International Journal of Innovative Technology and Interdisciplinary Sciences
https://journals.tultech.eu/index.php/ijitis
ISSN: 2613-7305
Volume 9, Issue 3
DOI: https://doi.org/10.15157/IJITIS.2026.9.3.1942-1971
Received: 07.06.2026; Revised: 09.07.2026; Accepted: 04.09.2026

Research Article

Predicting the Power Impact of Scheduling Decisions in Kubernetes Using Fine-Grained Monitoring and XGBoost

Megi Tartari* , **Genti Daci** , **Elinda Kajo Meçe**

Computer Engineering Department, Polytechnic University of Tirana, Tirana, Albania
***megi.tartari@fti.edu.al**

Abstract

The rapid growth of cloud data centres has increased their energy consumption and environmental footprint, highlighting the need for more energy-efficient resource management. Kubernetes has become a widely adopted container orchestration platform for automating the deployment, scaling, and management of containerized workloads. This study investigates the impact of Kubernetes scheduling decisions on cluster power consumption. A fine-grained monitoring system was implemented to characterize application behaviour and cluster state by collecting metrics at the node, shared-resource, and container levels. Controlled pod-placement scenarios were designed to evaluate how workload distribution topology, microservice affinity, and resource contention affect power consumption. Using the collected dataset, an XGBoost model was developed to predict cluster power consumption associated with pod placement based on pre-scheduling system-state metrics. The model achieved an (R^2) score of 93.2%, demonstrating high predictive accuracy. Building on these results, future work will focus on developing a customized Kubernetes scheduler based on reinforcement learning and integrating the power-prediction model to enable energy-aware pod placement. The proposed approach aims to support more sustainable and energy-efficient cloud data centre operations.

Keywords: Kubernetes; Scheduler; Microservice Affinity; Interference; Power Consumption; Monitoring; Prometheus.

INTRODUCTION

The dominant architectural model of large-scale systems deployed on cloud today is micro-service based, in which the system is decomposed into loosely coupled micro-services that provide independence in development, testing, deployment and scaling. The enabling technology of micro-services is containerization, representing a virtualization at the operating system level. The container encapsulates the application and its dependencies. Compared to VMs, containers provide lower resource consumption, faster startup and termination speed and higher portability.

Among the container orchestrators today, Kubernetes is distinguished as the de-facto platform used for large scale containerized workloads deployed on cloud today. It enables automation of deployment, scaling, and management during the container's lifecycle. A crucial component of Kubernetes is the scheduler, with its role on choosing the appropriate node for placing a container. The main criteria of selecting the best node is based on the static configurations of central processing unit (CPU), memory requests and limits per container, which often diverge from the application's real needs. This mismatch between the configured resources and the actual workload requirements leads to two problems: under-provisioning which results in performance degradation (due to pod evictions), and overprovisioning which causes inefficient resource utilization and, consequently, inefficient power consumption. Besides these problems, the default Kubernetes scheduler has limitations related to its unawareness on the dependency relationship between micro-services (micro-service affinity), unawareness on the applications interference patters and on the impact of the scheduling decision to the environment.

Cloud data centres consume substantial amounts of energy, resulting in a considerable environmental footprint due to their associated CO_2 emissions. Shaw et al. reported that data centres consumed approximately 7% of global electricity in 2022, a figure projected to reach 13% by 2030 [1]. The sector is also responsible for an estimated 2% of global CO_2 emissions, placing cloud infrastructure under increasing regulatory and societal pressure to reduce its carbon intensity.

Additionally, it is still an open topic for research community today, to improve the scheduler awareness towards minimizing environmental impact.

The contributions of our research work are as follows:

- A fine-grained monitoring framework was implemented to characterize cluster state and application behavior at multiple levels.
- Controlled pod-placement scenarios were designed to systematically cover different types and levels of resource interference, microservice affinity, and workload topology.
- A dataset comprising 4978 samples, representing individual pod-placement events, was constructed to analyze and predict the impact of scheduling decisions on cluster power consumption under specific pre-scheduling cluster states.
- The collected samples were analyzed at the scenario level, grouping pod placements associated with the same experimental scenario. This approach accounts for the fact that cluster power consumption is influenced not only by a newly placed pod but also by pods that are already running. Such scenario-level analysis provides a systematic approach for assessing the cumulative impact of scheduling decisions on cluster power consumption.
- Exploratory Data Analysis (EDA) and SHAP analysis were performed to identify the features with the greatest influence on the change in cluster power consumption (ΔPower).

- Baseline models and an XGBoost model were developed to predict the power consumption delta resulting from pod placement, defined as the difference between the cluster's mean power draw during and prior to the pod's lifecycle. The models use baseline system-state metrics collected before the scheduling decision. The XGBoost model outperformed the baseline models, achieving an (R^2) of 93%.

The remainder of this paper is organized as follows. The relevant literature and related work are first reviewed, followed by a description of the system design. The collected dataset and feature engineering approach are then presented, followed by an exploratory data analysis of the collected measurements. Next, an XGBoost model is developed and evaluated for predicting cluster power consumption. Finally, the main findings and conclusions are summarized, together with directions for future research.

RELATED WORKS

Optimizing Kubernetes scheduling while balancing conflicting objectives, such as the cloud provider's operational cost and the quality of service delivered to end users, remains an open topic within the research community today. To support more informed scheduling decisions, commonly used approaches include cluster monitoring, workload profiling, and the analysis of collected data using machine learning algorithms to derive valuable insights for decision-making.

Many scheduling solutions are proposed using energy efficiency and environmental sustainability as primary optimization goals [2-5, 11-13]. The study in [2] proposed a scheduler based on a Deep Q-Network (DQN). Their DRS scheduler monitors six resources simultaneously: CPU, memory utilization, network packet receive/transmit rate, disk read/write rate. DRS achieved more balanced workload distribution across the cluster and improved average resource utilization by 27.29 %. Authors in [14, 15] used deep learning and reinforcement learning for Kubernetes scheduling, reducing average CPU utilization per node by up to 20% compared with the default Kubernetes scheduler. These approaches show the potential of learning-based methods for power-aware pod placement [6].

The study in [7] proposed an energy-aware machine learning model for container placement. They collected CPU, memory, and I/O utilization metrics, enriched with physical sensor readings such as IPMI temperature and fan-speed measurements. They trained an XGBoost regression to predict the expected power consumption increase for each candidate host. The model achieved an R^2 score of 91.2% and reduced energy consumption by 3%. Similarly, in our work we collect a richer feature dataset, including resource utilization metrics, low-level hardware performance counters, pressure stall metrics, and we model the contextual dependencies between co-located pods of the workload episode.

In [8-10], authors introduce schedulers whose main objective is to identify and predict interference, to inform the scheduler about interference patterns and minimize performance degradation. These studies primarily rely on micro-architectural monitoring as the core mechanism for characterizing interference. Authors in [8] used Intel

Performance Counter Monitoring to collect metrics related to shared hardware resources, like L3 cache misses, memory bandwidth utilization, CPU core states, and instructions per cycle. Through these counters, the authors identify the micro-architectural resources under contention and evaluate the resulting performance impact. This finding is reinforced by [9], who performed a comprehensive analysis of co-located container workloads using hardware performance counter. Their study showed that different co-location patterns produce distinct micro-architectural signatures, recommending co-location combinations based on cache and memory bandwidth contention patterns. Additionally, the study in [10] applied hardware performance counter data and machine learning to predict workload interference in heterogeneous high-performance computing systems, in order to anticipate performance degradation caused by cache and memory bandwidth contention.

In addition to hardware counters for identifying interference patterns, software-level monitoring through workload profiling is also used as an alternative method. Authors in [12] addressed interference by classifying applications into high or low resource usage for CPU and disk I/O throughput. Then they constructed a penalty matrix to assigns weights to each co-location combination, representing the contention degree. The scheduler choses the node that minimizes the total interference penalty between the incoming pod and the applications already running on the candidate node. This study provides a practical approach to contention-aware placement, but it is limited to CPU and disk I/O, and it does not account for the network overhead caused when separating interdependent pods across different nodes.

The study in [3] presented a machine-learning-based Kubernetes scheduler to optimize micro-service placement for improved performance and reduced energy consumption in heterogeneous data centers. They implemented Bayesian Optimization to optimize pod replica scaling and pod placement, exploiting hardware heterogeneity by combining older and newer server within the cluster. The scheduler assigns latency-critical pods to newer servers and less demanding workloads to older machines. They achieved up to 45% lower power consumption and 2.3 times lower P99 latency compared with homogeneous clusters. Rao and Li in their approach analysed how frequently pods communicate with each other based on network traffic, and considered the resource usage of each micro-service when deciding pod placement. They also used an improved Sparrow Search Algorithm to pack pods in order to reduce energy consumption while still meeting SLA requirements. Their method reduced cluster energy usage by at least 5% compared with state-of-the-art container scheduling algorithms [4]. Authors in [5] integrated energy metrics into the Kubernetes scheduling framework. They designed a power-aware scheduler that dynamically evaluates node power consumption at scheduling time and places pods to the node with the lowest predicted power increase. The study in [13] extended the energy-awareness paradigm to edge-cloud orchestration, proposing methods to include computational and network energy metrics into existing Kubernetes scheduling approaches evaluated on heterogeneous ARM-based testbeds, with improvements in energy efficiency under high load scenarios.

Several studies have used time-series analysis on historical data to predict energy, power consumption, and CO_2 emissions [16, 17]. Authors in [16] predicted the hourly energy consumption of Docker containers using AR, ARIMA, and ETS models, with ETS giving the most accurate results. Their work shows that choosing the right forecasting model is important, but it mainly focuses on predicting past and future energy trends rather than helping the scheduler decide where to place a workload before deployment. In [17] used time-series prediction for carbon-aware scheduling. They predicted future CO_2 intensity and delayed non-critical jobs to time periods with lower expected emissions, while still meeting SLA deadlines. This approach reduced CO_2 emissions by 1.3% compared with the default Kubernetes scheduler.

