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, 29. Oktober 2025

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

KI-Disclaimer

Hiermit versichere ich, dass ich in der vorliegenden Arbeit ChatGPT 5, Version 1.2025.168 und DeepL Translator for Mac ausschließlich zur Verbesserung der sprachlichen Qualität verwendet habe. Die inhaltlichen Leistungen sind meine eigenen.

Die Satzung zur Sicherung guter wissenschaftlicher Praxis an der TU Berlin vom 8. März 2017[1] habe ich zur Kenntnis genommen.

(Unterschrift) Niklas Fomin, Berlin, October 29, 2025

[1] `https://www.static.tu.berlin/fileadmin/www/10000060/FSC/Promotion___Habilitation/Dokumente/Grundsaetze_gute_wissenschaftliche_Praxis_2017.pdf`

Acknowledgements

I would like to first and foremost thank my wife Jenny who not only supported me during the process of writing this thesis but guided me through a time of personal and professional change. I am thankful for a family that supported my decisions and gave me a place to rest and recalibrate. Through my friends I was able to endure this process. Special thanks to my supervisor Jonathan Bader for his caring guidance, that exceeded the scope of conducting this thesis.

Abstract

The combined contribution of processing units and memory devices to a data center's total energy consumption typically ranges between 70% and 80%. When scientific workflows are run in HPC data centers due to their parallelized execution multiple tasks can be co-located. Co-location can occur either on the same physical node or within the same virtual container which leads to sharing of the underlying compute and memory resources. A high level of resource sharing slows down the processing duration and increases the energy consumption compared to when executing them exclusively since they don't interfer with each other. Therefore, if possible workflow tasks with a high shared resource profile should be allocated to distinct compute resources to avoid contention, thus avoiding slowdown. In this thesis we present *ShaReComp*, a multi-staged approach that characterizes tasks in terms of temporal signatures and applies dissimilarity-based clustering to group tasks with complementary resource usage. We integrate *ShaReComp* into simulated workflow execution where we implement 4 co-location aware scheduling algorithms alongside co-location baselines, that we call *ShaRiff*. Across nine real-world scientific workflows, *ShaReComp* generally achieves higher intra-cluster dissimilarity than random clustering, confirming its ability to group tasks with complementary resource usage. When integrated into *ShaRiff* scheduling simulation, this approach reduced overall energy consumption by up to 18% and makespan by up to 15% compared to random co-location, demonstrating the effectiveness of informed, energy-aware task consolidation.

Kurzfassung

Der kombinierte Anteil von Prozessoren und Speichergeräten am Gesamtenergieverbrauch eines Rechenzentrums liegt in der Regel zwischen 70% und 80%. Wenn Datenanalyse-Workflows in HPC-Rechenzentren gestartet werden, können aufgrund ihres parallelisierten Ablaufs mehrere Analyse-Jobs gemeinsam ausgeführt werden. Die Zusammenlegung der Jobs kann entweder auf demselben physischen Server oder innerhalb desselben virtuellen Containers erfolgen, was zu einer gemeinsamen Nutzung der zugrunde liegenden Rechen- und Speicherressourcen führt. Eine hohe gemeinsame Nutzung von Ressourcen verlangsamt die Verarbeitungsdauer und erhöht den Energieverbrauch im Vergleich zur exklusiven Ausführung, da sich die Jobs gegenseitig beeinträchtigen. Daher sollten Workflow-Jobs mit einer hohen gemeinsamen Ressourcennutzung nach Möglichkeit verschiedenen Rechenressourcen zugewiesen werden, um Konflikte und damit Verlangsamungen zu vermeiden. In dieser Arbeit stellen wir *ShaReComp* vor, einen mehrstufigen Ansatz, der Workflow-Jobs anhand zeitlicher Verhaltens-Signaturen charakterisiert und eine auf Unähnlichkeit basierende Clusterbildung anwendet, um Jobs mit komplementärer Ressourcennutzung zu gruppieren. Wir implementieren die Umsetzung der Job Ko-lokations Methode *ShaReComp* in eine simulierte Workflow-Umgebung, in der wir neben algorithmischen Vergleichsgrundlagen vier ko-lokations-bewusste Scheduling-Algorithmen entwerfen, die wir *ShaRiff* nennen. In neun realen Datenanalyse-Workflows erzielt *ShaReComp* insgesamt eine höhere Intra-Cluster-Unähnlichkeit als zufälliges Clustering, was die rechnerische Fähigkeit bestätigt, Workflow-Jobs mit komplementärer Ressourcennutzung zu gruppieren. Bei der Integration in die *ShaRiff*-Simulation reduziert dieser Ansatz den Gesamtenergieverbrauch um bis zu 18% und die Gesamtausführungszeit um bis zu 15% im Vergleich zur zufälligen Kolokation, was die Wirksamkeit einer informierten, energie-effizienten Job-Konsolidierung demonstriert.

Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

1.1 Problem Motivation & Description

The rapid growth in data center power consumption has emerged as a pressing societal issue. As of March 2024, global statistics report more than 9,000 operational data centers worldwide, with this number continuing to rise steadily. Consequently, total annual energy consumption across data centers has been roughly doubling every four years with some now reaching capacities of up to 500 MW, comparable to the electricity demand of a city of one million residents [49]. 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 [58]. HPC data centers 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 does not only arise from computation but also from networking, cooling, and auxiliary equipment and makes them major consumers of electricity and significant contributors to climate change [57].

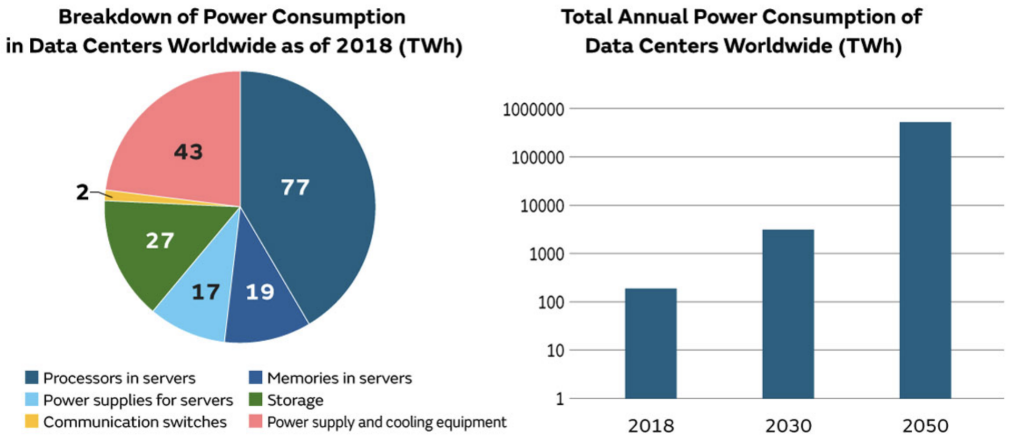

Figure 1.1: Breakdown of power consumption in data centers
Breakdown of power consumption in data centers and future outlook for power consumption.

A key driver of this trend is the exponential growth in data collection, storage, and processing across scientific disciplines. 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 [23]. 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. Nevertheless, practitioners face challenges in assessing which approaches for mitigation are most applicable and effective with concerns to multiple relevant objectives [59].

Arising from the demand for optimization, the central objective of this work is the energy-aware execution of scientific workflows. Tasks within data-driven workflows often appear in multiple instances and may vary substantially depending on input data, parameters,

or execution environments. As a consequence their patterns in resource demand, energy consumption result in fluctuating carbon emissions and runtimes rather than constant behavior. By investigating task behavior during parallel execution on shared resources we address an important aspect of workflow research that has received limited attention during the last years [43] [22] [25]. We formulate our approach by the need for continuous monitoring to enable fine-grained, time-dependent task models that accurately capture resource usage patterns. The resulting research task translates into using task time-series data to enhance the co-location process-being the intermediate stage between task-to-machine mapping and scheduling decisions during workflow execution. As a result, we expect that making informed decisions about which tasks should share resources can lead to more balanced performance, improved resource utilization, and greater energy efficiency. While related concepts have been explored in the context of co-scheduling at the operating system level or in cloud environments where batch jobs run alongside latency-critical services, the online co-location problem addressed here differs. In our setting, decisions about which tasks to execute together on the same virtual machine must be made dynamically at each scheduling interval, requiring continuous adaptation to the current workflow state and resource conditions. The importance of such decisions can be illustrated through an example by [12]. Consider two compute-bound programs (A1, A2) and two memory-bound programs (B1, B2) to be co-scheduled across two nodes. If both compute-bound tasks share one node while both memory-bound tasks share the other, the memory-intensive programs will heavily compete for bandwidth, causing severe slowdowns and doubling their execution time. In contrast, pairing each compute-bound task with a memory-bound one allows both to utilize different resources efficiently, resulting in no slowdown at all. This demonstrates two fundamental principles of co-location: (i) tasks that heavily use the same resource should not be co-located, and (ii) complementary resource usage—such as pairing CPU-intensive and memory-intensive workloads—can maximize system performance. Furthermore, previous studies have shown that effective co-scheduling of this kind can significantly enhance both energy efficiency and system throughput in high-performance computing environments. In the best cases, runtime can be reduced by up to 28% and total energy consumption by approximately 12% compared to isolated, dedicated execution. However, the extent of improvement depends strongly on the diversity and ratio of jobs in the scheduling queue [15]. To address this, we propose a hierarchical clustering approach that groups tasks by dissimilarity to encourage complementary co-location. Furthermore, we conduct experiments to verify the findings by [12]. Lastly, our co-location mechanism is integrated into a workflow simulation framework, where several scheduling algorithms are implemented to demonstrate how informed, runtime co-location decisions can enhance task mapping and overall workflow efficiency.

These challenges motivate the central goal of this work: modelling the complementary co-location of scientific workflow tasks using fine-grained, time-dependent resource usage profiles and embedding it into workflow execution for energy and performance aware decision-making at runtime.

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 leading to lower overall energy consumption?

RQ3 How can energy-aware co-location models be integrated into the workflow execution phase?

The resulting core contributions of this thesis are:

- The implementation of a monitoring client, capable of serving the relevant monitoring layers for scientific workflow execution by collecting fine-grained, time-dependent resource usage data for workflow tasks.
- The design and further development of a task co-location approach that utilizes time series data to compute for any given set of tasks clusters with complementary resource usage patterns.
- The development of novel scheduling algorithms that integrate the proposed co-location approach into the workflow execution phase.
- The conceptual implementation of a scheduler, able to predict the performance and energy consumption of co-located tasks using two machine learning models.
- The evaluation of the co-location approach in simulation using 9 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 performance monitoring, scientific workflow systems, the co-location problem, cluster resource management and machine learning. Chapter 3 introduces related work within the domain of optimizing Scientific Workflow execution and main objectives in line with these of this work. Furthermore, Chapter 4 highlights the approach to the problem in both a theoretical and practical manner and ultimately defines the realization of the approach. The corresponding implementation is elaborated in Chapter 5. Moreover, Chapter 6 evaluates the steps conducted to achieve the implementation of the overall approach using both empirical data and a simulation environment. Chapter 7 interprets the key findings and discusses some limitations of this work. Finally, Chapter 8 concludes this thesis and gives an outlook on the impact of this work for the future.

2 Background

This chapter provides the necessary background and concepts that are applied in Chapter 4 of this thesis. The sections will cover general concepts of High-Performance Computing and Scientific Workflows and will further elaborate on Monitoring, Co-location, and Machine Learning techniques applied in this work.

2.1 High Performance Computing

High-Performance Computing (HPC) encompasses a collection of interrelated disciplines that together aim to maximize computational capability at the limits of current technology, methodology, and application. At its core, HPC relies on specialized electronic digital machines, commonly referred to as supercomputers, to execute a wide variety of computational problems or workloads at the highest possible speed. The process of running such workloads on supercomputers is often called supercomputing and is synonymous with HPC. The fundamental purpose of HPC is to address questions that cannot be adequately solved through theory, empiricism, or conventional commercial computing systems. The scope of problems tackled by supercomputers extends beyond traditional scientific and engineering applications to include challenges in socioeconomics, the natural sciences, large-scale data management, and machine learning. An HPC application refers both to the problem being solved and to the body of code, or ordered instructions, that define its computational solution [58].

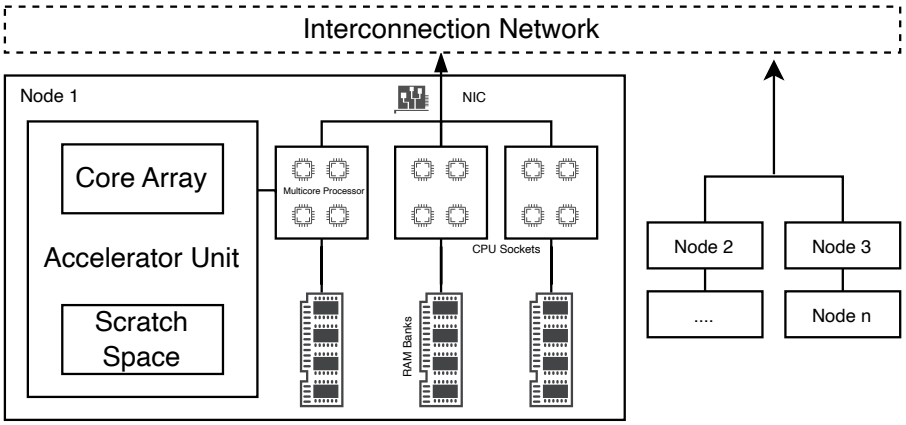

Figure 2.2: HPC Architecture
High-level Architecture of a HPC Data Center inspired by [58].

Figure 2.2 shows what distinguishes HPC systems from conventional computers is their organization, interconnectivity, and scale. According to [58] 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

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. The performance of an HPC system depends largely on the speed and coordination of its components, with processor clock frequency playing a central role. Modern 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 [58].

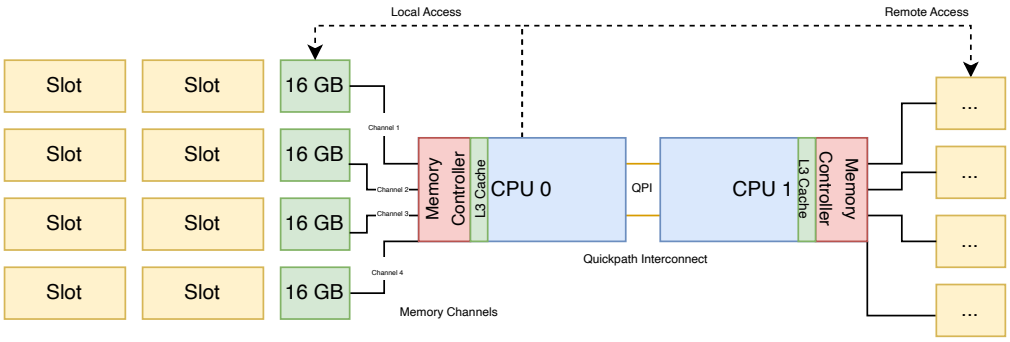

Figure 2.3: NUMA Cores
NUMA Core Architecture showing Local and Remote Memory Access.

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. Symmetric multiprocessors (SMPs) use a shared memory model accessible by all processors. 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. As visualized in 2.3, NUMA leverages fast local memory channels alongside slower global interconnects, enabling greater scalability than SMPs [52] [68].

2.1.2 Virtualization in HPC

With the ever-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 means to improve resource utilization in HPC environments by enabling 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 in deploying HPC applications [56]. 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 [27].

2.2 Scientific Workflows

2.2.1 Scientific Workflow Management Systems

Scientific Workflow Management Systems (SWMS's) enable the composition of complex workflow applications by connecting individual data processing tasks. Figure 2.4 shows an example of a SWMS architecture. 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 [59]. 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 [7].

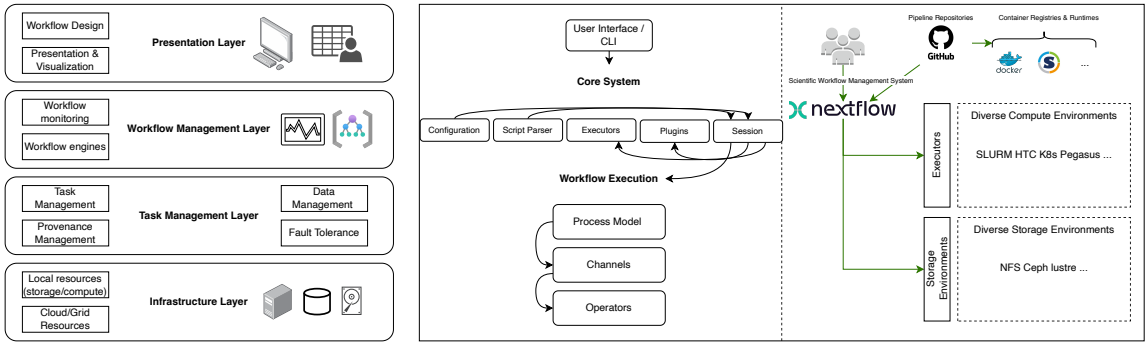

Figure 2.4: Scientific Workflow Management System
Structured Components of an exemplified Scientific Workflow Management System.

2.2.2 Scientific Workflow Tasks

Scientific workflows are compositions of sequential and concurrent data processing tasks, whose order is determined by data interdependencies. As 2.5 illustrates, 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 [18] [65]. 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 [59].

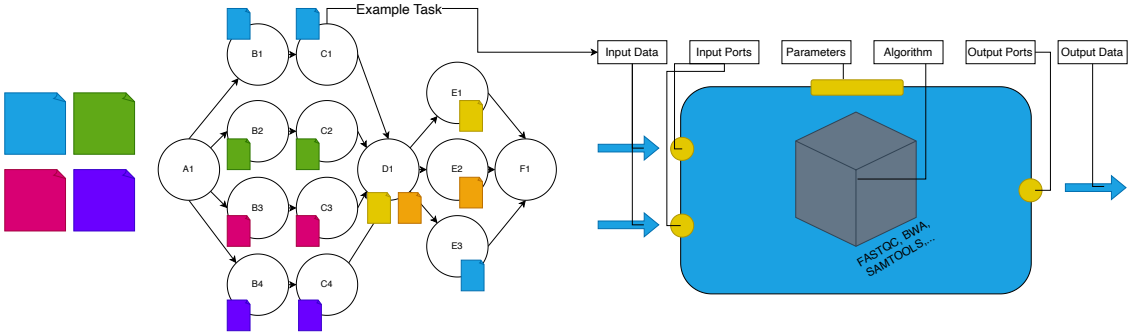

Figure 2.5: Workflow Task

A Scientific Workflow Task as part of a Workflow consuming input data and producing output data inspired by [18].

2.3 Monitoring of Scientific Workflows

Performance monitoring generally helps to prevent degradation 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 CPUs 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 [58] [65].

2.3.1 Distinct Monitoring Layers

Highl-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. 2.6 provides a high-level overview on the relevant monitoring layers identified by [7].

