizedenvironments, enablingaccuratemonitoringacrossVMsandcontaine$
$levelenergyattribution, futuredevelopmentsaimtoextenditscapabilitiestowardGPUandstoragedevicepowe$
$wideenergyconsumption[Cen+24]$.

cAdvisor (Container Advisor) is an open-source daemon for monitoring resource usage and performance of containers (focus here on Docker). It runs with root privileges, exposes a web UI and HTTP APIs (including Prometheus metrics), and continuously discovers containers via Linux cgroups under /sys/fs/cgroup. Once started, cAdvisor initializes in-memory storage, a central Manager, and HTTP handlers, then recovers the container hierarchy and begins housekeeping. Discovery and lifecycle tracking rely on inotify: the raw watcher subscribes to create/delete events in the cgroup filesystem, converts them to internal add/remove events, and (re)configures per-container handlers. Metrics originate from multiple layers: machine-level facts (CPU model/clock, memory, disks, NICs) parsed from /proc and /sys; container/process usage (CPU, memory, I/O, network) collected at cgroup boundaries; and optional accelerator/GPU hooks. The architecture comprises a cache, event channels, container watchers, per-runtime plugins (Docker, containerd, CRI-O, systemd), and HTTP handlers for both JSON APIs (e.g., /api/v1.0/containers) and raw metric endpoints. The web UI surfaces machine/FS status, running processes, and per-container time series (CPU, memory, disk, network). Programmatically, a Go client can query container specs, stats, events, and machine info. In practice, cAdvisor provides low-overhead, per-container telemetry suitable for local inspection or integration into observability stacks (e.g., Prometheus), with discovery via cgroups, change detection via inotify, and rich system context via /proc//sys. [Tol23].

Docker Activity is an early-stage, Rust-based container telemetry tool that augments Docker stats with per-container energy estimates. It requires access to the Docker engine socket and to /sys/class/powercap to read Intel/AMD RAPL counters; thus it currently supports power reporting only on RAPL-compatible CPUs. At runtime an orchestrator (using the Bollard Docker client) lists running containers, subscribes to Docker events, and spawns a lightweight watcher per container that streams docker stats-like samples. Each sample is mapped into a Record containing container ID/name, timestamp, CPU usage (computed from deltas of total vs. system CPU time and normalized by the online CPU count), PIDs, and memory usage/limits; when the optional enrichment-powercap feature is enabled, the tool reads package energy and attributes a share to the container by scaling total CPU energy with that container's instantaneous CPU fraction. Records are passed over an async channel to a pluggable exporter (e.g., JSON/Prometheus in future), enabling integration into monitoring stacks. In short, Docker Activity provides process/container-level observability with coarse energy attribution via RAPL, suitable for exploratory energy profiling of containerized workloads under realistic permissions and without external power meters.

The intuition behind DEEP-mon's per-thread power attribution method is 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. A key refinement of this method concerns Hyper-Threading (HT), where two logical threads share the same physical core. When two threads co-run on the same core, they do not each consume as much power as if they were running on separate cores. Experimental evidence shows that a physical core running two logical threads consumes about 1.15× the power of a single-threaded core. DEEP-mon incorporates this observation through a weighting factor (HTr), which adjusts the power attribution depending on whether a thread was running alone or sharing a core. If a thread runs concurrently with another on the same core, its share of the power is scaled down accordingly and divided equally between the two. 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 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 intuition behind DEEP-mon's per-thread power attribution method is 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. A key refinement of this method concerns Hyper-Threading (HT), where two logical threads share the same physical core. When two threads co-run on the same core, they do not each consume as much power as if they were running on separate cores. Experimental evidence shows that a physical core running two logical threads consumes about 1.15× the power of a single-threaded core. DEEP-mon incorporates this observation through a weighting factor (HTr), which adjusts the power attribution depending on whether a thread was running alone or sharing a core. If a thread runs concurrently with another on the same core, its share of the power is scaled down accordingly and divided equally between the two. 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 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 [BSS18].

