



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, 27. Oktober 2025

*https://www.static.tu.berlin/fileadmin/www/10000060/FSC/Promotion__Habilitation/Dokumente/Grundsatzgute_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. Softwareentwickler 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 Plattform 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 [thamsen2025energyawareworkflowexecutionoverview].

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 [Coleman_2021].

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 [Silva_2024].

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 [Kuity_2023].

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

flow. By combining coarse-grained and fine-grained tasks into clustered jobs, resource utilization becomes more balanced, and the performance of workflow execution can be significantly enhanced [Saadi_2023].

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 [5644899].

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

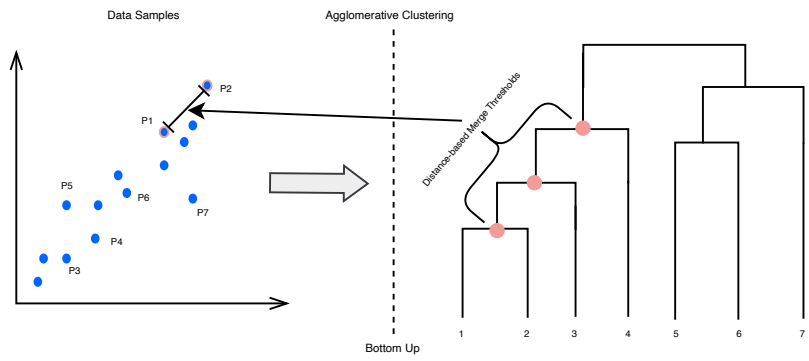


Figure 2.1: Clustering
Das ist eine Beschreibung der Abbildung.

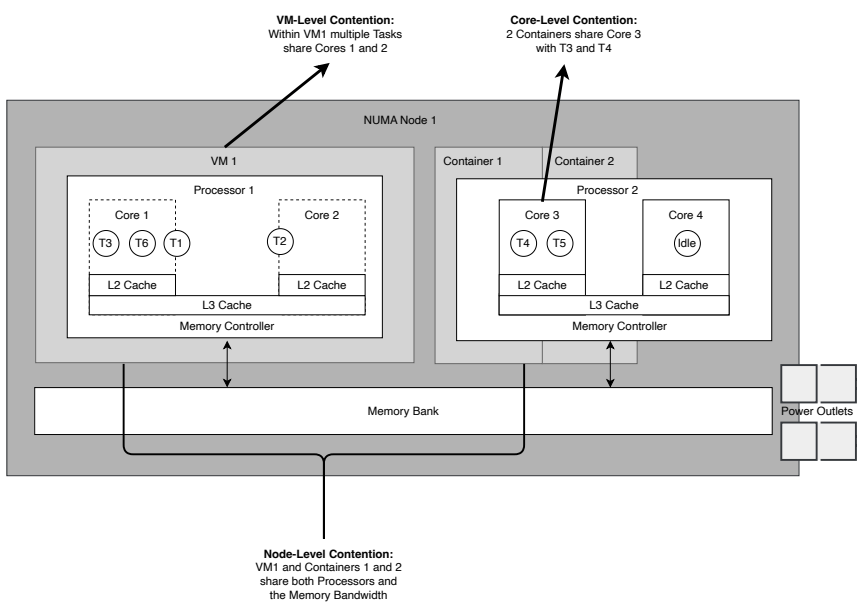


Figure 2.2: ClusteColoc Overview
Das ist eine Beschreibung der Abbildung.

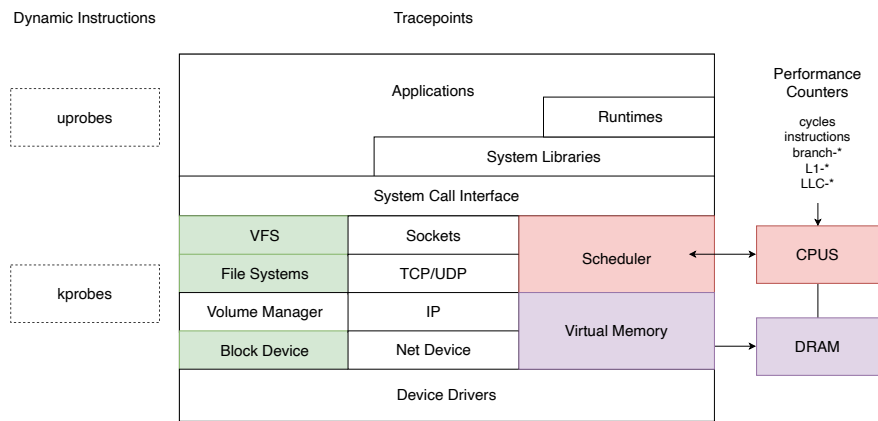


Figure 2.3: EBPF OS

Das ist eine Beschreibung der Abbildung.

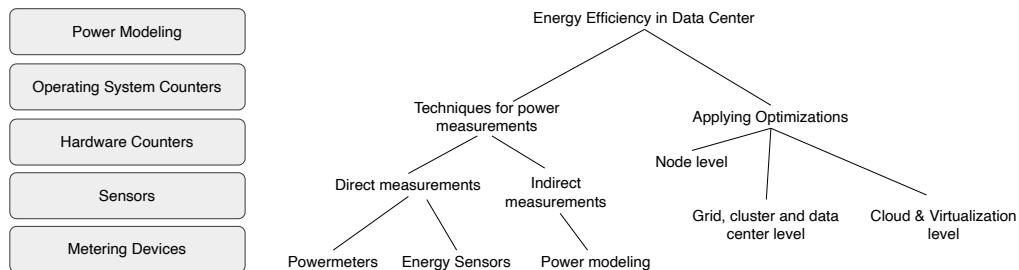


Figure 2.4: Energy Overview

Das ist eine Beschreibung der Abbildung.

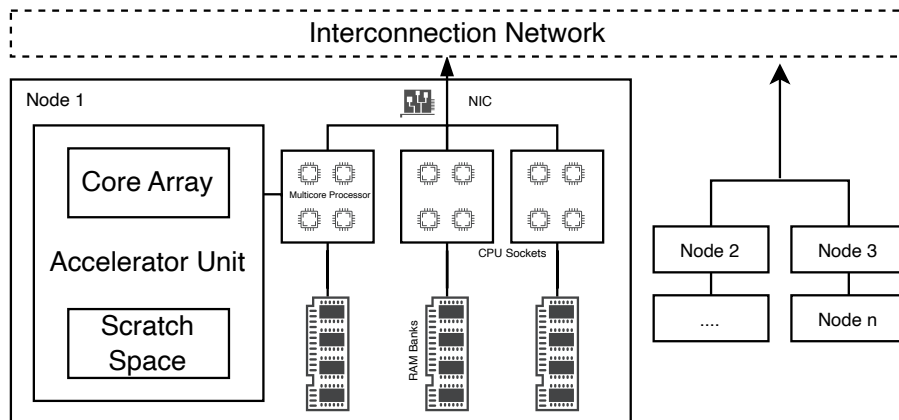


Figure 2.5: HPC Nodes

Das ist eine Beschreibung der Abbildung.

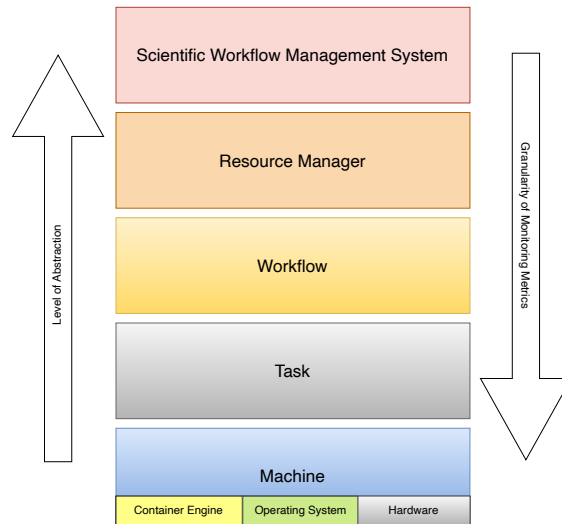


Figure 2.6: Monitoring Layers
Das ist eine Beschreibung der Abbildung.

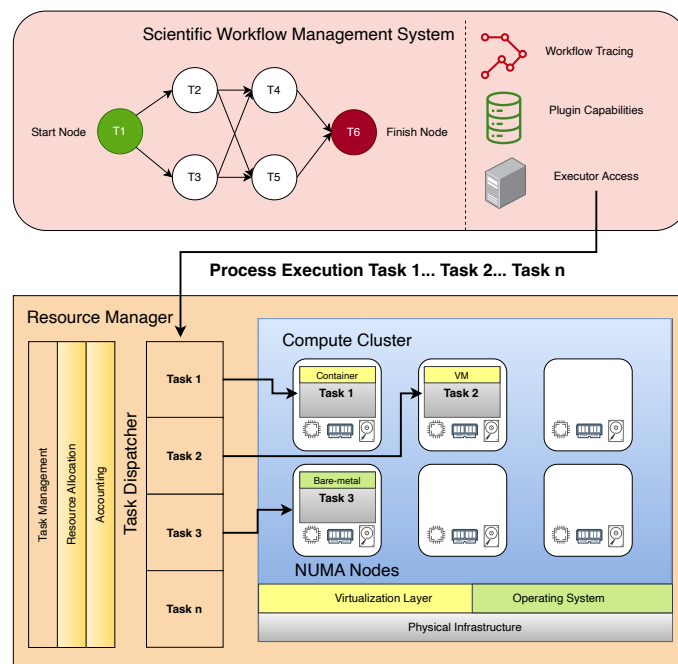


Figure 2.7: Monitoring Layers 2
Das ist eine Beschreibung der Abbildung.

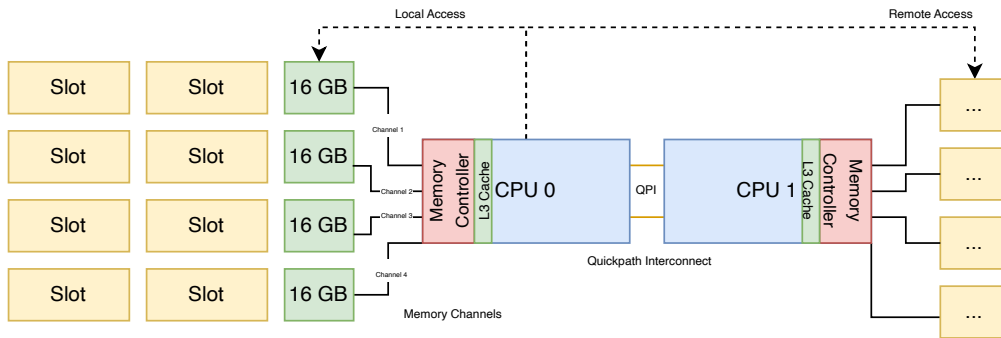


Figure 2.8: Numa Cores
Das ist eine Beschreibung der Abbildung.

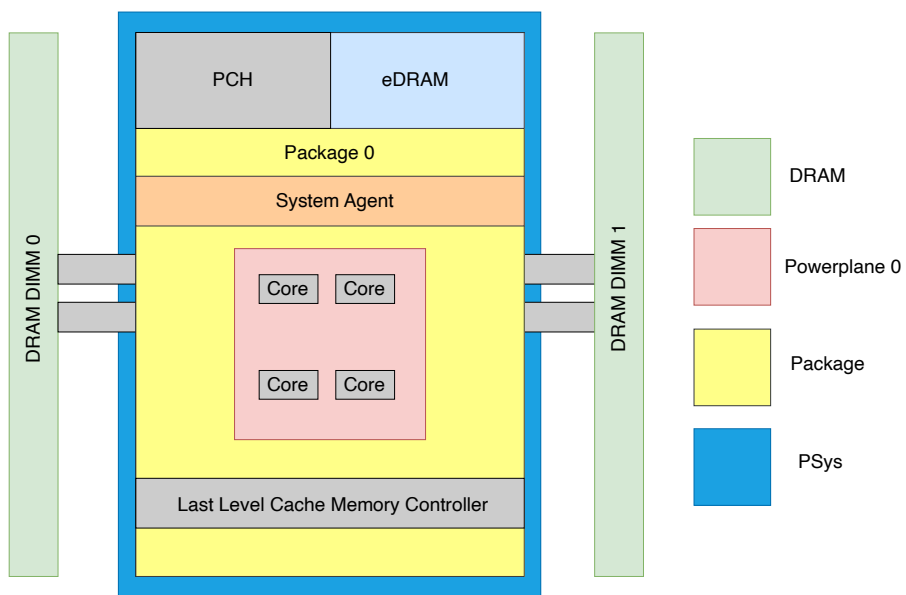


Figure 2.9: Rapl
Das ist eine Beschreibung der Abbildung.

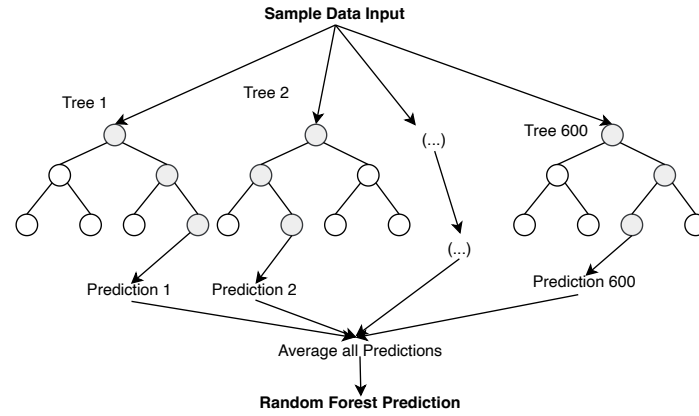


Figure 2.10: RF

Das ist eine Beschreibung der Abbildung.

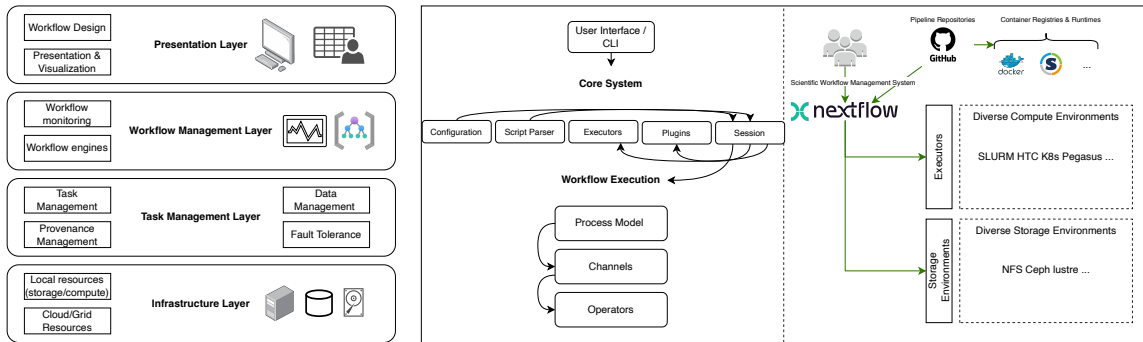


Figure 2.11: SWMS

Das ist eine Beschreibung der Abbildung.