The surveyed literature demonstrates that fine-grained system monitoring and application-level workload profiling are the foundations upon which power-aware scheduling solutions are built. System monitoring provides the real-time state of the cluster, encompassing resource utilization rates, hardware performance counter readings, physical sensor data, and power-related indicators. Workload profiling characterizes application behavior in terms of resource demand intensity, inter-service communication patterns, and sensitivity to resource contention.

However, several limitations remain. Many studies use non-realistic workloads on single-microservice, and they treat placement samples independently, or model each workload type separately. These assumptions reduce their ability to represent realistic cloud-native deployments composed of multiple heterogeneous, interacting micro-services.

In contrast, our work focuses on heterogeneous real workloads composed of several micro-services. Instead of usually analyzing pod placements as independent events, we consider groups of pods belonging to the same workload scenario, because previously placed pods affect the cluster power consumption. Additionally, besides using the pre-scheduling metrics that describe the current system state, it has been introduced episode-level context features describing interference weight, micro-service affinity and scenario topology. This makes the prediction problem more complex but also more representative of real Kubernetes environments, where the power impact of a new placement depends on both the target node condition and the behavior of existing, running services. Comparing to other works that trained workload-type models separately, limiting generalisation to mixed-workload scenarios, we trained a single XGBoost model on heterogeneous mixed workloads using group-aware cross-validation.

Research Gaps and Hypotheses

Based on the scheduling approaches analyzed in the literature review, we identified several research gaps that motivated this work:

First, existing performance-aware and power-aware Kubernetes schedulers mainly rely on general node-level utilization metrics, such as CPU, memory, disk, and network usage. However, they give limited consideration to low-level shared resources, microservice affinity, and detailed co-location patterns. Most scheduling approaches analyze each

placement decision independently, without explicitly modeling the temporal or episodic dependency between consecutive pod placements within the same workload scenario. In the context of power prediction, a pod placement should not be treated as an isolated event, because the resulting power variation also depends on the pods that are already running on the same server and on the interference conditions created by their co-location.

Second, interference has been mostly studied in terms of its impact on application performance degradation, while its relationship with server-level power consumption remains less deeply investigated. Therefore, there is a need to quantify how different interference types and intensities contribute to power variation after a scheduling decision.

Based on these gaps, we formulate the following research hypotheses:

- H1: Episode-context features improve the prediction of power consumption variation. This hypothesis evaluates whether including episode-level information, such as previously placed pods, event position, co-location counters, and scenario topology, improves the predictive performance of the model. To test this hypothesis, we compare the optimized XGBoost model trained on the full feature set with the same model trained on a reduced feature set where episode-context features are removed.
- H2: Interference-weight features have a statistically significant relationship with power consumption variation. This hypothesis evaluates whether different interference types are associated with changes in server-level power consumption. To test this hypothesis, we analyze the correlation between each interference-weight feature and the target variable, power_power_watts_diff, and apply statistical significance testing with FDR correction.
- H3: XGBoost provides better predictive performance than baseline models. This hypothesis evaluates whether the improvement achieved by XGBoost is systematic and not only caused by randomness in the data splits. To test this hypothesis, we compare XGBoost with Ridge Regression, Decision Tree, and Random Forest using GroupKFold validation results, paired t-tests, and Cohen's dz effect size.

SYSTEM DESIGN

Testbed

Our testbed consists of an HPE ProLiant DL380 Gen10 server with a dual-socket NUMA architecture, each socket providing eight physical cores with Hyper-Threading enabled (two threads per core) and 192 GB of RAM. KVM is installed as the hypervisor, and a Kubernetes cluster (version: v1.31.3) is deployed comprising one master node and five heterogeneous worker nodes, running Ubuntu 24.04.1 LTS, with resource capacities described in Table 1. Each Virtual Machine (VM) was configured using NUMA-aware CPU and memory affinity policies to ensure locality of computation and memory accesses (configuration details given in Figure 1).

Table 1. Kubernetes Cluster Configuration

Nodes	vCPU cores	RAM	Socket
Master node	4	16	0
Worker node1	4	32	0
Worker node2	8	32	0
Worker node3	4	8	1
Worker node4	4	16	1
Worker node5	8	64	1

```
megit@phdcloud:~$ virsh dumpxml worker2
<domain type='kvm' id='3'>
  <name>worker2</name>
  <uuid>5a98770d-d369-4761-9397-ebf4a2f0513c</uuid>
  <metadata>
    <libosinfo:libosinfo xmlns:libosinfo="http://libosinfo.org/xmlns/libvirt/dom
ain/1.0">
      <libosinfo:os id="http://ubuntu.com/ubuntu/24.04"/>
    </libosinfo:libosinfo>
  </metadata>
  <memory unit='KiB'>33554432</memory>
  <currentMemory unit='KiB'>33554432</currentMemory>
  <vcpu placement='static' cpuset='4-7'>8</vcpu>
  <cputune>
    <vcpupin vcpu='0' cpuset='4'/>
    <vcpupin vcpu='1' cpuset='20'/>
    <vcpupin vcpu='2' cpuset='5'/>
    <vcpupin vcpu='3' cpuset='21'/>
    <vcpupin vcpu='4' cpuset='6'/>
    <vcpupin vcpu='5' cpuset='22'/>
    <vcpupin vcpu='6' cpuset='7'/>
    <vcpupin vcpu='7' cpuset='23'/>
  </cputune>
  <numatune>
    <memory mode='strict' nodeset='0'/>
  </numatune>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='x86_64' machine='pc-q35-6.2'>hvm</type>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
    <apic/>
  </features>
  <cpu mode='host-passthrough' check='none' migratable='on'>
    <topology sockets='1' dies='1' cores='4' threads='2'/>
  </cpu>
```

Figure 1. Libvirt XML Configuration of Worker2

The libvirt mechanisms were configured to control CPU and memory placement at the NUMA level. Specifically, *cputune/vcpupin* was used to bind each VM vCPU to physical CPUs belonging to the intended NUMA node, thereby preventing uncontrolled migration across sockets or NUMA nodes. In addition, *vcpu placement='static'* was configured to ensure explicit CPU placement, while *numatune <memory mode='strict'>* constrained VM memory allocation to the corresponding NUMA node. Collectively, these configurations reduce cross-NUMA execution and remote memory accesses, thereby providing more controlled and predictable resource placement.

Deployed Applications

To experiment with the scheduling in our Kubernetes cluster, we chose heterogenous real-world conteinerized applications, in terms of their resource needs. It has been used *ffmpeg* with ultrafast preset and ffmpeg with medium preset (exposing distinct CPU utilisation profiles); the data-caching and in-memory analytics workloads from the CloudSuite benchmark suite. *Data-caching* is composed of three microservices: data caching server,the warmup job and the throughput job. *In memory analytics* is composed of two microservices: spark worker and in memory job.

Monitoring Framework

As part of the preliminary work, a real-time monitoring platform was implemented to collect fine-grained metrics characterizing workload behavior and Kubernetes cluster state across multiple system-resource dimensions, including CPU, memory, disk, network, L2 and L3 caches, memory bandwidth, and Intel Ultra Path Interconnect (UPI) [18]. Prometheus was configured as the central metrics scraper and time-series database, while Grafana was used for graphical visualization of the collected measurements. cAdvisor, integrated with the kubelet, was used to collect per-container resource-utilization metrics, while Node Exporter was deployed as a DaemonSet to expose node-level CPU, memory, disk, and network metrics.

Intel Performance Counter Monitor (PCM) was integrated to access hardware performance counters and characterize interference in low-level shared resources, including memory bandwidth, cache hierarchy, and the Ultra Path Interconnect between CPU sockets. Server power consumption was monitored through the integrated Intelligent Platform Management Interface (iLO). In addition, the Kubernetes Metrics Server was enabled, and Prometheus exporters were configured for each application deployed in the cluster to collect application-specific performance metrics in a centralized manner for subsequent analysis. The overall monitoring framework is illustrated in Figure 2.

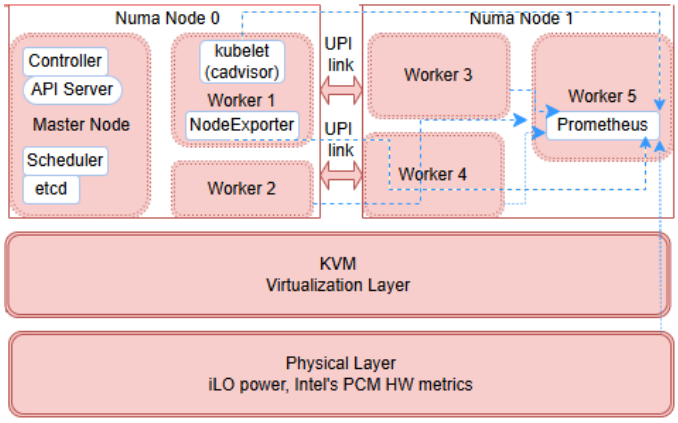


Figure 2. System Architecture

Experiments

In the first phase, it has been profiled the deployed applications to identify their real needs for resources like CPU, memory, disk and network by configuring as Burstable pods

and running them in isolation. We also recorded reference values for application-specific performance metrics.

In the second phase, it has been profiled each pod type (one pod per microservice) to record its pressure on low-level shared resources like L2/L3 cache and memory bandwidth.

The profiling results given in Table 2 were exploited to categorize applications in terms of their dominant resource utilization and to design co-location experiments to simulate different interference level for the 3rd phase.