4.4 Data Analysis

The monitoring data collected during workflow execution was systematically stored in a structured results directory to facilitate reproducible analysis and efficient data access. Each monitoring dimension—CPU, memory, disk, network, and energy—was written into separate subdirectories under a common results path. Within these directories, data were further organized according to the monitoring source or tool, allowing a clear separation of heterogeneous data types and measurement granularities. For example, the task$_cpu_datadirectorycontainsmeasurementsobtainedfromtheeBPF-basedmonitoringcomponent, whichc$
$containercycle-levelCPUactivity.Thisdataisstoredinthesubdirectoryebpf-mon/container_weighted_cycles$

In addition to these resource-specific folders, global logs such as started$_nextflow_containers.csvanddied_nextf$

4.4.1 Task Entity Matching

The first phase of the data analysis focuses on entity matching—the systematic alignment 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, ranging from container-level resource traces to process-level identifiers and workflow-specific metadata. To enable integrated analysis, these fragmented data streams must be correlated and matched against the task entities defined by the workflow management system (in this case, Nextflow).

The matching process begins with the Nextflow trace and container lifecycle records (started$_{n}extflow_{c}ontainers.csvanddied_{n}extflow_{c}ontainers.csv$), $whichprovideinformationontaskidentifi$
$levelentitiesandlow-levelmonitoringdata.Theanalysisframeworkfirstscopesthefullresultsdirectorytois$
$seriesdata.Thesedatafilesareorganizedhierarchicallyundersource-specificdirectoriesandoftencontainm$
$levelanalysis, thepipelinesplitstheseaggregatedtime-seriesdatasetsintoper-taskCSVfilesusingsource-$
$specificidentifiers(forinstance, containernamesortaskhashes).Thesplittingprocedureensuresthatallsubs$
$taskdatasetisenrichedwithcontextualmetadata.Thefirstenrichmentphaseattachesthecorrectworkingdirec$
$levelsemanticsbymatchingtheper-taskmonitoringdatawiththeworkflow'sownmetadata.Usingtheexporte$
$levelrecordsfromthemonitoringdatatoidentifywhichmonitoredcontainercorrespondstowhichlogicaltaskin$
$taskmonitoringfilesbyappendingthecorrectNextflowprocessname.$

4.5 Statistical Embedding and Supervised Learning

4.5.1 Data Preprocessing

We transform the heterogeneous, time-stamped monitoring traces into consistent task-level feature/label matrices suitable for statistical learning and KCCA. The pipeline proceeds in four steps: scoping harmonization, per-task time-series extraction, signature construction, and label assembly alignment.

1. Sampling smoothing.
2. Per-task time-series extraction.
3. Temporal signature construction (feature selection)

   Sampling and smoothing.

   Equal-length normalization.

   Container-wise collation.
4. Label assembly alignment.

   Power label extraction.

   Runtime normalization.

   Y matrix assembly.

   Index alignment.

This preprocessing yields: (i) a standardized, fixed-length, noise-robust feature matrix X that preserves per-metric usage distributions; (ii) a label matrix Y capturing runtime and energy; and (iii) a clean, task-aligned index. Together, these artifacts support downstream supervised modeling (e.g., regression, KCCA) and unsupervised analyses (e.g., clustering) without additional wrangling.

4.5.2 Predictive Modeling

In the final stage of the analysis pipeline, the collected and preprocessed task data were prepared for multiview learning. The goal was to learn relationships between the task-specific feature representations and the corresponding performance and energy outcomes.

To enable this, the data were divided into training and testing subsets, ensuring that roughly 70

Before training, both the feature data (X) and the target data (Y)—consisting of task runtime and energy consumption—were standardized independently. Each dataset was centered to a mean of zero and scaled to unit variance. This normalization step was necessary because the raw features and target values operate on very different numerical ranges, and kernel-based methods are highly sensitive to such scale differences. Standardization ensured that all variables contributed equally to the learning process and prevented features with large absolute values from dominating the optimization.