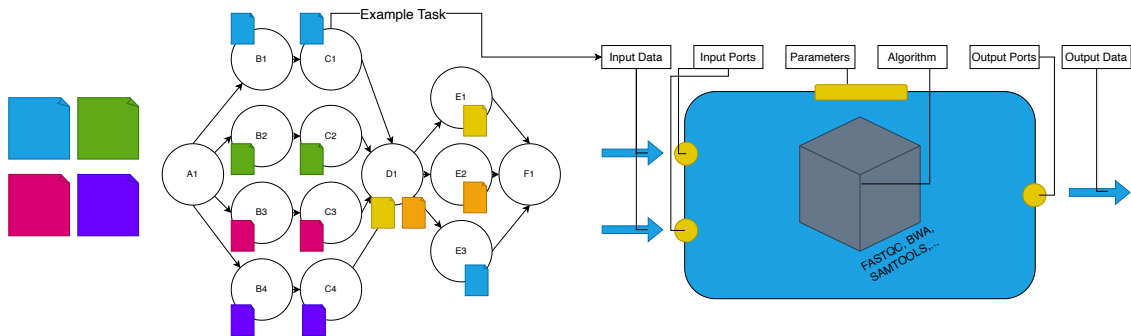


Figure 2.12: Workflow Task

Das ist eine Beschreibung der Abbildung.

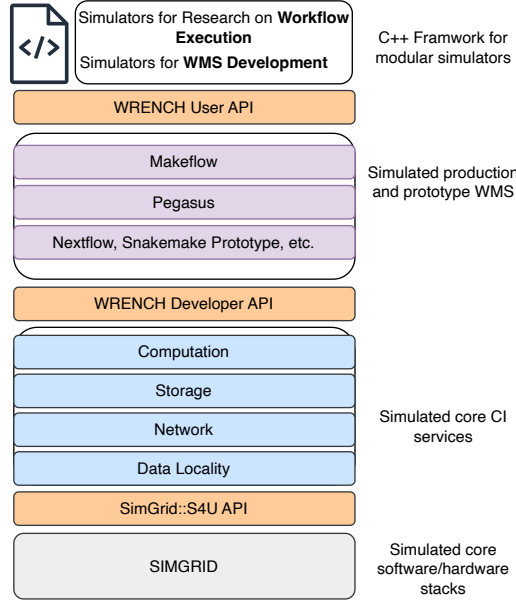


Figure 2.13: Wrench
Das ist eine Beschreibung der Abbildung.

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 synony-

mous with HPC. The fundamental purpose of HPC is to address questions that cannot be adequately solved through theory, empiricism, or conventional commercial computing systems. The scope of problems tackled by supercomputers extends beyond traditional scientific and engineering applications to include challenges in socioeconomics, the natural sciences, large-scale data management, and machine learning. An HPC application refers both to the problem being solved and to the body of code, or ordered instructions, that define its computational solution.

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

2.1.1 Modern HPC Hardware

High-performance computing (HPC) architecture defines how supercomputers are structured, how their components interact, and which instruction set architecture (ISA) they expose to the applications running on them. It is designed to use underlying hardware technologies efficiently to minimize time to solution, maximize computational throughput, and support large-scale, computation-intensive workloads. Increasingly, HPC systems are also used for data-intensive tasks such as big data analytics and graph processing. In both domains, the architecture is built to mitigate major performance bottlenecks such as latency, overhead, contention, and idle resources, while ensuring reliability and energy efficiency within the required performance and data constraints.

The performance of an HPC system depends largely on the speed and coordination of its components, with processor clock frequency playing a central role. A major challenge stems from the disparity between the rapid operation of processor cores and the comparatively slow access times of main memory (DRAM). To address this imbalance, modern architectures employ hierarchical memory systems that combine large DRAM capacities with faster SRAM cache layers. System performance is further influenced by communication efficiency, determined by bandwidth and latency, which vary across technologies and physical distances. Overall, HPC architectures aim to balance computation, memory, and communication performance while maintaining reasonable cost, power consumption, and ease of use to achieve high application throughput.

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. 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. 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.

2.1.2 Virtualization in HPC

With the growing demand for 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. 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.

Containers 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 [8397647]. In HPC the 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 through resource managers like Slurm. Container solutions support multiple container image formats, including compatibility with Docker images, and efficient integration 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 [thamsen2025energyawareworkflowexecutionoverview].

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 [Bader_2022].

2.2.2 Examples of Scientific Workflows

Scientific workflows are compositions of sequential and concurrent data processing tasks, whose order is determined by data interdependencies. 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. Scientific workflows exist at different levels of abstraction: abstract, concrete, and physical. An abstract workflow models data flow as a concatenation of conceptual processing steps. Assigning actual methods to abstract tasks results in a concrete workflow. 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 task mapping and results in a physical and executable workflow [Bux2013].

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. SWMS’s 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 like Ceph, HDFS, or NFS, where intermediate data is written and read between tasks [thamsen2025energyawareworkflowexecutionoverview].

2.3 Monitoring of Scientific Workflows

Performance monitoring is a critical stage in application development, extending beyond functional correctness and validation of results. Even after thorough testing with diverse datasets and computational modes, hidden inefficiencies may remain that limit the applications ability to fully exploit the underlying hardware. 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. Hardware-based approaches use modern

CPU to 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.

2.3.1 Distinct Monitoring Layers

High-level Monitoring While general HPC performance monitoring 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 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. 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. To deal with this lack of information, resource usage profiles from compute nodes must be correlated with metadata such as workflow logs or job orchestrator information [Witzke2024].

To bridge this gap, proposed an architectural blueprint for workflow monitoring organized 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

Low-level Resource Monitoring Monitoring scientific workflows requires tools that can capture and relate information across different components of a computing system, since no single perspective provides sufficient detail for effective optimization. Low-level system monitors can reveal CPU, memory, or I/O usage but lack awareness of which workflow tasks generate that load, while workflow management systems expose task-level metrics that are often too coarse to identify inefficiencies. In the following, we briefly discuss tools and approaches for monitoring the low layers.

Central among low-level techniques 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. 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.

CPU, memory, and disk are central components for understanding system behavior. Monitoring the CPU usage fundamentally relies on understanding how time is distributed across the 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. 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). Following CPU and memory, file system monitoring focuses 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 such as reads, writes, opens/closes and latency distributions for I/O, 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 [cite].

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. 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. 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. 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.

2.3.2 Monitoring Energy Consumption

Accurate and fine-grained energy monitoring is essential for improving the efficiency of HPC and data center systems. Traditional measurement methods, such as external power meters or chassis-level sensors, provide reliable but coarse readings at the node or rack level. These approaches are often expensive, intrusive, and insufficient for attributing power consumption to individual components or workloads. To achieve higher accuracy and temporal resolution, modern systems increasingly rely on hardware-assisted mechanisms and integrated interfaces that directly expose energy metrics from within the processor.

The most widely used of these interfaces is Intel’s Running Average Power Limit (RAPL), introduced with the Sandy Bridge architecture. RAPL provides direct access to cumulative energy consumption for various hardware domains, including the CPU package, cores, integrated GPU, and DRAM. In newer architectures such as Skylake, an additional PSys domain represents the overall system-on-chip energy use. Each domain reports energy data through model-specific registers (MSRs) that can be read at millisecond-level resolution. These measurements can be accessed using the Linux MSR driver, the sysfs interface, performance monitoring tools like perf, or libraries such as PAPI. RAPL’s low overhead and high sampling rate have made it a standard component of modern HPC monitoring frameworks, supporting tasks from energy profiling and performance analysis to power-aware scheduling. While originally an Intel-specific technology, RAPL functionality is also available on AMD Zen architectures, although with limited support for energy domains compared to Intel implementations. As containerization becomes standard in HPC and

cloud infrastructures, energy attribution must extend from node-level to container-level granularity. Workflows composed of numerous short-lived and heterogeneous tasks require monitoring that captures both process-level activity and resource-specific energy usage. Integrating RAPL data with container-level metrics bridges this gap, enabling per-task and per-workflow energy accounting. Tools such as DEEP-mon build on RAPL by combining kernel-level event tracing with container-level aggregation. They can attribute energy consumption to threads, processes, or containers in real time while maintaining low system overhead. Such tools extend traditional monitoring from static node metrics to dynamic, workload-aware measurements suitable for modern, containerized HPC environments. This integration of hardware-assisted measurement and software-level attribution forms the basis for energy-aware orchestration and supports the development of sustainable, performance-efficient scientific computing systems.

2.4 The Co-location Problem

The problem of 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 co-located workloads. Early work in operating systems introduced co-scheduling 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 co-located 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. 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 [5644899]

2.4.1 Impacts of Co-location on 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. Modern multi-core architec-

tures specifically 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. 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 [inproceedings] [10.1007/978-3-031-48803-0_31].

2.4.2 Shared aspects and distinctions from Scheduling & Task Mapping

2.4.3 Shared Aspects and Distinctions from Scheduling and Task Mapping

Co-location differs fundamentally from scheduling and task mapping, even though all three aim to improve performance and energy efficiency in workflow execution. Scheduling determines when each task runs, while mapping decides where each task is placed. In contrast, co-location focuses on how tasks interact when they are executed simultaneously on shared resources. Scheduling and mapping operate primarily at the planning level—deciding task order and resource assignment to optimize overall throughput, energy use, or completion time. Co-location, however, concerns the execution-level effects that arise once multiple tasks occupy the same physical resources. It addresses the contention for shared hardware components such as caches, memory bandwidth, interconnects, and I/O subsystems. While scheduling seeks to allocate time and space efficiently, co-location must manage interference that emerges during concurrent execution. Poor co-location decisions can negate the benefits of an otherwise optimal schedule by causing resource contention, latency, or performance degradation. Therefore, co-location is not an extension of scheduling but a complementary consideration that governs how shared resource usage affects the actual efficiency and energy behavior of scheduled tasks.

2.5 Machine Learning Techniques applied in HPC

2.5.1 Intelligent Resource Management

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

2.5.2 Utilized Statistical Models in this Thesis

Ordinary Least Squares and Extensions

2.5.3 1.3. Kernel Ridge Regression

Kernel Ridge Regression (KRR) builds on ordinary least squares (OLS) regression. OLS learns a linear relationship between input variables and a target variable by minimizing the squared difference between predictions and actual values. Ridge regression extends OLS by adding a penalty to the model coefficients, which controls overfitting and improves stability when features are correlated or data are noisy. KRR further extends ridge regression by applying the kernel trick. Instead of fitting a linear model directly in the input space, KRR implicitly maps the data into a high-dimensional feature space defined by a kernel function, such as a Gaussian or polynomial kernel. A linear model is then learned in this new space, which corresponds to a nonlinear function in the original space when a nonlinear kernel is used. Overall, Kernel Ridge Regression combines the simplicity and stability of ridge regression with the flexibility of kernel-based learning to model nonlinear relationships effectively.

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 [1202783] [5644899].

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 [Breiman2001RandomF].

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 Research-oriented Simulation of Distributed Computing

Scientific workflows have become essential in modern research across many scientific domains, supported by SWMSs 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 [wrench].

3 Related Work

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

Bader et al. present a conceptual framework for advanced monitoring in scientific workflow environments, emphasizing the heterogeneity of metrics and abstraction layers involved in large-scale workflow execution. Their work identifies four distinct monitoring layers—spanning from infrastructure-level telemetry to workflow-level behavior—and positions these as an architectural blueprint for integrating and correlating performance observations across distributed systems. The authors analyze several state-of-the-art workflow management systems to assess their monitoring capabilities and to highlight existing fragmentation in metric collection and interpretation. Their approach is primarily descriptive and infrastructural, focusing on how to organize and harmonize monitoring responsibilities across system components. In contrast, the work presented in this thesis goes beyond the architectural structuring of monitoring layers and instead operationalizes monitoring data for performance modeling and scheduling decisions. While Bader et al. focus on multi-layer metric aggregation, the present work uses collected measurements—such as runtime and power consumption under co-location—to infer quantitative models of interference and task affinity. Thus, where their contribution establishes the foundation for integrated observability in workflow systems, this thesis builds upon that idea by applying the monitored data directly to predictive, learning-driven scheduling mechanisms that adapt resource allocation based on empirically derived contention profiles [Bader_2022].

Witzke et al. address the persistent challenge of connecting low-level monitoring data with high-level workflow semantics in distributed scientific computing environments. Their work focuses on tracing I/O behavior across heterogeneous compute nodes, where workflow tasks may execute concurrently and share resources. By integrating system-level telemetry with metadata extracted from workflow logs and container orchestration frameworks such as Kubernetes, the authors propose a methodology to attribute observed resource consumption—particularly I/O activity—to specific workflow tasks. This enables a task-level analysis of performance bottlenecks and inefficiencies, offering valuable insights for optimization. In contrast to their work, which centers on establishing a traceability chain between metrics and logical workflow components, the approach presented in this thesis operates at a higher abstraction level by using modeled and predicted resource interactions to inform scheduling decisions dynamically. Rather than correlating observed traces post-execution, it leverages learned contention patterns and predictive models to influence online task placement, thus turning monitoring insights into proactive scheduling intelligence [Witzke2024].

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

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

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

Task co-location has become a central strategy for improving resource utilization and energy efficiency in large-scale distributed systems. Early work by Zhu et al. introduced pSciMapper, a consolidation framework designed for scientific workflows in virtualized environments. Their approach treated consolidation as a hierarchical clustering problem, using kernel canonical correlation analysis to model interference between CPU, memory, disk, and network resource profiles. By correlating resource requirements with task runtime and power consumption, pSciMapper achieved significant power savings with minimal slowdown. However, it operated primarily in an offline mode—optimizing workflow place-

ment decisions before execution—and focused on static virtualized environments rather than dynamic HPC workflows.

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

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

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

In the HPC domain, co-location has traditionally been explored through hardware-level modeling and scheduling heuristics. Zacarias et al. present an intelligent resource manager that predicts performance degradation between co-located HPC workloads using performance monitoring counters (PMCs) as model features. The scheduler then selects application mixes that minimize degradation while improving node utilization. This approach achieves measurable improvements over conventional batch scheduling, but it remains focused on pairwise job co-location at the job-manager level and lacks integration with workflow-level task orchestration. Complementary to this, Álvarez et al. introduce nOS-V, a system-wide scheduler enabling fine-grained co-execution of HPC applications at the task level. By dynamically managing shared node resources and bypassing traditional over-subscription mechanisms, nOS-V demonstrates notable gains in throughput and resource efficiency, though it does not incorporate predictive or learning-based adaptation.

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

at the level of system services or job managers rather than workflow-specific task scheduling.

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

Related Work: Energy Awareness

Energy measurement at fine granularity has been a recurring challenge across HPC and cloud environments. Raffin and Trystram dissect software-based CPU energy metering with a critical analysis of RAPL mechanisms and an exploration of eBPF as a low-overhead path for accurate, resilient measurements. Their Rust implementation emphasizes pitfalls in timing and counter handling, providing a blueprint for building correct energy profilers on modern x86 platforms. Complementing measurement, Arjona Aroca et al. conduct a component-level characterization of server energy, showing super-linear CPU power behavior and offering validated models that keep estimation error below 5% for real analytics workloads, while also identifying efficient operating points for NICs and disks. Warade et al. bring these concerns to containers, empirically breaking down the energy footprint of common Dockerized workloads to motivate energy-aware container development and deployment practices.

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

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

Energy awareness often intersects with interference management and QoS. Blagodurov and Fedorova advocate contention-aware HPC scheduling, arguing for cluster schedulers that reason about shared resource bottlenecks (e.g., caches, memory controllers) to avoid pathological slowdowns that also waste energy. Sampaio and Barbosa explicitly tie consolidation, slowdown, and power via a slowdown estimator feeding an interference- and power-aware scheduler; simulations based on cloud traces suggest SLA compliance with 12% cost reductions. Finally, several lines of work underscore that accurate performance/energy prediction is a precondition for effective energy-aware orchestration: from measurement robustness (Raffin and Trystram) and component characterization (Arjona Aroca et al.), to container-level prediction (Algarni et al., Warade et al.), to model-informed workflow scheduling (Silva et al., Coleman et al., Durillo et al.). Together, these studies advance the state of practice in measuring, modeling, and optimizing energy across the stack—from microarchitectural counters and container runtimes up to workflow-level schedulers and consolidation controllers.