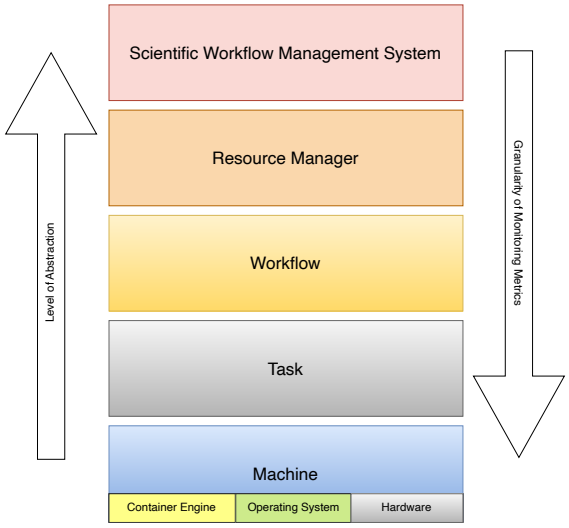

Figure 2.6: Monitoring Layers
Monitoring Layers in Scientific Workflow inspired by [7].

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 [65] [7].

To bridge this gap, [7] proposed an architectural blueprint for workflow monitoring that is 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. Higher layers provide increasingly abstract views, relying only on selected metrics from underlying layers while, in theory, being able to access all. Lower layers, as shown in Figure 2.7 show that the resource manager level aggregates metrics like task resource consumption or available machine resources. 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 [7].

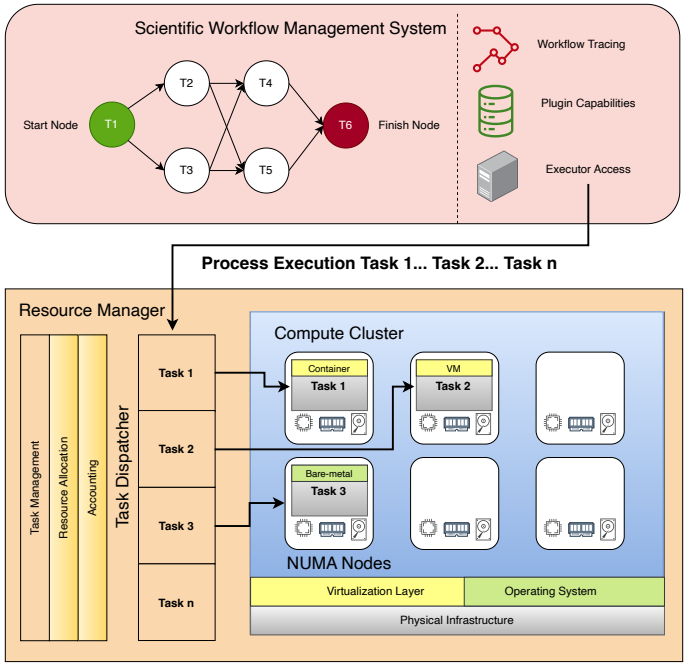

Figure 2.7: Monitoring Layers during Workflow Execution
Monitoring Layer during Workflow Execution inspired by [7].

Low-level Resource Monitoring

Low-level monitoring of 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 [65]. 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. The Berkeley Packet Filter (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. Together with fixed hardware or software counters, tracing and sampling form the backbone of observability. The BPF ecosystem provides several user-friendly front ends for tracing, most notably BCC (BPF Compiler Collection) and bpftrace [29]. Figure 2.8 illustrates common monitoring targets at the OS-level using eBPF.

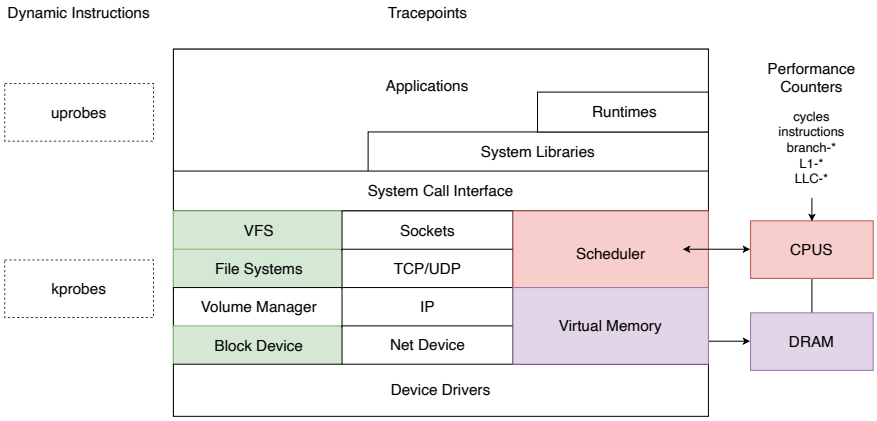

Figure 2.8: Low Level Monitoring Targets.
Low Level Monitoring Targets on the OS-Level using eBPF inspired by [29].

Low-level performance monitoring is extensively defined by [29]. 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 [29]. 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 [29].

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. 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 [29].

2.3.2 Monitoring Energy Consumption

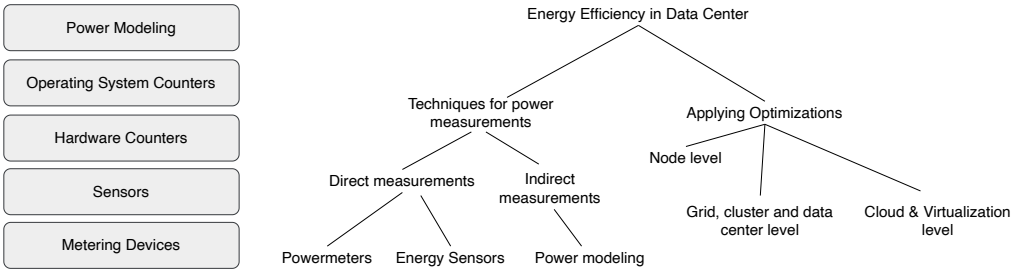

Figure 2.9: Means of Power Measurements in Data Centers
Different Levels and Means of increasing energy efficiency in data centers inspired by [35].

Conceptually shown in 2.9 the 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 [35].

The most widely used of these interfaces is Intel's Running Average Power Limit (RAPL), introduced with the Sandy Bridge architecture. RAPL, as visualized in 2.10, grants direct access to cumulative energy consumption for various hardware domains, including the CPU package, cores, integrated GPU, and DRAM. Each domain reports energy data through model-specific registers (MSRs) that can be read at millisecond-level resolution. 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 architectures, although with limited support for energy domains compared to Intel implementations [35].

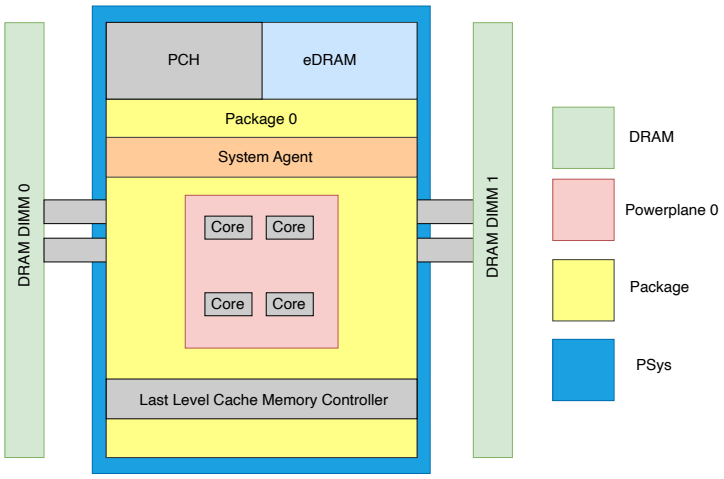

Figure 2.10: RAPL Domains
Overview of RAPL Energy Domains measured on a Processor inspired by [35].

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 [17].

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 [52]. 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. 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 [33] [36]. 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 [70]. However, consolidation is far from trivial: the resource usage of co-located 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 [70].

2.4.1 Impacts of Co-location on Resource Contention and Interference

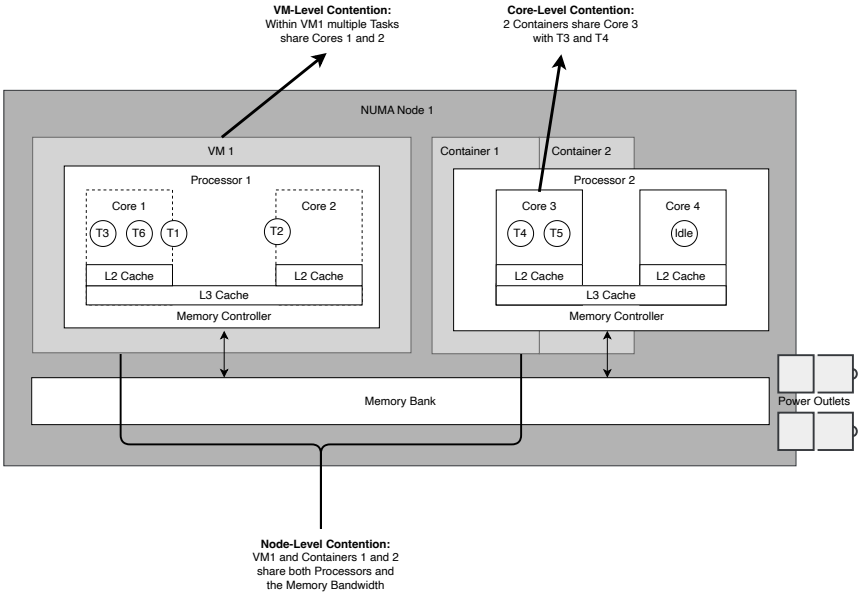

Figure 2.11: The Co-location Problem
Different Levels of Co-location on a Compute Node.

The fundamental motivation for addressing co-location in HPC and cloud environments lies in mitigating resource contention. Figure 2.11 illustrates three possible levels of contention that can occur on a single compute node. Tasks may share CPU cores within a virtual machine, multiple containers can be pinned to the same core, and even the placement of virtual machines and containers themselves introduces additional layers of co-location effects that influence overall performance and energy efficiency. When processes execute on different cores of the same server, they inevitably share hardware resources [11] [15]. Modern multi-core architectures 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. In extreme cases, memory traffic contention has been shown to cause slowdowns, where execution times more than double, making sequential execution more efficient than poorly chosen co-schedules. This contention can result in severe performance degradation, making it

critical to understand and predict the impact of co-location on application efficiency [9] [12] [60].

2.4.2 Shared Aspects and Distinctions from Scheduling and Task Mapping

Co-location differs fundamentally from scheduling and task mapping, even though all three can aim at improving performance and energy efficiency in workflow execution. Scheduling determines when each task runs, while mapping decides where each task is placed [43] [47]. In contrast, co-location focuses on how tasks interact when they are executed simultaneously on shared resources [69] [67] [24]. Scheduling and mapping operate primarily at the planning level—deciding task order and resource assignment to optimize overall throughput, energy use, or completion time [22]. 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 [70].

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 [5]. 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 [32] [13] [38]. 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 [6] [66].

2.5.2 Utilized Machine Learning Models in this Thesis

Ordinary Least Squares and Extensions

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 [73].

Kernel Canonical Correlation Analysis

Kernel Canonical Correlation Analysis (KCCA) 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. 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 [4] [70] [71].

Random Forest Regression

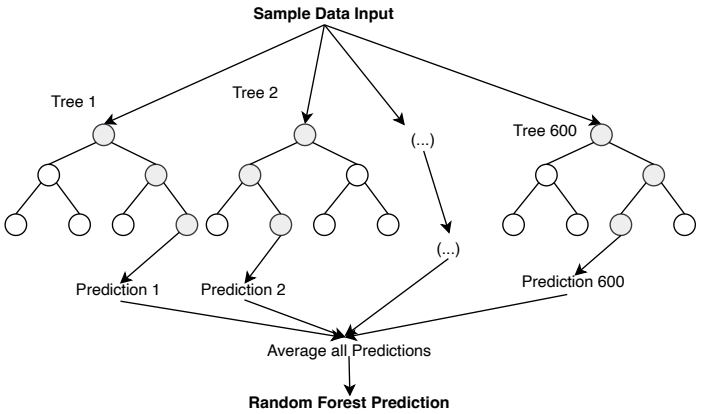

Figure 2.12: Random Forest Procedure
The iterative process of building a Random Forest for prediction.

Visualized in 2.12 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 [14] [74].

Agglomerative Clustering

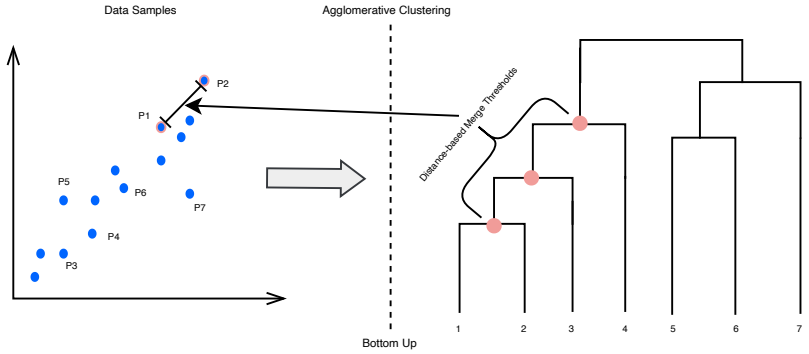

Figure 2.13: Agglomerative Clustering Procedure
Agglomerative Clustering applied on a distribution using a distance criterion.

Agglomerative clustering, also called hierarchical clustering is a family of algorithms that group data into nested clusters, represented as a tree-like structure called a dendrogram. Figure 2.13 provides an overview on the process from raw data to clusters. 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 [72].

2.6 Research-oriented Simulation of Distributed Computing

WRENCH was introduced as a simulation framework that provides accurate, scalable, and expressive experimentation capabilities. Figure 2.14 conceptualizes WRENCH's framework architecture. Building 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 sim-

ulated 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 [19].

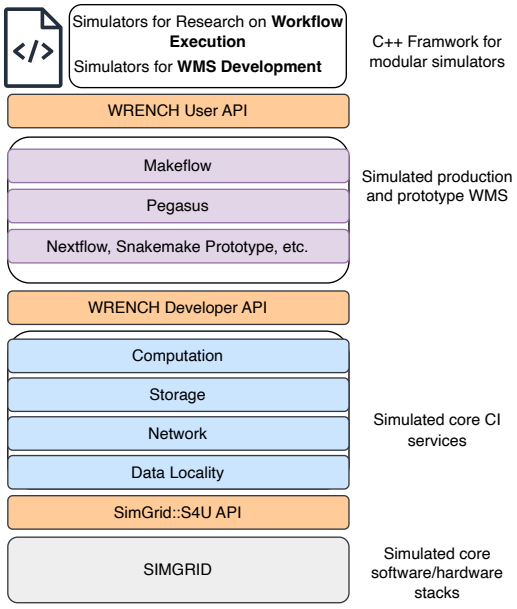

Figure 2.14: WRENCH
The WRENCH framework's layered architecture inspired by [19].

3 Related Work

This chapter presents and discusses research papers that we found to be important and related to this thesis. Additionally, we mention differences between the related work compared to ours. Firstly, in section 3.1 we introduce literature in which detailed approaches for overarching monitoring of HPC-application and specifically scientific workflows are presented. Afterwards, assessments on resource contention and co-location that served as a motivation and fundament for this work are summarized in 3.2. Lastly, task scheduling, and co-scheduling methods for energy- and performance-awareness during workflow execution are discussed in 3.3. Each work described in this chapter has contributed inspiration and conceptual outlines with influence on the approach developed in this thesis.

3.1 Monitoring of Scientific Workflows

We begin with a review on literature in the field of monitoring. The Authors of [3], [50] and [56] have their focus on HPC systems and identifying means to quantify power consumption by various components and virtualization technologies.

[3] conducted an empirical study to characterize power consumption across data center server components. They designed experiments to stress CPUs, network cards, and disks while measuring their energy usage under varying loads and frequencies. They identified optimal operational points and demonstrated that CPU power usage shows a super-linear relationship to load. Additionally, the study by [50] examine how HPC workloads consume power in real production environments by collecting data from two medium-scale clusters to capture job-level energy behavior. The work produced an open dataset and analysis framework intended to inform energy-efficient scheduling and resource management in future HPC systems. Their work helped in generally understanding energy consumption patterns in HPC systems thus bridging the knowledge gap and applying it to scientific workflows.

In [56], the authors examined the use of virtualization technologies in high-performance computing systems. They analyzed operating system-level and application-level virtualization approaches and their effects on performance and resource utilization. The paper discussed existing challenges posed by multi-user environments and customization needs in HPC and outlined directions for future development of virtualization in this context.

Shifting the focus to the use-case of scientific workflows, the works by [18] and [34] provided comprehensive reviews on workflow execution.

[18] specifically reviewed how scientific workflow systems handle parallelization in response to growing data volumes, especially in life sciences. The authors compared theoretical strategies with their practical realizations across existing systems. Their review guided us in understanding the parallel nature of workflow execution suitable for our monitoring and co-location.

Additionally, the paper by [34] focused on specifically characterizing scientific workflow tasks by choosing six scientific domains using profiling tools that capture I/O, memory, and computational behavior of workflow tasks. They identified dominant job types that consume most of the runtime and detected inefficiencies such as repeated data reads. They introduced new profiling techniques and provided workflow characterizations useful for generating realistic synthetic workflows for simulation and evaluation studies. Their insights on workflow behavior was found valuable for our work.

Papers [7] and [65] presented monitoring architectures and methods tailored for scientific workflows. The paper by [7] defines a four-layer monitoring architecture for scientific workflow executions. It motivates the need for automatic tracing of performance metrics, traces, and behavior across infrastructure, resource manager, workflow, and task layers. Available metrics available at each layer are explained with a focus on how the layers interact. It evaluates five state-of-the-art scientific workflow management systems to identify what is required to adopt the proposed four-layer approach. In our work, we use the identified layers to structure our monitoring approach while mapping specific monitoring tools instead of solely evaluating workflow management systems.

As for our empirical execution of real-world scientific workflows, we chose to use containerized pipelines. This choice imposed the necessity to specifically monitor containers. The authors in [64] analyzed the energy footprint of Docker containers under different workloads. They measured energy consumption in common containerized environments to assess performance and cost implications.

In [65], the authors developed methods to link low-level resource usage data with higher-level workflow tasks in distributed scientific workflows. They presented three approaches for detailed I/O monitoring and implemented one using eBPF. They proposed a method to associate monitoring data from system-level traces to workflow tasks, first mapping them to physical tasks and then to logical ones.

Building on the approach by [65] we leverage the system developed by [17]. They proposed DEEP-Mon, a monitoring tool that measures power consumption and attributes it to individual threads and application containers. The tool enables power usage analysis at the application level for cloud and HPC environments. They demonstrated that DEEP-Mon introduces negligible overhead on system performance and overcomes limitations of earlier monitoring solutions which made it a suitable choice for our work.