Kernel Canonical Correlation Analysis (KCCA) Following normalization, a Kernel Canonical Correlation Analysis (KCCA) model was trained to uncover shared structures between the feature space and the target space. In essence, KCCA finds correlated projections of both datasets onto a common latent space, allowing the model to identify nonlinear relationships between resource usage patterns and energy or runtime behavior. To determine the most suitable kernel function for this dataset, several alternatives such as linear, cosine, radial basis function (RBF), Laplacian, and polynomial kernels were evaluated using cross-validation. The kernel that maximized the inter-view correlation on the training data was selected for further use. Once the KCCA model was fitted, it was extended into a predictive framework. The latent representation learned from the feature data was used to train a simple linear regression model that maps the learned feature embeddings to the corresponding runtime and energy targets. This enabled the model not only to describe correlations but also to predict task outcomes directly from unseen feature data. Finally, the predictive performance was evaluated on the test data by comparing the model's estimates against the actual runtime and energy measurements. The evaluation relied on metrics such as mean squared error (MSE) and the coefficient of determination (R^2), which together quantify how well the model explains variability in the observed data and how accurately it generalizes to new tasks. Through this process, the preprocessing and multiview modeling steps provided a systematic way to connect measured system behavior to its energy and performance implications.

Random Forest Regression To complement the multiview model, we trained two nonparametric regressors based on Random Forests—one to predict mean task power and one to predict task runtime from the same preprocessed feature matrix. As a sanity check, we established simple baselines by predicting the training-set mean of the target (once for power, once for runtime) on the test split and reporting the corresponding absolute errors. These baselines quantify the minimum improvement any learned model must exceed. For each target, we split the data into training and test partitions (70/30). We then fit a RandomForestRegressor on the training set only, using the task feature vectors as inputs and either mean power or runtime as the scalar target. Because tree ensembles are scale-invariant, no additional normalization of X was required beyond the preprocessing used to build the feature matrix; targets were kept in their physical units to preserve interpretability. Model capacity was tuned with randomized hyperparameter search under K-fold cross-validation. The search space spanned the number of trees, maximum tree depth, feature subsampling at split time, minimum samples per split/leaf, sampling fraction per tree, and splitting criterion. This procedure balances bias–variance trade-offs, encourages decorrelated trees through feature subsampling, and mitigates overfitting via depth and sample constraints. The best configuration (by cross-validated error on the training set) was then refit on the full training partition. We evaluated generalization on the held-out test set using: (i) mean absolute error (MAE) to report average deviation in natural units; (ii) mean absolute percentage error (MAPE) to provide a scale-free notion of accuracy; and (iii) coefficient of determination (R^2) to quantify explained variance. We compared these metrics directly to the mean-predictor baseline to show absolute and relative gains. For transparency, we also report the model's score (R^2) from scikit-learn, which matches our R^2 computation. When predicting power, the forest was trained against the mean per-task energy-rate labels and scored on the test split; an analogous model was trained for runtime. Hyperparameters were tuned separately for each target, as optimal depth, tree count, and sampling often differ between the (potentially noisier) energy signal and runtime. Finally, because forests offer native feature-importance measures, the trained models can be probed to identify which components of the task signatures contribute most to power vs. runtime predictions—useful later for scheduler heuristics and for validating that the learned signals align with domain expectations.

4.6 Unsupervised Learning

4.6.1 Task Clustering

We cast consolidation as a clustering problem with a twist: instead of grouping similar tasks, we deliberately group tasks that are dissimilar in their resource demands so that colocated tasks complement each other and reduce contention. To do this, we build a task–task distance that grows when two tasks exercise the same resource in the same way, and shrinks when their peak-usage patterns are complementary.

1. Peak-pattern construction. For every task and monitored workload type (CPU, memory, file I/O), 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. This emphasizes contention-relevant bursts while smoothing short noise spikes. Each series is then normalized onto a relative time axis (seconds since first sample) to make tasks of different absolute start times comparable. When two peak series must be compared, we truncate both to the shorter length (or interpolate to a common grid when needed) so that correlation is computed on aligned vectors without padding artifacts.