4 Approach

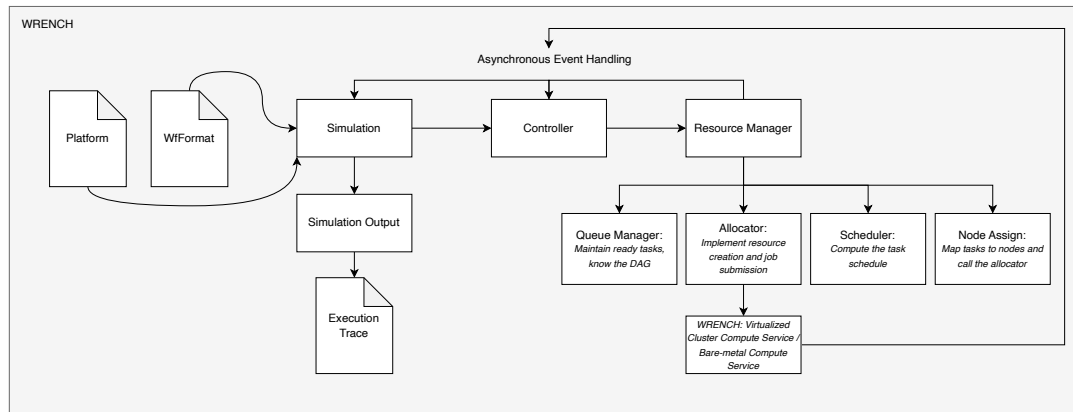


Figure 4.14: Approach Simulation
Das ist eine Beschreibung der Abbildung.

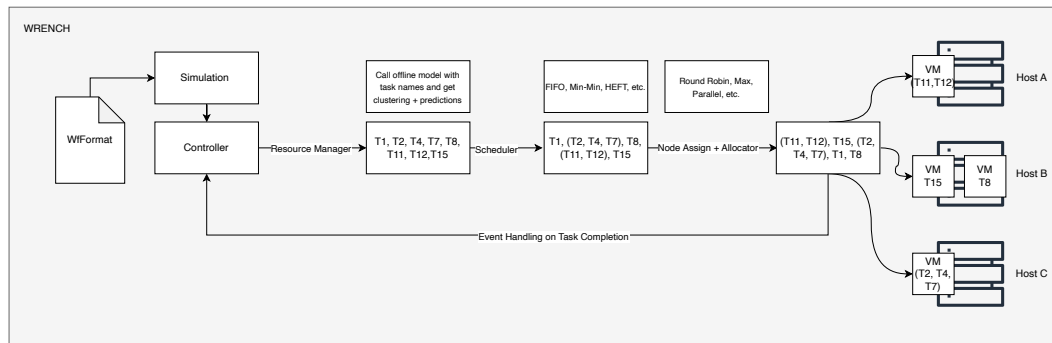


Figure 4.15: Coloc Problem
Das ist eine Beschreibung der Abbildung.

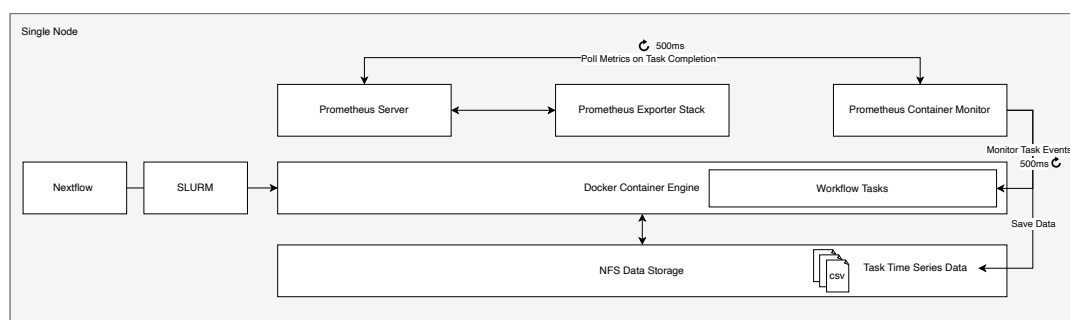


Figure 4.16: Monitoring
Das ist eine Beschreibung der Abbildung.

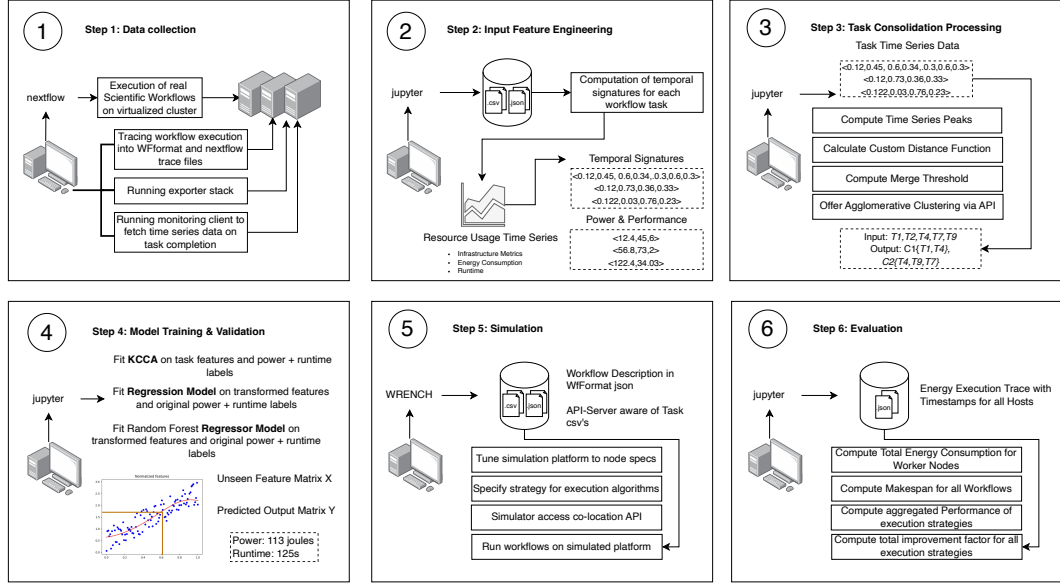


Figure 4.17: Overview
Das ist eine Beschreibung der Abbildung.

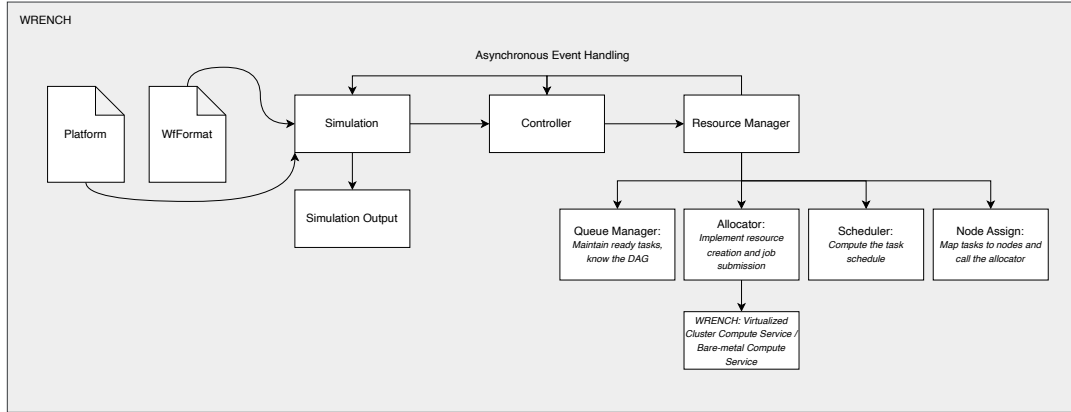


Figure 4.18: Wrench design
Das ist eine Beschreibung der Abbildung.

The following section outlines the methodological approach of this thesis. While concrete implementation details are discussed in the subsequent chapter, this section focuses on establishing the theoretical foundation that builds upon the concepts introduced in the background. The structure of this chapter is organized according to the steps in ?? and begins with outlining the central objective of the thesis. There follows a set of assumptions to delimit the scope, applicability, and limitations of the proposed approach before the different steps are explained in detail.

4.1 Central Objective - Modelling of Task Co-location

The central objective of this work is to formally model the co-location of workflow tasks and further determine how this information can be applied during workflow execution, thereby achieving improved energy efficiency. While this work builds upon the fundamental approach to co-location of [5644899] we want to specifically examine how co-location can

be directly integrated into the dynamic scheduling and task mapping process, arguing that co-location and scheduling are inherently interdependent and should be addressed in a unified way. Consequently, we characterize workflow tasks 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. Furthermore, we extend the question of which tasks to co-locate by the need of knowing how the performance behavior of co-located tasks can be learned and show possible means of making this knowledge accessible to resource managers.

The proposed research problem as introduced in ?? is decomposed into six steps, seen in ?. First, a data collection phase captures detailed execution metrics from scientific workflow runs. Second, this collected data undergoes structural treatment and formatting to identify relevant performance characteristics. Thirdly, the data is used to represent temporal behavior of a workflow task. This representation allows for comparison between a set of tasks to capture differences in their resource usage profiles over time and thus cluster tasks if they differ in their behavior. Step 4 arranges the collected data from step 1 in a slightly different way suitable for learning the relationship between task behavior and the resulting runtime and energy consumption. Finally, step 5 and 6 discover how the theoretical approach of step 3 can be brought into action during workflow execution and how the potential benefits of this approach can be evaluated.

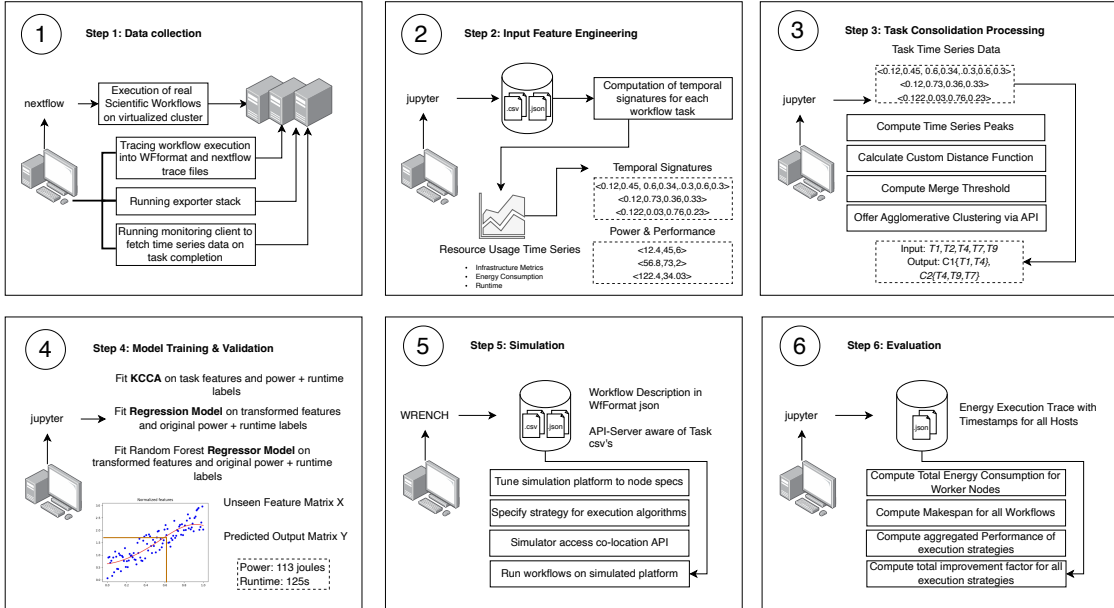


Figure 4.19: Overview of the approach consisting of 6 steps.

4.1.1 Assumptions

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

1. **Monitoring Configuration Limits:** Workflow tasks are described by 80 monitoring features. This work does not investigate the influence of monitoring data dimensionality on clustering and predictive performance.

2. **Monitoring Data Coverage:** Short-lived tasks under one second are only partially captured or occasionally missed due to high system load and sampling intervals exceeding one second during workflow execution.
3. **Offline Data Analysis:** The data preprocessing and predictive model fitting are performed offline after workflow execution. The co-location clustering algorithm is evaluated offline but also 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. It is expected that the overall behavior observed in simulation aligns with real-world execution trends.
5. **Simulation Capabilities and Contention Modelling:** The WRENCH framework currently supports simulation of memory contention by limiting per-Host memory consumption, where exceeding the limit results in extended task execution times. Similarly, CPU contention is modeled through proportional increases in task runtime. Other low-level contention effects like cache interference, interconnect bottlenecks, or I/O contention are not modeled in this iteration. The energy model provided by SimGrid is assumed to realistically approximate energy consumption variations when tasks with differing resource usage profiles are co-located on the same virtual machine. The impact on energy efficiency is attributed to CPU utilization behavior and derived from the platform description where different load levels map to consumed energy amount.

4.2 Online Task Monitoring

The online task monitoring approach builds on a hierarchical architecture that tries to capture a wide spectrum of metrics relevant to the execution of scientific workflows. The design is inspired by [Bader_2022] targeting a four-layered structure consisting of the Resource Manager, Workflow, Machine, and Task layers. Each layer represents a distinct abstraction level in the workflow execution environment and provides valuable insights into different aspects of performance and resource utilization.

Starting with the Resource Manager layer a coarse-grained view related to cluster or partition status, resource allocation, and job management is provided. The Workflow layer operates on the logical workflow representation, maintaining execution progress, task dependencies, and overall runtime statistics. Below, the Machine layer captures system-level performance data such as CPU, memory, and storage utilization, as well as hardware-specific configurations. At the lowest level, the Task layer delivers fine-grained, time-resolved monitoring data, including per-task resource consumption, low-level kernel metrics, and execution traces. [Bader_2022].

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

To enable access to detailed performance metric in time-series format we chose to use Prometheus as the central time-series database. As shown in the ??, we consequently focused on tools that are interoperable with Prometheus as data exporters and examine

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

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

their level of operation in layered monitoring framework. Due to different reasons certain exporters that are listed in ?? were excluded from further use. The Slurm exporter, a custom implementation, was originally intended to retrieve job-to-process mappings from the resource manager. However, equivalent information could also be obtained directly from the workflow engines tracing mechanism, and because the programmatic interaction with Slurm’s daemons introduced considerable overhead, the exporter was not used in later stages. The process exporter was also omitted due to difficulties in collecting and aggregating all process IDs associated with a single container. In many cases, only the initial shell process of a workflow task was captured, while its spawned subprocesses remained untracked. Scaphandre, despite appearing promising at first, showed limited compatibility with different hardware architectures. The tool also struggled to capture short-lived workflow tasks leading to incomplete datasets. Docker-Activity imposed the same limitations as scaphandre.

The most consistent and comprehensive results were achieved using cAdvisor and a custom fork of the DEEP-mon system, presented in [8425477]. These two tools demonstrated the best stability and coverage across different task durations and resource types and were therefore selected for the final monitoring setup.

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

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

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

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

prominently container and process usage collected at cgroup boundaries. In practice, cAdvisor provides low-overhead, per-container telemetry.