Accompanying the research by [17], [51] analyzed the use of Intel's RAPL interface for energy measurement in CPUs and examined common mistakes in existing tools. The authors revisited RAPL's underlying mechanisms and evaluated its behavior with different processor models. They explored the use of eBPF for interacting with RAPL and implemented a Rust-based solution that improves correctness and timing accuracy. The doctoral thesis by [35] investigated energy consumption in server-based computing systems across HPC, scientific, and cloud environments. The work employed Intel's RAPL to measure, model, and analyze power efficiency at both component and system levels. Multiple modeling strategies were explored, including regression, statistical, and non-linear additive approaches, validated using data center logs and EC2 instances.

3.2 Implications of Resource Contention and Mitigations

The study conducted by [30] researched on the performance impact of resource contention in multicore processors using a differential analysis approach. The authors compared MPI process bindings to isolate the effects of shared memory hierarchy resources. They evaluated benchmarks across multiple quad-core architectures. The findings provided insights into how shared hardware resources influence application performance and system efficiency in high-end computing environments.

[8], [9], [15], [60] and [16] focused on various aspects of measuring, quantifying, and mitigating resource contention in HPC systems. Even though, their focus was not specifically

on scientific workflows, their findings still hold true for data-intensive applications and can be generally summarized as effects occurring in HPC environments.

The paper by [8] addressed performance bottlenecks in modern HPC clusters arising from contention for shared resources both within and across nodes. The authors noted that current schedulers lack mechanisms to account for such interference. To mitigate this, they introduced a set of metrics designed to model shared resource contention and describe job-level utilization and communication patterns. These metrics rely on data collected from hardware performance counters and interconnect monitoring tools.

Similar to [8], [9] presented a virtualized HPC cluster framework designed to recognize and manage contention for shared resources such as caches, memory buses, and controllers. The authors aimed to improve workload performance and stability by introducing contention awareness into cluster operation, addressing a limitation in existing HPC systems.

Paper [15] examines how co-scheduling memory-bound and compute-bound applications can enhance performance and energy efficiency on supercomputers. To support this, the authors introduced auto pin+, a tool for monitoring and optimizing co-scheduled workloads. Their experiments showed that coordinated co-scheduling can reduce runtime by up to 28% and energy consumption by twelve percent relative to dedicated runs. They also proposed an adaptive scheduling strategy that adjusts to the composition of jobs in the queue.

The study from [60] explored topology-aware process-to-core mappings as a means to improve performance in co-allocated HPC workloads. The authors implemented several core enumeration strategies that enable multiple parallel applications to share compute nodes without explicit application awareness. Their experiments showed that such mappings significantly influence memory bandwidth and overall execution time. By analyzing both individual job durations and overall makespan, the work evaluated topology-aware co-allocation as a practical alternative to exclusive node allocation policies in HPC clusters.

The paper by [16] investigated job striping as an alternative to the traditional exclusive-node scheduling policy in supercomputers. By allowing pairs of jobs from different users to share nodes, the technique aimed to boost overall throughput and energy efficiency.

[63], [27] and [42] focus specifically on investigating contention effects on certain hardware resources, as well as containerized environments, which are relevant to our work.

In [63], the authors developed Contenders, a method for predicting performance loss caused by cache contention when applications share servers. Their approach profiles each application using custom micro-benchmarks to model cache usage and estimate resulting cache misses and execution times under co-scheduling. Evaluations on various application sets showed that Contenders achieves higher prediction accuracy than existing tools.

The study from the paper [27] examined interference among Docker containers running on shared Linux hosts. Through targeted experiments on different resource types, the authors observed that containers contend for shared resources and that part of the interference originates from host operating system overhead. Their findings suggest that workload-aware container placement is essential to maintain stable and predictable performance.

Both [42] and [55] are concerned with slowdown effect occurring when co-locating workloads in virtualized environments. By introducing scheduling latency as a new metric to capture interference between online and offline services co-located in containerized cloud environments [42] found a more precise performance impact on latency-sensitive online

workloads. The authors combined this metric with machine learning models to predict interference and guide scheduling decisions. They further developed a predictive scheduler that allocates CPU and memory resources based on workload characteristics and query load.

The research introduced in [55] a mechanism to maintain performance guarantees in virtualized clusters where virtual machines contend for shared resources. A slowdown estimator and a power- and interference-aware scheduler formed the core of the approach. The estimator predicted potential deadline violations from noisy slowdown data, prompting rescheduling when required. Through simulations derived from Google Cloud trace-logs, the method demonstrated effective SLA compliance and achieved roughly twelve percent cost savings.

[20], [45], [44] and [26] take the concept of predicting resource contention, also referred to as performance degradation further by leveraging predictive frameworks.

Lastly, the central approach for contention-avoidance was strongly guided by the following works that applied clustering techniques to identify suitable co-location candidates.

[41] presented ScalCCon1, a correlation-aware VM consolidation approach designed to handle scalability challenges in large data centers. The method applied a two-phase clustering scheme to efficiently group virtual machines while minimizing the computational cost of correlation analysis. Experimental results showed that this approach significantly reduced execution time, physical machine usage, and performance violations compared to existing one-phase clustering and correlation-based methods.

With pSciMapper by [70], a power-aware framework for consolidating scientific workflows in virtualized cloud environments. The approach formulated consolidation as a hierarchical clustering problem using an interference-based distance metric and applied kernel canonical correlation analysis to link resource usage with performance and power consumption. Experiments with real and synthetic workflows showed that pSciMapper can cut power usage by more than half while keeping performance degradation below fifteen percent. The framework also demonstrated low scheduling overhead and strong scalability for large workflow workloads.

3.3 Energy-Aware Scheduling, Co-Scheduling and Task Mapping in HPC Workflows

The chapter by [59] examined the environmental impact of scientific workflows by estimating the carbon footprint of three real-world applications from different domains. It discussed strategies for reducing energy use and emissions at both task and workflow levels. The proposed techniques included exploiting energy-efficient hardware, optimizing code generation, adjusting processor frequencies, consolidating workloads, and applying energy-aware scheduling to improve overall sustainability in large-scale computational research.

As mentioned in the previous section 3.2, the work by [70] achieved significant power savings with minimal slowdown by applying the approach outlined in [59]. However, it operated primarily in an offline mode, optimizing workflow placement decisions before execution and focused on static virtualized environments rather than dynamic HPC workflows. Although adapting their concepts of temporal signature computation and application of dissimilarity-based task clustering we extend the approach to an online scheduling context.

More recent research has expanded co-location techniques toward data- and container-level optimization. WOW by [40] 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 [53] 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.

Energy-aware scheduling for scientific workflows has been studied from both modeling and algorithmic angles.

The work by [62] conducted an extensive comparative study of list-scheduling and cluster-scheduling heuristics for makespan minimization in DAG-based parallel programs. By analyzing algorithms across rather than within categories, the study clarified how each approach performs under varying processor constraints.

[61] analyzed scheduling algorithms for scientific workflows across various computing environments. Covering studies from 2010 to 2023, it categorized algorithms by optimization goals, techniques, and evaluation criteria. The findings showed that cloud platforms dominate current research while areas such as stream processing, energy efficiency, and hybrid environments remain underexplored.

The survey by [31] reviewed task scheduling approaches for workflow executions in cloud environments. The authors focused on balancing the objectives of users and providers while maintaining cost efficiency and service quality. Their work introduced a taxonomy that classifies existing algorithms by scheduling strategy, optimization goals, and evaluation metrics. The survey also discussed current challenges and research gaps, outlining directions for improving task scheduling mechanisms in future studies.

Building on the discussion of scheduling strategies in cloud-based workflows, the paper by [43] the authors extended the analysis to large-scale scientific workflows operating over big data environments. This work emphasized the trade-off between makespan and monetary cost as central factors in workflow scheduling. The authors proposed a taxonomy of algorithms reflecting these priorities and evaluated key workflow management systems in terms of usability and performance. Their study complemented prior surveys by providing a more focused and comparative examination of leading scheduling approaches and systems. From an optimization perspective, [25] 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. [48] pursue metaheuristics to jointly minimize makespan, cost, and energy for scientific workflows. [22] 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. [54] 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. To specifically target the energy-efficiency during workflow execution, several approaches were proposed for both cloud and traditional cluster environments. [39], [21], [10], [1] and [37] use the performance degradation imposed by close workload co-location as means for optimizing the placement.

Lastly, the following works include machine learning techniques into their approaches for either predicting performance degradation or optimizing scheduling decisions.

Lotaru, introduced by [5], is a local runtime prediction method for scientific workflows operating on heterogeneous compute clusters. Unlike traditional models requiring historical data, Lotaru predicts task runtimes using microbenchmarks and reduced input profiling during early execution. The method employs Bayesian linear regression to estimate runtimes and quantify prediction uncertainty, supporting adaptive scheduling decisions. [2] explored fine-grained energy prediction for Docker-based applications in cloud data centers. Using time series models they forecasted container-level power consumption over hourly intervals. The analysis compared model performance across multiple containers running diverse workloads and evaluated prediction accuracy. Results showed that ETS achieved the most accurate forecasts for several container types, while ARIMA performed better for others, emphasizing the importance of model selection according to container behavior and workload characteristics. [46] introduced a machine learning-based container scheduling and consolidation method for cloud environments. The approach dynamically adjusts the number of active nodes in response to container submission patterns to reduce power consumption. By analyzing historical submission data, it identifies high, medium, and low activity periods and adapts resource usage accordingly. Implemented within Docker Swarmkit, the method showed effective energy savings and adaptability across varying workload scenarios.

4 Approach

The following chapter presents the methodological approach of this thesis. While the subsequent chapter discusses the concrete implementation aspects, this section establishes the theoretical foundation that builds upon the concepts introduced in chapter 2. The structure of this chapter follows the sequence of steps illustrated in Figure 4.15, beginning with an outline of the central objective of the thesis. This is followed by a set of assumptions that define the scope, applicability, and limitations of the proposed approach. The remainder of the chapter then describes each step in detail, explaining how the individual components contribute to the overall objective of enabling energy-aware and contention-conscious co-location of scientific workflow tasks.

4.1 Central Objective

The central objective of this work is to model the co-location of scientific workflow tasks with a focus on achieving energy-awareness and determining how this knowledge can be applied during workflow execution to improve resource utilization and overall energy efficiency. To accomplish this, we introduce two key concepts: (i) *ShaReComp* and (ii) *ShaRiff*. Both address the idea of shared resource usage but from different perspectives. *ShaReComp* defines the modeling approach for resource sharing during computation. It focuses on understanding and representing co-location with respect to resource usage, contention, and energy behavior. In contrast, *ShaRiff* introduces an algorithmic framework that applies the *ShaReComp* model during workflow execution to enable informed and feasible resource sharing. Together, these concepts provide both the theoretical foundation and practical mechanism for energy-aware co-location in scientific workflows. *ShaReComp* is inspired by the work presented in [70], which introduced a mathematical formulation of the co-location problem. However, the implementation of *ShaReComp* differs significantly in its scope and application. By combining *ShaReComp* with *ShaRiff*, this work aims to investigate how co-location can be directly integrated into the dynamic scheduling and task mapping process, emphasizing that co-location and scheduling are inherently interdependent and should be addressed jointly. To achieve this, workflow tasks are characterized prior to execution, enabling contention-aware co-location decisions at runtime. In this setting, co-location is implemented at the virtual container level, focusing on virtual machines deployed on physical hosts. Additionally, this work extends the scope of co-location from merely determining which tasks should share resources to understanding how the performance behavior of co-located tasks can be modeled and predicted, thereby providing actionable insights for resource managers in future workflow executions.

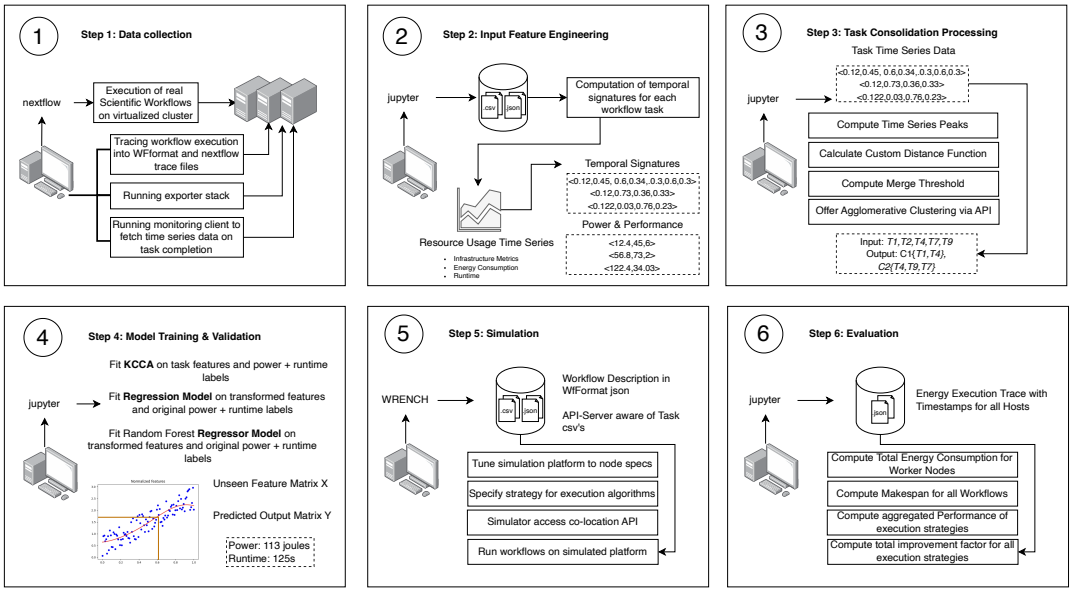

Figure 4.15: Overview of the Proposed Approach
The proposed approach consists of 6 steps.

The proposed objective, introduced in Section 1.1, is decomposed into six main steps, as illustrated in Figure 4.15. Step 1, described in Section 4.2, involves a data collection phase that captures detailed execution metrics from scientific workflow runs. In Step 2, this collected data undergoes structural processing and formatting to extract relevant performance characteristics, as outlined in Section 4.3. The raw data treatment of Step 2 is performed immediately after data collection, while Steps 3 and 4 require more specialized data preparation tailored to their respective analytical goals. Details for these stages are provided in Sections 4.3.1 and 4.3.2. Step 3 focuses on consolidating time-series data by identifying tasks with complementary resource usage patterns through clustering, as discussed in Section 4.3.1. Step 4 reorganizes the data collected in Step 1 into a format suitable for learning the relationship between task behavior, runtime, and energy consumption, as described in Section 4.3.2. Finally, Steps 5 and 6, presented in Section 4.4, demonstrate how the theoretical concepts developed in Step 3 can be applied during workflow execution and how their potential benefits in terms of performance and energy efficiency can be evaluated through simulation.

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 designed to capture a broad range of metrics relevant to the execution of scientific workflows. Its design is inspired by [7] and follows a four-layered structure comprising the Resource Manager, Workflow, Machine, and Task layers. Each layer represents a distinct level of abstraction within the workflow execution environment, providing insights into specific aspects of system performance and resource utilization.

Table 4.1 provides an overview of the tools and data collection options that we considered for achieving our monitoring objectives, outlining how different metrics could be gathered and integrated across the hierarchical layers to form a comprehensive view of workflow behavior.

Metric Category	cAdvisor	Deep-Mon	docker-activity	scaphandre	slurm-exporter	process-exporter	cgroups-exporter
Workflow-Level Metrics							
Infrastructure status					x		
Running workflows					x		
Workflow ID					x		
File system status					x		
Machine-Level Metrics							
Status				x	x		
Machine type				x	x		
Hardware specification				x	x		
Available resources				x	x		
Used resources				x	x		
Requested and consumed resources				x	x		x
Resource usage metrics				x	x		x
Power consumption				x	x		
Fault diagnosis					x		
Task-Level Metrics							
Task status	x	x		x	x		
Task ID					x		x
Task duration	x	x		x	x		
Requested and consumed resources	x	x	x	x	x		x
Resource usage metrics	x	x	x	x	x		x
Process metrics		x		x		x	
Power consumption		x	x	x	x		

Table 4.1: Overview of monitored metrics and their data sources inspired by [7].

To enable access to detailed performance metrics in time-series format, Prometheus was selected as the central time-series database. As shown in Table 4.1, the monitoring setup therefore focused on tools that are interoperable with Prometheus as data exporters and that operate effectively within the layered monitoring framework. During the evaluation of several potential exporters, some were excluded due to limited compatibility, excessive overhead, or incomplete metric coverage—particularly for short-lived or multi-process workflow tasks. After comparative testing, the combination of cAdvisor and a custom fork of the Deep-Mon system [17] produced the most stable and comprehensive results across varying workloads and resource types. These two tools were therefore selected as the foundation of 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.

Software Tool	**Used Metrics**
nextflow	trace file summary
cAdvisor	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
Deep-Mon	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

Table 4.2: Metrics included in the workflow monitoring approach

cAdvisor is an open-source daemon for monitoring resource usage and performance of containers. It continuously discovers containers via Linux cgroups under the path /sys/fs/cgroup. Once started, cAdvisor subscribes to create/delete events in the cgroup filesystem, converts them to internal add/remove events, and configures per-container handlers. Metrics originate from machine-level facts parsed from /proc and /sys directories and most prominently container and process usage collected at cgroup boundaries. In practice, cAdvisor provides low-overhead, per-container telemetry [28].

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 [17].

Based on the identification of relevant metrics and the selection of appropriate monitoring technologies, we designed an architectural model and an accompanying algorithm to define how the individual components interact. The resulting architecture integrates the selected exporter stack into a coherent monitoring workflow, ensuring that metric collection, event handling, and data aggregation are performed in a structured and automated manner. Figure 4.16 provides an overview of the system components required for this setup, illustrating how they interact to realize the monitoring logic proposed in this work. The technical implementation details are discussed in Section 5.

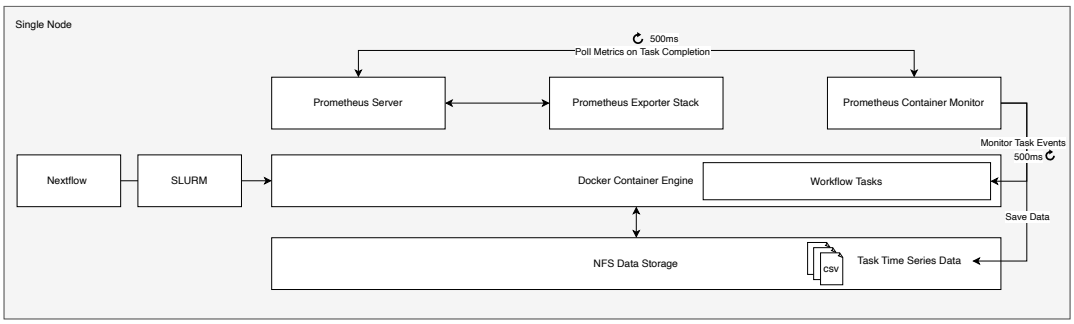

Figure 4.16: Monitoring Client
Schematic Overview of the Components of the Monitoring Client.

Figure 4.16 and Algorithm 1 show the behavior of the monitoring client.