2. Workload-type affinity. Different resource domains interfere to different degrees (e.g., CPU vs CPU peaks are typically more contentious than CPU vs file I/O). We encode this with an empirical "affinity score" between workload types (cpu–cpu, cpu–mem, mem–io, etc.). High affinity means higher potential interference when peaks align; low affinity reflects benign coexistence.

3. 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–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.

4. Sanity filtering. Some tasks (e.g., very short-lived or purely memory-resident ones) may produce constant peak series, which makes correlation ill-defined. We detect and

drop such pathological series from distance estimation for robustness; this prevents NaNs from contaminating the matrix.

5. Threshold selection. Because the distance matrix is data-dependent, we estimate a merge threshold directly from its empirical distribution (lower triangle, excluding the diagonal). A quantile (e.g., the 40th 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. This adaptive choice avoids brittle, hand-tuned cutoffs.

6. 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.

7. From clusters to co-location candidates. Each cluster defines a candidate co-location set. To make these cluster-level entities usable by downstream predictors (e.g., KCCA or Random Forest), we construct cluster feature vectors by flattening and concatenating the per-task temporal signatures (pattern vectors for CPU/memory/file I/O) of all members and summing them dimension-wise. This simple, permutation-invariant aggregation approximates the combined load shape the scheduler would see if the cluster's tasks were run together on the same host. (In future iterations, this can be refined to weight members by peak overlap fraction and dwell time in peak regions.)

4.7 Simulation Environment

4.7.1 Design Pillars

The simulator for co-location strategies builds upon three fundamental design pillars that reflect the main optimization opportunities identified in the co-scheduling problem: resource allocation, queue ordering, and job placement. The simulator aims to reproduce these decision dimensions in a controlled environment, allowing systematic evaluation of co-scheduling strategies under varying workload and system conditions.

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, but the co-scheduling paradigm assumes that multiple applications can coexist efficiently if they do not saturate the same resources simultaneously. The simulator therefore models node-sharing policies where jobs may share cores, memory bandwidth, or caches depending on their resource profiles.

The second pillar addresses queue ordering and dispatching, which influence throughput and fairness. Since the effectiveness of co-scheduling depends on the characteristics and arrival times of jobs, the simulator explores alternative queue management strategies—ranging from conventional FIFO ordering to heuristic reordering that prioritizes beneficial pairings. This enables analysis of trade-offs between system-level metrics, such as makespan and utilization, and user-oriented metrics, such as slowdown or turnaround time. The third pillar focuses on job placement within available resources, emphasizing how specific pairings or groupings affect performance. The simulator integrates the concept of pair matching, as identified in the paper, where dissimilar workloads are co-located to minimize contention and maximize utilization. It supports both random pairing and heuristic approaches that aim to exploit workload complementarity. Following the rationale of the original work, the simulator evaluates co-scheduling effectiveness through key performance metrics: makespan speedup, weighted mean job speedup (WMJS), utilization, and mean slowdown. It thereby enables the quantification of trade-offs between performance improvement and fairness. Moreover, the simulator design incorporates the statistical modeling perspective proposed in the paper, allowing the examination of correlations between these metrics to identify conditions where co-scheduling is advantageous. Finally, the simulator supports experimentation with opportunistic and greedy scheduling strategies, reflecting the pragmatic orientation of the co-scheduler research. Exhaustive optimization is computationally infeasible; hence, the simulator adopts scalable heuristics for dynamic pairing, bulk scheduling, and queue reordering. This design allows evaluation of how localized scheduling decisions affect global system performance and resource efficiency, ultimately enabling controlled exploration of the principles behind practical HPC co-scheduling [Tri+25].

4.7.2 Heuristic Design

The simulator follows a heuristic-based design philosophy. Instead of formulating the scheduling process as an optimization problem defined by a cost function or a multi-objective fitness model, the approach aims to achieve energy awareness through embedded 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 and structured criteria—such as task priorities, resource affinities, or workload complementarities—to produce acceptable solutions efficiently without guaranteeing global optimality.