DEEP-Mon is a per-thread power attribution method to translate coarse-grained hardware power measurements into fine-grained, thread-level energy estimates by exploiting hardware performance counters. The Intel RAPL interface provides power readings per processor package or core, but it cannot distinguish how much of that energy was consumed by each thread. DEEP-Mon bridges this gap by observing how actively each thread uses the processor during each sampling interval. It does so by monitoring the number of unhalted core cycles—a counter that measures how long a core spends executing instructions rather than idling. Since power consumption correlates almost linearly with unhalted core cycles, the fraction of total cycles attributed to each thread provides a reasonable proxy for its share of energy usage. 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 DEEP-mon tool was modified in this work to export container-level metrics directly to Prometheus.

Based on this identification of relevant metrics and the according technologies, we now introduce the monitoring algorithm built on top of the selected exporter stack and the resulting overview of the needed components. It outlines how these components were combined to realize the monitoring logic in this work, while the technical implementation details are described in ??.

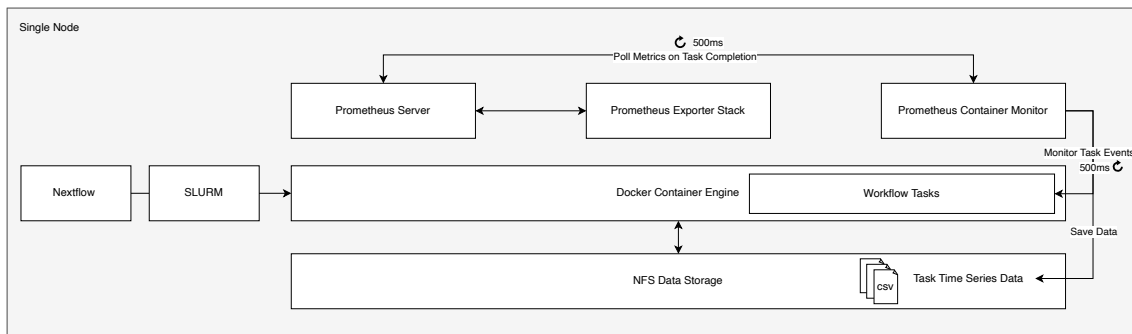


Figure 4.20: The monitoring client alongside the needed system components.

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

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

Algorithm 1: Event-Driven Monitoring and Metric Aggregation Framework

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

```

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

```

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

4.3 Modelling the Co-location of Task Behavior based on Time-Series

This section covers step 2, 3 and 4 seen in ?? and describes how the collected monitoring data is processed to derive task behavior representations suitable for co-location clustering and predictive modeling.

General Preprocessing of Raw Time-Series Data

The collected monitoring data consists of raw time-series files in CSV format. Therefore, an initial step of pre-processing is necessary.

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

During workflow execution, diverse monitoring tools and system components produce data at different abstraction levels. We use container lifecycle records, which provide information on task identifiers, container names, process hashes, and working directories of the workflow task. These files serve as the primary link between workflow-level entities and

low-level cotainer monitoring data. Each monitoring source is traversed recursively to locate and load the associated time-series data. These data files are organized hierarchically under source-specific directories. To facilitate task-level analysis, these aggregated time-series datasets need to be split into per-task CSV files. We attach the correct working directory to every container trace and extract task names and their corresponding working directories. These entries are matched against container-level records from the monitoring data to identify which monitored container corresponds to which task in the workflow.

4.3.1 Task Clustering for Contention-Aware Consolidation

After concluding the initial preprocessing of the raw time-series data we proceed with further refinement of the data according to the needs of performing task clustering.

Preprocessing Raw Time-Series Data for Task-Clustering

We apply the following two main steps prior to the clustering algorithm.

1. **Peak-pattern construction.** For every task and monitored workload type, we first derive a peak time series: the raw per-second resource signal is resampled into three-second buckets and the maximum per bucket is retained. When two peak series must be compared, we truncate both to the shorter length so that correlation is computed on aligned vectors without padding artifacts.
2. **Computing Workload-type affinity.** Different resource domains interfere to different degrees such as CPU vs CPU peaks are typically more contentious than CPU vs file I/O. We encode this by computing an affinity score between workload types which is described in ???. High affinity means higher potential interference when peaks align; low affinity reflects benign coexistence.

Measuring Resource Interference of Co-located Benchmarks The measurement of resource contention follows a two-stage protocol that first establishes isolated baselines for each workload class and then repeats the same workloads under controlled co-location. In the baseline stage, CPU-bound, memory-bound, and file-I/O-bound benchmark containers are executed one at a time on pinned logical CPUs. Each run records a start and finish timestamp at microsecond resolution. In parallel, Deep-Mon is used to record the power time series for each container. After a run finishes, the power streams of the containers are retained, aligned to the containers lifetime, and summarized to a representative mean value. After isolated benchmark execution we replay the same benchmarks in pairs to expose interference effects by CPU pinning. The experiment binds pairs of benchmarks to siblings on the same physical core to amplify shared-core effects. Each co-located container is measured in exactly the same way as in isolation, producing a matched set of durations and power summaries.

In order to derive a workload affinity from the contention experiments we define contention effects to occur when co-located workloads exceed their duration and power consumption compared to their isolated measurements.

Isolated and Co-located Metrics.

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

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

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

Per-workload Slowdown Factors.

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

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

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

The per-workload slowdowns are defined as

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

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

The mean slowdowns across the workload pair are

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

A weighted average combines both effects:

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

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

The final combined slowdown factor is

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

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

Affinity Score.

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

First, compute pairwise affinity ratios:

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

A weighted average combines both effects:

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

The normalized affinity score is then:

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

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

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

Algorithm ?? summarizes the main stages of the task consolidation procedure used in this work. The approach begins by computing pairwise similarities between task signatures, where each signature represents a multi-dimensional profile of resource usage over time. The similarity computation incorporates resource affinity weights derived from the contention experiments. This yields a resource-aware similarity matrix that reflects how well tasks can coexist on the same node without interference. Next, a percentile-based threshold is applied to the similarity matrix. This adaptive rule determines when clustering should stop by selecting only the most suitable task pairs according to the defined percentile. In this way, the algorithm avoids arbitrary distance cutoffs and instead adapts to the underlying distribution of similarity values. Finally, agglomerative clustering is performed using a specified linkage criterion, such as average or complete linkage, to iteratively merge tasks into clusters.

Algorithm 2: ShaReComp - Task Consolidation Algorithm

Input: task signatures sig , affinity weights w , percentile τ , linkage

Output: clusters \mathcal{C}

```

 $S \leftarrow \text{compute\_similarity}(\text{sig}, w);$            // resource-aware similarity
 $\theta \leftarrow \text{percentile\_threshold}(S, \tau);$        // percentile-based stopping rule
 $\mathcal{C} \leftarrow \text{agglomerative\_merge}(S, \theta, \text{linkage});$  // agglomerative clustering
return  $\mathcal{C}$ 

```

With ?? we introduce the ShaReComp algorithm. ShaReComp stands for share resources for computation and summarizes the goal of the method: to group tasks that can share resources effectively when co-located. The key steps of the algorithm are as follows:

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

Inter-Task Distance and Resource Correlation Model

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

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

where:

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

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

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

Interpretation

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

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

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

$$D_{i,j} < \text{merge_threshold}.$$

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

2. **Computing the merge threshold.** Because the distance matrix is data-dependent, we estimate a merge threshold directly from its empirical distribution. In our approach we select the 20th percentile on the raw distances as an automatic cut-level: any pair below this threshold is safe enough to consider for co-location, while pairs above it are kept apart.
3. **Agglomerative clustering with precomputed distances.** We run average-linkage agglomerative clustering on the precomputed distance matrix with the chosen distance threshold and no preset cluster count. This yields variable-sized clusters whose members are mutually non-contentious under our metric. Because we use a threshold rather than a fixed k , the method adapts to each workload mix, producing more or fewer groups as warranted by the observed interference structure.

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

4.3.2 Predicting the Runtime and Energy Consumption of Task Clusters with KCCA and Random Forest Regression

Preprocessing Raw Time-Series Data for Predictive Modelling

Differing from the preprocessing ??, we define the following actions as a preparation for step 4 in ??.

1. Temporal signature construction (feature selection).

Sampling and smoothing.

Equal-length normalization.

Container-wise collation.

2. Model Input Construction.

Normalization and Scaling.

Extraction of Input Features and Output Labels.

Temporal Signature and Model Input Construction.

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

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

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

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

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

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

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

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

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

denotes the complete input matrix used for model training.

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

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

Example.

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

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

Concatenating these signatures yields the input matrix

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

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

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

Building upon ?? from ??, we suggest another ShaReComp algorithm that allows to model the temporal behavior of consolidated tasks using the extracted features per task from ??.

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

Input: task clusters \mathcal{C} , per-task signatures sig , trained model \mathcal{M} (KCCA or Random Forest)

Output: predicted runtime energy pairs $\hat{Y} = \{(\hat{t}_k, \hat{E}_k)\}$

foreach cluster $C_k \in \mathcal{C}$ **do**

$F_k \leftarrow \text{sum_cluster_features}(\{\text{sig}[t_i] \mid t_i \in C_k\})$; // sum task signatures to form cluster feature

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

foreach cluster feature $X_k \in X$ **do**

$(\hat{t}_k, \hat{E}_k) \leftarrow \text{predict_runtime_and_energy}(X_k, \mathcal{M})$

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

The predictors used in ?? are described in the following.

Kernel Canonical Correlation Analysis KCCA wants to identify relationships between task-specific features and their corresponding performance and energy characteristics. To achieve this, the dataset is divided into two parts: approximately 70% of the tasks are used for training the models, while the remaining 30% are reserved for testing and validation.

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

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

Kernel Canonical Correlation Analysis (KCCA).

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

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

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

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

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

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

Illustrative Example.

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

$$x_1 = [0.40, 0.75, 0.90, 0.35, 0.55, 0.60], \quad x_2 = [0.20, 0.50, 0.70, 0.25, 0.40, 0.50], \quad x_3 = [0.30, 0.65, 0.85, 0.10, 0.45, 0.70].$$

Their corresponding runtime power outcomes are:

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

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

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

Example. Assume two clusters:

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

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

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

Cluster features are summed:

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

Model predictions from both KCCA or Random Forest Regressor yield

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

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

4.4 Simulation of Task Co-location during Workflow Execution

Outline of Simulation Components

To accomplish step 5 and 6 depicted in ??, we build upon specific design pillars that structure the simulation environment.

The design of distributed HPC systems centers on three core aspects: resource allocation, queue ordering and dispatching, and job placement. In ?? we adapt these core aspects and design a controller component that is responsible for invoking a central resource manager. The resource management component itself is steering the scheduling of tasks, the DAG-aware queue management, the assignment of tasks to the available compute nodes and eventually the allocation and execution on resources. Since we aim to gain insight into the effectiveness of task co-location the processing engine of the simulator needs to offer an interface for alternative scheduling, task mapping and allocation strategies, ranging from conventional First-in-First-out (FIFO) ordering to heuristic reordering and placement that prioritizes beneficial pairings.

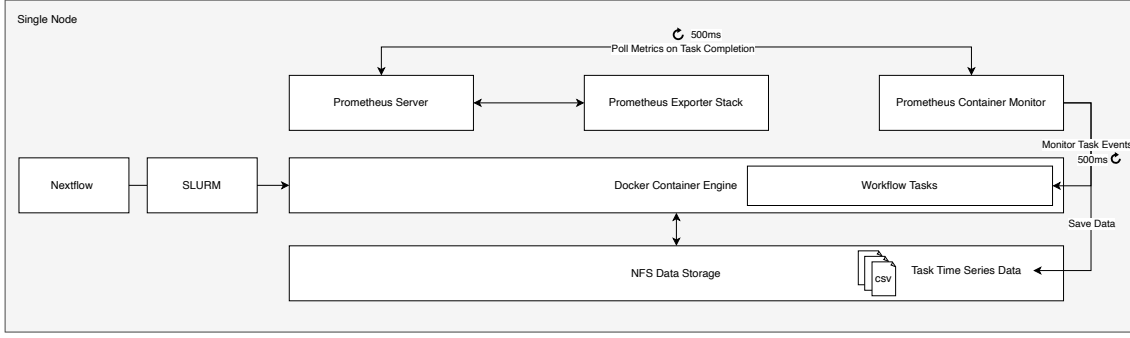


Figure 4.21: The monitoring client alongside the needed system components.

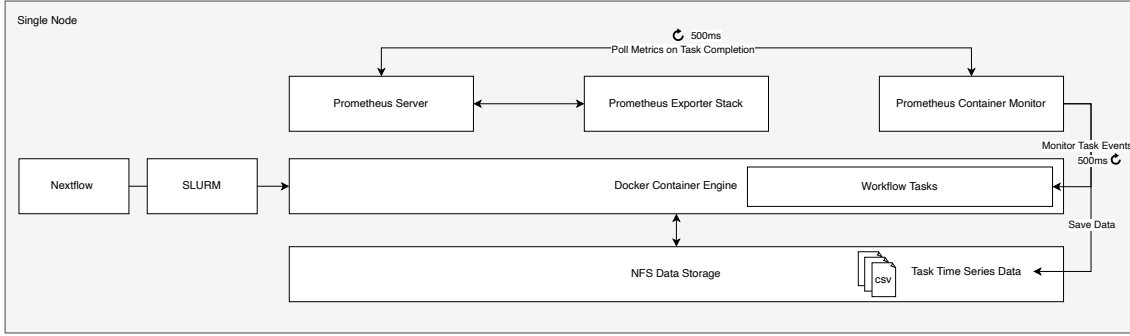


Figure 4.22: The monitoring client alongside the needed system components.

4.4.1 Embedding Task Co-location into Scheduling Heuristics

Our algorithmic approach to define the behavior of the component's resource manager, allocator and scheduler introduced in ?? is grounded in heuristic principles. We choose to not formulate the embedding of co-location into the scheduling process as an optimization problem but through decision-making within simple, interpretable 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. We divide the task-to-node mapping into two steps. ?? illustrates this approach. The simulation launches the controller component which invokes our resource manager. Through the queue management the resource manager retrieves a set of ready-to-run tasks without predecessors, that they depend on. In our example task 1,2,4,8,11,12,15 are to be scheduled. For the scheduling we choose to adopt list-scheduling algorithms, which form one of the most established heuristic approaches for workflow scheduling. They assign a priority or ranking to each task based on topological and performance factors (e.g., critical path length, execution cost, or communication overhead), producing a priority list. They then iteratively select the highest-priority unscheduled task and map it to a available resources. As we are interest in mapping not singel tasks but clusters of tasks we identify to embed the infomation retrieval on beneficial task consolidation at this exact stage during the scheduling process. The co-location hints my yield in consolidating T1, (T2, T4, T7), T8, (T11, T12) and T15. Lastly, we need to decide which host can accomodate the formed task clusters the best. We again define an interface for node assignment strategies either prioritize the host with the most idle compute cores first or tries to distribute the task clusters among all the available hosts in a fair manner. Finally the allocaiton component creates virtual machines that contain the task clusters and launch them on the assigned

host. By this exemplified design, we create a ground to not only embed and study the co-location problem in an execution environment but also gain the possibility of implementing different means of comparison through the interfaces per component. The next section formally introduce the simulation environment, consisting of the scientific workflow as the application that executes tasks and a system model that we use to represent our algorithmic approach to the co-location problem.

Simulation System Model

Workflow Properties

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

$$G = (T, E)$$

where

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

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

Task Properties

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

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

Infrastructure Model

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

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

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

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

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

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

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

Resource Assignment

A mapping of tasks to hosts and VMs is represented as

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

where

- $h \in \mathcal{H}$ is the assigned host,

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

The set of mappings for an allocation interval is written as

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

Task Co-location

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

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

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

Oversubscription

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

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

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

Execution Dynamics

At runtime:

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

Using the modelling introduced above, we construct an algorithm that depicts the overall behavior of our simulation approach.