Table 2. Profiling Results

pod_type	cpu_usage_millicores	memory_usage_gb	network_tx_bps	network_rx_bps	diff_l3_miss_rate	dram_diff
ffmpeg-fast	4604.75	0.628	1034.223	584112.4	1756390.28	5821720355
ffmpeg-med	6737.046	0.6812	783.7162	341332.1	1492560.659	1831148026
dc-server	1251.5941	4.19255	37023786.92	134691120	104009.6504	416824175.5
dc-warmup	4516.7513	1.8138	341506459	8462626.796	316385.197	1886293449
dc-throughput	1239.9171	1.067	14010267.58	80532030.53	336788.9533	740804541.8
spark-worker	1356.2355	0.65	69746.894	165032.9725	482071.2876	776628738.8
inmemory-job	1054.3839	0.411	159461.1356	67929.9202	239727.6209	590286407.6

In the third phase it has been designed 81 controlled placement scenarios, by varying:

- the workload mix, defined by the total number of instances per application type.
- the node placement of every pod of each instance. The placement was guided from our main goal of simulating the necessary use cases to test interference types, the violation/non-violation of microservice affinity within an application, different scenario topology (pods placed in the same node, same socket, across sockets).

Table 3 provides a complete scenario catalogue grouped by workload composition. Each group contains scenarios with the same application types, the same number of instances per each type. The episode size is defined as the total number of pods launched in the scenario. Scenarios within the same workload-composition group differ in their pod placement strategy, which was designed according to three main objectives. First, different co-location combinations were used to simulate varying levels and types of resource contention. Second, placement decisions were configured to model microservice affinity within the same application by co-locating or separating related service instances. Third, we varied scenario topology, including same-node, same-socket, and cross-socket. After each scenario completed, a 10 s cooldown interval was applied before starting the next scenario to reduce residual effects from the previous execution.

During the 3rd phase, we did not set the CPU limits to avoid enabling CFS throttle, so to detect the effect of interference on performance degradation.

Table 3. Scenarios Catalogue

index	workload_composition	scenario_ids	episode_size
0	1 ffmpeg-fast, 1 ffmpeg-med	1	2
1	2 ffmpeg-fast	42	2
2	2 ffmpeg-med	43	2
3	1 ffmpeg-fast, 1 inmemory	67,68,71,74,76	3
4	1 ffmpeg-fast, 2 ffmpeg-med	81	3
5	1 ffmpeg-med, 1 inmemory	15,16,17,18,69,70,72,73,75	3
6	2 ffmpeg-fast, 1 ffmpeg-med	2	3
7	3 ffmpeg-fast	3,4,5,44	3
8	3 ffmpeg-med	45,46	3
9	1 ffmpeg-fast, 1 data-caching	54,56,57,59	4
10	1 ffmpeg-med, 1 data-caching	13,14,55,58,60,61,63,64,65	4
11	2 inmemory	28,29,30,31,32,33,34,35,36,37,38,39, 40,41,80	4
12	1 ffmpeg-fast, 1 ffmpeg-med, 1 data-caching	62	5
13	1 inmemory, 1 data-caching	7,8,9,10,11,47,48,49,50,51,52,53,82	5
14	1 ffmpeg-med, 1 inmemory, 1 data-caching	19,20,21	6
15	2 data-caching	12,22,23,24,25,26,27	6
16	2 inmemory, 2 data-caching	6	10
17	2 ffmpeg-fast, 2 ffmpeg-med, 3 inmemory, 2 data-caching	79	16
18	2 ffmpeg-fast, 3 ffmpeg-med, 3 inmemory, 2 data-caching	77	16
19	3 ffmpeg-fast, 3 ffmpeg-med, 3 inmemory, 2 data-caching	78	18

DATASET CONSTRUCTION

Methodology

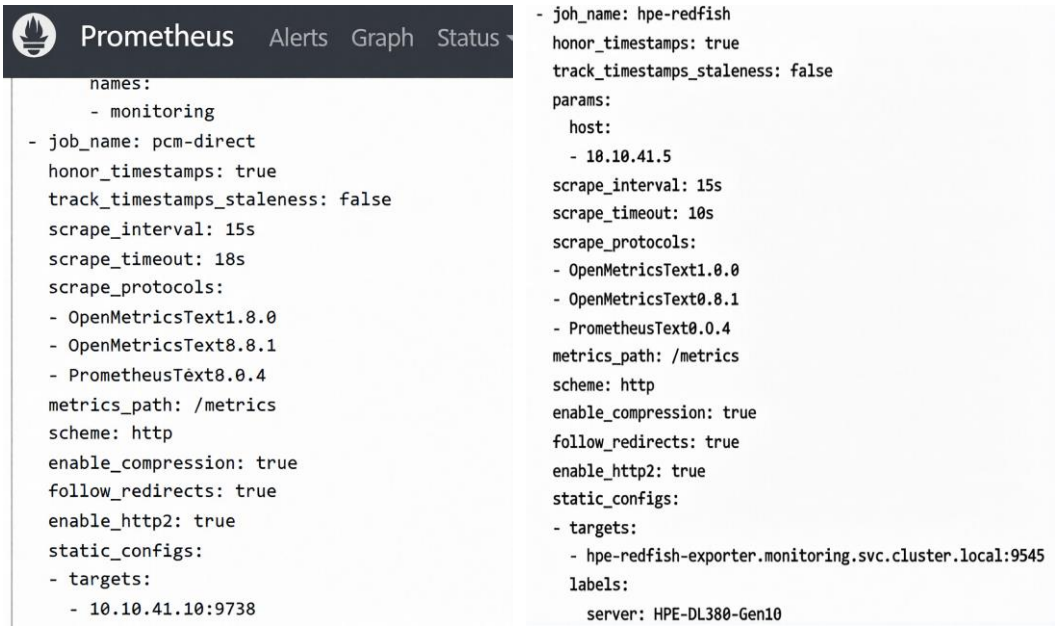
The focus of our work is to analyze the impact of the scheduling decisions to the power consumption rate of change in our cluster. For this we built a dataset with the steps below:

- The set of 81 controlled placement scenarios was executed and repeated ten times to reduce measurement noise and improve the robustness of the measurements. Because Prometheus, Grafana, Flannel, kube-proxy, and Ceph pods continuously run as background processes on the worker node, and because each scenario leaves a residual workload footprint that can affect the cluster state before the subsequent scenario, the execution order of the scenarios was randomized. This is important to eliminate this ordering bias and ensure that the background process load and any thermal carry-over from a preceding scenario are distributed randomly across scenarios. Across these repetitions, we noticed the range of the C0StateResidency

metric from {0.03–0.44} to {0.03–0.726}, so of introduced 4 additional repetitions specifically to extend the observed range to {0.03–0.726}, ensuring that the training data covers the full spectrum of CPU active-state conditions and that the predictive model can learn the influence of this metric on power consumption under identical workloads. C0-State Residency metric is a hardware-level indicator that reflects the fraction of time CPU cores spends in actively executing instructions.

- Per each new pod placement within our scenarios, it has been collected metrics grouped in the following categories: (i) *application's resource needs*, captured during 1st profiling phase; (ii) *application's pressure on low-level shared resources*, captured during 2nd profiling phase; (iii) *node resource utilization*; (iv) *node pressure stall metrics*; (v) *Intel Performance Counter Monitors(PCM)* for low-level shared resource monitoring (e.g., memory bandwidth, L3 miss rate, LLC occupancy, UPI traffic); (vi) *pod resource utilization*; and (vii) *cluster power consumption*. The complete set of metrics is reported in Table 4.

As we previously mentioned, we used Prometheus as our centralized monitoring service, with a scrape interval of 10 seconds to collect cluster/node and workload metrics. We also configured scrape jobs, to import to Prometheus specific performance metrics from deployed applications (in-memory-analytics, data-caching), power readings from iLO, low-microarchitecture performance counters from Intel's PCM. In Figure 3, we show the configuration of the *hpe-redfish* scrape job used to export power reading from the iLO , and of the Intel's PCM scrape job, both with scrape interval of 15 seconds.



(a) (b)

Figure 3. a) Prometheus Scrape-Job Configuration of PCM; b) Prometheus Scrape-Job Configuration of iLO

Metric samples were retrieved using a 10-second query step and subsequently aggregated over the corresponding pod execution interval using mean values. The data collection, preprocessing, and feature-engineering procedure comprised the following stages:

1. *Metric collection and temporal synchronization.* To enable an in-depth analysis of feature relationships and their temporal dynamics, metrics were collected over two distinct time windows per placement event: cluster baseline metrics, recorded prior to the new pod's placement, and during metrics, recorded throughout the new pod's lifecycle and averaged over this period to describe the impact of the placement.

   Synchronization between cluster/workload's metrics and power readings was performed timestamp-wise: baseline metrics were computed over a fixed 60 seconds window before pod placement, while "during pod execution" metrics and iLO power samples were aligned over the corresponding pod lifecycle execution interval. Considering the diverse lifecycle between pod types, for our Prometheus queries, we set specific rate windows for counter-based metrics, reducing the likelihood of empty samples for short-lived pods (detailed on Figure 4).

```
APP_RATE_WINDOWS = {
    "ffmpeg-fast":    "40s",
    "ffmpeg-med":     "40s",
    "inmemory-job":   "35s",
    "dc-throughput": "25s",
    "dc-server":      "30s",
    "dc-warmup":      "40s",
    "spark-worker":   "35s",
}
```

Figure 4. Pod-Specific Time Window for Metrics Collected During Pod Lifecycle

2. *Dataset construction and missing-value filtering.* All collected metrics were consolidated into a structured dataset in which each row represents a single pod placement event, encoding both the baseline and during-period metrics alongside the corresponding power consumption observations. Regarding missing values, 70 rows were removed based on the following criteria:
   - missing baseline and/or during-execution power readings, which represent the target values for the prediction model, or
   - missing values accounting for at least 80% of the total input feature set.

   These cases were primarily associated with temporary Prometheus service, HPE iLO interface unavailability. After applying these filtering rules, the resulting dataset contained 4,978 samples and 202 features.

3. *Pre-scheduling feature selection.* To create a dataset suitable for analysing and predicting the impact of scheduling decisions on power consumption, the original dataset was filtered to retain only features available at the time of the scheduling decision. This resulted in a subset of 73 pre-scheduling features, corresponding exclusively to baseline metrics. The resulting feature set was subsequently used to identify the most influential pre-scheduling characteristics associated with variations in cluster power consumption and to develop a prediction model suitable for future integration into a power-aware scheduler.
4. *Feature engineering.* It was performed feature engineering to handle two problems of our dataset: i) the high dimensionality of the feature space, which is known to degrade the learning quality of ML model; ii) the contextual nature of scheduling decisions.