In this context, the heuristic scheduler is designed to incorporate energy-related considerations directly into basic scheduling steps such as task ordering and resource selection. Rather than seeking a mathematically optimal solution, it focuses on guiding the scheduling process toward energy-efficient outcomes through lightweight decision rules. These rules are derived from observed workload behaviors and co-location characteristics, ensuring that the resulting mappings balance computational load, minimize interference, and reduce redundant energy usage. The heuristic operates deterministically: once a feasible schedule is reached, the process terminates. This design choice provides a practical balance between computational efficiency, interpretability, and responsiveness—qualities that are essential for evaluating co-location strategies under realistic conditions without the complexity of meta-heuristic or multi-objective optimization frameworks [HS24]. Machine learning–based and list-scheduling approaches represent two major paradigms in workflow scheduling. Machine learning–driven scheduling builds on data-driven models that learn decision policies from historical workflow executions or runtime observations. Methods such as deep learning, Q-learning, and reinforcement learning are primarily used to model task execution patterns, predict failures, and adapt scheduling decisions dynamically. For example, the Deep Q-learning Heterogeneous Earliest Finish Time (DQ-HEFT) algorithm integrates reinforcement learning into the classical HEFT framework, using a neural network to model interactions with the cloud environment and to optimize task–processor mappings under cost, time, and budget constraints. Other studies employ supervised models such as decision trees to predict task or job failures based on workload traces from large systems like Google or Alibaba, allowing proactive rescheduling. More advanced models, such as the HunterPlus framework, combine recurrent neural networks (GRUs) with workflow graph inputs to optimize energy efficiency and minimize SLA violations. Generally, machine learning–based scheduling enables adaptive and predictive behavior by continuously refining mappings and execution orders based on performance feedback. Beyond per-task list-scheduling, we also support cluster-scheduling, which first forms task clusters (to keep heavy communicators together), then merges clusters onto a limited processor set, and finally orders tasks within each cluster. For clustering, we include representatives spanning the design space: linear/critical-path–driven approaches (e.g., LC), dominant-sequence strategies that greedily place a task with its constraining parent and optionally relocate other parents when beneficial (DSC), and dynamic-critical-path variants (DCP) that update allocated levels iteratively and allow “squeezing” a critical task into existing timelines when this does not increase the schedule length. For cluster merging, we provide (a) simple load-balancing (assign largest clusters first to the least-loaded processor), (b) guided load-balancing that merges in order of clusters' earliest start times to reduce temporal interference, and (c) a list-scheduling adaptation that assigns entire clusters by evaluating the resulting schedule length per processor and picking the best. Finally, intra-cluster task ordering can follow allocated bottom-level (criticality-aware) or a ready-critical-path/ETF-like strategy to suppress idle gaps. Cluster-scheduling represents a hierarchical approach to task scheduling that first groups interdependent or communication-intensive tasks into clusters before assigning them to processors. It operates in three major phases: clustering, cluster merging, and intra-cluster task ordering. In the clustering phase, algorithms such as Linear Clustering (LC), Dominant Sequence Clustering (DSC), and Dynamic Critical Path (DCP) progressively group tasks based on their critical paths and communication dependencies. LC iteratively extracts the longest remaining path in the task graph, while DSC and DCP extend this idea by incorporating additional heuristics that relocate tasks or dynamically update critical paths to minimize communication delays and start times. In the cluster-merging phase, the initially unlimited clusters are mapped onto a limited set of processors. The simplest merging method applies load balancing, where clusters are sorted by computational weight and assigned to the least loaded processor. More advanced methods, such as Guided Load Balancing (GLB), consider the temporal alignment of processor loads, merging clusters based on their earliest start times to better reflect real execution overlap. Alternatively, a list-scheduling adaptation can be used, where each cluster is assigned according to the minimal increase in total schedule length estimated from current processor allocations. Finally, in the task-ordering phase, tasks within clusters are sequenced according to established priority rules from list-scheduling. These can include static metrics such as bottom-level order or dynamic rules such as the Ready Critical Path method, which prioritizes tasks that can start earliest while minimizing idle time. Together, these stages form an integrated scheduling pipeline that balances communication efficiency, parallelism, and processor utilization by combining clustering heuristics with list-based ordering principles. In contrast, list-scheduling algorithms form one of the most established heuristic approaches for workflow scheduling. These algorithms operate in two stages: first, 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 that minimizes a defined objective function such as earliest finish time. The classical Heterogeneous Earliest Finish Time (HEFT) algorithm exemplifies this approach by ordering tasks based on bottom-level ranking and assigning them to processors that minimize completion time. Numerous variants extend this principle, incorporating multi-objective functions for reliability, energy, or cost optimization. For example, energy-aware list schedulers integrate dynamic voltage and frequency scaling (DVFS) to reduce power consumption, while reliability-aware schedulers replicate critical tasks across multiple processors to enhance fault tolerance. Hybrid methods also exist, such as HH-LiSch or HEFT-TD, which combine ranking heuristics with duplication or insertion strategies to reduce makespan and improve resource utilization [**8301529**].