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 algorithm outlined in Algorithm 1 is designed to minimize monitoring overhead while ensuring that only relevant data is collected during workflow task execution. When a container start event is detected, the monitoring client extracts essential metadata—such as the container ID, the associated workflow label, and the start timestamp—and registers this information in an internal mapping to maintain a link between container identifiers and workflow tasks. When a termination event occurs, the client initiates a targeted data collection phase, querying only the relevant metrics for the specific container and execution period.

This event-driven approach ensures efficient monitoring by restricting data retrieval to the active lifecycle of workflow tasks. Furthermore, the monitoring client supports dynamic configuration of the metrics listed in Table 4.2, allowing users to define which metrics are collected and analyzed. This flexibility enables fine-grained, task-specific monitoring while maintaining low overhead and adaptability to different experimental requirements.

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

This section corresponds to Steps 2, 3, and 4 in Figure 4.15 and explains how the collected monitoring data is transformed into structured representations of task behavior. The focus lies on preparing and processing the raw time-series metrics to generate meaningful feature sets that can be used for two main purposes: identifying complementary tasks for co-location through clustering and building predictive models that capture the relationship between task behavior, runtime, and energy consumption.

General Preprocessing of Raw Time-Series Data

The collected monitoring data is obtained as raw time-series files in CSV format and must therefore undergo several preprocessing steps before further analysis. The goal of this stage is to transform heterogeneous monitoring outputs into a consistent and structured representation that links system-level metrics to individual workflow tasks. The general preprocessing workflow includes the following steps:

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, data is produced at multiple abstraction levels—from the resource manager down to container metrics. To transform this heterogeneous data into a unified, task-level format, a structured preprocessing pipeline was developed. First, all monitoring outputs are recursively traversed and scoped according to the desired data sources and metric types. The resulting directory tree is then filtered to retain only relevant metrics, such as task metadata and power measurements. Each dataset is subsequently split into per-task CSV files based on container identifiers. As an anchor for this process, the started and died container tracking files are used, which record general container lifecycle information such as container names, identifiers, and timestamps. To establish a consistent mapping between workflow tasks and their corresponding container traces, entity matching is performed. Here, container lifecycle records—capturing task identifiers, container names, and working directories—serve as the linking mechanism

between workflow-level entities and container-level monitoring data. The working directories are attached to every container trace, enabling a reliable cross-reference with workflow metadata extracted from SWMS trace files. This ensures that each monitored container is correctly associated with the corresponding workflow task, providing a coherent, task-centric dataset for subsequent analysis.

4.3.1 Task Clustering for Contention-Aware Consolidation

Following the preprocessing and structuring of raw time-series data, we continue to address step 3 of the overall approach outlined in 4.15. This step focuses *ShaReComp's* mechanics - task clustering as a means of identifying tasks that can be co-located without causing severe resource contention. The goal is to refine the processed monitoring data into representations that capture each task's characteristic resource usage patterns and to use these representations for forming complementary task groups. This clustering step forms the conceptual foundation for contention-aware consolidation.

Preprocessing Raw Time-Series Data for Task-Clustering

To prepare the data for contention-aware task clustering, the processed time-series must be transformed into a compact yet expressive representation of each task's resource usage behavior. This ensures that the clustering algorithm can meaningfully capture similarities and differences between tasks across multiple resource domains. We therefore apply the following two main preprocessing steps prior to executing 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 4.3.1. High affinity means higher potential interference when peaks align; low affinity reflects benign coexistence.

While the first step focuses on constructing consistent temporal representations of task behavior, the second step introduces a quantitative measure of how different workload types interact when co-located. To provide a clearer understanding of this relationship and its role in the clustering process, we examine the computation of workload-type affinity in more detail.

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 container's 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

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

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

Per-workload Slowdown Factors

The per-workload slowdowns are defined as

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

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

The mean slowdowns across the workload pair are

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

A weighted average combines both effects:

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

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

The final combined slowdown factor is

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

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

Affinity Score

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

First, compute pairwise affinity ratios:

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

A weighted average combines both effects:

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

The normalized affinity score is then:

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

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

Algorithmic Approach to Task Consolidation

Building on the previously derived peak time-series representations and the computed workload-type affinity scores, we propose an algorithmic formulation of the consolidation, process. 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 that quantifies how strongly workloads interfere when sharing resources, the clustering process incorporates this measure directly into its distance metric. The distance between tasks increases when two tasks exert pressure on the same resources simultaneously, indicating potential contention, and decreases when their resource usage peaks complement one another.

Algorithm 2 showcases the *ShaReComp* task consolidation procedure developed in this work, integrating both the peak-pattern representations and affinity scores into a unified clustering process. The method first computes pairwise similarities between all task signatures, where each signature encodes the temporal evolution of multiple resource metrics. These similarities are weighted by the experimentally derived affinity values to reflect the degree of potential interference between workloads. The resulting similarity matrix thus provides a contention-aware view of task compatibility. A percentile-based threshold is then applied to determine the stopping condition for cluster formation, ensuring that only task pairs with sufficiently low contention potential are merged. Finally, agglomerative clustering is executed using average linkage criterion to form consolidated task groups that exhibit complementary resource utilization.

Algorithm 2: ShaReComp - Task Consolidation Algorithm

Input: task signatures sig, affinity weights w, percentile τ, linkage
Output: clusters $\mathcal{C}$

```
S ← compute_similarity (sig, w) ;            // resource-aware similarity
θ ← percentile_threshold (S, τ) ;      // percentile-based stopping rule
C ← agglomerative_merge (S, θ, linkage) ;      // agglomerative clustering
return C
```

We examine each stage of Algorithm 2 in turn.

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 $R = \{\text{CPU}, \text{Memory}, \text{Disk}\}$, yielding in total ten pairwise combinations of resource types across two tasks. The distance term combines the precomputed affinity score with the correlation of peak resource intensities.

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

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.

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.

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.

4.3.2 Predicting the Runtime and Energy Consumption of Task Clusters

Following the reasoning of [70], we investigate whether the formed task clusters exhibit shared structures in terms of correlations between their time-series behavior and their corresponding energy and performance metrics. To this end, we adopt their proposed approach for preprocessing time-series data into temporal signatures on a per-task basis.

Preprocessing Raw Time-Series Data for Predictive Modelling

Unlike the preprocessing described in Section 4.3.1, the following procedure prepares the data specifically for step 4 of the overall approach shown in Figure 4.15, focusing on transforming raw time-series inputs into a form suitable for predictive modeling.

1. Temporal signature construction.

   Sampling and smoothing.

   Equal-length normalization.

   Container-wise collation.

2. Model Input Construction.

   Normalization and Scaling.

   Extraction of Input Features and Output Labels.

Temporal Signature and Model Input Construction

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

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

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

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

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

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

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

where $d_x = 3 \times 10 = 30$ in this example configuration.

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

denotes the complete input matrix used for model training.

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

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

Example

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

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

Concatenating these signatures yields the input matrix

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

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

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

Building upon Algorithm 5 in Section 4.3.1, we extend the *ShaReComp* concept by linking task clustering with predictive modeling. This step introduces a follow-up procedure that

uses the clustered task groups and their extracted temporal features from Section 4.3.2 to model and predict the expected runtime and energy behavior of consolidated task clusters.

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

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

foreach *cluster* $C_k \in \mathcal{C}$ **do**
  $F_k \leftarrow$ `sum_cluster_features` $(\{\, \mathrm{sig}[t_i] \mid t_i \in C_k \,\})$; `// sum task signatures to form cluster feature`
$X \leftarrow$ `build_feature_matrix` $(\{F_k\})$; `// construct consolidated feature matrix`
foreach *cluster feature* $X_k \in X$ **do**
  $(\hat{t}_k, \hat{E}_k) \leftarrow$ `predict_runtime_and_energy` $(X_k, \mathcal{M})$
return $\hat{Y} = \{(\hat{t}_k, \hat{E}_k)\}_{k=1}^{|\mathcal{C}|}$

The statistical models used in Algorithm 3 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.

KCCA itself does not perform prediction but rather uncovers shared structures between task feature data performance outcomes. Through the data preprocessing described in the previous step, we prepare the feature matrices so that a regression model can generalize from the learned shared space to unseen inputs—for instance, the aggregated features of a new task cluster. We use KCCA to project both input and target data into a common latent space that captures nonlinear relationships between task resource usage and performance behavior.

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_xA and K_yB. This is expressed as the generalized eigenvalue problem:

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

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

$$X' = K_xA, \qquad Y' = K_yB,$$

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

During training, we then fit a Kernel Ridge Regression model on the latent-space representations of the input features and the original, un-normalized target values Y (runtime and energy). This approach leverages KCCA's ability to transform unseen data into the same latent space, allowing KRR to predict concrete runtime and energy values for previously unseen X inputs. KRR is chosen because, as outlined in chapter 2.5.2, it extends ordinary least squares regression by capturing nonlinear dependencies through kernel-based feature mappings.

Illustrative Example

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

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

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.

To illustrate this concept, we consider the following example.

Example on Predicting the Performance of Task Cluster with ShaReComp

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:

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

Applying 3 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 Kernel Ridge Regression or the Random Forest Regressor yield

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

representing predicted runtime (seconds) and energy (joules) per cluster.

4.4 Simulation of Task Co-location during Workflow Execution

We advance to steps 5 and 6 of the overall approach shown in Figure 4.15, where the previously developed concepts are embedded into a workflow execution environment. In this stage, the focus shifts from modeling and clustering individual tasks to simulating their co-location during execution, eventually allowing us to evaluate how these strategies affect overall performance and energy efficiency.

Outline of Simulation Components

We build upon generic design pillars that structure the simulation environment.

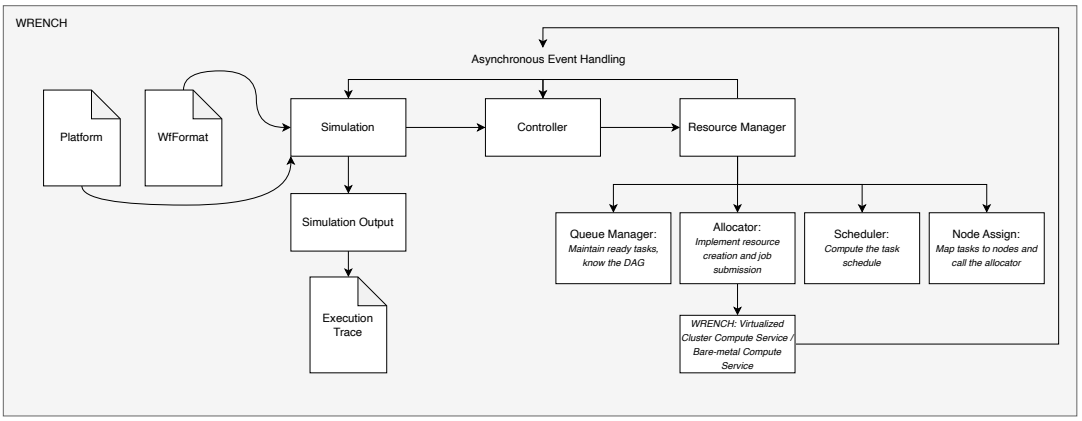

Figure 4.17: High-level Simulator-Design
Simulator Design using WRENCH for studying Energy-aware Co-location of Scientific Workflow Tasks.

The design of distributed HPC systems typically revolves around three fundamental mechanisms: resource allocation, queue management and dispatching, and job placement. As illustrated in Figure 4.17, these principles are adapted and integrated into our simulator through a dedicated controller component that interacts with a central resource manager. This resource manager orchestrates task scheduling, DAG-aware queue handling, task-to-node assignment, and the eventual allocation and execution of tasks on compute resources. Because the objective of this work is to analyze the effects of task co-location, the simulator's processing engine is designed with extensible interfaces that support alternative scheduling, mapping, and allocation strategies—from traditional First-In-First-Out (FIFO) execution to heuristic or energy-aware methods that emphasize beneficial co-location patterns.

4.4.1 Embedding Task Co-location into Scheduling Heuristics

Our algorithmic design for the resource manager, allocator, and scheduler components shown in Figure 4.17 follows heuristic principles. Instead of formulating co-location within the scheduling process as an explicit optimization problem, we embed it through a set of simple, interpretable decision rules. We divide the task-to-node mapping process into two distinct stages, as illustrated in Figure 4.18. The simulation begins by launching the controller component, which invokes the resource manager. Through its queue management mechanism, the resource manager identifies all ready-to-run tasks that have no remaining dependencies. In the depicted example, tasks 1, 2, 4, 8, 11, 12, and 15 are selected for scheduling. For this process, we adopt list-scheduling algorithms, one of the most established heuristic approaches in workflow scheduling. These algorithms assign each task a priority or ranking based on topological and performance characteristics such as critical path length, estimated execution time, or communication cost. Tasks are then scheduled iteratively, with the highest-priority unscheduled task being mapped to an available resource until all ready tasks have been assigned.

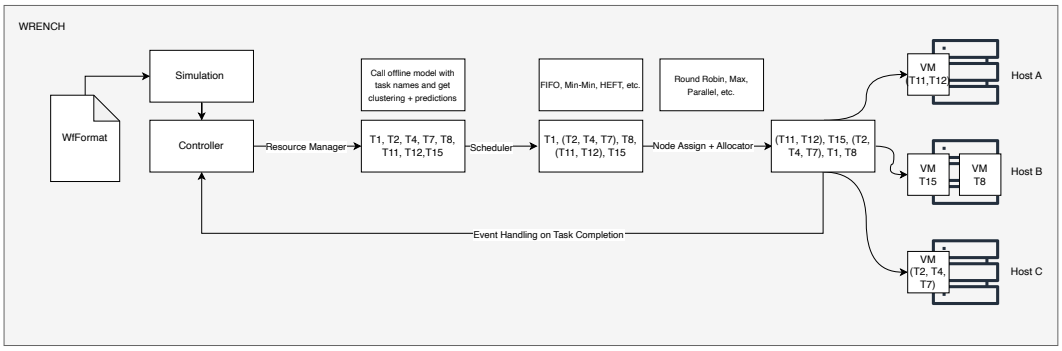

Figure 4.18: Co-location embedded into Workflow Execution
This figure shows a schematic example on how co-location is embedded in workflow execution.

Since our goal is to map not individual tasks but clusters of tasks, we embed the retrieval of co-location information derived from *ShaReComp* directly into this stage of the scheduling process. In the illustrated example, the co-location hints result in consolidated groups such as T1, (T2, T4, T7), T8, (T11, T12), and T15. After clustering, the next step is to determine which compute host can best accommodate each task group. To this end, we define an interface for node assignment strategies that can either prioritize hosts with the most idle compute cores or distribute task clusters evenly across all available nodes to ensure balanced utilization. Finally, the allocation component creates virtual machines for each task cluster and launches them on their designated hosts, completing the mapping and allocation workflow. Through this exemplified design, we establish a foundation that not only enables the embedding and study of the co-location problem within an execution environment but also allows for systematic comparison through modular component interfaces. The following section formally introduces the simulation environment and the system model that represents our algorithmic formulation of the co-location problem.

Simulation System Model

Workflow Properties

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

$$G = (T, E)$$

where

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

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

Task Properties

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

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

Infrastructure Model

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

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

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

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

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

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

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

Resource Assignment

A mapping of tasks to hosts and VMs is represented as

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

where

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

The set of mappings for an allocation interval is written as

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

Task Co-location

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

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

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

Oversubscription

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

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

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

Execution Dynamics

At runtime:

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

We present the unified algorithm that governs the overall behavior of our simulation framework. This algorithm formalizes how workflow tasks are scheduled, assigned, and executed under different resource management strategies, including optional co-location support through *ShaReComp*.