Table 4. Raw Metrics Collected

Feature groups	Raw Metrics
Application's resource needs (1st Profiling results) / container resource utilization during 3rd phase	container_cpu_usage_seconds_total container_cpu_cfs_throttled_periods_total container_memory_working_set_bytes container_network_transmit_bytes_total container_network_receive_bytes_total container_fs_reads_bytes_total container_fs_writes_bytes_total
Application's pressure on low-level shared resources (2nd Profiling results) / Intel's PCM for cluster state during 3rd phase	L2_Cache_Misses {core="",thread=""} L3_Cache_Misses{core="",thread=""} DRAM_Reads{socket=""} DRAM_Writes{socket=""} pcm_upi_in_{socket=””} pcm_upi_out_{socket=””} pcm_cfreq_ghz{core=””}
Node resource utilization	node_cpu_seconds_total node_memory_MemAvailable_bytes node_memory_MemTotal_bytes node_disk_read_bytes_total node_disk_written_bytes_total node_network_receive_bytes_total
Node pressure	node_pressure_cpu_waiting_seconds_total node_pressure_io_waiting_seconds_total node_pressure_memory_waiting_seconds_total node_pressure_io_stalled_seconds_total node_pressure_memory_stalled_seconds_total

Feature Engineering

Our original dataset based on raw metrics contained a total of 202 features, so we needed to reduce redundancy across the large set of collected metrics. Initially, we aggregated some of the granular PCM metrics as below:

- *L2/L3 miss rate, C0-state residency, and core frequency*: values were averaged across the CPU cores of the node on which the new pod was placed.
- *DRAM reads and writes:* values from both CPU sockets were summed to obtain node-level measurements.
- *UPI incoming and outgoing traffic:* traffic values were summed across the two UPI links connecting the CPU sockets, separately for incoming and outgoing traffic.

The goal of our dataset is to build a ML model to predict the impact of the scheduling decision to the cluster's power consumption rate of change. A key methodological contribution of this work is the introduction of episode context features. Since every pod placement belongs to a scenario episode (a sequence of instances, decomposed of a set of placements) sharing the same run identifier, the power consumption of the server during a new pod's lifecycle is influenced not only by that pod's resource demand but also by the cumulative effect of all previously placed pods within the same episode. Modelling placements as independent events, as is common in the literature, ignores this dependency. We added episode context features to give the model knowledge about the interference between collocated pods, microservice affinity for that application, and the scenario topology. The following episode context features are formally defined.

Let an episode be $E = \{p_1, \dots, p_n\}$, where each row/pod placement p_i has scheduling time s_i, end time e_i, node N_i, socket S_i, pod type T_i, and instance name I_i. Each pod type T_i is assigned an offline vector:

$$\mathbf{r}(T_i) = [c_i, \ell_i, m_i^{bw}, m_i^{cap}, d_i, n_i] \qquad where: c_i, \ell_i, m_i^{bw}, m_i^{cap}, d_i, n_i \in \{0,1,2\} \tag{1}$$

represent CPU, cache, memory-bandwidth, memory-capacity, disk, and network intensity levels. The low-intensity value 0, is replaced by 0.2, to handle the case of co − location of several low intensity pods. These metadata are used to calculate the interference weight measurable at the new pod's placement time, by initially identifying the overlapping pods. For a pod p_i, an already-running pod p_j, is considered an existing overlap if:

$$s_j \leq s_i \wedge e_j > s_i \wedge j \neq i \tag{2}$$

For a resource type r, the *interference weight* of pod p_i is:

$$W_i^{(r)} = \sum_{p_j \in O_i^{(r)}} \frac{\phi(a_i^{(r)}) \cdot \phi(a_j^{(r)})}{a_{\max}} \tag{3}$$

where:

$$\phi(a) = \begin{cases} 0.2, & a = 0 \\ a, & a > 0 \end{cases} \quad (4)$$

- $a_i^{(r)}$ is the intensity category of pod p_i for resource r
- $a_j^{(r)}$ is the intensity category of overlapping pod p_j for resource r
- $a_{\max} = 2$, since the maximum intensity category is 2
- $\mathcal{O}_i^{(r)}$ is the subset of overlapping pods relevant for resource r, selected according to topology

The topology-aware overlapping set is:

$$\mathcal{O}_i^{(r)} = \{p_j \in \mathcal{O}_i \mid \tau_r(p_i, p_j) = 1\} \quad (5)$$

where $\tau_r(p_i, p_j)$ is a topology relevance function defined as:

$$\tau_r(p_i, p_j) = \begin{cases} 1, & r \in \{\text{CPU,memory capacity}\} \wedge N_i = N_j \\ 1, & r \in \{\text{cache,memory bandwidth}\} \wedge (N_i = N_j \vee S_i = S_j) \\ 1, & r \in \{\text{disk,network}\} \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

where N_i and S_i denote the node and socket of pod p_i, respectively.

Additionally, in compact form:

$$W_i^{(r)} = \sum_{p_j \in \mathcal{O}_i} \tau_r(p_i, p_j) \cdot \frac{\phi(a_i^{(r)}) \cdot \phi(a_j^{(r)})}{4} \quad (7)$$

Episode size, as the total number of pod placement events in episode. This metric encodes the overall workload complexity. *Event Index* metric represents the sequential position of pod p within an episode.

We added the running pods counters, specific per each topology category: *current_active_pods_same_node_count,current_active_pods_same_socket_count, current_active_pods_accross_socket_count.*

We also added microservice affinity counters, to encode whether microservices of the same application instance are co-located on the same node,samesocket, or across sockets: *same_instance_active_same_node_count,same_instance_active_same_socket_count, same_instance_active_accrosssockets_count.*

EXPLANATORY DATA ANALYSIS

The data analysis started by examining the distribution of the target variable, ΔPower, across all 4,978 pod-placement samples, as illustrated by the histogram with a kernel density estimate (KDE) overlay in Figure 5.

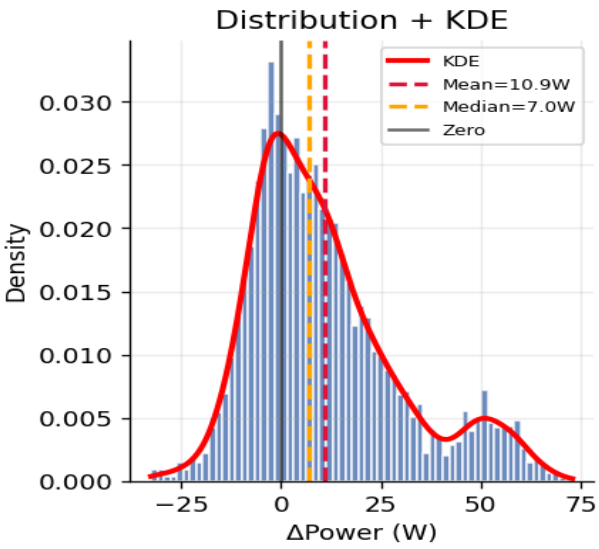


Figure 5. Distribution of Target Variable

The distribution is right-skewed and non-normal, with the central mass concentrated between 0 and 20, indicating that the majority of pod placement events result in a moderate positive power increase. A lower-density plateau is visible in the 40–60 W range, which is attributable to placements within large episodes involving concurrent cpu, cache-intensive and disk-intensive workloads, where cumulative LLC eviction pressure and PCIe controller saturation produce disproportionate power increases. Approximately 28–31% of placement events produce a negative ΔPower, extending to −32.5 W. They are explained by two distinct profiling-grounded mechanisms. In scenarios dominated by ffmpeg-fast (highest DRAM, highest L3 miss), when a node's memory bandwidth ceiling is already reached by prior episode placements, adding another ffmpeg-fast pod triggers DRAM queuing stalls, where active cores enter wait states, reducing CPU power despite nominally running more work. In scenarios dominated by dc-server (with the highest memory footprint), physical DRAM pages become fully occupied, leading to page-fault induced stalling across all co-located workloads. The low-density tails at both extremes of the distribution(below −20 W and above 50 W), do not represent statistical outliers. They correspond to a subset of experimental with limited number of distinct placement combinations.

Following we analyze the impact of application type, scenario topology and interference to the power change of the cluster.

Application type

An important aspect of our analysis was to investigate the impact of the application type on power consumption. The analysis was initially conducted at the application level and subsequently extended to a finer granularity by examining the microservices comprising each application. As shown in Figure 6, *ffmpeg* applications exhibit the strongest impact on server power because they stress a broader set of hardware resources simultaneously, including CPU, cache, memory bandwidth, disk, and network activity.

This wider resource footprint explains why their power distribution is generally higher and more variable than that of the other application groups. Within the ffmpeg category, ffmpeg-fast tends to dominate ffmpeg-medium, which can be attributed to its higher pressure on memory bandwidth and disk activity. In contrast, when comparing data-caching with in-memory analytics, the higher power levels observed for the medium data-caching configuration can be explained by the pod-level profiling results: the dc-warmup pod exhibits strong utilization of CPU, memory bandwidth, and network resources, reaching levels comparable to ffmpeg workloads. Meanwhile, the in-memory analytics workloads show a more moderate and less intensive resource usage pattern, which is reflected in a comparatively lower power impact.