Algorithm 4: ShaReComp Simulation - WRENCH Framework

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \dots, h_m\}$, scheduling policy π , oversubscription factor α , optional co-location API \mathcal{S}
Output: workflow executed with policy-driven scheduling, node assignment, and adaptive resource management

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

To compare the effectiveness of using the ShaReComp approach we define multiple algorithms for the scheduling and node assignment phase that either employ exclusive resource assignment, assign tasks to multiple hosts in parallel or cluster random tasks into virtual machines. Generically all algorithms build on our simulation blueprint showed above ???. We call these algorithms baselines. In fact, we implement two baselines where each task gets executed in it's own VM, so no co-location is applied at all. Baselines 3, 3.1 and 3.2 co-locate tasks onto virtual machines and experiment with prioritizing the available compute hosts according to different strategies. Baselines 4 and 4.1 employ oversubscription, meaning that more tasks than reserved compute resources get assigned to a VM, strategically forcing resource contention. All baseline algorithms are included in the appendix of this thesis. In ?? we include more detailed descriptions on the behavior and choice for the baseline algorithms.

Algorithmic Examples of Task Mapping with guided Co-location

This chapter on our approach concludes with introducing our main contribution of this thesis. We use our simulation framework to implement the ShaReComp approach directly into the workflow execution process. Therefore, we design four algorithms in 4 different versions of ShaRiff. ShaRiff is an acronym for share resource if feasible. Each ShaRiff version implements a different strategy of mapping consolidated tasks onto available compute hosts while the decision making for the co-location itself is always guided by the ShaReComp approach, introduced in ??. Table ?? provides an overview of the key differences between the ShaRiff algorithms.

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

Table 4.3: Overview of ShaRiff and ShaReComp Scheduling Algorithms.

In the following we will in-depth describe the behavior of the ShaRiff algorithms one by one.

Algorithm 5: ShaRiff 1 — Contention-aware Max-Parallel VM Co-Location Scheduling

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

Output: all tasks $t_i \in T$ executed with contention-aware co-location for improved efficiency and utilization

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

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

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

 Wait until any task t_r completes; Release its cores:

$c(h(t_r)) \leftarrow c(h(t_r)) + \text{req_cores}(t_r)$; For each successor t_s of t_r , enqueue t_s into \mathcal{Q} if all predecessors are completed; **continue**

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

 Initialize empty host task mapping list \mathcal{M} ;

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

 Select up to $c(h)$ ready tasks from \mathcal{Q} into \mathcal{T}_h ; Compute file-location map $\Phi(\mathcal{T}_h)$; Query ShaReComp API: $\text{colocMap} \leftarrow \mathcal{S}(\mathcal{T}_h)$; // **returns** co-location groups

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

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

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

$C_{\text{req}} \leftarrow \text{sum}(\text{req_cores}(\mathcal{C}_k))$; $M_{\text{req}} \leftarrow \text{sum}(\text{req_mem}(\mathcal{C}_k))$; Allocate $v_h = (C_{\text{req}}, M_{\text{req}}, h)$; Launch all $t \in \mathcal{C}_k$ on v_h ;

$c(h) \leftarrow \max(0, c(h) - C_{\text{req}})$; Remove \mathcal{C}_k from \mathcal{Q} ;

 Wait until any task t_r completes; Release its cores:

$c(h(t_r)) \leftarrow c(h(t_r)) + \text{req_cores}(t_r)$; **if** no active tasks remain on its VM **then**

 Destroy VM

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

return workflow complete

This variant implements ShaRiff 1, which augments the FIFO pipeline with the external co-location adviser ShaReComp and a cluster-aware allocator. Tasks are dequeued in strict arrival order. Before placement, the scheduler invokes ShaReComp with the current set of ready tasks and receives clusters of jobs that are computed to co-locate well. The node-assignment stage then ranks hosts by descending idle-core capacity and fills the largest host first: it forms a batch of up to that hosts idle cores and attaches the ShaRiff cluster map to the batch; if tasks remain, it proceeds to the next host in the ranked list. A small-queue fast path ensures dispatch even when only a few tasks are available. The allocator realizes the advisers plan one VM per recommended cluster on the chosen host. For each multi-task cluster, it provisions a VM whose vCPU count and memory equal the sum of the clustered tasks declared requirements, starts the VM, and submits the tasks to that same virtual compute service. Singleton clusters are grouped into a shared VM on the host 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, thereby reducing contention and improving per-host utilization without oversubscription. When the adviser yields single tasks, the system still shares if feasible by pooling them into a shared VM, maintaining the same deterministic provisioning and lifecycle rules.

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

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

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

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

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

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

 Wait until any task t_r completes Release its cores:

$c(h(t_r)) \leftarrow c(h(t_r)) + \text{req_cores}(t_r)$ For each successor t_s of t_r , enqueue t_s into \mathcal{Q} if all predecessors are completed

continue

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

 Initialize empty host task mapping list \mathcal{M}

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

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

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

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

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

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

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

else

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

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

 Wait until any task t_r completes Release its cores:

$c(h(t_r)) \leftarrow c(h(t_r)) + \text{req_cores}(t_r)$ **if** no active tasks remain on its VM **then**

 Destroy VM

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

return workflow complete

This variant retains the ShaRiff-augmented FIFO pipeline but enables controlled oversubscription during placement and VM sizing. Tasks are dequeued in arrival order. Before dispatch, the scheduler queries ShaReComp with the current ready set and receives clusters of tasks pcomputed to co-locate well. Hosts are ranked by descending idle-core capacity; the assigner then fills the largest host first with a batch whose size may exceed the hosts

free cores by a fixed factor of in our example 25%. If tasks remain, it proceeds to the next host and repeats. The allocator implements the advisers plan one VM per cluster on the chosen host, but with oversubscription semantics. For multi-task clusters, it provisions a VM whose vCPU and memory equal the sum of the clusters requests—even if that exceeds the hosts currently free cores. For single task clusters collected on the same host, it provisions a shared VM and caps vCPUs at the hosts free cores when necessary. 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. Because ShaReComp groups complementary tasks, oversubscription holds the potential to translate into higher throughput and energy efficiency. However, when clustered tasks are less complementary, contention can surface, making this variant an explicit trade-off between utilization and interference.

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

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

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

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

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

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

 Wait until any task t_r completes Release its cores:

$c(h(t_r)) \leftarrow c(h(t_r)) + \text{req_cores}(t_r)$ For each successor t_s of t_r , enqueue t_s into \mathcal{Q} if all predecessors are completed

continue

Select next host $h = \mathcal{H}[r \bmod |\mathcal{H}|]$ Update round-robin index:

$r \leftarrow (r + 1) \bmod |\mathcal{H}|$ Retrieve available cores $C = c(h)$ Select up to C ready tasks from \mathcal{Q} into \mathcal{T}_h Query ShaReComp API: $\text{colocMap} \leftarrow \mathcal{S}(\mathcal{T}_h)$;

// **returns** co-location groups

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

$C_{\text{req}} \leftarrow \text{sum}(\text{req_cores}(\mathcal{C}_k))$ $M_{\text{req}} \leftarrow \text{sum}(\text{req_mem}(\mathcal{C}_k))$ Allocate

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

 Remove \mathcal{C}_k from \mathcal{Q}

Wait until any task t_r completes Release its cores:

$c(h(t_r)) \leftarrow c(h(t_r)) + \text{req_cores}(t_r)$ **if** *no active tasks remain on its VM* **then**

 Destroy VM

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

return *workflow complete*

ShaRiff 3 preserves FIFO dequeuing but combines round-robin first-fit placement with ShaReComp-guided intra-VM co-location. The scheduler releases tasks strictly in arrival order. The node-assignment component scans hosts in round-robin/first-fit fashion and picks the first host reporting at least one idle core. It then pulls up to that hosts idle-core capacity worth of ready tasks and queries ShaReComp for a co-location plan over this batch. The allocator realizes ShaReComps plan one VM per suggested cluster on the chosen host. For each multi-task cluster, it sizes the VM by summing vCPU and memory requirements of the clusters tasks, starts the VM, and submits all cluster tasks to the

same virtual compute service. Tasks that ShaReComp leaves as singletons are grouped onto an additional shared VM on that host; its size is the aggregate of the singletons requests. Conceptually, the policy is first-fit host, adviser-driven packing. 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 ShaReComp’s clustering to raise utilization and efficiency through informed co-location.

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

5 Implementation

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

5.1 System Architecture

5.2 Technology and Design Choices

Table 5.4: Monitoring features and associated technical components.

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

5.3 Decoupled Design for further Research

The extensibility of the overall system is achieved through a 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 interfaces rather than rewriting existing code.

5.3.1 Configurable Monitoring Client

The monitoring client exemplifies this philosophy by relying on a flexible configuration-driven architecture. Using a YAML-based configuration file, it dynamically defines which metrics to collect from heterogeneous data sources such as Prometheus exporters, cAdvisor, eBPF probes, or SNMP-based sensors. The configuration specifies not only the metric names and queries but also the identifiers and units, allowing seamless adaptation to different environments or workflow engines. This separation of logic and configuration enables the same client to operate across diverse infrastructures without recompilation. The 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.

Table 5.5: Adaptable Monitoring Configuration Overview.

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

5.3.2 Enabling Access to Co-location Hints in the Simulator

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

Table 5.6: Overview of REST API Endpoints Exposed by the ShaReComp Service.

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

5.3.3 Used Simulator Components

The simulator setup further demonstrates the benefits of this decoupled design. The resource management layer of the simulator exposes generic interfaces for allocators, schedulers, and node assigners, allowing any of them to be replaced or combined dynamically at runtime. This separation allows the same simulator to execute both baseline and experimental resource allocation strategies without modifying the controller logic. Through this design, the simulator can execute diverse workflow types 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.

```

1 int main(int argc, char **argv)
2 {
3     auto simulation = wrench::Simulation::createSimulation();
4
5     simulation->init(&argc, argv);

```

```

6
7  auto workflow = wrench::WfCommonsWorkflowParser::createWorkflowFromJSON
    (workflow_file, "100Gf", true);
8
9  simulation->instantiatePlatform(platform_file);
10
11  std::set<std::shared_ptr<wrench::StorageService>> storage_services;
12  auto storage_service = simulation->add(wrench::SimpleStorageService::
    createSimpleStorageService(
13      "WMSHost", {"scratch/"}, {{wrench::SimpleStorageServiceProperty::
        BUFFER_SIZE, "50MB"}}, {}));
14
15  std::vector<std::string> hostnames = {"VirtualizedClusterHost1", "
    VirtualizedClusterHost2", "VirtualizedClusterHost3"};
16
17  std::cerr << "Instantiating an EnergyMeterService on WMSHost that
    monitors VirtualizedClusterHost1 every 10 seconds ..." << std::endl
    ;
18  auto energy_meter_service = simulation->add(new wrench::
    EnergyMeterService("WMSHost", hostnames, 10));
19
20  simulation->getOutput().enableEnergyTimestamps(true);
21
22  auto virtualized_cluster_service = simulation->add(new wrench::
    VirtualizedClusterComputeService(
23      "VirtualizedClusterProviderHost", virtualized_cluster_hosts, "",
        {}, {}));
24  auto wms = simulation->add(
25      new wrench::Controller(workflow, virtualized_cluster_service,
        storage_service, "WMSHost"));
26
27  auto file_registry_service = new wrench::FileRegistryService("WMSHost")
    ;
28  simulation->add(file_registry_service);
29
30  simulation->launch();
31
32  auto energy_trace = simulation->getOutput().getTrace<wrench::
    SimulationTimestampEnergyConsumption>();
33
34  simulation->getOutput().dumpHostEnergyConsumptionJSON("../results/{
    strategy}_{workflow}rnaseq_host_energy_consumption.json", true);
35
36  return 0;
37 }

```

Listing 1: WRENCH C++ Simulation File

6 Evaluation

6.1 Evaluation Setup

6.1.1 Infrastructure

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

6.1.2 Workflows

Table 6.7: Overview of evaluated nf-core workflows and their input/output characteristics.

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

6.1.3 Monitoring Configuration

Table 6.8: Adaptable Monitoring Configuration Overview.

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

6.1.4 Implemented Models for Task Clustering and Prediction

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

6.1.5 Simulation Setup

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

1 <?xml version='1.0'?>


```

2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
3 <platform version="4.1">
4   <zone id="AS0" routing="Full">
5     <host id="VirtualizedClusterHost1" speed="32Gf,28Gf,20Gf,12
6       Gf,8Gf" pstate="0" core="24">
7       <prop id="ram" value="188GiB" />
8       <!-- The power consumption in watt for each pstate (
9         idle:all_cores) -->
10      <prop id="wattage_per_state" value="116:133:220,
11        114:129:202, 111:125:186, 108:121:170,106:118:156"/>
12      <prop id="wattage_off" value="10" />
13    </host>
14    <host id="VirtualizedClusterHost2" speed="32Gf,28Gf,20Gf,12
15      Gf,8Gf" pstate="0" core="24">
16      <prop id="ram" value="188GiB" />
17      <!-- The power consumption in watt for each pstate (
18        idle:all_cores) -->
19      <prop id="wattage_per_state" value="116:133:220,
20        114:129:202, 111:125:186, 108:121:170,106:118:156"/>
21      <prop id="wattage_off" value="10" />
22    </host>
23    <host id="VirtualizedClusterHost3" speed="32Gf,28Gf,20Gf,12
24      Gf,8Gf" pstate="0" core="24">
25      <prop id="ram" value="188GiB" />
26      <!-- The power consumption in watt for each pstate (
27        idle:all_cores) -->
28      <prop id="wattage_per_state" value="116:133:220,
29        114:129:202, 111:125:186, 108:121:170,106:118:156"/>
30      <prop id="wattage_off" value="10" />
31    </host>
32  </zone>
33 </platform>

```

Listing 2: Example XML Configuration File

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

Table 6.9: Overview of Baseline Scheduling Algorithms.

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

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

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

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

6.2 Experiment Results

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

6.2.1 Measuring Interference during Benchmark Executions

Table 6.10: Summary of Synthetic Benchmarks Used in Evaluation

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

For measuring resource contention, as described in Chapter 4, we use the stress-ng tool with the commands listed in ???. The benchmark code is executed inside Docker containers, and the Docker API is used to capture the execution time of each benchmark. Since the benchmark instructions are fixed and not time-dependent, this provides consistent and

comparable measurements. To record the containers’ energy consumption, we again use the ebpf-based Deep-Mon tool.

?? shows the results of running each benchmark sequentially on different CPU cores of the node. This reveals how long a container pinned to a single core takes to complete the specified benchmark. Next, we execute all possible pairs of workloads, running each pair sequentially on the same pinned CPU core. For each pair, we plot the runtime of both benchmarks in grouped bars, where the grey area represents the runtime of the benchmark when executed in isolation. The same procedure is repeated for collecting and visualizing the corresponding energy consumption data.