Algorithm 4: ShaReComp Simulation - WRENCH Framework

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

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

```
Initialize system state: host capacities, ready queue Q, and monitoring layer
while workflow G not completed do
    Update Q with all newly ready tasks
    Perform task scheduling: prioritize tasks in Q according to policy π
    Perform node assignment: select suitable host(s) h ∈ H using π
    Perform resource allocation: determine allowed capacity n_max = f(c(h), α)
      and reserve resources
    if policy π supports co-location then
        Optionally query 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 evaluate the effectiveness of the *ShaReComp* approach, we define several scheduling and node assignment algorithms that differ in their treatment of resource sharing and task placement. All of these algorithms build upon the unified simulation blueprint presented in Algorithm 4 and serve as comparative baselines. The first category of baselines executes each task in its own virtual machine, thereby avoiding any form of co-location. The second category enables task co-location by grouping multiple tasks within shared virtual machines and applying different host prioritization and allocation strategies. A third category extends this idea through controlled oversubscription, where more tasks are assigned to a virtual machine than the available reserved resources allow, deliberately inducing resource contention to test system robustness. Detailed algorithmic definitions are provided in the appendix, while 6 discusses their comparative behavior in depth. The *ShaRiff* algorithms build directly on top of these baselines, extending them with co-location awareness derived from *ShaReComp*. In doing so, they replace random or static task grouping with data-driven consolidation decisions that account for resource affinity and contention characteristics.

Algorithmic Examples of Task Mapping with guided Co-location

This section concludes our approach by introducing the integration of the *ShaReComp* methodology into the workflow execution process. To this end, we design four scheduling algorithms collectively referred to as *ShaRiff* (Share Resources if Feasible). Each *ShaRiff* variant implements a distinct strategy for mapping consolidated task clusters onto avail-

able compute hosts, while the co-location decisions themselves are consistently guided by the *ShaReComp* approach introduced in Section 4.3.1. An overview of the four *ShaRiff* variants and their respective strategies is provided in Table 4.3.

Algorithm	**Type**
ShaRiff 1	Biggest Host first, parallel Host-backfilling and mapping of Task Clusters
ShaRiff 2	Biggest Host first, parallel Host-backfilling and mapping of Task Clusters with controlled VM Oversubscription
ShaRiff 3	Round-Robin Assignment of Task Clusters, No parallelism
ShaRiff MinMin	Biggest Host first, parallel Host-backfilling and mapping of ordered Task Clusters

Table 4.3: Overview of the *ShaRiff* scheduling variants that make use of ShaReComp.

In the following, we provide a detailed description of the behavior of each *ShaRiff* algorithm, followed by its formal algorithmic representation.

Algorithm 5: ShaRiff 1 — Biggest Host first, parallel Host-backfilling and mapping of Task Clusters

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

Output: 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: colocMap $\leftarrow \mathcal{S}(\mathcal{T}_h)$; `// returns co-location groups`

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

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

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

      $C_{\text{req}} \leftarrow \text{sum}(\text{req\_cores}(\mathcal{C}_k))$; $M_{\text{req}} \leftarrow \text{sum}(\text{req\_mem}(\mathcal{C}_k))$; 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 a 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 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. Conceptually, *ShaRiff* preserves FIFO ordering and capacity-ranked host filling, but replaces random batching with adviser-driven clustering.

Algorithm 6: ShaRiff 2 — Biggest Host first, parallel Host-backfilling and mapping of Task Clusters with controlled VM Oversubscription

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

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

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

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

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

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

    continue

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

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

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

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

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

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

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

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

      else

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

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

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

    Destroy VM

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

return *workflow complete*

This variant extends *ShaRiff* 1 by 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. 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 Assignment of Task Clusters, No parallelism

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

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

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

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

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

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

    continue

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

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

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

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

    Destroy VM

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

return *workflow complete*

ShaRiff 3 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 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 *ShaReComp's* plan one VM per suggested cluster on the chosen host. For each multi-task cluster, it sizes the VM by summing vCPU and memory requirements of the clusters tasks, starts the VM, and submits all cluster tasks to the same virtual compute service. Conceptually, the policy is first-fit host, adviser-driven packing. Unlike the other 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 variant extends the *ShaRiff* framework with a MinMin ordering layer, a classical heuristic from list scheduling. Instead of operating on individual tasks, it applies the MinMin principle to task clusters generated by *ShaReComp's* co-location analysis. At each scheduling interval, the scheduler requests a new clustering of ready tasks, queries the prediction service using the models introduced in 4.3.2 with 3 for estimated runtimes, and orders clusters in ascending order of predicted execution time. The *ShaRiff* node assignment and VM allocator then manage placement and resource provisioning. In essence, this forms a MinMin scheduler over co-located clusters, combining *ShaRiff's* affinity-based co-location with prediction-guided execution ordering to minimize queueing delays and improve overall workflow completion time.

Algorithm 8: ShaRiff 1 MinMin — Biggest Host first, parallel Host-backfilling and ordered mapping of Task Clusters

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

Output: all tasks $t_i \in T$ executed with predictive ordering and contention-aware 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)

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: colocMap $\leftarrow \mathcal{S}(\mathcal{T}_h)$; `// returns co-location groups`

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

  `// Predictive MinMin ordering based on estimated runtimes`

  Query prediction API: predictions $\leftarrow \mathcal{P}(\mathcal{M})$ Sort mappings $\mathcal{M}$ by ascending predicted runtime: $\mathcal{M}_{sorted} = \text{sort}_{asc}(\mathcal{M}, \text{key} = r_i)$

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

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

      $C_{\text{req}} \leftarrow \text{sum}(\text{req\_cores}(\mathcal{C}_k))$ $M_{\text{req}} \leftarrow \text{sum}(\text{req\_mem}(\mathcal{C}_k))$ 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*

5 Implementation

This chapter details the technical implementation of the concepts introduced in the previous section. While chapter 4 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. 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

Figure 5.19 illustrates the procedural flow of our system architecture.

Technology and Design Choices

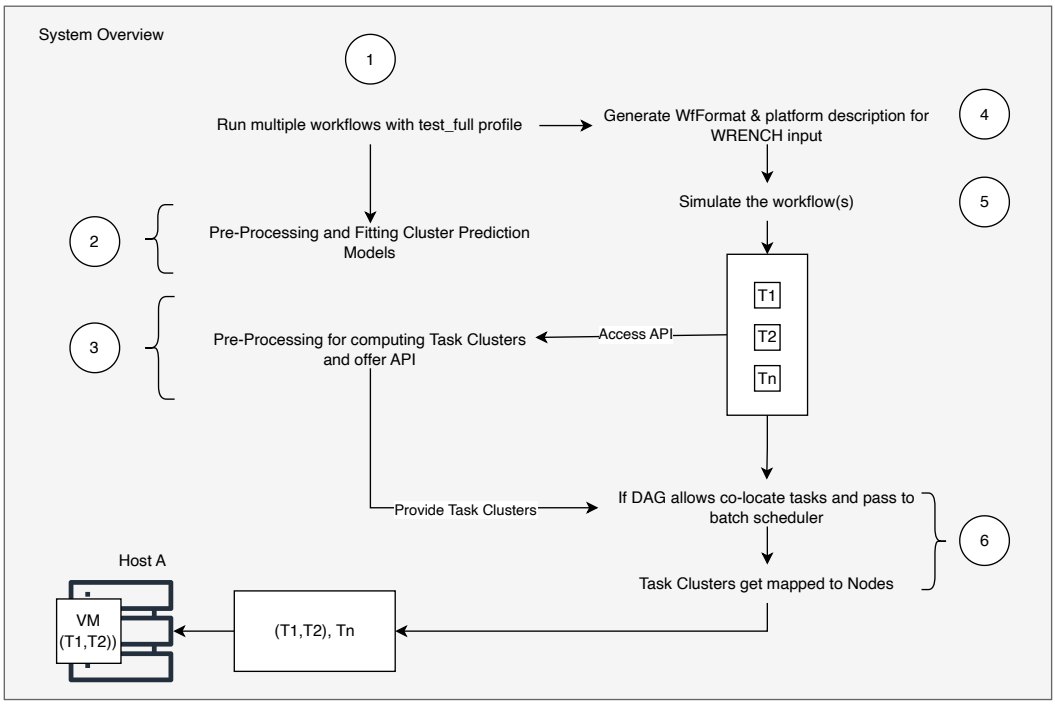

Figure 5.19: System Overview
A procedural view on the System Architecture.

Each implemented component in 5.19 is denoted with a number. These numbers correspond with the following table 5.4 and mention the technology and design choices.

Component Category	Software Used	Comments / Functionality
(1) Workflow Execution		
Operating System	Ubuntu 22.04.5 LTS	Base system environment for all components.
Kernel	Linux 5.15.0-143-generic	Linux Kernel.
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 container environments for task execution.
Resource Manager	Slurm 24.05.2	Allocates CPU and memory resources dynamically across hosts.
(1) Monitoring System		
Time-Series Database	Prometheus Server 3.3.1	Central Time-series database.
Monitoring Client	Go 1.24.1	Collects per-container resource metrics.
(2) 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.
(2,3) Co-location Modelling		
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.
(4) 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.

Table 5.4: Technology and Design Choices of the System Architecture

5.2 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.

5.2.1 Configurable Monitoring Client

The monitoring client exemplifies this by relying on a flexible configuration-driven architecture. Using the YAML configuration file shown underneath, 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 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, scientists can extend the monitoring framework simply by adding new data sources or metrics to the configuration file, without altering the underlying collection logic.

```
server_configurations:
  config_path:
  prometheus:
    target_server:
      controller:
      workers:
      address:
      timeout: 5s
monitoring_targets:
  cpu:
    enabled: true
    data_sources:
    - source: cAdvisor
      labels: [ id, image, job, name ]
      identifier: name
      metrics:
      - name: container_cpu_user_seconds_total
        query: container_cpu_user_seconds_total
        unit: seconds
    - source: ebpf-mon
      labels: [ container_id, name ]
      identifier: name
      metrics:
      - name: container_weighted_cycles
        query: container_weighted_cycles
        unit: counter
  memory:
    enabled: true
    data_sources:
    - source:
      labels:
      identifier:
      metrics:
```

5.2.2 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.

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

Table 5.5: Overview of REST API Endpoints Exposed by the *ShaReComp* Service.

5.2.3 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. The file underneath shows the C++ code used to setup the generic WRENCH simulator for the experiments in this thesis. Through invocation of all simulation specific classes, the controller can hold user-specific abstractions and logic depending on the experimental scope without interfer-

ing with the core simulator functionalities.

```cpp
int main(int argc, char **argv)
{
    auto simulation = wrench::Simulation::createSimulation();

    simulation->init(&argc, argv);

    auto workflow = wrench::WfCommonsWorkflowParser::createWorkflowFromJSON(workflow_file, "100Gf", true);

    simulation->instantiatePlatform(platform_file);

    std::set<std::shared_ptr<wrench::StorageService>> storage_services;
    auto storage_service = simulation->add(wrench::SimpleStorageService::createSimpleStorageService(
        "WMSHost", {"/scratch/"}, {{wrench::SimpleStorageServiceProperty::BUFFER_SIZE, "50MB"}}, {}));

    std::vector<std::string> hostnames = {"VirtualizedClusterHost1", "VirtualizedClusterHost2", "VirtualizedClusterHost3"};

    std::cerr << "Instantiating an EnergyMeterService on WMSHost that monitors VirtualizedClusterHost1 every 10 seconds ..." << std::endl;
    auto energy_meter_service = simulation->add(new wrench::EnergyMeterService("WMSHost", hostnames, 10));

    simulation->getOutput().enableEnergyTimestamps(true);

    auto virtualized_cluster_service = simulation->add(new wrench::VirtualizedClusterComputeService(
        "VirtualizedClusterProviderHost", virtualized_cluster_hosts, "", {}, {}));
    auto wms = simulation->add(
        new wrench::Controller(workflow, virtualized_cluster_service, storage_service, "WMSHost"));

    auto file_registry_service = new wrench::FileRegistryService("WMSHost");
    simulation->add(file_registry_service);

    simulation->launch();

    auto energy_trace = simulation->getOutput().getTrace<wrench::SimulationTimestampEnergyConsumption>();

    simulation->getOutput().dumpHostEnergyConsumptionJSON("../results/{strategy}_{workflow}rnaseq_host_energy_consumption.json", true);

    return 0;
}
```

Listing 1: WRENCH C++ Simulation File

6 Evaluation

This chapter presents the evaluation setup and results of the task co-location approach alongside the developed scheduling algorithms for nine heterogeneous, real-world scientific workflows executed in a simulated environment.

6.1 Evaluation Setup

First, this section articulates the evaluation setup used in the experiments.

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. 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

Nine real-world, openly available scientific workflows from the nf-core repository have been used for monitoring data collection and been integrated into simulation to evaluate the performance of our proposed approach. Specifically, the following workflows have been tested:

Workflow	Number of Tasks	Input Files	Output Files	Data Profile
`atacseq`	268	32	437	Analysis pipeline used for ATAC-seq data
`chipseq`	506	34	777	analysis pipeline used for Chromatin Immuno-Precipitation sequencing
`rnaseq`	295	30	631	Analysis of RNA sequencing data
`scnanoseq`	210	6	470	Analysis pipeline for 10X Genomics single-cell/nuclei RNA-seq data
`smrnaseq`	908	65	1524	Small RNA sequencing
`pixelator`	55	12	172	Analysis of Molecular Pixelation
`methylseq`	58	20	166	Analysis pipeline used for Methylation (Bisulfite) sequencing data
`viralrecon`	2835	113	9790	Analysis pipeline used to perform assembly and low-frequency variant calling for viral samples
`oncoanalyser`	43	64	357	Analysis of cancer DNA

Table 6.6: Overview of evaluated nf-core workflows

6.1.3 Monitoring Configuration

The execution of the workflows and the accompanying monitoring data collection have been performed in a high-throughput computing environment with the following monitoring metrics being collected:

Monitoring Target	Enabled	Supported Data Sources	Collected Metric Types / Adaptability Notes
Task Metadata		nextflow	tracefile
CPU		cAdvisor, Deep-Mon	container_cpu_usage_seconds_total container_weighted_cycles
Memory		cAdvisor, Deep-Mon	container_memory_usage_bytes container_mem_rss
Disk		cAdvisor, Deep-Mon	container_blkio_device_usage_total container_num_reads container_num_writes
Energy		Deep-Mon	container_power
Prometheus Configuration			
Prometheus backend collecting all metrics at 500ms intervals.			

Table 6.7: Monitoring Configuration Overview

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 API service is based on a Jupyter Notebook that contains all implementations and evaluations of our approach and were executed on the infrastructure described in Section 6.1.1.

6.1.5 Simulation Setup

Simulated Platform Configuration

The infrastructure described in Section 6.1.1 was replicated within the WRENCH 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.

```xml
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
<platform version="4.1">
    <zone id="AS0" routing="Full">
        <host id="VirtualizedClusterHost1" speed="32Gf,28Gf,20Gf,12Gf,8Gf" pstate="0" core="24">
            <prop id="ram" value="188GiB" />
            <!-- The power consumption in watt for each pstate (idle:all_cores) -->
            <prop id="wattage_per_state" value="116:133:220, 114:129:202, 111:125:186, 108:121:170,106:118:156"/>
            <prop id="wattage_off" value="10" />
        </host>
        <host id="VirtualizedClusterHost2" speed="32Gf,28Gf,20Gf,12Gf,8Gf" pstate="0" core="24">
            <prop id="ram" value="188GiB" />
            <!-- The power consumption in watt for each pstate (idle:all_cores) -->
            <prop id="wattage_per_state" value="116:133:220, 114:129:202, 111:125:186, 108:121:170,106:118:156"/>

            <prop id="wattage_off" value="10" />
        </host>
        <host id="VirtualizedClusterHost3" speed="32Gf,28Gf,20Gf,12Gf,8Gf" pstate="0" core="24">
            <prop id="ram" value="188GiB" />
            <!-- The power consumption in watt for each pstate (idle:all_cores) -->
            <prop id="wattage_per_state" value="116:133:220, 114:129:202, 111:125:186, 108:121:170,106:118:156"/>
            <prop id="wattage_off" value="10" />
        </host>
</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.

Algorithm	Type
Baseline 1	FIFO Scheduling, Exclusive Round-Robin Host Assignment, Exclusive VM Allocation
Baseline 2	FIFO Scheduling, Exclusive Host-Backfilling, Exclusive VM Allocation
Baseline 3	Round-Robin Assignment of Task Clusters, No parallelism
Baseline 3.1	Biggest Host first, Mapping of random Task Clusters
Baseline 3.2	Biggest Host first, parallel Host-backfilling and mapping of random Task Clusters
Baseline 4	Round-Robin Host-Mapping of random Task Clusters with Oversubscription
Baseline 4.1	Biggest Host first, parallel Host-backfilling and mapping of random Task Clusters with VM Oversubscription

Table 6.8: Overview of Baseline Scheduling Algorithms

Baselines without Co-location

The **baseline** scheduling algorithms implement 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 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.

Baselines with Co-location

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 with 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. **Baseline 4** retains the same FIFO scheduler but introduces a node assignment and allocation policy focused on oversubscribing round-robin selected hosts. 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 with the highest idle capacity first and grouping an amount of ready tasks into a VM that exceeds the available host-cores by a determined oversubscription factor. Once the first host is filled, the process continues when the ready task queue is updated. The **last baseline** further extends the previous one by allowing controlled CPU over-subscription during co-location with parallel host-assignment. 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 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 random task co-location with controlled oversubscription.

6.2 Experiment Results

The section on the experimental results is organized as follows. We revisit the approach introduced in Chapter 4.3.1 by examining the workload experiments and the resulting measurements that form the basis for subsequent evaluation steps in 6.2.1. We then present the outcomes of the statistical methods applied in this work, starting with an in-depth analysis of the task consolidation approach in 6.2.2. Building on these results, we continue with an interpretation of the outcomes from training two predictive models on the clustering data in 6.2.3. Finally, we integrate all components into a unified simulation framework. Using this setup, we evaluate the results of the simulation in section 6.1.2.

6.2.1 Measuring Interference during Benchmark Executions

Benchmark	Execution Command	Behavior Type	Comments
CPU	`stress-ng -cpu 1 -cpu-method matrixprod -cpu-ops 100000 -metrics-brief`	CPU-bound, matrix computation kernel.	Used to emulate high arithmetic intensity workloads.
Memory (VM)	`stress-ng -vm 1 -vm-bytes 18G -vm-ops 1000 -metrics-brief`	Memory-bound workload.	Tests VM allocation, memory contention, and NUMA effects.
File I/O	`fio -name seqread -rw read -bs 1M -size 18G -numjobs 1 -readonly=1 -direct=1 -iodepth=32 -ioengine=io_uring -group_reporting`	I/O-intensive sequential read.	Evaluates disk and I/O scheduling performance.

Table 6.9: Summary of Synthetic Benchmarks Used in Evaluation.

For measuring resource contention we use the stress-ng tool with the commands listed in 6.9. The benchmark code is executed inside Docker containers, and the Docker API is used to capture the execution time of each benchmark. To record the containers' energy consumption, we again use the ebpf-based Deep-Mon tool. Figure 6.20 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 gray area on each plot represents the runtime of the benchmark when executed in isolation.

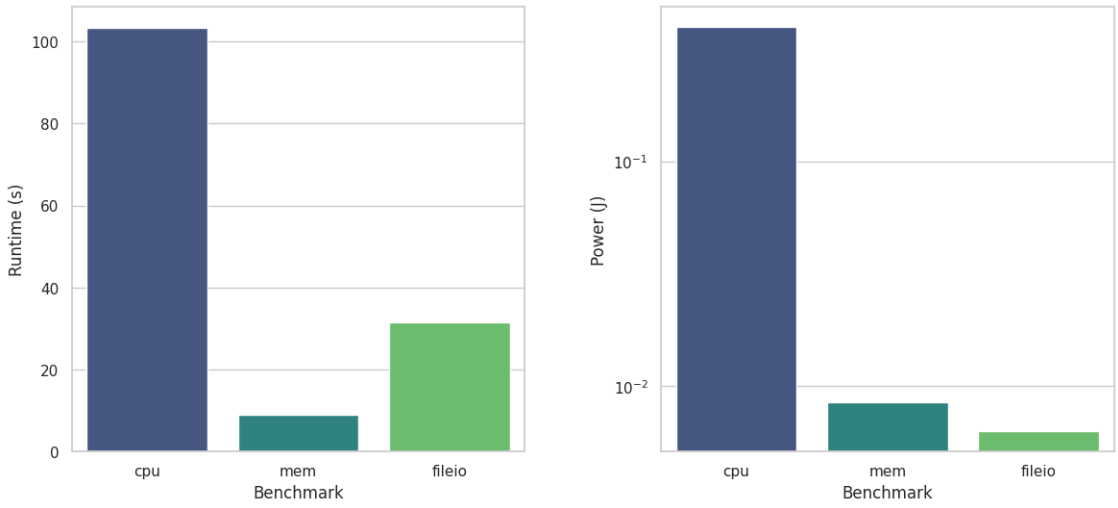

Figure 6.20: Runtime and Power measurements for different Benchmark Executions
This barplot compares the runtime of isolated benchmarks combinations with their power consumption.

While the information in 6.20 concerning the runtime is only dependent on the load of the benchmark code we focus on the right side of the plot and note that the power consumption is heavily dominated by compute heavy workloads, on a big magnitude followed by the

memory benchmark. File I/O shows the least power consumption. This behavior is expected as the experiments are conducted on AMD's Zen 4 architecture that has limited support for the DRAM-RAPL domain leading to processor-heavy workloads dominating the power measurements. We observe in 6.21 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.