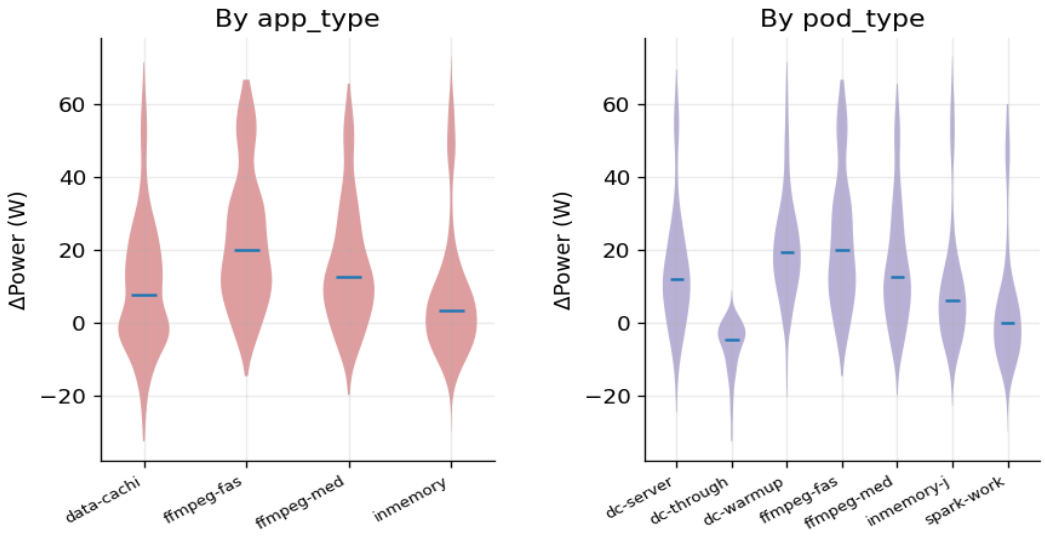


Figure 6. (a)-Power Distributions Grouped by Application Type. (b)-Grouped by Pod Type.

Scenario Topology

One of the key factors that significantly affects power consumption is the scenario topology, so how microservices are distributed within an episode. The impact of this factor is presented in Figure 7. Scenarios with pod placements in the same node (Topology 0) show the most symmetric power distribution in the range of {-24;+30 W}, with a median of 3.8 W. This topology results with the smallest power impact per new placement, since the workload are limited on the node's local resources, without any inter-node communication overhead and also because the interference pattern has already been established by prior placements, and incremental additions to a locally saturated node produce diminishing power returns. Topology 1 (Same Socket, Different Nodes) shows a moderately wider distribution with median 5.26 W. Co-location across different nodes within the same NUMA socket introduces additional memory controller and shared LLC pressure, but retains the efficiency advantage of intra-socket memory access without UPI traversal. The slightly higher median relative to topology 0 reflects the fact that topology 1 spreads pods

across worker nodes that share the same physical CPU socket, but engaging more physical cores to serve the expanded pod set. Another fact that contributes to the increase in power consumption, is the additive cost of virtual inter-node communication and cross-node memory access within the same socket. For topology 2 (Across Sockets) we get a median of 11.89 W, a mean of 16.69 W. The upper tail extends beyond 65 W. This increase is explained by the fact that a server running workloads on both sockets draws more power than when the workload is consolidated onto one socket, because the second socket's uncore, memory subsystem, and inter-socket fabric must all be powered and clocked. Also, once both sockets are active, cross-socket memory accesses traverse the inter-socket interconnect, adding the UPI (Ultra Path Interconnect) power overhead. This finding provides strong experimental evidence that NUMA-aware scheduling is a crucial way for server power reduction in a Kubernetes cluster.

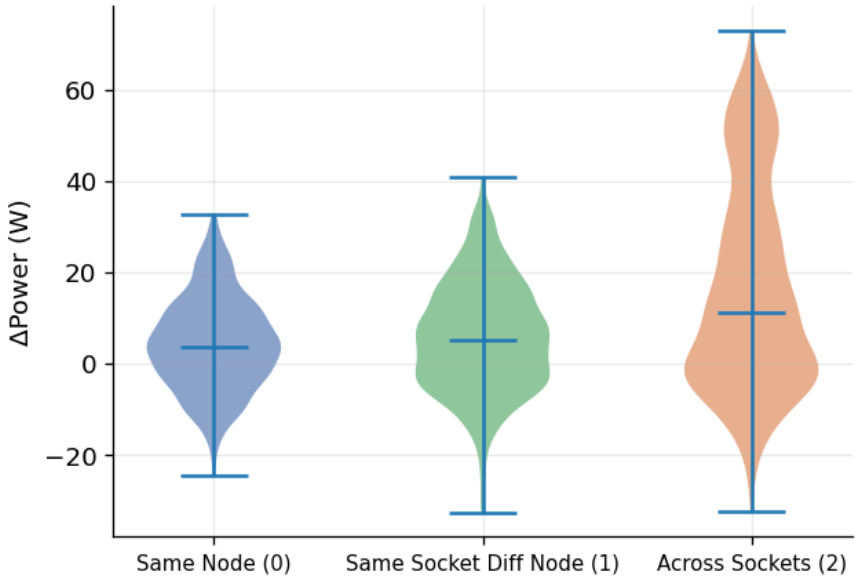


Figure 7. Power Distribution by Scenario Topology

Interference

An important goal to us was to examine if the type and intensity level of interference have a direct impact on power consumption, considering that most existing studies in the literature have primarily investigated the direct effect of interference on application performance. This analysis is supported by the results presented in Figure 8. It can be observed that cache, CPU, disk, and memory-bandwidth interference show a positive relationship with the target power variation, where higher interference weights generally correspond to larger increases in power consumption. Disk interference weight shows the strongest and consistent positive association with ΔPower ($r = +0.460$, 95% Confidence Interval CI {0.438, 0.481}), with the mean trend rising steeply and near-monotonically from initially 6 W at low weights to 45–58 W at high weights, confirming that the progressive accumulation of disk co-located pods drives the largest marginal power increases among

all interference categories. Cache interference follows as the second strongest positive predictor (r = +0.392, 95% CI {0.368, 0.415}), with a similarly rising trend that reflects the incremental exhaustion of last-level cache capacity as more cache-thrashing workloads accumulate in the episode, forcing all co-located pods into high-frequency DRAM accesses and elevating memory-controller power. It has been noticed that memory bandwidth interference ((r = +0.169)) shows a weaker impact on power increase, but has directionally consistent positive trend (with a mean of +12 W for most of the interference weight ranges). Compared to the previous interference types, network interference (r = +0.120) exhibits a different behavior. In the first half of weight ranges {0.1-5}, we notice a mean power increase of 15 W, followed with a rising mean through mid-range weight values, finally declining at the highest observed weights. CPU interference weight has the weakest effect of all six types (r = +0.089). More CPU-intensive pods competing on the same node does raise power slightly, but the effect is small and does not rapidly increase like cache or disk interference does. The reason is that CPU contention slows down all competing pods proportionally, so the power increase stays modest and bounded. There is no cascade effect where one resource shortage triggers another, unlike what happens with cache misses driving DRAM accesses. Memory capacity interference weight is the only type with a negative overall correlation (r = −0.182, 95% CI {-0.209, -0.155}). At low-to-moderate weight the mean ΔPower is still positive, meaning some memory pressure exists but the node is still functioning normally. But once the accumulated weight crosses a certain threshold, the mean ΔPower turns negative because of memory capacity saturation from large footprint pods, causing page faults and forcing co-located workloads into wait states, suppressing active CPU power and reducing Power.

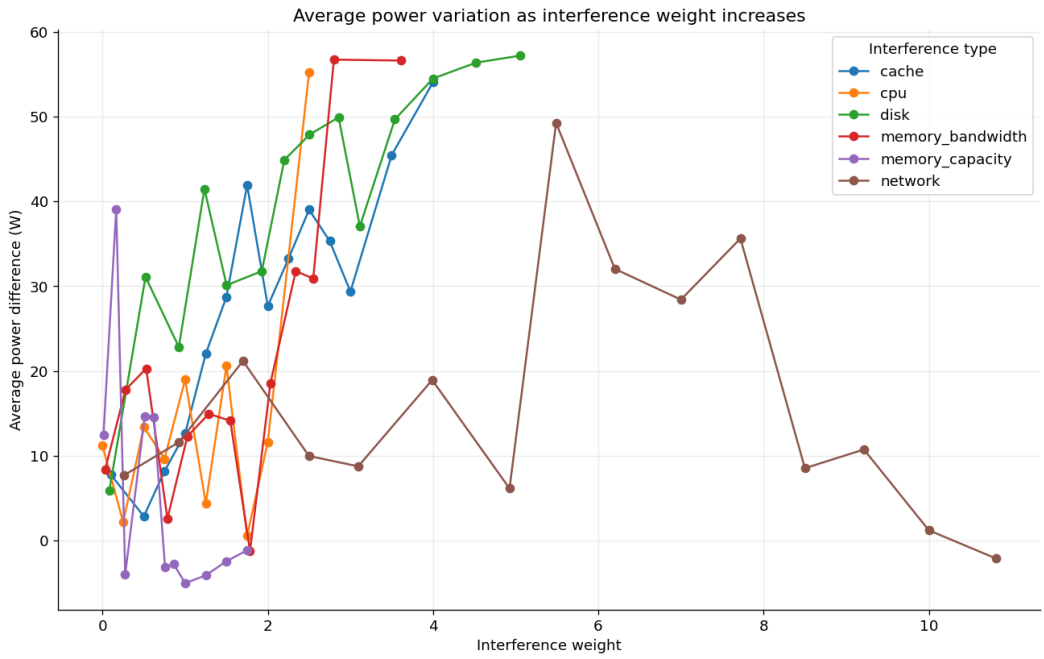


Figure 8. Interference Type & level impact on ΔPower

The complete statistical results of the Pearson correlation analysis are reported in Table 5, including the correlation coefficient, raw p-value, 95% confidence interval, and Benjamini–Hochberg false discovery rate adjusted p-value (FDR). The results show that all interference weight features have a statistically significant relationship with power consumption variation, since all p-values remain significant after FDR correction at 0.05 level. This analysis represents an important contribution, since the research has been mainly focused on analyzing the direct impact of interference on workload performance degradation.