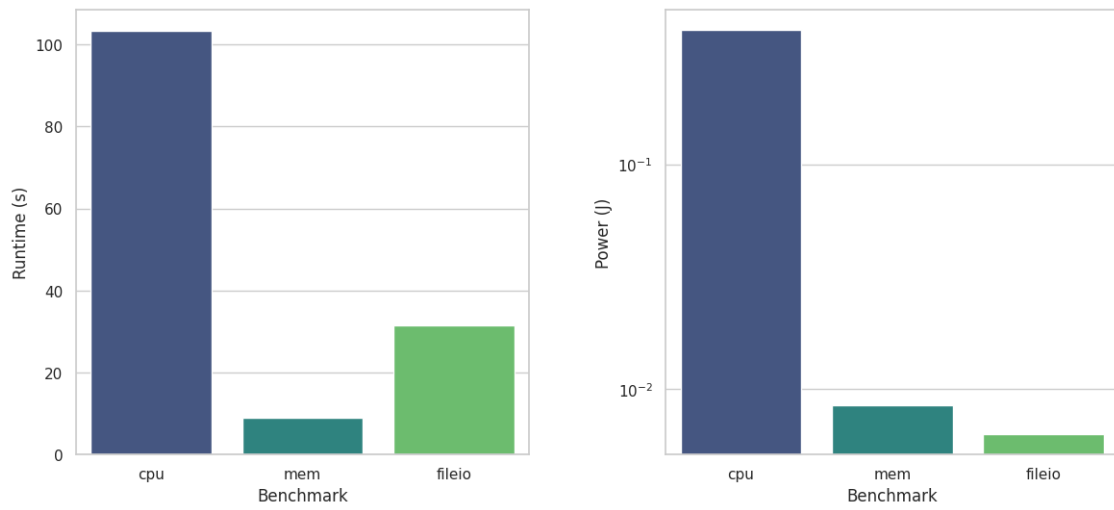
We observe in ?? that the runtime increases for every co-located pair of workloads. While memory-intensive benchmarks show only a slight increase, file I/O workloads tend to take roughly one quarter longer than their isolated runs. The most pronounced effect appears when compute-heavy workloads are co-located, where the execution time nearly doubles. This confirms that CPU-bound applications experience the strongest interference when sharing the same physical core.

The power measurements for the co-located workloads, however, show much higher variance and inconsistency across repeated runs. For example, in the CpuCpu case, the isolated power draw averages around 3.43W, whereas the co-located readings drop to approximately 0.008 joules. Similarly, for FileIOFileIO, the measured power decreases from 0.054W in isolation to about 0.009W when co-located. These values are clearly unrealistic and indicate that the RAPL-based measurements on AMD hardware are unreliable in this setup.

This discrepancy stems from the fact that AMD’s RAPL interface does not expose per-core or per-container energy readings in a consistent manner. When two containers share a single CPU core, RAPL counters may fail to attribute power consumption correctly between them, resulting in near-zero or erratic readings. In contrast, for the CpuMem combination, the measured power values 27.93 joules and 15.58 joules are much higher and closer to expected magnitudes. This pairing likely produced a more stable measurement because the CPU and memory subsystems were stressed differently, allowing RAPL to record distinct and more accurate energy counters.

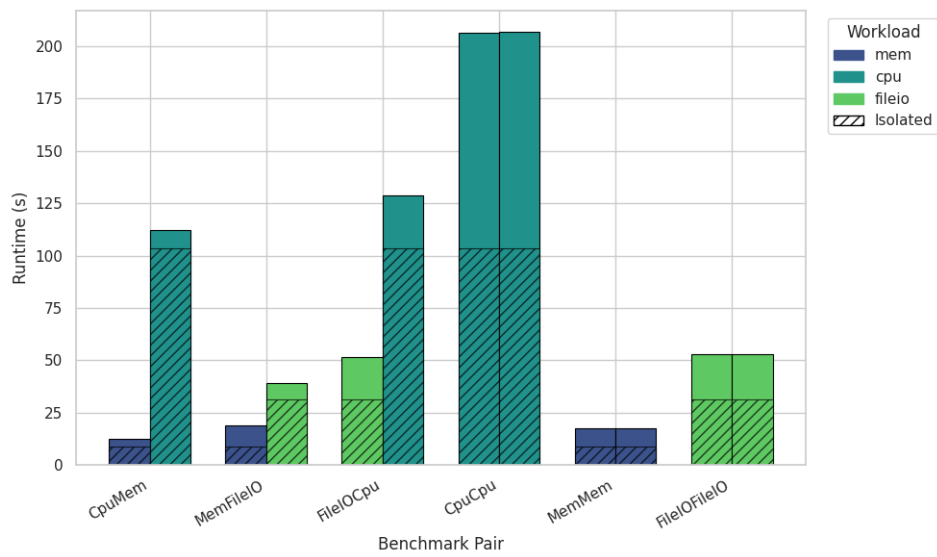
Overall, while the runtime data clearly demonstrate the expected contention effects, the power data highlight a measurement limitation. We therefore do not include the plotted power measurements for the co-located benchmark executions but account for the impact of contention in the following evaluation.

Figure 6.23: Bar Plot iso



Das ist eine Beschreibung der Abbildung.

Figure 6.24: Bar Plot colo



Das ist eine Beschreibung der Abbildung.

We calculated the affinity scores between workload pairs according to ??.

Table 6.11: Affinity scores between workload types indicating co-location compatibility.

Workload 1	Workload 2	Affinity Score	Comment
mem	cpu	0.736	Very Low compatibility; memory and CPU workloads can share resources with serious contention occurring.
mem	fileio	0.293	High compatibility; low interference due between I/O and memory bandwidth pressure.
fileio	cpu	0.272	High compatibility; CPU workloads cause low contention for shared I/O buffers.
cpu	cpu	0.503	Low compatibility CPU-bound tasks compete for cores and effect thread scheduling.
mem	mem	0.566	Low compatibility; High memory contention under shared caching.
fileio	fileio	0.414	Limited compatibility; file I/O contention degrades throughput under co-location.

The results shown in the previous figures are consistent with the affinity scores presented in table ???. When both CPU and memory benchmarks are co-located, their runtimes increase significantly, resulting in the highest affinity scores that indicate strong contention and poor compatibility. This pattern is followed by other same-type co-locations, such as CPU–CPU and Mem–Mem, where scores around 0.5 also reflect noticeable interference, as seen in the near doubling of execution times in the earlier plots. File I/O workloads, by contrast, show comparatively high compatibility. Their co-location with CPU or memory benchmarks causes only minor slowdowns, confirming that I/O-bound tasks exert limited pressure on shared compute or memory resources.

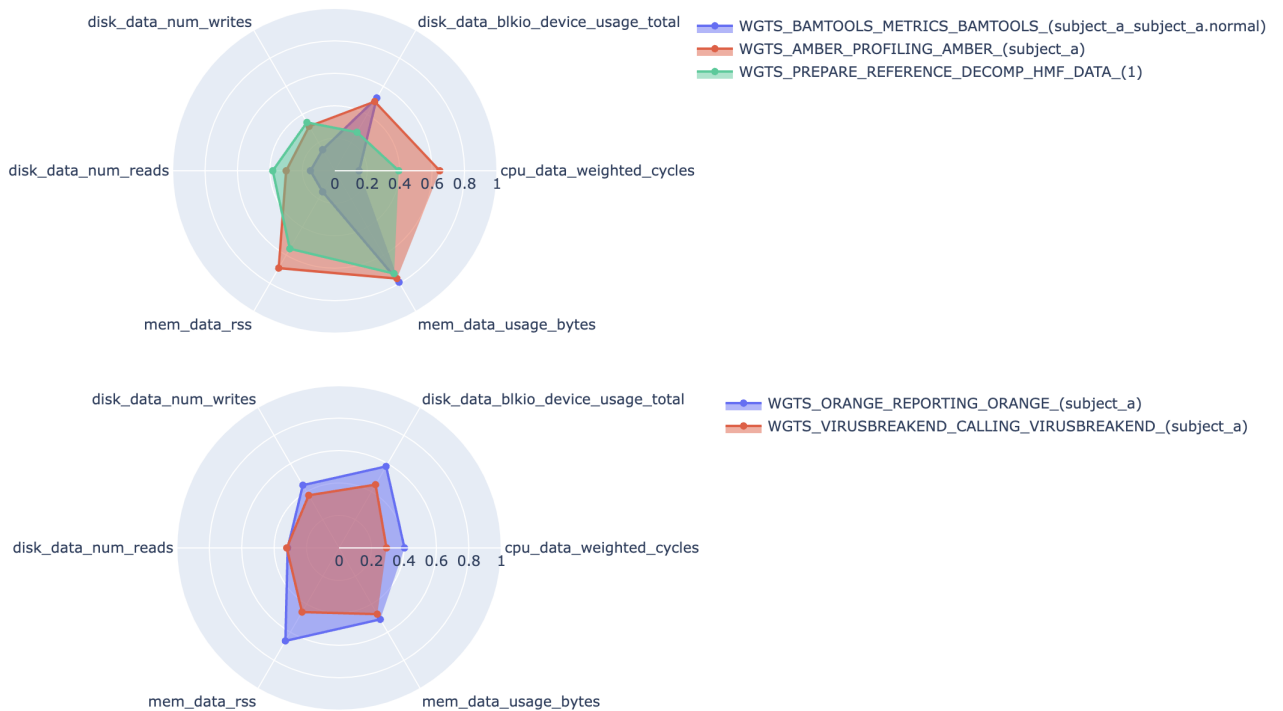
It is worth noting, however, that while the runtime degradation for the CPUMemory pair was moderate, this combination exhibited the largest increase in power consumption. This behavior aligns with the earlier observation that RAPL-based power measurements on AMD hardware tend to be unstable. In calculating the affinity scores, we therefore introduced a weighting parameter $\alpha = 0.7$ to assign greater importance to runtime than to power consumption, compensating for the measurement inconsistencies discussed previously. The resulting affinity scores thus primarily reflect the runtime interference between workloads, which will serve as input for computing task dissimilarities in the following section.

6.2.2 Dissimilarity-based Task Clustering

We evaluate the task clustering procedure in two stages. First, we randomly sample a subset of tasks from the workflow executions and preprocess them for clustering as described in Chapter 4. Specifically, we select 34 random tasks from the OncoAnalyser workflow and perform two clustering variants: a random baseline clustering and the ShaReComp-based clustering, which incorporates dissimilarity distances influenced by the affinity scores presented in the previous table. For each cluster, we compute the scale normalized average

temporal signatures of the monitored performance metrics, as defined by the monitoring configuration in Chapter 6. These averages are then visualized using radar plots. Each radar plot represents the tasks grouped into a single cluster and thus identifies potential candidates for co-location. The purpose of this visualization is to illustrate the effect of the dissimilarity-based distance formulation and the use of a clustering threshold set to the 25th percentile of the overall distance distribution. This threshold ensures that only sufficiently dissimilar tasks are clustered together. Additionally, because the distance measure is adjusted by the correlation between resource usage patterns, tasks with high correlation in their workload behavior tend to be separated, reflecting the influence of the previously computed affinity scores.

Figure 6.25: Radarplot Random



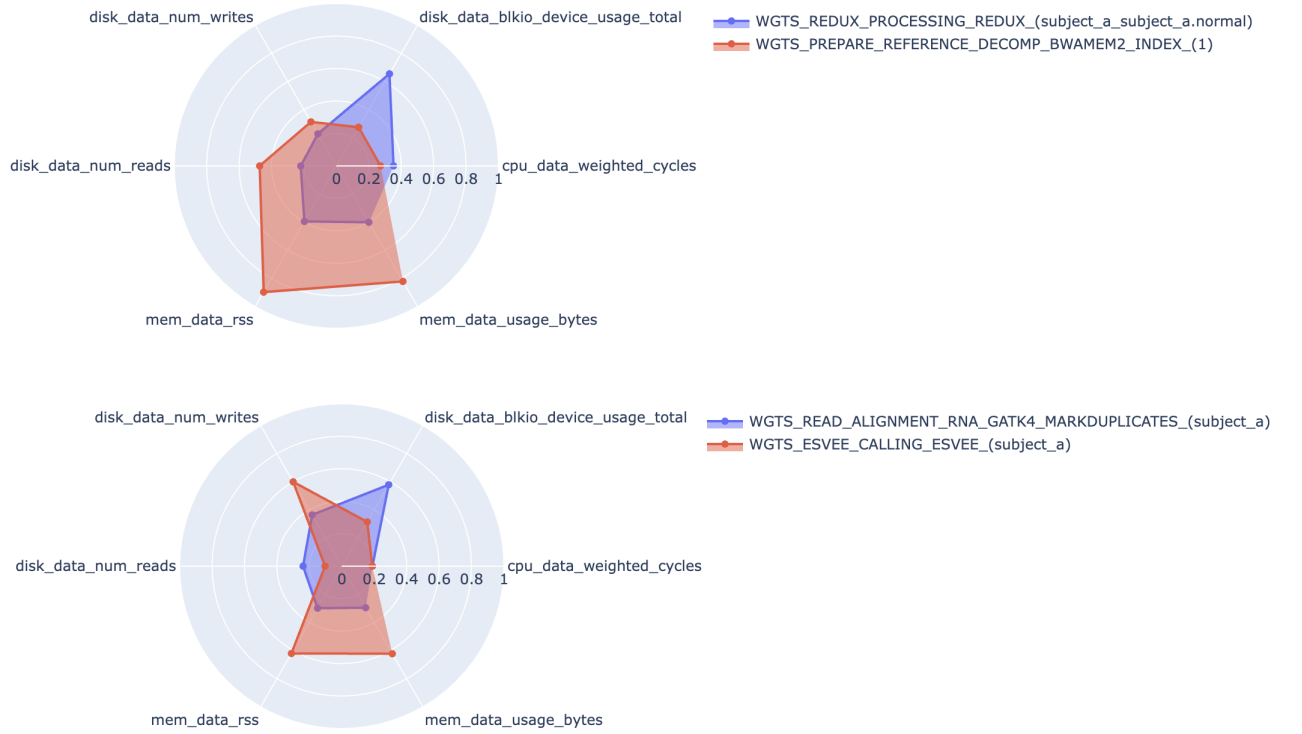
Das ist eine Beschreibung der Abbildung.

We first examine the distribution of task characteristics in the clusters formed by the completely random clustering over the selected probe of oncoanalyser tasks. The first cluster contains three tasks, while the second cluster includes two. At first glance, the radar plots show that the resource profiles of the tasks overlap strongly across all six dimensions of the task resource signatures. Most notably, the bamtools metrics, amber profiling, and prepare reference tasks exhibit almost identical patterns in their average memory allocation over time and resident memory usage RSS. The same overlap appears in their block I/O activity, reflected by nearly identical numbers of file reads and writes, indicating similar file access behavior. Although the average CPU usage differs slightly between these tasks, the variations remain small in scale, suggesting that when such tasks are co-located, contention effects are likely to occur. Overall, this clustering outcome contradicts the affinity findings summarized in Table ??, where memory-memory, CPU-CPU, and mixed memory-CPU combinations showed the highest contention potential. A similar

issue can be observed in the second radar plot, which represents another randomly formed cluster consisting of two tasks. Here again, the task profiles overlap substantially, and the resource dimensions are distributed within the same scale range. This overlap suggests a comparable risk of resource contention, confirming that purely random clustering disregards workload diversity and leads to groupings that do not respect the resource affinity relationships observed earlier.