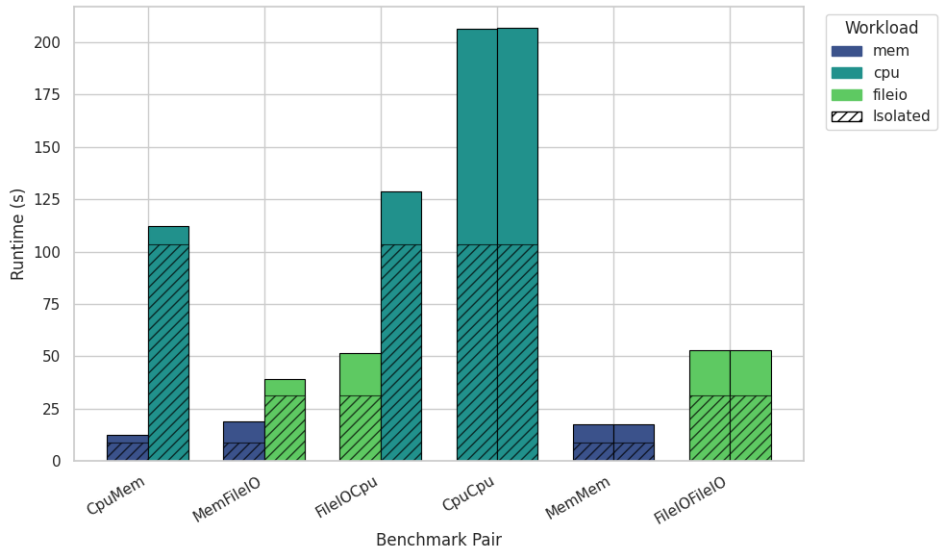

Figure 6.21: Runtime Degradation of different Benchmark Combinations under Co-location
This barplot compares the runtime of co-located benchmark combinations against their isolated executions.

The power measurements obtained for co-located workloads exhibit considerably higher variance and inconsistency across repeated experiments. For instance, when co-locating CPU-intensive workloads, the average isolated power draw is approximately 3.43 joules, whereas the corresponding co-located readings decrease to about 0.008 joules. A similar pattern is observed for FileIO-FileIO benchmarks, where power consumption drops from 0.054 joules in isolation to roughly 0.009 joules under co-location. These deviations are implausibly large and suggest that the RAPL-based power measurements on AMD hardware are unstable or unreliable under the given experimental configuration.

We argue that this discrepancy stems from the fact that AMD's RAPL interface does not expose per-core energy readings consistently. 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 CPU-Memory benchmark 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.

We calculated the affinity scores between workload pairs according to 4.3.1.

Workload 1	Workload 2	Affinity Score	Comment
mem	cpu	0.736	Very Low compatibility; memory and CPU workloads can share resources with serious contention occuring.
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.

Table 6.10: Affinity scores between workload types indicating co-location compatibility.

The results shown in the previous figures are consistent with the affinity scores presented in table 6.10. 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 CPU-Memory 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 scaled, 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 20-th 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.

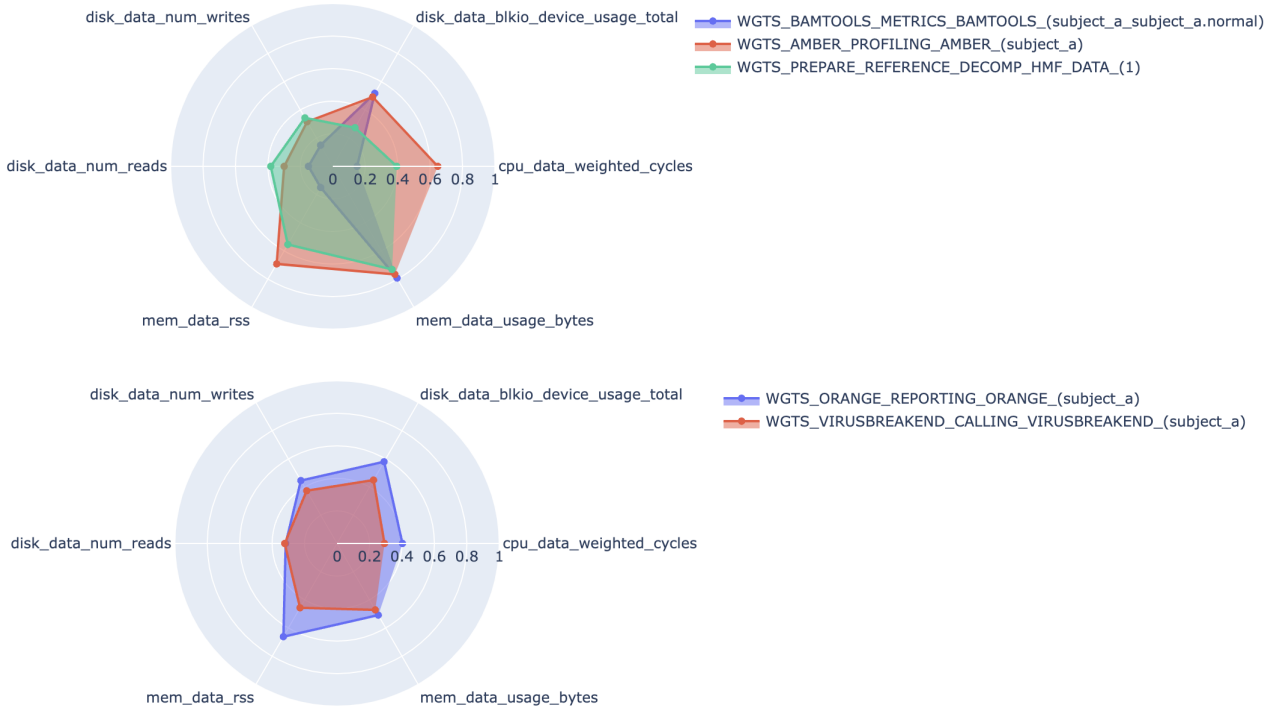

Figure 6.22: Random Clustering Results on oncoanalyser workflow tasks
This radar shows how random task clustering performed on a random task sample.

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. 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 6.10, 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 compare the clusters formed by the *ShaReComp* approach. At first, 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, which are visibly offset from one another.

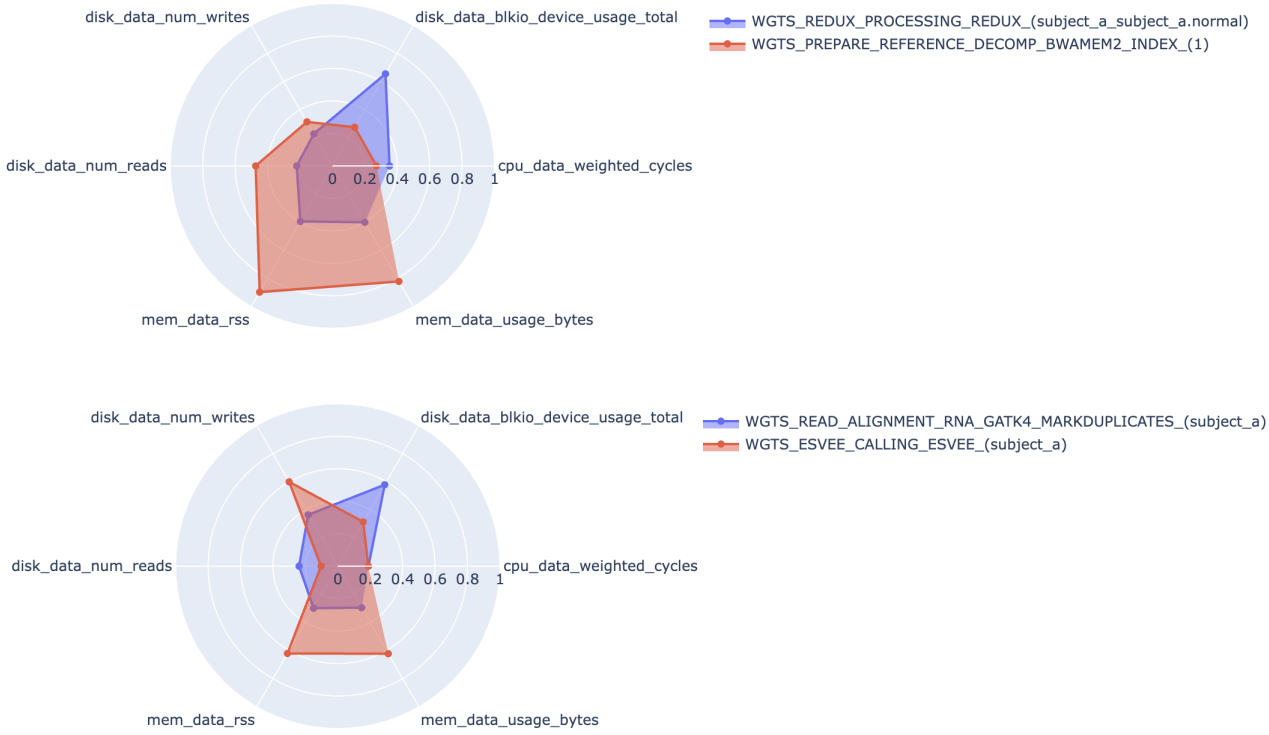

Figure 6.23: ShaReComp Clustering Results on oncoanalyser workflow tasks
This radar shows how the ShaReComp approach performed clustering on a random task sample.

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.

To conclude the evaluation of the task dissimilarity approach, we provide a summary statistic that illustrates *ShaReComp's* performance across all 9 nf-core workflows. For this purpose, we define probe sizes of 20, 30, and 40 tasks per workflow to undergo analysis. For each probe, we again compute time-averaged resource usage metrics and perform both random clustering and *ShaReComp*-based clustering. For each value of the temporal signature within a cluster, pairwise differences between all task signatures are computed and aggregated into a single absolute measure representing the total intra-cluster variation in resource usage. Averaging this measure across all probes yields one comparable metric for both random clustering and *ShaReComp*. Conceptually, a higher intra-cluster distance implies that *ShaReComp* successfully grouped tasks with dissimilar resource usage profiles, whereas smaller distances indicate overlapping or redundant resource behavior, as typically observed in random clustering. As shown in Table 6.11, *ShaReComp* consistently produces higher intra-cluster distances than random clustering for nearly all workflows, demonstrating its ability to capture meaningful dissimilarities among task resource patterns. Notably, workflows such as `rnaseq`, `smrnaseq`, and `methylseq` exhibit the most pronounced improvements, suggesting that *ShaReComp* is particularly effective when workflows contain a diverse mix of compute- and I/O-intensive tasks. In contrast, workflows like `scnanoseq` and `pixelator` show less or inverse variation, which may indicate more homogeneous resource behavior across tasks or limited temporal diversity in their signatures. Overall, these findings confirm that *ShaReComp* enhances the clustering of complementary tasks and generalizes well across different workflow structures and task counts.

Workflow	**Number of Tasks**	**Avg. *ShaReComp* Cluster Difference**	**Avg. Random Cluster Difference**
`atacseq`	72	25,256	24,191
`chipseq`	68	15,652	13,771
`rnaseq`	54	26,712	22,254
`scnanoseq`	83	1,006	1,526
`smrnaseq`	59	28,244	25,424
`pixelator`	44	17,823	22,003
`methylseq`	65	23,650	20,9500
`viralrecon`	51	19,839	16,754
`oncoanalyser`	97	16,552	13,821

Table 6.11: Average inter-cluster difference comparison between *ShaReComp* and random clustering across workflows

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. Whenever applicable, we calculated the R^2 and Mean Absolute Error (MAE) score to assess explanatory model performance and the average absolute difference between predictions and true values in target units.

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 RandomizedSearchCV	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 RandomizedSearchCV	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.

Table 6.12: Summary of model configurations and performance metrics for task-cluster prediction

The 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, 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 with 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 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

Results of Makespan and Energy Consumption

In this section, we evaluate the integration of the co-location-aware *ShaReComp* approach using the *ShaRiff* algorithms. We test the nine nf-core workflows 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`, `smrnaseq`, 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.

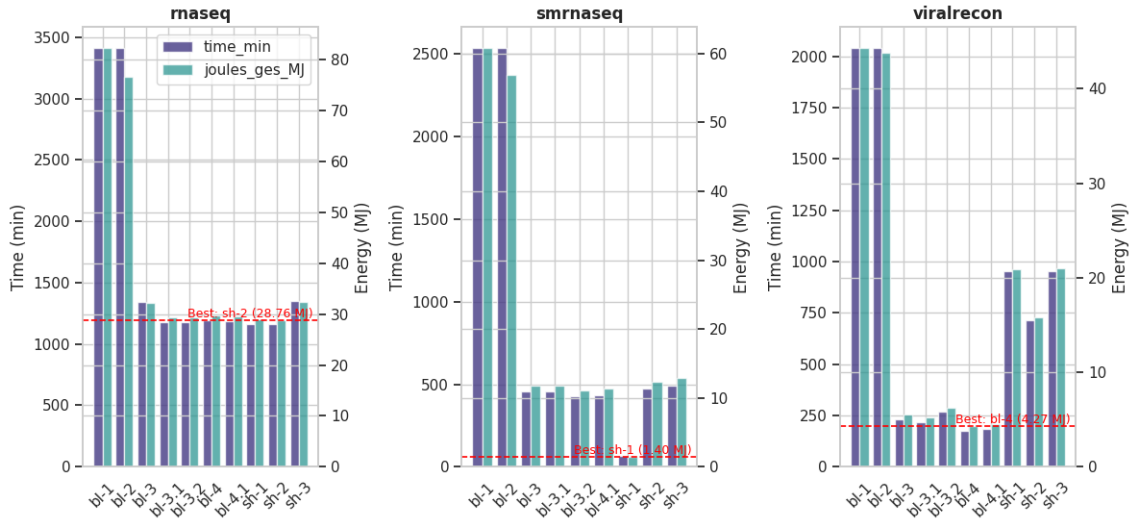

Figure 6.24: Makespan and Energy Consumption per Scheduling Algorithm for 3 Workflows

This figure shows the achieved makespan and energy consumption per scheduling algorithm for 3 selected workflows with highlighting of the best performing variant.

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 ac-

curately monitored resource usage allowed the clustering and task-distance calculations to produce effective co-location decisions. Since the simulation environment 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 `smrnaseq`, *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 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.

Distribution of Makespan and Energy Consumption for all Workflows

While the remaining grouped bar plots are provided in Appendix 8, this section concentrates on the overall performance of the implemented strategies across all evaluated workflows, specifically analyzing their achieved makespan and total 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.

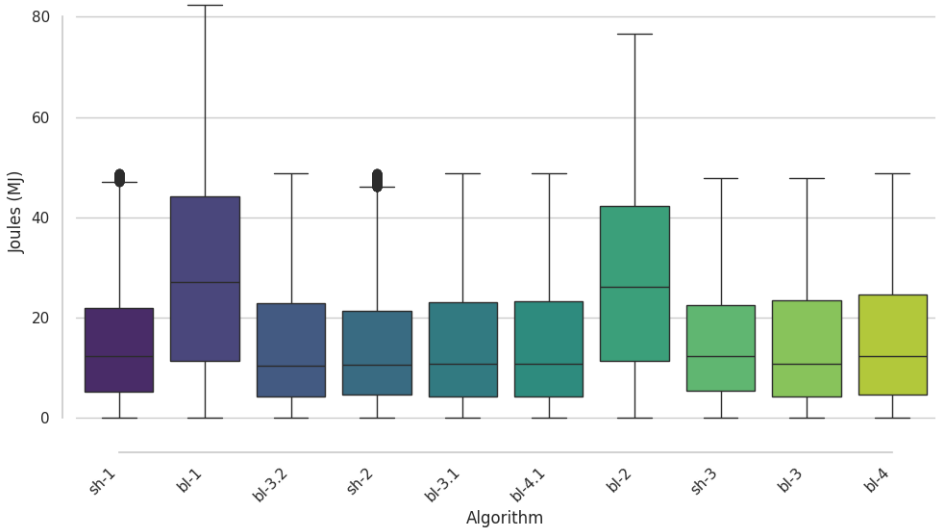

Figure 6.25: Total Energy Consumption per Scheduling Algorithm over 9 Workflows

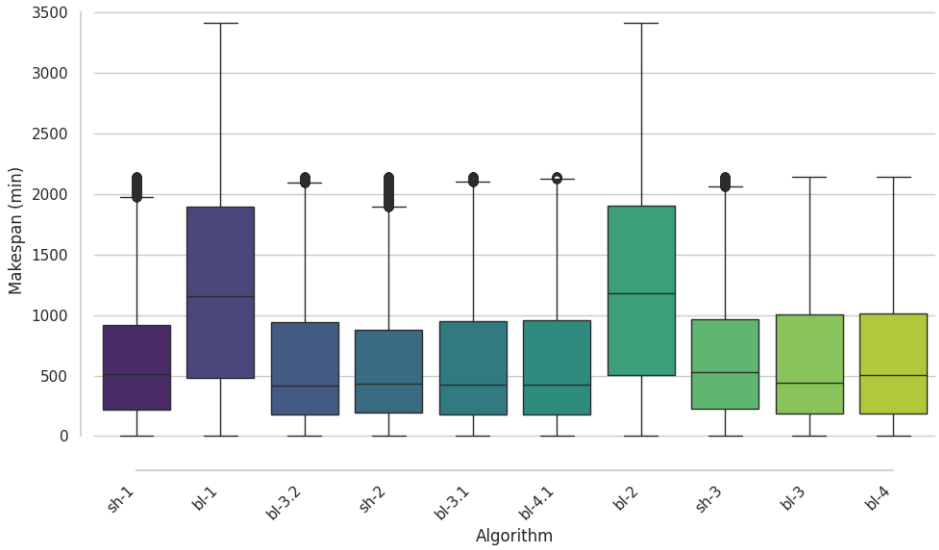

Figure 6.26: Total Makespan per Scheduling Algorithm over 9 Workflows

This figure shows the distribution for both makespan and total energy consumption of all evaluated scheduling algorithm across the 9 nf-core workflows.

A similar pattern appears in the makespan distribution. 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. While all remaining grouped bar plots are presented in Appendix 8, the following paragraph will summarize the evaluation of all implemented strategies across workflows, with particular emphasis on their achieved energy-efficiency, makespan and total energy consumption. 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 improvement for the raw energy consumption and makespan values themselves to conclude the evaluation of the *ShaRiff* algorithms.

Summary Statistics of Energy Efficiency, Makespan, and Energy Consumption

Table 6.13 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%, likely due to the benefits of oversubscribing complementary workloads. Negative improvement values, as seen for `methylseq`, `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.

Workflow	Best Approach	Avg. Baseline Efficiency	Best Efficiency	Improvement
atacseq	ShaRiff-3	0.023304	0.022627	2.906
chipseq	ShaRiff-3	0.024819	0.024764	0.225
methylseq	ShaRiff-1	0.022601	0.022726	-0.556
oncoanalyser	ShaRiff-3	0.022109	0.021989	0.540
pixelator	ShaRiff-1	0.024160	0.024735	-2.379
rnaseq	ShaRiff-2	0.024388	0.024759	-1.521
scnanoseq	ShaRiff-3	0.022203	0.022186	0.076
smrnaseq	ShaRiff-1	0.025061	0.022142	11.649
viralrecon	ShaRiff-2	0.023236	0.022030	5.191

Table 6.13: Efficiency over time improvement of the best *ShaRiff* approach compared to the average baseline efficiency per workflow

Table 6.14 presents the improvement in energy consumption achieved by the best-performing *ShaRiff* variant compared to the average baseline consumption for each workflow. The values indicate how much more energy the workflows consumed when using the corresponding *ShaRiff* algorithm. Higher improvement percentages reflect less consumed power and thus

better energy-awareness. Across all workflows, the results show that *ShaRiff* consistently reduces energy consumption 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 energy consumption by almost 39%, demonstrating strong optimization of co-location and task placement. *ShaRiff*-1 achieves the best performance for `methylseq`, `pixelator`, and `smrnaseq`. The improvement for `smrnaseq` is the most pronounced overall, reaching almost 95%, indicating that this variants 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 energy consumption. 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.