Table 5. Pearson Correlation Analysis: Interference Weights vs Δpower

feature	r	ci95_low	ci95_high	p_value	p_fdr_bh
iw_disk_intensive	0.46	0.44	0.48	p < 0.001	p < 0.001
iw_cache_intensive	0.39	0.37	0.42	p < 0.001	p < 0.001
iw_memory_capacity_intensive	-0.18	-0.21	-0.15	p < 0.001	p < 0.001
iw_memory_bandwidth_intensive	0.17	0.15	0.2	p < 0.001	p < 0.001
iw_network_intensive	0.12	0.09	0.15	p < 0.001	p < 0.001
iw_cpu_intensive	0.09	0.06	0.12	p < 0.001	p < 0.001

Feature Importance

In our large feature dataset, we used SHAP analysis to identify the variables with the largest contribution to the prediction of the target variable, ΔPower. The resulting top-25 SHAP ranking from the Figure 9, shows that *power_watts_baseline* is the most impactful feature (mean |SHAP| ≈ 5.2 W), followed by *episode_size* at around 4.6 W.

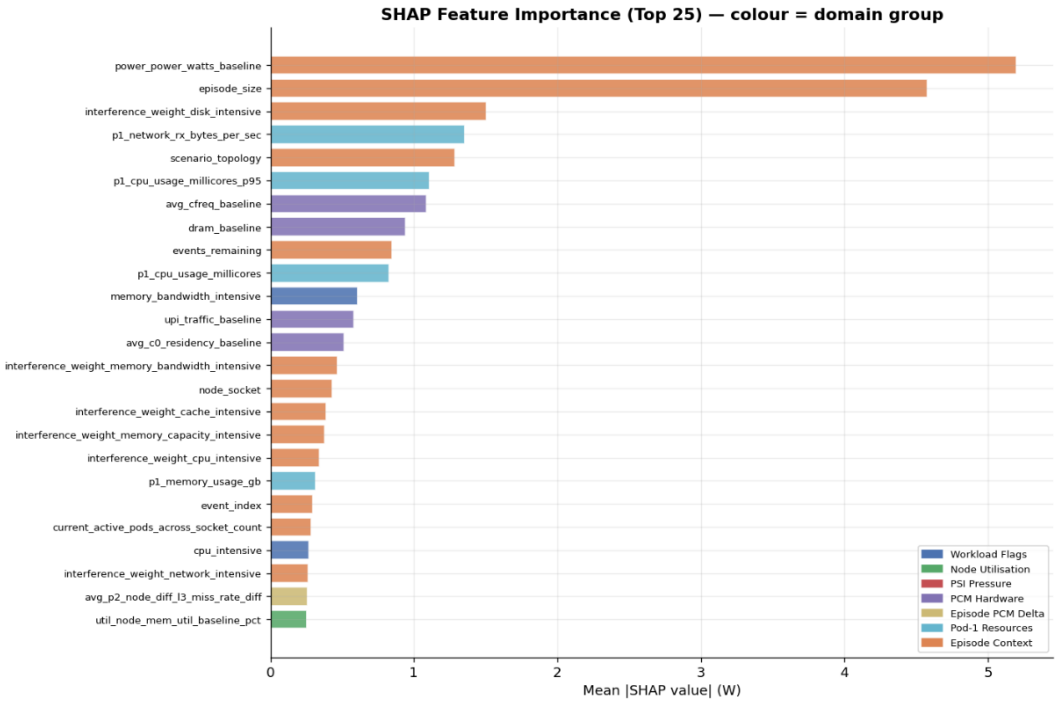


Figure 9. SHAP Feature Importance (Top 25)

The reason behind why *power_watts_baseline* is the most important, is because whenever baseline server power exceeds the thermal headroom threshold, this single feature pushes the prediction strongly toward zero or negative ΔPower. The next most relevant features include *interference_weight_disk_intensive*, *p1_network_rx_bytes_per_sec*, and *scenario_topology*, suggesting that disk-related interference, network receive activity, and microservice placement structure also have a meaningful effect on ΔPower. Hardware-state features such as *avg_cfreq_baseline*, *dram_baseline*, and *upi_traffic_baseline* further indicate that the model captures the influence of CPU frequency, memory subsystem activity, and inter-socket communication on power behavior.

POWER CONSUMPTION PREDICTIVE MODEL

The final processed dataset comprised 73 features, of which 17 were metadata features, while the predictor matrix (X) contained 55 features. The target variable, *power_watts_diff*, represents the difference between the average server power consumption during the lifecycle of the newly placed pod and the baseline power consumption measured prior to its placement. Multiple supervised regression models were trained and evaluated to predict the variation in power consumption associated with the placement of a new pod, while accounting for the episode context in which the placement occurred.

Rationale for Selecting Regression Models

XGBoost was selected as the primary model for predicting power consumption. The selection of XGBoost is grounded in the empirical study of [19], who benchmarked tree-based models against deep learning across 45 medium-sized tabular datasets and consistently found tree-based models superior. The main advantages of XGBoost are the following:

a) tree-based models outperformed deep learning across 45 medium-sized tabular datasets, containing heterogeneous columns, smaller sample sizes, extreme values, and features with very different meanings.

b) tree-based models showed better at capturing irregular relationships between features and targets.

c) Tree-based models handled irrelevant features better, through feature selection at splits.

d) Tree-based models preserve feature orientation.

e) The XGBoost objective function incorporates built-in L1 and L2 regularisation, preventing overfitting without a separate regularisation pipeline.

f) XGBoost supports early stopping, halting training automatically when validation error stops improving, and its tree structure enables exact SHAP-based feature attribution, expressing each feature's contribution directly in Watts.

For comparative evaluation, we trained on our dataset, also the following baseline models, with different model complexities:

1. Ridge Regression, a regularized linear regression, chose to evaluate if power variation could be explained by linear relationships among the input features. Its L2 regularization penalizes large coefficients, reducing overfitting and improving coefficient stability in the case of multicollinearity among metrics.
2. Decision Tree, chose as a simple nonlinear baseline model, capable of capturing threshold-based feature interactions.
3. Random Forest, chose as an ensemble learning alternative based on bagging. Multiple decision trees are trained independently and their predictions are averaged to reduce overfitting.

Compared to our baseline models, XGBoost is based on gradient boosting, in which trees are built sequentially to correct the prediction errors of previous trees.

Dataset Structure

The dataset comprises 4978 pod placements collected across 81 controlled scenarios. Each record is organized into three distinct column groups:

- 17 Metadata columns: episode and scenario identifiers, pod arrival timestamp in the waiting queue, scheduling and termination time stamp, pod name, instance name, node name retained for traceability but excluded from model training.
- 55 features in the X set: the complete set of predictive inputs available at scheduling time, subdivided into:
  - *Numeric features:* node utilization metrics, Linux PSI pressure counters, Intel PCM hardware counters (L2/L3 miss rates, DRAM bandwidth, UPI traffic, C0 residency, core frequency), per-pod resource consumption (CPU, memory, network), and episode context features (episode size, event index, interference weights, microservice affinity counters, co-location counts).
  - *Categorical features:* app_type (4 categories: ffmpeg-fast, ffmpeg-med, data-caching, in-memory), pod_type (7 categories), and scenario_topology (0 = same node, 1 = same socket different node, 2 = across sockets). We applied one-hot encoding.
- Target variable: *power_power_watts_diff*, defined as the difference between the mean server power measured over the pod's lifecycle and the baseline server power recorded immediately before the placement event, expressed in Watts.

Group-Aware Dataset Partitioning

Our dataset has a hierarchical structure, where rows are nested within episodes. Treating rows as independent and applying random train–test splits would cause data leakage in case when rows from the same episode could appear in both partitions, allowing the model to learn episode-specific patterns rather than generalisable scheduling rules. To prevent this, all splits are performed at the episode group level using the *GroupShuffleSplit* and *GroupKFold* primitives from scikit-learn, with group_key = episode_id. The group-level

split ensures complete episodes appear in exactly one partition, so the model is evaluated exclusively on scheduling scenarios unseen during training.

The dataset was divided into training and testing subsets using an 80% : 20% split ratio, while preserving episode-level separation between the two partitions.

Hyperparameter Tuning and Evaluation

To optimize the predictive performance of the XGBoost regression model, hyperparameter tuning was performed using GridSearchCV. The search was applied only on the training partition in order to avoid information leakage from the test set. Hyperparameter tuning was performed over the following parameter space for each of our selected models:

- Ridge: *alpha* ∈ {0.01, 0.1, 1, 10, 50, 100, 500, 1000}
- Decision Tree: *max_depth* ∈ {3, 5, 7, 10, None}, *min_samples_leaf* ∈ {5, 10, 20, 50}, *min_samples_split* ∈ {2, 10, 20}, the *random seed* at 42.
- Random Forest: *n_estimators* ∈ {100, 200, 300}, *max_depth* ∈ {5, 8, 12, None}, *min_samples_leaf* ∈ {5, 10, 20}, the *random seed* at 42.
- XGBoost: *max_depth* ∈ {3, 5}, *subsample* ∈ {0.7, 0.9}, *colsample_bytree* ∈ {0.6, 0.8}, *min_child_weight* ∈ {1, 3, 5}, *reg_alpha* ∈ {0, 0.1, 1}, and *reg_lambda* ∈ {1, 3}. The number of trees, *n_estimators*, was fixed at a ceiling of 1,000 trees with *early_stopping_rounds* = 50, meaning training halted automatically when the validation R^2 showed no improvement for 50 consecutive rounds. The *learning rate* was fixed at 0.05, the loss objective at *reg:squarederror*, and the *random seed* at 42.

During hyperparameter optimization, cross-validation (CV) was performed using GroupKFold with five folds. During each CV fold, the model was trained on the 4- folds training subset, while the corresponding 1-fold validation subset was used for early stopping. The hold-out test set (from the preliminary split of train-test 80%:20%), was used only for final model evaluation.