We now move on to compare the clusters formed by the ShaReComp approach. At first glance, the radar plots already differ notably in shape compared to those produced by random clustering. The task profiles appear shifted relative to one another rather than overlapping, suggesting that the clustering process has effectively separated tasks with similar resource usage. Focusing on the affinities, we observe that both CPU and memory utilization differ clearly in magnitude between tasks within the same cluster. The same applies to the memory-related metrics, including total memory usage and resident set size (RSS), which are visibly offset from one another. This separation is also reflected in the file I/O dimensions. Although moderate contention potential remains, the average values differ sufficiently to indicate that these tasks do not compete heavily for the same I/O resources. Consequently, the higher peaks in memory and disk I/O observed in the radar plots do not imply significant interference, as their activity patterns occur in distinct resource dimensions. The second radar plot shows a similar trend. Tasks with low CPU utilization are clustered together with tasks exhibiting higher memory usage, yet their respective memory signatures remain clearly separated. This indicates that the ShaReComp method successfully avoids grouping tasks that would likely contend for the same resources. From this comparison, we conclude that dissimilarity-based clustering informed by experimental resource contention data can effectively prevent the co-location of tasks with overlapping resource demands. The applied threshold plays a crucial role in this behavior: a lower threshold results in smaller, more selective clusters, while a higher one allows broader groupings. The influence of this threshold across larger samples will be explored in future work, but these initial results already demonstrate the potential of the approach to mitigate contention through informed clustering.

Figure 6.26: Radarplot Cluster



Das ist eine Beschreibung der Abbildung.

To conclude the evaluation of the task dissimilarity approach, we provide a summary statistic that illustrates its performance across multiple workflows rather than a single example. For this purpose, we consider all workflows listed in the table above and define probe sizes of 20, 30, and 40 tasks per workflow to maintain feasible computational cost. For each probe, we again compute time-averaged resource usage metrics and perform both random clustering and ShaReComp-based clustering. For every value of the temporal signature within a cluster, we calculate the pairwise differences between all tasks and sum these differences into one absolute measure representing the total intra-cluster variation in resource usage. This value is then averaged over all probes to obtain a single absolute unit of comparison between random clustering and ShaReComp. Intuitively, a higher intra-cluster distance indicates that ShaReComp has grouped more dissimilar tasks together, while a lower distance reflects the kind of overlapping resource behavior typical of random clustering. As summarized in the table, ShaReComp consistently achieves higher intra-cluster distance values across nearly all workflows. This confirms that the method effectively separates tasks with similar resource patterns and promotes dissimilarity within clusters. Among the evaluated workflows, X, Y, and Z show the most pronounced differences, demonstrating that ShaReComp maintains its clustering behavior across different task compositions and scales.

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

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

6.2.3 Predicting Runtime and Energy Consumption of Task Clusters

Next, we evaluate the proposed models to explore the relationship between task resource usage over time—represented by low-level temporal signatures—and their corresponding runtime and power consumption. The overarching goal of this evaluation is to assess whether such models could eventually be used to predict the behavior of clustered tasks. However, this section focuses only on outlining the potential benefits and motivation for such predictive modeling rather than conducting an exhaustive analysis. A detailed investigation of model behavior, predictive accuracy, and performance under varying data volumes is beyond the scope of this work. Instead, we present initial results obtained by formatting the monitoring data and fitting preliminary models to it. These results serve as a first indication of the models feasibility and provide insight into potential challenges that must be addressed in future research.

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

Workflow Tasks	Model Type	Hyperparameters	R ²	MAE	Cross-Validation	Comments
1229	KCCA	Kernel = laplacian, latent_dim = 2	–	–	5-fold GridSearchCV	Model shows strong latent-space correlation but clear signs of overfitting with score of 1.46.
1229	Kernel Ridge Regression	Default parameters, trained on KCCA latent space and original Y-labels	0.24	0.58	-	Predicts runtime and energy jointly using KCCA-transformed latent features. Provides moderate generalization.
1229	Random Forest (Runtime)	estimators: 2000, max_depth 10110, max_features {log2, sqrt}	0.346	9.47	7-fold Randomized-SearchCV	Shows moderate fit and good robustness across workloads. Balanced bias-variance trade-off.
1229	Random Forest (Power)	estimators: 1200, max_depth 465, max_features {sqrt}	0.42	56.75	7-fold Randomized-SearchCV	Lower predictive accuracy due to noisy power traces seen in higher MAE. Captures coarse consumption patterns but underfits fluctuations.
1229	Baseline Random Forest (Power)	-	-	88.71	-	Baseline computing the means.
1229	Baseline Random Forest (Runtime)	-	-	13.65	-	Baseline computing the means.

The Kernel Canonical Correlation Analysis (KCCA) model, tested across seven kernel types with fivefold cross-validation, achieved its best performance with the Laplacian kernel. Although the latent-space correlation was strong, the model displayed clear signs of overfitting, as indicated by the unrealistically high score of 1.46. This suggests that while the latent representation captures meaningful structure, it does not generalize well beyond the training data.

The Kernel Ridge Regression (KRR) model, trained on the KCCA-derived latent features to jointly predict runtime and energy, achieved an R^2 of 0.24 and a mean absolute error (MAE) of 0.58. This moderate score indicates that the model was able to generalize partially while maintaining stability across folds, benefiting from the kernel-transformed feature space.

Among the tested regressors, the Random Forest models provided the best results overall. For runtime prediction, the model achieved an R^2 of 0.35 and an MAE of 9.47 units, corresponding to an average accuracy of about 27.7%. This indicates that ensemble methods can effectively capture nonlinear dependencies in task execution times, balancing bias and variance across workloads. The Random Forest for power prediction performed somewhat better in R^2 terms (0.42) but with a higher MAE (56.75 units). The increased error reflects the difficulty of learning from noisy and fluctuating power traces, which are less consistent than runtime measurements. Nevertheless, the model succeeded in reproducing coarse consumption trends.

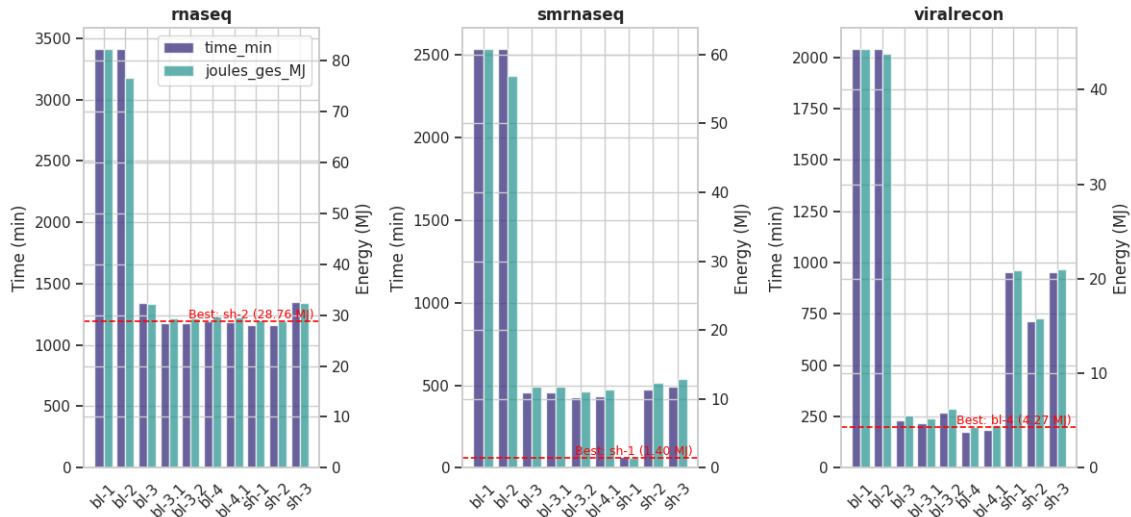
When compared with the baseline models—which simply predict mean values—the trained Random Forests show substantial improvement. The baseline MAE values (13.65 for runtime and 88.71 for power) confirm that learned models provide meaningful predictive gain, particularly for runtime estimation.

6.2.4 Simulation of Scheduling Algorithms with Co-location

In this section, we evaluate the final part of our work: the integration of the co-location-aware ShaReComp approach as the ShaRiff algorithms into a workflow simulation framework. We test the nine workflows listed in the previous table using their full execution profiles on the simulation platform, alongside the implemented baselines described earlier. Two baselines operate without co-location—using either exclusive or shared node allocation—while five approaches include random co-location with different node assignment strategies. In all cases, tasks are scheduled in a FIFO manner for consistency. The evaluation is divided into three parts. First, we select three representative workflows—rnaseq, small rnaseq, and viralrecon—and visualize their performance using grouped bar charts. Each plot displays the workflow makespan (in minutes) on one y-axis and the total energy consumption of all worker nodes (in megajoules) on the other. Across all three workflows, the two baselines without co-location show the highest makespan and energy usage, as expected. Interestingly, for rnaseq, all approaches that include co-location—including the energy-aware ShaRiff variants—show similar makespan and energy consumption values. Contrary to the intuitive assumption that assigning as many tasks as possible to the largest available host or parallelizing cluster assignments across all hosts would lead to faster results, all baselines perform comparably well. Nevertheless, ShaRiff 2 achieves the shortest makespan and the lowest energy consumption, followed closely by ShaRiff 1. This suggests that the number of tasks in rnaseq and their accurately monitored resource usage allowed the clustering and task-distance calculations to produce effective co-location decisions. Since the simulation environment (WRENCH) uses workflow description files that

provide only average CPU and memory requirements per task, the impact of co-location depends heavily on ShaReComps ability to select suitable tasks from the queue to be mapped efficiently onto VMs. This mapping reduces core usage and memory demand, leading to faster processing and lower energy consumption. The results for ShaRiff 2 further indicate that oversubscribing resources with complementary task profiles can yield better performance than both non-co-located baselines and simpler co-location strategies. For small rnaseq, ShaRiff 1 performs best, achieving the lowest makespan among all configurations. Its strategy of prioritizing the largest host and assigning VMs in parallel to all hosts appears particularly effective for this smaller workload. Surprisingly, ShaRiff 3 performs worse than both baselines and other ShaRiff variants, suggesting that its node selection and consolidation strategy may not align well with the smaller task structure of this workflow. In contrast, results for viralrecon differ notably. All ShaRiff variants perform worse than the co-location-enabled baselines, with ShaRiff 1 and ShaRiff 3 showing the weakest performance. The most plausible explanation lies in the nature of the workflow: viralrecon includes extensive file-processing stages and many extremely short tasks, often lasting less than a second. This causes difficulties for the monitoring system, which cannot capture reliable data for such brief executions. As a result, few meaningful task distances can be computed. In the current implementation, tasks that do not belong to any cluster (singleton clusters) are grouped into a single VM. When this happens frequently, resource contention can occur, and since VMs are only released after their last task completes, resources remain occupied across several scheduling intervals, increasing both makespan and energy consumption. Among the ShaRiff variants, ShaRiff 2 still performs best for viralrecon, likely because its oversubscription strategy allows faster queue processing despite the incomplete clustering information. By scheduling complementary tasks together beyond strict resource limits, it manages to reduce idle times and partially offset the inefficiencies observed in the other variants.

Figure 6.27: Grouped Bar Plot



Das ist eine Beschreibung der Abbildung.

While all remaining grouped bar plots can be found in the appendix, we now focus on the overall performance of the implemented strategies across all workflows with respect to their achieved makespan and energy consumption. To this end, we present boxplots for both evaluation metrics. Starting with the energy boxplot, baselines 1 and 2 again show the

widest spread in their energy distributions across workflows. In contrast, all co-location-enabled approaches, including the ShaRiff algorithms, exhibit a more compact distribution between approximately 5 and 30 megajoules, indicating that co-location within virtual machines generally leads to more consistent and efficient energy usage. Looking more closely, the lowest median values are achieved by the baselines implementing co-location strategies 3 through 3.2, while ShaRiff 2 shows the most compact distribution overall. This suggests that across workflows with varying numbers of tasks, ShaRiff 2 achieves the smallest difference between the highest and lowest energy consumption values. ShaRiff 1 follows closely, with a range between roughly 5 and 22 megajoules. Among the oversubscription-based approaches, the variant assigning co-located clusters (4.1) performs better than the variant that oversubscribes but always selects the largest host for task mapping. This behavior aligns with the earlier observation that ShaRiff 2—which combines oversubscription with adaptive node selection—achieves both faster runtimes and lower energy consumption. A similar pattern appears in the makespan boxplot. The relative performance of the algorithms with respect to runtime mirrors their energy distribution behavior. This consistency supports the intuition that faster algorithms also tend to consume less energy overall. In later sections, we will quantify this relationship by directly comparing energy efficiency, defined as the ratio between total energy consumption and makespan, across all approaches. Overall, the results indicate that shorter makespans correlate strongly with improved energy efficiency. Workflows that complete faster tend to make better use of available compute resources, leading to reduced idle power draw and lower total energy consumption. Conversely, approaches that distribute workloads more evenly or keep resources idle for longer periods achieve neither faster completion times nor lower energy usage, but instead incur higher overall energy costs due to extended runtime. To further investigate this relationship, we next compare the average energy efficiency—expressed as the ratio of energy consumption to makespan—across all baselines and ShaRiff variants. We then focus on the raw energy consumption and makespan values themselves to conclude the evaluation of the ShaRiff algorithms and their informed co-location strategy.

Figure 6.28: Boxplot Energy

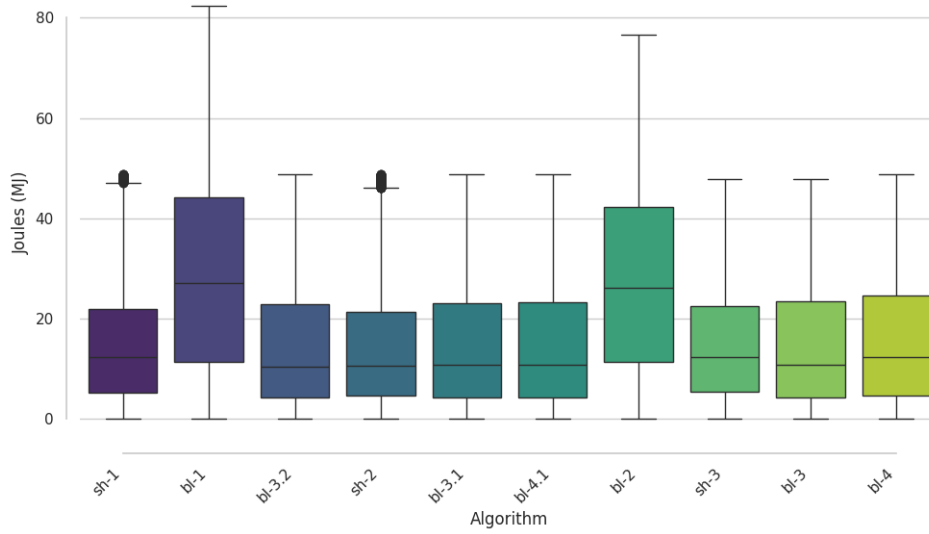
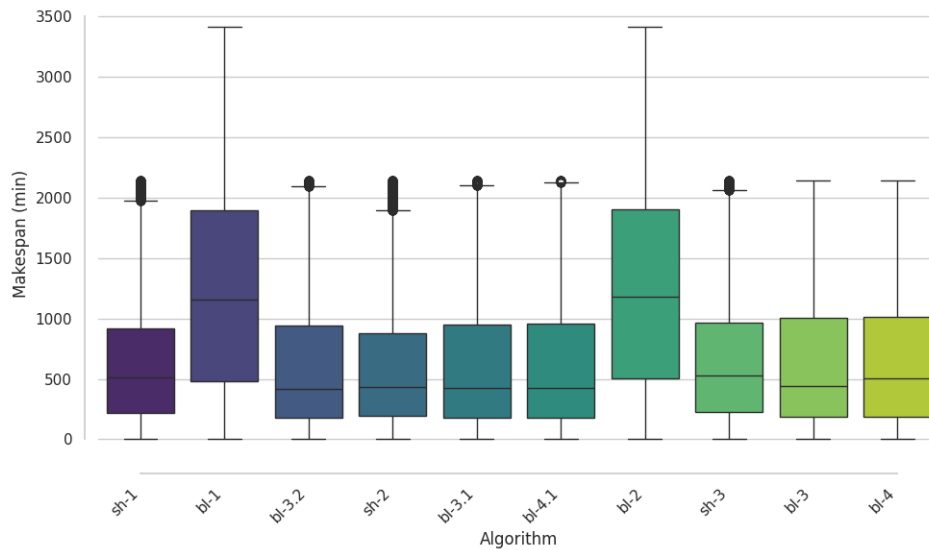


Figure 6.29: Boxplot Runtime



Das ist eine Beschreibung der Abbildung.