Workflow	Best ShaRiff Approach	Avg. Energy Consumption MJ	Best ShaRiff Energy Consumption	Improvement
`atacseq`	ShaRiff-3	9.919	8.036	18.984
`chipseq`	ShaRiff-3	31.602	19.419	38.552
`methylseq`	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 6.14: Energy Consumption improvement of the best *ShaRiff* approach compared to the average baseline efficiency per workflow.

Table 6.15 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`, `methylseq`, `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 hows 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 and partic-

ularly task heterogeneity, resource complementarity, and average task duration. Overall, the results confirm that informed co-location enables more efficient resource utilization and faster completion times compared to baseline scheduling.

Workflow	**Best ShaRiff Approach**	**Avg. Baseline Makespan**	**Best ShaRiff Makespan**	**Improvement**
`atacseq`	ShaRiff-1	427.167	355.167	16.855
`chipseq`	ShaRiff-3	1305.667	784.167	39.941
`methylseq`	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 6.15: 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 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 alpha when combining runtime and energy in the affinity computation. This refinement would likely produce more reliable and interpretable affinity scores, better reflecting the resource interaction between co-located tasks.

The previously discussed aspect also directly relates to the results obtained from the predictor training phase. 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 low-level features per task, also introduces complexity in feature selection. Deciding which metrics to retain and understanding their respective contributions to prediction accuracy remains an important direction for further investigation. 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 coming 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 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 can potential only contain a small number of tasks due to data interdependencies. To address this, we already 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 to reduce resource contention and improve 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 that extends and adapts an existing formulation of the co-location problem by framing it as a consolidation process to group dissimilar workloads. 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 KCCA together with a regression model 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 should focus on several directions to extend and refine the proposed framework. First, the monitoring system can be improved to achieve broader task coverage, reduced overhead, and more accurate energy estimation, for instance by enhancing eBPF-based measurements and incorporating vendor-independent predictive energy models based on external powermeters. Second, the processing of time-series data should be deepened to generate more expressive and representative feature vectors, supported by dimensionality reduction and feature selection techniques that prioritize features with the highest explanatory power. Third, the current affinity score calculation could be advanced through more detailed, low-level interference measurements, while also improving the handling of static or less dynamic time-series data in distance computations. These key directions will 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 for the successful operation of the *ShaRiff* algorithms as they depend 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 because 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 hold potential. The current design limits resource utilization, as virtual machines cannot be resized dynamically. As a consequence 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 further reduce makespan and energy usage. Leveraging workflow DAG information available in WRENCH alongside SimGrid's detailed idle-core time metrics could further improve scheduling precision and provide

more accurate insights into resource utilization. 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. We plan to extend our simulation by expanding the host pool and adopting alternative resource management strategies to provide richer insights into how co-location and *ShaReComp* improve the performance of workflow execution systems. From a systems perspective, integrating explicit cost functions into the co-location strategies would allow the formulation of optimization problems that support both single- and multi-objective trade-offs. The simulation framework should also be extended and calibrated with realistic energy sources to better reflect actual data center behavior. Finally, future research could reformulate the co-location problem itself from task consolidation towards predicting degradation through resource contention directly thereby moving from analytical modeling to real-time decision-making.

A Workflow-Specific Execution Results of Baseline and Co-location Algorithms

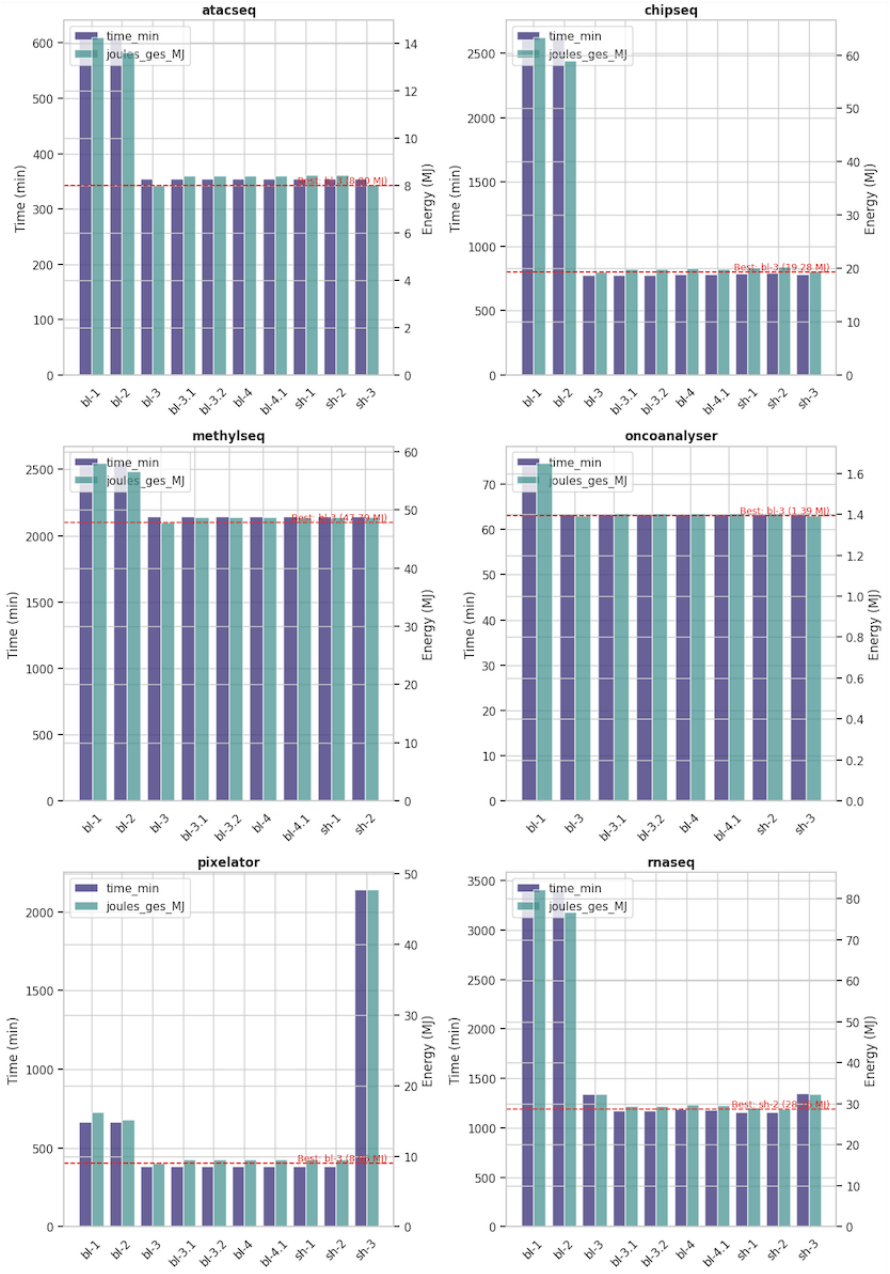

Figure A.27: Execution Results of Baseline and Co-location Algorithms

Execution Results of Baseline and Co-location Algorithms

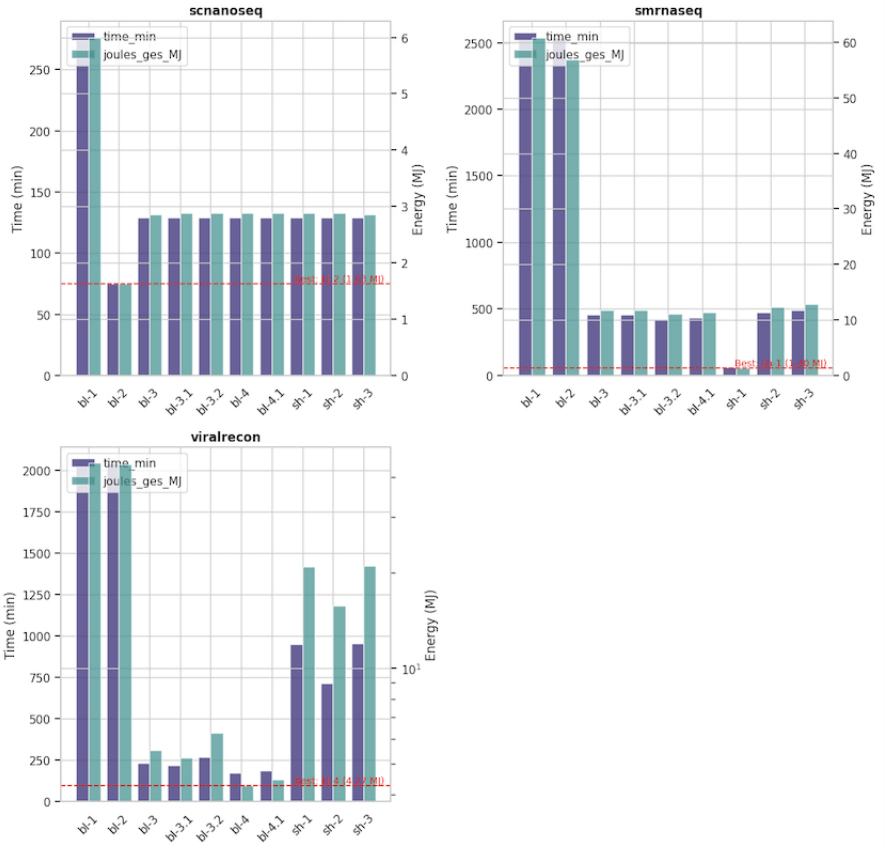

Figure A.28: Execution Results of Baseline and Co-location Algorithms

Execution Results of Baseline and Co-location Algorithms

B Blueprint Baseline Algorithms for the *ShaRiff* Approach

Algorithm 9: Baseline 1 — FIFO Scheduling with Round-Robin Host Assignment

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$
Output: each task $t_i \in T$ executed once all dependencies are resolved

Initialize idle cores $c(h_j) \leftarrow C_j$ for all $h_j \in \mathcal{H}$ Initialize empty ready queue $\mathcal{Q}$ Enqueue all source tasks (no predecessors) into $\mathcal{Q}$ in FIFO order Set host index $k \leftarrow 1$

while *not all tasks* $t_i \in T$ *completed* **do**
    while $\mathcal{Q}$ *not empty* **do**
        Let t_i be the head of $\mathcal{Q}$ Select host h_k in round-robin order from $\mathcal{H}$ **if** $c(h_k) > 0$ **then**
            Reserve one core: $c(h_k) \leftarrow c(h_k) - 1$ Launch VM $v = (1,\, M_{t_i},\, h_k)$ to execute t_i Dequeue t_i from $\mathcal{Q}$
        Advance to next host index $k \leftarrow (k \bmod m) + 1$
    Wait until a running task t_r completes Release its host core: $c(h(t_r)) \leftarrow c(h(t_r)) + 1$ For each successor t_s of t_r, enqueue t_s into $\mathcal{Q}$ if all predecessors are completed
return *workflow complete*

Algorithm 10: Baseline 2 — FIFO Scheduling with Host Backfilling

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$
Output: each task $t_i \in T$ executed once all dependencies are resolved

Initialize idle cores $c(h_j) \leftarrow C_j$ for all $h_j \in \mathcal{H}$ Initialize empty ready queue $\mathcal{Q}$ Enqueue all source tasks (no predecessors) into $\mathcal{Q}$ in FIFO order

while *not all tasks* $t_i \in T$ *completed* **do**
    while $\mathcal{Q}$ *not empty* **do**
        Let t_i be the head of $\mathcal{Q}$ Find the first host $h_j \in \mathcal{H}$ with $c(h_j) > 0$ **if** *such* h_j *exists* **then**
            Reserve one core: $c(h_j) \leftarrow c(h_j) - 1$ Launch VM $v = (1,\, M_{t_i},\, h_j)$ to execute t_i Dequeue t_i from $\mathcal{Q}$
        else
            break ; `// no idle host available, defer assignment`
    Wait until a running task t_r completes Release its core: $c(h(t_r)) \leftarrow c(h(t_r)) + 1$ For each successor t_s of t_r, enqueue t_s into $\mathcal{Q}$ if all predecessors are completed
return *workflow complete*

Algorithm 11: Baseline 3 — FIFO Scheduling with VM Co-Location

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$
Output: all tasks $t_i \in T$ executed, with ready tasks co-located on shared VMs when possible

Initialize idle cores $c(h_j) \leftarrow C_j$ for all $h_j \in \mathcal{H}$ Initialize empty ready queue $\mathcal{Q}$ Enqueue all source tasks (no predecessors) into $\mathcal{Q}$ in FIFO order
while *not all tasks* $t_i \in T$ *completed* **do**
  while $\mathcal{Q}$ *not empty* **do**
    Find first host h_j with $c(h_j) > 0$ **if** *no such host exists* **then**
      break ; `// wait for resource release`
    Select subset $\mathcal{T}_j \subseteq \mathcal{Q}$ such that sum(req_cores($\mathcal{T}_j$)) $\leq c(h_j)$ Allocate VM $v_j = (\text{sum(req\_cores}(\mathcal{T}_j)), \text{sum(req\_mem}(\mathcal{T}_j)), h_j)$
    Launch all tasks $t_i \in \mathcal{T}_j$ on v_j Update $c(h_j) \leftarrow c(h_j) - \text{cores}(v_j)$ and remove $\mathcal{T}_j$ from $\mathcal{Q}$
  Wait until a task t_r completes Release its cores: $c(h(t_r)) \leftarrow c(h(t_r)) + \text{req\_cores}(t_r)$ **if** *all tasks on* v_j *finished* **then**
    Destroy VM v_j
  For each successor t_s of t_r, enqueue t_s into $\mathcal{Q}$ if all predecessors are completed
return *workflow complete*

Algorithm 12: Baseline 3.1 — FIFO Scheduling with Max-Core VM Co-Location

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$
Output: all tasks $t_i \in T$ executed, co-located on VMs at hosts with maximum idle capacity

Initialize idle cores $c(h_j) \leftarrow C_j$ for all $h_j \in \mathcal{H}$ Initialize empty ready queue $\mathcal{Q}$ Enqueue all source tasks (no predecessors) into $\mathcal{Q}$ in FIFO order
while *not all tasks* $t_i \in T$ *completed* **do**
  while $\mathcal{Q}$ *not empty* **do**
    Select host $h_j = \arg\max_{h \in \mathcal{H}} c(h)$ **if** $c(h_j) = 0$ **then**
      break ; `// no host available, wait for completion`
    Select subset $\mathcal{T}_j \subseteq \mathcal{Q}$ such that sum(req_cores($\mathcal{T}_j$)) $\leq c(h_j)$ Allocate VM $v_j = (\text{sum(req\_cores}(\mathcal{T}_j)), \text{sum(req\_mem}(\mathcal{T}_j)), h_j)$
    Launch all tasks $t_i \in \mathcal{T}_j$ on v_j Update $c(h_j) \leftarrow c(h_j) - \text{cores}(v_j)$ and remove $\mathcal{T}_j$ from $\mathcal{Q}$
  Wait until a task t_r completes Release its resources: $c(h(t_r)) \leftarrow c(h(t_r)) + \text{req\_cores}(t_r)$ **if** *all tasks on* v_j *finished* **then**
    Destroy VM v_j
  For each successor t_s of t_r, enqueue t_s into $\mathcal{Q}$ if all predecessors are completed
return *workflow complete*

Algorithm 13: Baseline 3.2 — FIFO Scheduling with Max-Parallel VM Co-Location

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$
Output: all tasks $t_i \in T$ executed with host-level parallel assignment and intra-host co-location

Initialize idle cores $c(h_j) \leftarrow C_j$ for all $h_j \in \mathcal{H}$ Initialize empty ready queue $\mathcal{Q}$ Enqueue all source tasks (no predecessors) into $\mathcal{Q}$ in FIFO order
while *not all tasks* $t_i \in T$ *completed* **do**
  Build available-host list $L = \{\, h \in \mathcal{H} \mid c(h) > 0 \,\}$, sorted by $c(h)$ descending
  if L *is empty* **then**
    Wait for any running task to finish; update $c(\cdot)$; **continue**
  Initialize empty host-batch mapping set $\mathcal{M}$ **foreach** *host* $h \in L$ ***and*** *while* $\mathcal{Q}$ *not empty* **do**
    Select subset $\mathcal{T}_h \subseteq \mathcal{Q}$ in FIFO order such that sum(req_cores($\mathcal{T}_h$)) $\leq c(h)$
    if $\mathcal{T}_h \neq \emptyset$ **then**
      Add mapping $(h, \mathcal{T}_h)$ to $\mathcal{M}$
  foreach *mapping* $(h, \mathcal{T}_h) \in \mathcal{M}$ **do**
    Allocate VM v_h = (sum(req_cores($\mathcal{T}_h$)), sum(req_mem($\mathcal{T}_h$)), h)
    Launch all $t \in \mathcal{T}_h$ on v_h Update $c(h) \leftarrow c(h) - \text{cores}(v_h)$ and remove $\mathcal{T}_h$ 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** *its VM* v_h *has no active tasks* **then**
    Destroy v_h
  For each successor t_s of t_r, enqueue t_s into $\mathcal{Q}$ if all predecessors have completed
return *workflow complete*

Algorithm 14: Baseline 4 — FIFO Scheduling with VM Co-Location and Controlled Over-Subscription

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$, oversubscription factor α
Output: all tasks $t_i \in T$ executed, allowing limited CPU oversubscription on selected hosts

Initialize idle cores $c(h_j) \leftarrow C_j$ for all $h_j \in \mathcal{H}$ Initialize empty ready queue $\mathcal{Q}$
Enqueue all source tasks (no predecessors) into $\mathcal{Q}$ in FIFO order
while *not all tasks* $t_i \in T$ *completed* **do**
  if $\mathcal{Q}$ *is empty* **then**
    Wait for any running task to complete; update $c(h_j)$; **continue**
  Select host $h_j = \arg\max_{h \in \mathcal{H}} c(h)$; `// host with most idle cores`
  Determine permitted task capacity $n_{\max} = \lceil c(h_j) \times (1 + \alpha) \rceil$ Select up to $n_{\max}$ ready tasks from $\mathcal{Q}$ into set $\mathcal{T}_j$ Compute total requested cores $C_{\text{req}} = \text{sum(req\_cores}(\mathcal{T}_j))$ Compute total requested memory $M_{\text{req}} = \text{sum(req\_mem}(\mathcal{T}_j))$ **if** $C_{req} > c(h_j)$ **then**
    Allocate VM $v_j = (c(h_j),\, M_{\text{req}},\, h_j)$; `// oversubscription active`
  else
    Allocate VM $v_j = (C_{\text{req}},\, M_{\text{req}},\, h_j)$
  Launch all tasks $t \in \mathcal{T}_j$ on v_j Update $c(h_j) \leftarrow \max(0,\, c(h_j) - C_{\text{req}})$ and remove $\mathcal{T}_j$ 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** *all tasks on* v_j *finished* **then**
    Destroy VM v_j
  For each successor t_s of t_r, enqueue t_s into $\mathcal{Q}$ if all predecessors are completed
return *workflow complete*

Algorithm 15: Baseline 4.1 — FIFO Scheduling with Max-Parallel VM Co-Location and Controlled Over-Subscription

Input: workflow $G = (T, E)$, hosts $\mathcal{H} = \{h_1, \ldots, h_m\}$, oversubscription factor α
Output: all tasks $t_i \in T$ executed with inter-host parallel assignment, intra-host co-location, and controlled CPU overcommitment

Initialize idle cores $c(h_j) \leftarrow C_j$ for all $h_j \in \mathcal{H}$ Initialize empty ready queue $\mathcal{Q}$; enqueue all source tasks in FIFO order
while *not all tasks* $t_i \in T$ *completed* **do**
  Build available-host list $L = \{\, h \in \mathcal{H} \mid c(h) > 0 \,\}$, sorted by $c(h)$ descending **if** L *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
  Initialize empty host-batch mappings $\mathcal{M}$
  foreach *host* $h \in L$ *and while* $\mathcal{Q}$ *not empty* **do**
    $n_{\max} \leftarrow \lceil c(h) \cdot (1 + \alpha) \rceil$ Select $\mathcal{T}_h \subseteq \mathcal{Q}$ (FIFO) with at most $n_{\max}$ tasks **if** $\mathcal{T}_h \neq \emptyset$ **then**
      add $(h, \mathcal{T}_h)$ to $\mathcal{M}$
  foreach *mapping* $(h, \mathcal{T}_h) \in \mathcal{M}$ **do**
    $C_{\text{req}} \leftarrow \text{sum}(\text{req\_cores}(\mathcal{T}_h))$ $M_{\text{req}} \leftarrow \text{sum}(\text{req\_mem}(\mathcal{T}_h))$ **if** $C_{req} > c(h)$ **then**
      Allocate $v_h = (c(h),\, M_{\text{req}},\, h)$; `// oversubscription active`
    else
      Allocate $v_h = (C_{\text{req}},\, M_{\text{req}},\, h)$
    Launch all $t \in \mathcal{T}_h$ on v_h $c(h) \leftarrow \max(0,\, c(h) - C_{\text{req}})$ Remove $\mathcal{T}_h$ 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** *its VM* v_h *has no active tasks* **then**
    Destroy v_h
  For each successor t_s of t_r, enqueue t_s into $\mathcal{Q}$ if all predecessors are completed
return *workflow complete*

Bibliography

[1] Foudil Abdessamia, Wei-Zhe Zhang, and Yu-Chu Tian. "Energy-efficiency virtual machine placement based on binary gravitational search algorithm". In: *Cluster Computing* 23.3 (2020), pp. 1577–1588.