The best hyperparameter configuration was selected according to the highest mean validation R^2 across the five folds. The same grouping key used in the train–test partitioning was also applied during cross-validation, ensuring that all repetitions of a given scenario remained within the same fold. The best hyperparameter configurations identified by the grid search for each of our selected models were as following:

- Ridge: *{ 'alpha' : 1}*
- Decision Tree*: {'max_depth': 10, 'min_samples_leaf': 10, 'min_samples_split': 2}*
- Random Forest: *{'max_depth': None, 'min_samples_leaf': 5, 'n_estimators': 300}*
- XGBoost: {*'colsample_bytree': 0.6, 'max_depth': 5, 'min_child_weight': 3, 'reg_alpha': 1, 'reg_lambda': 1, 'subsample': 0.9}*

Based on the cross-validation results computed on the validation folds (given in Table 6), it has been noticed that XGBoost achieved the best predictive performance among baseline, across all metrics. It obtained the highest mean validation R^2 of 94.15% ± 0.44%,

the lowest MAE of 3.49 W ± 0.15 W, RMSE of 4.65 W ± 0.27 W. The low standard deviation across folds, indicates that the model performance remained highly stable across the different GroupKFold splits, showing a good generalization ability.

Our optimal XGBoost model, achieved its highest performance with a median best iteration across folds of 639 trees. Early stopping helped control unnecessary additional boosting rounds and reduced the risk of overfitting. In the Figure 10 it has been seen the evolution of RMSE during training and validation across iterations.

Using the best parameter set, the final XGBoost model was trained on the full training subset and evaluated on the held-out test subset and achieved a test R^2 of 93.2%, a MAE of 3.23 W and RMSE of 4.26 W.

Table 6. Cross-validation evaluation results

Model	CV R2 mean	CV R2 std	CV MAE mean (W)	CV MAE std (W)	CV RMSE mean (W)	CV RMSE std (W)
Ridge	0.8645	0.0017	5.4176	0.175	7.0869	0.1433
Dec. Tree	0.8747	0.0117	5.0097	0.1216	6.8033	0.2293
Rand. Forest	0.9177	0.0081	4.0974	0.1981	5.5155	0.2314
XGBoost	0.9415	0.0044	3.4975	0.1576	4.6574	0.2739

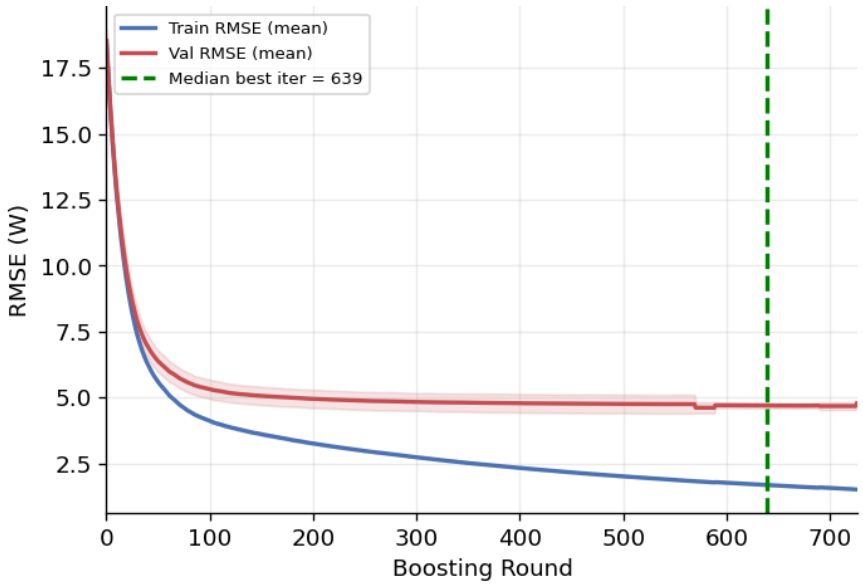


Figure 10. XGBoost Convergence Using RMSE

Figure 11 shows the overfitting analysis for all our trained models, by comparing the performance of the model on the training set with the prediction performance on the held-out test set. As an indicator of the generalization gap, we used ΔR^2. We notice that XGBoost and Rigde show the smallest gap between the training and test prediction results, with ΔR^2 values of 0.063 and 0.061, respectively, emphasizing the advantages of these models on minimizing overfitting. For Ridge, this stability is mainly explained by the L2

regularization; for XGBoost, the low gap is explained by the combination of regularization mechanisms, including L1/L2 penalties, subsampling, column sampling, tree-depth control, minimum child-weight constraints etc. Decision Tree shows the largest gap (ΔR^2 = 0.128), because the model builds a single tree, which can easily adapt to local variations in the training set. This train-test gap is reduced in Random Forest (ΔR^2 = 0.089) through ensemble learning.

In summary, XGBoost achieved the highest test R^2 and the lowest train–test gaps compared to our baseline models, providing the best balance between predictive accuracy and overfitting control for the ΔPower prediction task.

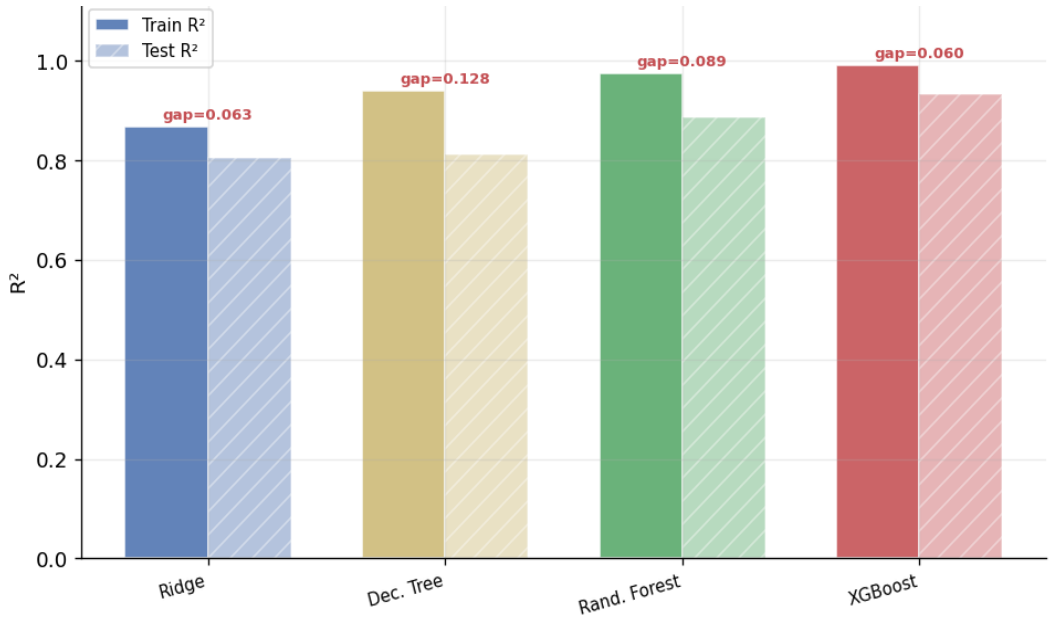


Figure 11. Overfitting Analysis Based on ΔR^2: Train Vs. Test

Significance Test

To verify that the superiority of XGBoost quality prediction over the baseline models is reliable and not affected by randomness introduced by the CV splits, we performed significance tests on the cross-validation results. Since all models were evaluated on the same GroupKFold splits, we used a paired t-test to compare the fold level MAE values of XGBoost against each baseline mode. In addition, we used Cohen's dz, to quantify the magnitude of the observed improvement beyond its statistical significance.

As shown in the Table 7, XGBoost achieved a statistically significant reduction in MAE compared with all baseline models. XGBoost reduced the mean MAE by 1.92 W, 1.51 W, 0.60 W compared respectively with Ridge Regression, Decision Tree and Random Forest. In all cases, the p-value was lower than 0.001, indicating that the improvement of XGBoost across the GroupKFold validation folds is statistically significant. Also, Cohen's dz values higher than 0.8, show that this improvement is large in effect size.

Table 7. Significance Tests Results

Compared models	Baseline mean MAE (W)	XGBoost mean MAE (W)	Mean MAE reduction (W)	t-statistic	p-value	Cohen's dz
XGBoost vs Ridge	5.42	3.50	1.92	51.95	$p < 0.001$	23.2339444
XGBoost vs Dec. Tree	5.01	3.50	1.51	20.95	$p < 0.001$	9.370214257
XGBoost vs Rand. Forest	4.10	3.50	0.60	8.00	$p < 0.001$	3.579858838

Episode Context Features Impact on the Model Performance

To showcase the importance of the added episode-context features in describing the system state before a scheduling event, we trained the optimized XGBoost model on two different feature sets:

1. The Full dataset, which includes all 55 input features in the X set
2. The reduced dataset composed of 36 input features, obtained after removing 19 episode-context features as following:
   - 6 interference weight features
   - 6 episode's composition and progress features('episode_size','event_index','events_remaining', 'instance_stage_index','instance_total_stages')
   - 1 'Scenario_topology' feature
   - 6 Counter based features, describing colocation through the number of active pods on the same node, same socket, and across sockets ('current_active_pods_same_node_count', 'current_active_pods_same_socket_diff_node_count', 'current_active_pods_across_socket_count', 'current_same_instance_active_same_node_count', 'current_same_instance_active_same_socket_diff_node_count', 'current_same_instance_active_across_socket_count')

The results showed that removing these features led to a considerable decrease in the model's predictive performance. When trained on the full dataset, XGBoost achieved an R^2 of 94%, while on the reduced dataset, it achieved an R^2 of 72%, corresponding to a decrease of 22 %. In addition, the MAE increased from 3.5 W to 7.2 W, with an increase of 3.7 W, while the RMSE increased from 4.7 W to 10.1 W, with an increase of 5.4 W. This degradation confirms the contribution of the proposed feature design to the prediction of power variation, showing that episode-level context features help the model learn the effect of scenario topology, colocation patterns on the resulting power variation.