Baseline 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. For each assigned task, the allocator component handles all aspects of resource creation and management. It first determines the file locations of the task's input and output data, ensuring correct data placement and accessibility. Then, it instantiates a virtual machine on the selected host, configured according to the task's resource requirements. The task is packaged as a job, submitted to the virtual machine for execution, and monitored until completion. After execution, the allocator triggers the controlled shutdown and destruction of the virtual machine, releasing the allocated resources before moving to the next task. This composition—FIFO scheduling, round-robin host assignment, and VM-based task allocation—defines a static and reproducible baseline. It excludes adaptive heuristics, predictive optimization, and parallelism, emphasizing clear causality and isolation. The resulting execution model establishes a neutral reference for measuring the impact of more advanced heuristic or co-location-aware scheduling strategies developed later in the work. This 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 (i.e., idle$_c$*ores* >
0), *withoutrequiringthehosttobecompletelyidle.Theallocatorthenprovisionsa*1−*vCPUVMonthechosenhos*
locatedonthesamehostuptoitscorecapacity, increasinginstantaneousparallelismandutilization.Thepolicyis
itignoresNUMA/bandwidthconsiderations, doesnotrotatethestartinghost(*thusbehavinglikegreedyfirst*−
fitratherthanload−*balancedround*−*robin*), *andreliesonthecomputeservicetoenforcecapacity.Asaresult, B*
*locationunderFIFOarrivalwhilekeepingresourceprovisioningandI/OhandlingidenticaltoBaseline*1.*This*
location˘awarecomponents.TasksarestilldequeuedinstrictarrivalorderbytheFIFOscheduler.Whenthenod
coreavailability, itagainselectsthefirsthostwithavailablecores.However, insteadoflaunchingoneVMperta
corecapacityaregroupedintoasinglebatch.Thesetasksarethenco−*locatedinsideonesharedVMinstancethati*
VMco−*location, wheremultipleindependenttasksssharethesamevirtualmachineinsteadofbeingdistributed*
fithostselectionbutaltersthe allocationgranularityfromıoneVMpertaskȷtoıoneVMperbatch.ȷTheresultisa
locationmodelthatincreasesper−*hostutilizationwhilemaintainingdeterministicschedulingorderandconsis*
location˘awarecomponents.TasksarestilldequeuedinstrictarrivalorderbytheFIFOscheduler.Whenthenod
coreavailability, itagainselectsthefirsthostwithavailablecores.However, insteadoflaunchingoneVMperta

corecapacityaregroupedintoasinglebatch.Thesetasksarethenco−locatedinsideonesharedVMinstancethati
VMco−location, wheremultipleindependenttasksssharethesamevirtualmachineinsteadofbeingdistributed
fithostselectionbutaltersthe allocationgranularityfromıoneVMpertaskȷtoıoneVMperbatch.ȷTheresultisa
locationmodelthatincreasesper−hostutilizationwhilemaintainingdeterministicschedulingorderandconsis

This variant keeps the FIFO scheduler (tasks are dequeued strictly in arrival order) and the co-location allocator (multiple tasks share a single VM), but replaces the placement policy with a max-idle host selector. 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 (up to its idle-core count), preserving FIFO order within the batch. The co-location allocator provisions a single VM on that host sized to the batch's aggregate demand (vCPUs = sum of task core requirements; memory = sum of task memory requirements), starts it, and submits each task as an independent job to the same VM. The VM remains active until all tasks mapped to it have completed, after which it is torn down. Conceptually, this policy implements best-fit by capacity at the host level (always fill the roomiest host first) combined with intra-VM co-location for the selected batch. 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.