[2] Abdulmohsen Algarni, Iqrar Shah, Ali Imran Jehangiri, Mohammed Alaa Ala'Anzy, and Zulfiqar Ahmad. "Predictive Energy Management for Docker Containers in Cloud Computing: A Time Series Analysis Approach". In: *IEEE Access* 12 (2024), pp. 52524–52538. DOI: 10.1109/ACCESS.2024.3387436.

[3] Jordi Arjona Aroca, Angelos Chatzipapas, Antonio Fernández Anta, and Vincenzo Mancuso. "A Measurement-Based Characterization of the Energy Consumption in Data Center Servers". In: *IEEE Journal on Selected Areas in Communications* 33.12 (2015), pp. 2863–2877. DOI: 10.1109/JSAC.2015.2481198.

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

[5] Jonathan Bader, Fabian Lehmann, Lauritz Thamsen, Ulf Leser, and Odej Kao. "Lotaru: Locally predicting workflow task runtimes for resource management on heterogeneous infrastructures". In: *Future Generation Computer Systems* 150 (2024), pp. 171–185. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.08.022.

[6] Jonathan Bader, Kathleen West, Soeren Becker, Svetlana Kulagina, Fabian Lehmann, Lauritz Thamsen, Henning Meyerhenke, and Odej Kao. *Predicting the Performance of Scientific Workflow Tasks for Cluster Resource Management: An Overview of the State of the Art.* 2025. arXiv: 2504.20867 [cs.DC].

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

[8] Sergey Blagodurov and Alexandra Fedorova. "In search for contention-descriptive metrics in HPC cluster environment". In: *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering.* ICPE '11. Karlsruhe, Germany: Association for Computing Machinery, 2011, pp. 457–462. ISBN: 9781450305198. DOI: 10.1145/1958746.1958815.

[9] Sergey Blagodurov and Alexandra Fedorova. "Towards the contention aware scheduling in HPC cluster environment". In: *Journal of Physics: Conference Series* 385.1 (2012), p. 012010. DOI: 10.1088/1742-6596/385/1/012010.

[10] Sergey Blagodurov, Alexandra Fedorova, Evgeny Vinnik, Tyler Dwyer, and Fabien Hermenier. "Multi-objective job placement in clusters". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC15. ACM, 2015. DOI: 10.1145/2807591.2807636.

[11] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. "Contention-Aware Scheduling on Multicore Systems". In: *ACM Transactions on Computer Systems* 28.4 (2010), pp. 1–45. ISSN: 1557-7333. DOI: 10.1145/1880018.1880019.

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

[13] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. "Predictive Modeling for Job Power Consumption in HPC Systems". In: *High*

Performance Computing. Springer International Publishing, 2016, pp. 181–199. ISBN: 9783319413211. DOI: 10.1007/978-3-319-41321-1_10.

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

[15] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. "Case Study on Co-scheduling for HPC Applications". In: *2015 44th International Conference on Parallel Processing Workshops*. 2015, pp. 277–285. DOI: 10.1109/ICPPW.2015.38.

[16] Alex D. Breslow, Leo Porter, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Dean M. Tullsen, and Allan E. Snavely. "The case for colocation of high performance computing workloads". In: *Concurr. Comput.: Pract. Exper.* 28.2 (Feb. 2016), pp. 232–251. ISSN: 1532-0626. DOI: 10.1002/cpe.3187.

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

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

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

[20] Ruobing Chen, Wangqi Peng, Yusen Li, Xiaoguang Liu, and Gang Wang. "Orchid: An Online Learning Based Resource Partitioning Framework for Job Colocation With Multiple Objectives". In: *IEEE Transactions on Computers* 72.12 (2023), 3440 $ÌÍ MßHB. ISSN: 2326-3814. DOI: 10.1109/tc.2023.3303959.

[21] Ruobing Chen, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. "OLPart: Online Learning based Resource Partitioning for Colocating Multiple Latency-Critical Jobs on Commodity Computers". In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys-23. ACM, 2023, pp. 347–364. DOI: 10.1145/3552326.3567490.

[22] Anita Choudhary, Mahesh Chandra Govil, Girdhari Singh, Lalit K. Awasthi, and Emmanuel S. Pilli. "Energy-aware scientific workflow scheduling in cloud environment". In: *Cluster Computing* 25.6 (2022), pp. 3845–3874. ISSN: 1573-7543. DOI: 10.1007/s10586-022-03613-3.

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

[24] Daniel Dauwe, Eric Jonardi, Ryan D. Friese, Sudeep Pasricha, Anthony A. Maciejewski, David A. Bader, and Howard Jay Siegel. "HPC node performance and energy modeling with the co-location of applications". In: *The Journal of Supercomputing* 72.12 (2016), pp. 4771–4809. ISSN: 1573-0484. DOI: 10.1007/s11227-016-1783-y.

[25] Juan J. Durillo, Vlad Nae, and Radu Prodan. "Multi-objective energy-efficient workflow scheduling using list-based heuristics". In: *Future Generation Computer Systems* 36 (2014), pp. 221–236. ISSN: 0167-739X. DOI: 10.1016/j.future.2013.07.005.

[26] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. "A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads". In: *SC '12: Proceedings of*

the International Conference on High Performance Computing, Networking, Storage and Analysis. 2012, pp. 1–11. DOI: 10.1109/SC.2012.11.

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

[28] Google. *cAdvisor — google/cadvisor (GitHub)*. https://github.com/google/cadvisor. Accessed: 27 Oct 2025. Oct. 2025.

[29] Brendan Gregg. *BPF Performance Tools*. 1st. Editor-in-Chief: Mark L. Taub; Series Editor: Brian Kernighan; Executive Editor: Greg Doench; Senior Project Editor: Lori Lyons. Boston, MA: Pearson Education, Inc., 2020. ISBN: 9780136554820.

[30] Robert Hood, Haoqiang Jin, Piyush Mehrotra, Johnny Chang, Jahed Djomehri, Sharad Gavali, Dennis Jespersen, Kenichi Taylor, and Rupak Biswas. "Performance impact of resource contention in multicore systems". In: *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470399.

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

[32] Zhengxiong Hou, Hong Shen, Xingshe Zhou, Jianhua Gu, Yunlan Wang, and Tianhai Zhao. "Prediction of job characteristics for intelligent resource allocation in HPC systems: a survey and future directions". In: *Frontiers of Computer Science* 16.5 (2022). ISSN: 2095-2236. DOI: 10.1007/s11704-022-0625-8.

[33] Yichao Jin, Yonggang Wen, and Qinghua Chen. "Energy efficiency and server virtualization in data centers: An empirical investigation". In: *2012 Proceedings IEEE INFOCOM Workshops*. 2012, pp. 133–138. DOI: 10.1109/INFCOMW.2012.6193474.

[34] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. "Characterizing and profiling scientific workflows". In: *Future Generation Computer Systems* 29.3 (2013). Special Section: Recent Developments in High Performance Computing and Security, pp. 682–692. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2012.08.015.

[35] Nizam Kashif Khan. "Energy Measurement and Modeling in High Performance Computing with Intel's RAPL". Electronic ISBN: 978-952-60-7892-2; Supervising Professor: Antti Ylä-Jääski; Thesis Advisor: Jukka Nurminen. Doctoral dissertation (article). Espoo, Finland: Aalto University, 2018. ISBN: 978-952-60-7891-5.

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

[37] Sachin Kumar, Saurabh Pal, Satya Singh, Raghvendra Pratap Singh, Sanjay Kumar Singh, and Priya Jaiswal. "An Energy & Cost Efficient Task Consolidation Algorithm for Cloud Computing Systems". In: *Advancements in Smart Computing and Information Security*. Ed. by Sridaran Rajagopal, Parvez Faruki, and Kalpesh Popat. Cham: Springer Nature Switzerland, 2022, pp. 446–454. ISBN: 978-3-031-23092-9.

[38] Justin Kur, Jingshu Chen, Ji Xue, and Jun Huang. "Resolution Matters: Revisiting Prediction-Based Job Co-location in Public Clouds". In: *2022 IEEE/ACM 15th In-*

ternational Conference on Utility and Cloud Computing (UCC). 2022, pp. 163–166. DOI: 10.1109/UCC56403.2022.00029.

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

[40] Fabian Lehmann, Jonathan Bader, Friedrich Tschirpke, Ninon De Mecquenem, Ansgar Lößer, Soeren Becker, Katarzyna Ewa Lewińska, Lauritz Thamsen, and Ulf Leser. "WOW: Workflow-Aware Data Movement and Task Scheduling for Dynamic Scientific Workflows". In: *2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2025, pp. 525–538. DOI: 10.1109/CCGRID64434.2025.00021.

[41] Xi Li, Anthony Ventresque, Jesus Omana Iglesias, and John Murphy. "Scalable correlation-aware virtual machine consolidation using two-phase clustering". In: *2015 International Conference on High Performance Computing and Simulation (HPCS)*. 2015, pp. 237–245. DOI: 10.1109/HPCSim.2015.7237045.

[42] Xiang Li, Linfeng Wen, Minxian Xu, and Kejiang Ye. "An Interference-aware Approach for Co-located Container Orchestration with Novel Metric". In: *2023 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. 2023, pp. 600–607. DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics60724.2023.00111.

[43] Junwen Liu, Shiyong Lu, and Dunren Che. "A Survey of Modern Scientific Workflow Scheduling Algorithms and Systems in the Era of Big Data". In: *2020 IEEE International Conference on Services Computing (SCC)*. 2020, pp. 132–141. DOI: 10.1109/SCC49832.2020.00026.

[44] Maicon Melo Alves and LÃºcia Maria de AssumpÃ§Ã£o Drummond. "A multivariate and quantitative model for predicting cross-application interference in virtual environments". In: *Journal of Systems and Software* 128 (2017), pp. 150–163. ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.04.001.

[45] Maicon Melo Alves, Luan Teylo, Yuri Frota, and Lúcia M.A. Drummond. "An Interference-Aware Virtual Machine Placement Strategy for High Performance Computing Applications in Clouds". In: *2018 Symposium on High Performance Computing Systems (WSCAD)*. 2018, pp. 94–100. DOI: 10.1109/WSCAD.2018.00024.

[46] Tarek Menouer and Patrice Darmon. "Containers Scheduling Consolidation Approach for Cloud Computing". In: *Pervasive Systems, Algorithms and Networks*. Ed. by Christian Esposito, Jiman Hong, and Kim-Kwang Raymond Choo. Cham: Springer International Publishing, 2019, pp. 178–192. ISBN: 978-3-030-30143-9.

[47] Andreas Merkel, Jan Stoess, and Frank Bellosa. "Resource-conscious scheduling for energy efficiency on multicore processors". In: *Proceedings of the 5th European conference on Computer systems*. EuroSys-10. ACM, 2010, p. 150. DOI: 10.1145/1755913.1755930.

[48] Ali Mohammadzadeh, Mohammad Masdari, Farhad Soleimanian Gharehchopogh, and Ahmad Jafarian. "A hybrid multi-objective metaheuristic optimization algorithm for scientific workflow scheduling". In: *Cluster Computing* 24.2 (2020), pp. 1479–1503. ISSN: 1573-7543. DOI: 10.1007/s10586-020-03205-z.

[49] Murata Manufacturing Co., Ltd. *Data Center 4*. Accessed: 15 Oct 2025. Oct. 2025.

[50] Tirthak Patel, Adam Wagenhäuser, Christopher Eibel, Timo Hönig, Thomas Zeiser, and Devesh Tiwari. "What does Power Consumption Behavior of HPC Jobs Reveal? :

Demystifying, Quantifying, and Predicting Power Consumption Characteristics". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2020, pp. 799–809. DOI: 10.1109/IPDPS47924.2020.00087.

[51] Guillaume Raffin and Denis Trystram. "Dissecting the software-based measurement of CPU energy consumption: a comparative analysis". In: (2024). arXiv: 2401.15985 [astro-ph.IM].

[52] Jitendra Kumar Rai, Atul Negi, Rajeev Wankar, and K.D. Nayak. "Performance Prediction on Multi-core Processors". In: *2010 International Conference on Computational Intelligence and Communication Networks*. 2010, pp. 633–637. DOI: 10.1109/CICN.2010.125.

[53] Thomas Renner, Lauritz Thamsen, and Odej Kao. "CoLoc: Distributed data and container colocation for data-intensive applications". In: *2016 IEEE International Conference on Big Data (Big Data)*. 2016, pp. 3008–3015. DOI: 10.1109/BigData.2016.7840954.

[54] Youssef Saadi, Soufiane Jounaidi, Said El Kafhali, and Hicham Zougagh. "Reducing energy footprint in cloud computing: a study on the impact of clustering techniques and scheduling algorithms for scientific workflows". In: *Computing* 105.10 (2023), pp. 2231–2261. ISSN: 1436-5057. DOI: 10.1007/s00607-023-01182-w.

[55] Altino M. Sampaio and Jorge G. Barbosa. "Estimating Effective Slowdown of Tasks in Energy-Aware Clouds". In: *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*. 2014, pp. 101–108. DOI: 10.1109/ISPA.2014.22.

[56] Mingtian Shao, Kai Lu, and Wenzhe Zhang. "Survey on Virtualization of High Performance Computing". In: *2021 International Conference on Intelligent Computing, Automation and Applications (ICAA)*. 2021, pp. 39–41. DOI: 10.1109/ICAA53760.2021.00015.

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

[58] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. "Chapter 2 - HPC Architecture 1: Systems and Technologies". In: *High Performance Computing*. Ed. by Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. Boston: Morgan Kaufmann, 2018, pp. 43–82. ISBN: 978-0-12-420158-3. DOI: https://doi.org/10.1016/B978-0-12-420158-3.00002-2.

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

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

[61] Aurelio Vivas, Andrei Tchernykh, and Harold Castro. "Trends, Approaches, and Gaps in Scientific Workflow Scheduling: A Systematic Review". In: *IEEE Access* 12 (2024), pp. 182203–182231. DOI: 10.1109/ACCESS.2024.3509218.

[62] Huijun Wang and Oliver Sinnen. "List-Scheduling versus Cluster-Scheduling". In: *IEEE Transactions on Parallel and Distributed Systems* 29.8 (2018), pp. 1736–1749. DOI: 10.1109/TPDS.2018.2808959.

[63] Ze Wang, Tim Suß, Andre Brinkmann, and Lars Nagel. "Contenders: Predicting Cache Contention of Co-Scheduled Applications". In: *2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2025, pp. 343–352. DOI: 10.1109/CCGRID64434.2025.00017.

[64] Mehul Warade, Kevin Lee, Chathurika Ranaweera, and Jean-Guy Schneider. "Monitoring the Energy Consumption of Docker Containers". In: *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2023, pp. 1703–1710. DOI: 10.1109/COMPSAC57700.2023.00263.

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

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

[67] Zhiheng Zhong, Jiabo He, Maria A. Rodriguez, Sarah Erfani, Ramamohanarao Kotagiri, and Rajkumar Buyya. "Heterogeneous Task Co-location in Containerized Cloud Computing Environments". In: *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. 2020, pp. 79–88. DOI: 10.1109/ISORC49007.2020.00021.

[68] Huanzhou Zhu, Ligang He, Bo Gao, Kenli Li, Jianhua Sun, Hao Chen, and Keqin Li. "Modelling and Developing Co-scheduling Strategies on Multicore Processors". In: *2015 44th International Conference on Parallel Processing*. 2015, pp. 220–229. DOI: 10.1109/ICPP.2015.31.

[69] Jianyong Zhu, Renyu Yang, Chunming Hu, Tianyu Wo, Shiqing Xue, Jin Ouyang, and Jie Xu. "Perph: A Workload Co-location Agent with Online Performance Prediction and Resource Inference". In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021, pp. 176–185. DOI: 10.1109/CCGrid51090.2021.00027.

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

[71] cca_zoo contributors. *KCCA — cca_zoo documentation*. Accessed: 21 Oct 2025. Oct. 2025.

[72] scikit-learn developers. *Hierarchical clustering (Clustering) — scikit-learn*. Accessed: 7 Oct 2025. Oct. 2025.

[73] scikit-learn developers. *Linear models — scikit-learn*. https://scikit-learn.org/stable/modules/linear_model.html. Accessed: 3 Oct 2025. Oct. 2025.

[74] scikit-learn developers. *RandomForestRegressor — scikit-learn*. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html. Accessed: 12 Oct 2025. Oct. 2025.