SUMMARY AND CONCLUSION

This paper presented a methodology for predicting the server power impact of Kubernetes scheduling decisions through a combination of fine-grained cluster and workload monitoring, feature engineering, and gradient-boosted regression.

In the infrastructure level, a comprehensive monitoring framework was deployed on a heterogeneous Kubernetes cluster comprising five worker nodes. The framework integrated node-level utilization metrics, Linux Pressure Stall Information (PSI), Intel Performance Counter Monitor (PCM) hardware counters, per-container resource-consumption metrics, and server-level power measurements obtained through HPE iLO. A set of realistic containerized applications was deployed, and controlled pod-placement scenarios were designed to provide systematic coverage of different interference types and levels, workload topologies, and microservice-placement configurations. The resulting dataset contained 4,978 samples and 55 predictor features describing the cluster state and workload characteristics available around each scheduling decision, together with metadata and the corresponding power-consumption target.

At the feature-representation level, a key methodological contribution was the introduction of episode-context features to encode the accumulated state of a workload scenario at the time a new pod-placement decision was made. These features included interference-weight vectors quantifying the accumulated resource pressure of co-located episode pods across six resource dimensions, microservice-affinity counters describing the distribution of same-instance pods across node, socket, and cross-socket topology levels, scenario-topology metadata, episode-structural features representing episode progress, and hardware-counter deltas characterizing the resource pressure associated with the running episode. By representing pod placements as elements of a structured episode rather than as independent events, the proposed feature representation captures the accumulated workload context that can influence the power response to subsequent scheduling decisions. The importance of this contextual representation was further supported by an ablation analysis. When the 19 episode-context features were removed, leaving 36 predictor features, the optimized XGBoost model exhibited an approximately 22-percentage-point reduction in (R^2) relative to the 93.2% obtained using the complete 55-feature representation. This result indicates that episode-level contextual information provides substantial predictive value for characterizing power variation associated with pod placement.

At the modelling level, an XGBoost regression model was trained to predict the cluster power-consumption difference ((Δ Power)) associated with each scheduling decision using scenario-grouped cross-validation. The model achieved an (R^2) of 93.2% and a mean absolute error (MAE) of 3.23 W. XGBoost was selected based on its superior predictive performance among the evaluated models and its ability to represent nonlinear relationships within the dataset. These relationships include the diminishing marginal increase in power at higher episode sizes and the nonlinear response associated with memory-capacity interference. For comparative evaluation, three baseline regression

models representing different levels of model complexity were also trained: Ridge regression, Decision Tree, and Random Forest. On the held-out test set, these models achieved (R^2) values of 80.4%, 81.2%, and 88.5%, respectively. XGBoost therefore achieved the highest predictive performance among the evaluated models when using the complete 55-feature representation.

SHAP-based feature-importance analysis further supported the importance of workload interference and accumulated cluster state in explaining power variation. In particular, disk and cache interference exhibited substantial predictive contributions, indicating that the resource pressure generated by co-located episode workloads can be more influential than the resource footprint of the newly placed pod alone. Baseline server power emerged as the strongest individual predictor, reflecting the dependence of the power response on the pre-scheduling operating state of the server. Scenario topology and NUMA distance between co-located workloads also exhibited substantial predictive importance. Cross-socket placement was associated with a 3.4-fold higher mean power impact than same-node placement, coinciding with increased inter-socket UPI traffic and the additional resource activity associated with cross-socket execution. These findings extend existing interference-aware scheduling research, which has predominantly focused on application-level performance degradation, by demonstrating that workload interference and placement topology also have directly measurable consequences for server power consumption.

Several limitations should be considered when interpreting these results. First, the experimental evaluation was conducted on a limited physical infrastructure consisting of a single physical server hosting a Kubernetes cluster with one control-plane node and five worker nodes. Consequently, the findings may not generalize directly to larger or geographically distributed data-center environments. Second, although the study employed real containerized applications with different resource-intensity profiles, the diversity of workloads and application configurations remains limited relative to production environments. Future studies should therefore incorporate a broader range of real-world microservice applications, workload arrival patterns, cluster sizes, and hardware configurations. Such extensions would enable more comprehensive validation of the proposed feature representation and predictive model under diverse production-like conditions.

Future work will focus on integrating the trained XGBoost power-prediction model into a customized reinforcement-learning-based Kubernetes scheduler. The objective is to use the predicted power response as part of the scheduling decision process, enabling power-aware pod placement at the cluster level rather than optimizing individual placement events independently. This extension will provide an opportunity to evaluate whether the proposed predictive framework can support more energy-efficient scheduling policies under dynamic workload conditions.

AUTHOR CONTRIBUTIONS

Conceptualization, M.T., and E.K.M.; Methodology, M.T. and G.D.; Software and Computational Modelling, M.T.; Validation, M.T. and G.D.; Formal Analysis, M.T.; Investigation, M.T.; Resources, M.T.; Data Curation, M.T. and E.K.M.; Writing- Original Draft Preparation, M.T.; Writing- Review & Editing, M.T. and E.K.M.; Visualization, M.T.; Supervision, G.D. Both authors have read and agreed to the published version of the manuscript.

DATA AVAILABILITY STATEMENT

Data is contained within the article.

CONFLICT OF INTERESTS

The authors confirm to have no conflict of interest associated with this publication.

REFERENCES

1. Shaw, R., Howley, E., & Barrett, E. Applying reinforcement learning towards automating energy efficient virtual machine consolidation in cloud data centers. *Information Systems*, **2022**, *107*, 101722.
2. Jian, Z., Xie, X., Fang, Y., Jiang, Y., Lu, Y., Dash, A., et al. DRS: A deep reinforcement learning enhanced Kubernetes scheduler for microservice-based system. *Software: Practice and Experience*, **2024**, *54*(10), 2102-2126.
3. Dong, H., Singh, P., Awad, Y., George, F., Narayanam, K., Arora, S., & Appavoo, J. Towards performance and energy aware kubernetes scheduler. *ACM SIGENERGY Energy Informatics Review*, **2025**, *5*(2), 69-75.
4. Rao, W., & Li, H. Energy-aware scheduling algorithm for microservices in Kubernetes clouds. *Journal of Grid Computing*, **2025**, *23*(2), 2.
5. Kumari, K., Dakhane, D. Real-Time Node's Power-Aware Kubernetes Scheduler in a Cloud Environment. In *Artificial Intelligence Based Smart and Secured Applications*; Springer: Cham, **2025**, pp. 357–372
6. Rjoub, G., Bentahar, J., Abdel Wahab, O., Saleh Bataineh, A. Deep and reinforcement learning for automated task scheduling in large-scale cloud computing systems. *Concurrency and Computation: Practice and Experience*. **2021**, *33*, e5919.
7. Albuquerque, R., & Jaumard, B. Power prediction and energy aware placement of containers over virtual machines. *Computer Communications*, **2025**, *248*, 108340.
8. Tzenetopoulos, A., Masouros, D., Xydis, S., Soudris, D. Interference-Aware Orchestration in Kubernetes. In *Proceedings of the International Conference on High-Performance Computing*; Springer International Publishing: Cham, **2020**, pp. 321–330
9. Chen, W.Y., Ye, K.J., Lu, C.Z., Zhou, D.D., Xu, C.Z. Interference Analysis of Co-Located Container Workloads: A Perspective from Hardware Performance Counters. *J. Comput. Sci. Technol.* **2020**, *35*(2), 322–336.

10. Shubham, Takahashi, K., Takizawa, H. Leveraging Hardware Performance Counters for Predicting Workload Interference in Vector Supercomputers. In: Li, Y., Zhang, Y., Xu, J. (eds) Parallel and Distributed Computing, Applications and Technologies. PDCAT 2024. Springer, Singapore, **2025**, pp. 553-565.
11. Parikh, M., Soni, A.A., Shah, S.M., Jha, A. R. Big Data Workload Profiling for Energy-Aware Cloud Resource Management. *arXiv* **2026**, arXiv:2601.11935.
12. Medel, V., Tolón, C., Arronategui, U., Tolosana-Calasanz, R., Bañares, J.Á., & Rana, O.F. (Client-Side Scheduling Based on Application Characterization on Kubernetes. In *Proceedings of the International Conference on the Economics of Grids, Clouds, Systems, and Services*; Springer International Publishing: Cham, **2017**, pp. 162–176
13. Ali, D., Sofia, R.C. Experimenting with Energy-Awareness in Edge-Cloud Containerized Application Orchestration. *arXiv* **2025**, arXiv:2511.09116.
14. Xu, Z., Gong, Y., Wang, Q., Wu, W., & Du, X. Enhancing Kubernetes automated scheduling with deep learning and reinforcement techniques for large-scale cloud computing optimization. *Proc. SPIE 13291, Ninth International Symposium on Advances in Electrical, Electronics, and Computer Engineering*, **2024**, 132916E.
15. Zhou, H., Xu, M., Zhang, Y., Li, X., & Wang, Z. A Kubernetes custom scheduler based on reinforcement learning for compute-intensive pods. *Proceedings of the 2025 10th International Conference on Cloud Computing and Internet of Thin*, **2025**, pp. 82-91.
16. Algarni, A., Shah, I., Jehangiri, A.I., Ala'Anzy, M.A., & Ahmad, Z. Predictive energy management for Docker containers in cloud computing: A time series analysis approach. *IEEE Access*, **2024**, *12*, 52524-52538.
17. Piontek, T., Haghshenas, K., & Aiello, M. Carbon emission-aware job scheduling for Kubernetes deployments: T. Piontek et al. *The Journal of Supercomputing*, **2024**, *80*(1), 549-569.
18. Tartari, M., Daci, G. *Profiling-Driven Kubernetes Scheduling:* Early Steps Toward Power-Efficient and Interference-Aware Resource Allocation. *International Conference on Information Technologies and Educational Engineering (ICITEE 2025)*, FTI, **2025**, ISBN 9789928481474.
19. Grinsztajn, L., Oyallon, E., & Varoquaux, G. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in Neural Information Processing Systems*, **2022**, *35*, 507-520.