This variant retains the same FIFO scheduler but introduces a node assignment and allocation policy focused on maximizing parallel host utilization. Tasks are still dequeued strictly in arrival order by the FIFO scheduler. 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 host's 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 VM's 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. This ensures that resource lifetime is tied to the collective completion of all tasks sharing the same virtual machine. Conceptually, this variant extends the intra-VM co-location principle by scaling it across multiple hosts in parallel. It preserves FIFO task ordering but replaces simple first-fit placement with a capacity-ranked packing policy that fills the most capable hosts first. The result is a maximally parallel co-location strategy that maintains deterministic scheduling behavior while improving system-wide resource utilization through host-level batching and VM sharing.

This variant keeps the FIFO scheduler and the capacity-ranked batching strategy, but adds deliberate oversubscription during co-location. Tasks are still dequeued strictly in arrival order. 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 (e.g., 25%): the batch limit is set to idle$_{c}$*ores*(1+*oversub*$_{f}$*actor*)*.Theprocedurecontinuesdowntherank*
locatedtaskscomplete, thenitisshutdownanddestroyed.Crucially, thedegreeofcontention¯andthusrealized
locatedtaskprofiles.WhenCPU−, memory−, andI/O−intensivephasesoverlapunfavorably(e.g., severalC
boundtasks), oversubscriptionamplifiesinterferenceandqueueingonscarcevCPUs.Whenprofilesarecomp

boundwithI/O−boundormemory−latency˘dominatedtasks), thesameoversubscriptionadmitsmoreusefulc
hostthroughput.Conceptually, thisvariantimplementsparallel, capacity−rankedco−locationwithcontrolle

Co-location strategies 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 host's 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 adviser's 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.

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 host's free cores by a fixed factor (e.g., +25%). If tasks remain, it proceeds to the next host and repeats. The allocator implements the adviser's 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 cluster's requests—even if that exceeds the host's currently free cores (hard oversub). For singleton clusters collected on the same host, it provisions a shared VM and caps vCPUs at the host's 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.

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 host's idle-core capacity worth of ready tasks and queries ShaRiff for a co-location plan over this batch. The allocator realizes ShaRiff's 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 cluster's 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 ShaRiff's clustering to raise utilization and efficiency through informed co-location.

This scheduler extension augments the ShaRiff-based variants with a Min–Min 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 ShaRiff's 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 ShaRiff's 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

5.3 Extensibility through Decoupled Design

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 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 client's 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.

5.3.2 Statistical Modeling

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.

5.3.3 Simulator Setup

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 Discussion

7 Conclusion and Future Work

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. "DEEP-Mon: 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, 2709â2715. 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.

[Cen+24] Carlo Centofanti, José Santos, Venkateswarlu Gudepu, and Koteswararao Kondepu. "Impact of power consumption in containerized clouds: A comprehensive analysis of open-source power measurement tools". In: *Computer Networks* 245 (2024), p. 110371. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2024.110371.

[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 â ICCS 2021.* Springer International Publishing, 2021, 183â197. 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 Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud 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), 2611â2634. 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), 2231â2261. 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].

[Tol23] Nanik Tolaram. “cadvisor”. In: *Software Development with Go: Cloud-Native Programming using Golang with Linux and Docker*. Berkeley, CA: Apress, 2023, pp. 347–376. ISBN: 978-1-4842-8731-6. DOI: 10.1007/978-1-4842-8731-6_18.

[Tri+25] Nikolaos Triantafyllis, Athanasios Tsoukleidis-Karydakis, Efstratios Karapanagiotis, Myrsini Kellari, Georgios Goumas, and Nectarios Koziris. *Outlining an HPC Co-Scheduler*. June 2025. DOI: 10.5281/zenodo.15543144.

[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.