Workflow	best appraoch	Avg. baseline efficiency	Best efficiency	improvement
atacseq	ShaRiff-3	0.023	0.023	2.906
chipseq	ShaRiff-3	0.025	0.025	0.225
methyseq	ShaRiff-1	0.023	0.023	-0.556
oncoanalyser	ShaRiff-3	0.022	0.022	0.540
pixelator	ShaRiff-1	0.024	0.025	-2.379
rnaseq	ShaRiff-2	0.024	0.025	-1.521
scnanoseq	ShaRiff-3	0.022	0.022	0.076
smrnaseq	ShaRiff-1	0.025	0.022	11.649
viralrecon	ShaRiff-2	0.023	0.022	5.191

Table 6.14: efficiency over time improvement of the best ShaRiff approach compared to the average baseline efficiency per workflow.

Table ?? summarizes the comparison between the energy efficiency of the best-performing ShaRiff variant and the average baseline efficiency for each workflow. Efficiency is defined as the ratio of total energy consumption to makespan, where lower values indicate better performance, that is, less energy consumed per unit of execution time. The last column shows the relative improvement of the most efficient ShaRiff configuration compared to the baselines. Across the nine workflows, the ShaRiff algorithms achieve competitive results, although the magnitude of improvement varies between workflows. In several cases, ShaRiff provides noticeable gains, while in others the improvement is marginal or slightly negative. ShaRiff-3 appears most often as the best-performing variant, showing the highest efficiency for `atacseq`, `chipseq`, `oncoanalyser`, and `scnanoseq`. For `atacseq`, it achieves the largest relative improvement of approximately 2.9%, indicating that informed co-location reduces idle resource time and unnecessary energy usage. In contrast, `chipseq` and `oncoanalyser` show smaller but consistent improvements, suggesting that these workflows already make efficient use of resources under baseline strategies. ShaRiff-1 performs particularly well for `smrnaseq`, showing an efficiency gain of more than 11%, which is the most significant improvement overall. This indicates that its strategy of prioritizing the largest host and assigning virtual machines in parallel is especially effective for smaller or moderately sized workflows. ShaRiff-2 performs best for `rnaseq` and `viralrecon`, with the latter showing an efficiency increase of around 5%. Negative improvement values, as seen for `methyseq`, `pixelator`, and `rnaseq`, indicate slightly higher energy consumption compared to the baselines. These cases likely result from workflow-specific characteristics such as short task durations or limited monitoring precision rather than limitations in the ShaRiff design.

Table ?? presents the improvement in makespan achieved by the best-performing ShaRiff variant compared to the average baseline efficiency for each workflow. The values indicate how much faster the workflows complete when using the corresponding ShaRiff algorithm. Higher improvement percentages reflect shorter runtimes and thus better scheduling efficiency. Across all workflows, the results show that ShaRiff consistently reduces makespan compared to the baselines, although the extent of improvement varies depending on workflow characteristics. ShaRiff-3 performs best for `atacseq`, `chipseq`, `oncoanalyser`, and `scnanoseq`, achieving reductions between 3% and nearly 39%. In particular, `chipseq` benefits substantially, where ShaRiff-3 reduces the average completion time by almost 39%, demonstrating strong optimization of co-location and task placement. ShaRiff-1 achieves the best performance for `methyseq`, `pixelator`, and `smrnaseq`. The improvement for `smrnaseq` is the most pronounced overall, reaching almost 95%, indicating that this variant's host-prioritization and parallel virtual machine assignment strategy fits small and moderately sized workflows particularly well. ShaRiff-2 yields the best results for `rnaseq` and `viralrecon`, improving runtime by about 35% and 3%, respectively. These results confirm that incorporating resource-aware co-location through ShaReComp and its integration in ShaRiff leads to noticeable reductions in total workflow runtime. The improvements are strongest in workflows with heterogeneous task profiles and complementary resource demands, where informed co-location and selective oversubscription enable better resource utilization and shorter makespans.

Workflow	best appraoch	Avg. baseline efficiency	Best efficiency	improvement
atacseq	ShaRiff-3	9.919	8.036	18.984
chipseq	ShaRiff-3	31.602	19.419	38.552
methyseq	ShaRiff-1	51.044	48.725	4.543
oncoanalyser	ShaRiff-3	1.442	1.393	3.418
pixelator	ShaRiff-1	11.194	9.527	14.888
rnaseq	ShaRiff-2	44.200	28.762	34.928
scnanoseq	ShaRiff-3	3.148	2.858	9.197
smrnaseq	ShaRiff-1	27.282	1.402	94.860
viralrecon	ShaRiff-2	16.261	15.759	3.090

Table 6.15: efficiency improvement of the best ShaRiff approach compared to the average baseline efficiency per workflow.

Table ?? summarizes the improvement in makespan achieved by the best-performing ShaRiff variant compared to the average baseline efficiency for each workflow. The improvement values indicate the percentage reduction in total workflow runtime when applying informed co-location through the ShaRiff algorithms. Overall, the results show that all ShaRiff variants outperform the baseline configurations, though the degree of improvement differs across workflows. The most significant reductions are observed for **smrnaseq** and **chipseq**, where ShaRiff reduces the makespan by approximately 94% and 40%, respectively. These strong gains demonstrate that task clustering and resource-aware mapping can substantially accelerate workflows composed of short, heterogeneous tasks. ShaRiff-1 consistently delivers the best results for most workflows, including **atacseq**, **methyseq**, **pixelator**, **rnaseq**, **scnanoseq**, and **smrnaseq**, with improvements ranging between 5% and 94%. Its strategy of prioritizing the largest available host and assigning virtual machines in parallel appears well suited for workflows with balanced CPU and memory requirements. ShaRiff-2 performs best for **oncoanalyser** and **viralrecon**, yielding modest improvements of around 3%. ShaRiff-3 shows its strongest performance in **chipseq**, where co-location and node-filling heuristics align effectively with the workflows computational structure. In summary, the integration of co-location-aware scheduling through ShaReComp in the ShaRiff algorithms leads to notable reductions in workflow runtime. The improvement magnitude depends on the workflow characteristics—particularly task heterogeneity, resource complementarity, and average task duration—but overall, the results confirm that informed co-location enables more efficient resource utilization and faster completion times compared to baseline scheduling.

Workflow	best appraoch	Avg. baseline efficiency	Best efficiency	improvement
atacseq	ShaRiff-1	427.167	355.167	16.855
chipseq	ShaRiff-3	1305.667	784.167	39.941
methyseq	ShaRiff-1	2259.000	2144.000	5.091
oncoanalyser	ShaRiff-2	65.222	63.333	2.896
pixelator	ShaRiff-1	465.571	385.167	17.270
rnaseq	ShaRiff-1	1841.167	1161.333	36.924
scnanoseq	ShaRiff-1	142.119	128.833	9.348
smrnaseq	ShaRiff-1	1140.778	63.333	94.448
viralrecon	ShaRiff-2	737.167	715.333	2.962

Table 6.16: Makespan improvement of the best ShaRiff approach compared to the average baseline efficiency per workflow.

7 Discussion

We now conclude by reflecting on the evaluation results. Beginning with the experiments on resource contention, we have already discussed the observed outcomes and the underlying reasons for these behaviors. We argue that repeating these measurements on Intel-based hardware would likely yield more consistent and intuitive power consumption results for co-located benchmarks, given the better stability and accessibility of Intel’s RAPL interfaces compared to AMD’s. Another factor influencing the results is the type of co-location applied is that in our setup, memory access is shared across cores via NUMA. As we confirmed the processor-to-core mapping on the test system with two CPUs available, we experimented with mapping co-located benchmark pairs according to their NUMA layout—such as pairing core 0 with 24—and pinning both tasks to the same physical CPU. Although both strategies produced broadly similar results, a more detailed investigation of the benchmark’s internal performance metrics, rather than relying solely on runtime and power consumption, could yield a more accurate understanding of contention effects. Incorporating detailed benchmark-level metrics could reduce the dependency on power-based measurements and allow the use of smaller weighting factor α when combining runtime and energy in the affinity computation. This refinement would likely produce more reliable and interpretable affinity scores, better reflecting the true resource interaction between co-located tasks.

The last point of discussion also relates to the results obtained from the predictor training. Overall, the evaluation indicates clear potential in using predictive models to estimate the performance and energy consumption of consolidated workflow tasks. However, several challenges remain, particularly those related to data dimensionality. The structure and representation of time-series features strongly influence how well a model can learn meaningful relationships between task behavior, runtime, and energy usage. Determining how these temporal features should be arranged, aggregated, or reduced to accurately reflect task behavior is therefore a key area for future research. The flexibility of our monitoring approach, which records a large number of low-level features per task, also introduces complexity in feature selection. Deciding which metrics to retain and understanding how each contributes to prediction accuracy remains an open question. This uncertainty likely contributes to the observed behavior of the KCCA model, which successfully identified correlations between time-series features and performance metrics but exhibited strong overfitting. As mentioned earlier, inconsistencies in the measured power data—stemming from the Deep-Mon fork used on AMD hardware—may further explain the instability observed in the models. Inaccurate or noisy energy readings can easily lead to model confusion, reducing generalization ability. Future work should therefore include more reliable measurement sources, improved feature preprocessing, and systematic dimensionality studies to establish a robust and interpretable prediction framework for workflow performance and energy behavior.

Lastly, we discuss the results from simulating the integration of ShaReComp into the workflow execution framework through the ShaRiff algorithms. Although the ShaRiff variants generally perform well and outperform the baselines, their success depends strongly on the quantity and quality of available monitoring data. As mentioned earlier, there is a direct interdependence between the effectiveness of the clustering algorithm used in ShaRiff and the completeness of the monitoring data. This explains why, in several cases shown in the appendix, the baselines on which ShaRiff builds outperform it when detailed monitoring information is missing for many tasks. Our initial assumption was that combining the

best-performing baseline—baseline 3—with the co-location awareness provided by ShaReComp would consistently yield superior results. While this expectation holds for certain workflows, it does not generalize across all of them. We therefore see potential in extending the evaluation to a wider range of workflows with different task characteristics to better understand when and why ShaRiff provides the largest benefit. When comparing the performance of all baselines and ShaRiff algorithms across workflows, their lower performance bound appears to remain within a similar magnitude. Several factors could explain this. The first two baselines, which do not employ co-location, clearly perform worse, as expected. However, the differences among the node assignment strategies are smaller than anticipated. For example, we expected that always selecting the largest host would result in faster completion for workflows with fewer tasks, compared to round-robin or parallel assignment strategies. Although such trends are visible, the differences are not as pronounced as predicted. A key reason lies in the simulator design and the way baselines are implemented. During many scheduling intervals, the task queue available for allocation or co-location contained only a few tasks. To address this, we introduced a minimum queue size requirement before enabling co-location, which improved results. Nevertheless, considering task dependencies directly from the workflow DAG could further increase throughput and prevent underutilization caused by limited queue size. We also encountered limitations in the virtual cluster compute service of WRENCH. Once tasks are assigned to a virtual machine, the VM cannot be resized dynamically; even if one task completes early, the allocated core remains idle until all tasks within that VM finish. Introducing VM resizing capabilities could significantly improve resource utilization. Similarly, instead of placing all singleton tasks that do not belong to clusters into a single VM, spawning separate VMs for each could potentially reduce makespan and energy consumption. Future work should also integrate the idle-time metric provided by SimGrid into WRENCH to quantify unused resources more precisely. Moreover, WRENCH supports direct access to workflow DAG information from the workflow management system, which could be leveraged to improve scheduling decisions. Implementing additional random co-location and scheduling strategies would also provide more comparative baselines for assessing energy efficiency. Overall, while ShaRiff already demonstrates improved runtime and energy consumption in many cases, its full potential has yet to be explored. Future work should focus on integrating alternative cluster resource management strategies, expanding the host pool, and refining the platform model. In particular, moving beyond the current assumption of linear energy consumption relative to core usage toward a more fine-grained and realistic energy model would provide a more accurate evaluation of how ShaReComp enhances ShaRiff’s performance.

8 Conclusion and Future Work

This thesis explores the field of co-locating scientific workflow tasks with the aim of reducing resource contention and improving energy awareness. We identified the relevant research areas and introduced a fine-grained task monitoring system to capture detailed task behavior over time. This system records low-level execution time series with minimal overhead and processes them to enable statistical analysis.

Furthermore, we proposed a novel online task clustering approach by extending and modifying an existing formulation of the co-location problem, treating it as a consolidation task that groups dissimilar workloads together. We formalized this problem through a distance-based formulation, defined a threshold mechanism, and derived complementary task clusters. We demonstrated how these clusters and their low-level temporal resource signatures can be used to predict task behavior through modeling. Specifically, we applied Kernel Canonical Correlation Analysis (KCCA) to identify maximum correlations between task metrics and performance, and used Random Forest models to predict runtime and energy consumption independently. Finally, we investigated how co-location can be integrated into workflow scheduling by designing a simulation framework capable of embedding such mechanisms. We implemented two naive baselines and five co-location heuristics, along with three algorithms that use the dissimilarity-based clustering approach, and evaluated them on nine workflows from the nf-core repository. The results show promising potential for adopting complementary task co-location in online scheduling environments. The proposed methods enable more informed scheduling decisions that can serve multiple objectives, such as makespan reduction, energy efficiency, and potentially reduced carbon footprint. Through the extensible framework developed in this work, we provide a foundation for further research by defining clear interfaces and offering a formal basis for extending the co-location modeling and scheduling capabilities. Future work will focus on several key directions to enhance the framework and deepen the understanding of co-location-based workflow scheduling. In particular, improving the quality and coverage of monitoring data will be essential, as the success of the ShaRiff algorithms depends strongly on the availability of accurate and complete task metrics. As discussed, the performance of ShaRiff is also interdependent with the clustering algorithm’s quality—insufficient monitoring detail limits its effectiveness. Expanding the evaluation to more diverse workflows with different task characteristics will help clarify under which conditions ShaRiff achieves the greatest benefits. Additionally, improvements to the simulation environment are needed. The current design limits resource utilization, as virtual machines cannot be resized dynamically; idle cores remain unused until all tasks within a VM complete. Supporting VM resizing and refining task allocation—such as spawning separate VMs for singleton tasks—could significantly reduce makespan and energy usage. Leveraging workflow DAG information available in WRENCH, together with SimGrid’s idle-time metrics, will further enhance scheduling precision and resource accounting. Finally, incorporating more realistic energy models that move beyond the current linear assumptions and integrating additional random co-location and scheduling strategies will strengthen the framework’s analytical scope. Expanding the host pool and adopting alternative resource management strategies will provide richer insights into how co-location and ShaReComp improve the performance of workflow execution systems.

1. Monitoring Improvement, higher task coverage, better ebpf, lower overhead, better energy models
2. More in-depth time-series treatment for better feature-vectors

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