Learning to Schedule: A Supervised Learning Framework for Network-Aware Scheduling of Data-Intensive Workloads

Sankalpa Timilsina
Tennessee Technological University
Cookeville, TN, USA
stimilsin43@tntech.edu

Susmit Shannigrahi
Tennessee Technological University
Cookeville, TN, USA
sshannigrahi@tntech.edu

ABSTRACT

Distributed cloud environments hosting data-intensive applications often experience slowdowns due to network congestion, asymmetric bandwidth, and inter-node data shuffling. These factors are typically not captured by traditional host-level metrics like CPU or memory. Scheduling without accounting for these conditions can lead to poor placement decisions, longer data transfers, and suboptimal job performance. We present a network-aware job scheduler that uses supervised learning to predict the completion time of candidate jobs. Our system introduces a prediction-and-ranking mechanism that collects real-time telemetry from all nodes, uses a trained supervised model to estimate job duration per node, and ranks them to select the best placement. We evaluate the scheduler on a geo-distributed Kubernetes cluster deployed on the FABRIC testbed by running network-intensive Spark workloads. Compared to the default Kubernetes scheduler, which makes placement decisions based on current resource availability alone, our proposed supervised scheduler achieved 34–54% higher accuracy in selecting optimal nodes for job placement. The novelty of our work lies in the demonstration of supervised learning for real-time, network-aware job scheduling on a multi-site cluster.

CCS CONCEPTS

• **Networks** → **Network experimentation**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

network-aware, scheduling, supervised learning, data-intensive, telemetry

ACM Reference Format:
Sankalpa Timilsina and Susmit Shannigrahi. 2025. Learning to Schedule: A Supervised Learning Framework for Network-Aware Scheduling of Data-Intensive Workloads. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3731599.3767457

SC Workshops '25, November 16–21, 2025, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767457

1 INTRODUCTION

Scheduling is the process of assigning tasks or jobs to resources over time. An efficient scheduler improves system utilization, reduces job latency, and balances load across nodes. The rise of data-intensive applications across science, analytics, and AI/ML has led to growing demand for scalable, distributed infrastructure [12, 16]. These workloads generate and process massive datasets, making them sensitive to how jobs are scheduled. Poor or random scheduling decisions can lead to increased job completion times, congested network paths, and under- or over-utilized nodes [15]. Informed scheduling is key to performance. For instance, if a Spark shuffle-heavy job, where large volumes of intermediate data must be exchanged across nodes, is scheduled on a node experiencing high outbound network traffic, it may significantly delay subsequent stages, even if the node has idle CPU cores.

Recent works have explored reinforcement learning (RL) and heuristic-based scheduling for telemetry-aware orchestration [4, 14]. However, RL methods are often sample-inefficient, require reward engineering, and are difficult to interpret [9]. Heuristics, while simpler, may not generalize across workload types or cluster states [21].

In this work, we propose a network-aware, supervised learning–based scheduler that predicts job completion time per node using real-time cluster telemetry. Our methodology is modular, generic, and should be broadly applicable across diverse cluster and workload environments. Unlike RL-based approaches, our method does not require online exploration. And unlike static heuristics, it is trained on real executions, which allows it to learn how both current network conditions and resource load affect job runtime, and adapt accordingly.

The scheduler collects live metrics such as inbound and outbound network traffic volume for each node, using telemetry data exposed by node-level exporters. In addition, it gathers host-level metrics such as CPU and memory usage to avoid partial observability [17]. The framework uses this live telemetry to predict job completion time across all nodes and schedules each job to the one with the shortest predicted duration. Our contributions are:

- We design a network-aware scheduling system that uses supervised learning to predict job completion time per node from real-time telemetry and job configuration to guide placement decisions.
- We present a preliminary evaluation of our framework on the FABRIC testbed by deploying data-intensive workloads on a geo-distributed Kubernetes cluster spanning multiple sites.

- We show that our approach improves node selection accuracy by 34–54% compared to the default Kubernetes scheduler for job placement.

2 MOTIVATION

We establish our motivation based on the need for network-awareness in data-intensive applications, our choice of Kubernetes and Spark as representative platforms, the rationale for using supervised learning, and considerations around deployability and adaptability of our approach in production environments.

2.1 Network-awareness for Data-Intensive Workloads

Data-intensive applications involve massive data shuffles and I/O operations that make the network a critical performance factor, often more so than compute [11]. For example, in a network-intensive large-scale data-processing engine like Apache Spark, datasets are often cached in memory, which in turn makes network throughput the limiting resource for shuffles and distributed operations [26]. Unbalanced data locality or poor placement of tasks can lead to severe network congestion and degrade cluster performance [14]. Thus, network-aware scheduling policies that account for real-time network conditions are needed to avoid unnecessary data transfers and prevent network bottlenecks from undermining application performance.

2.2 Insurgence of Cloud-Native Applications

Cloud-native applications are highly distributed (often composed of many microservices or tasks) and are generally network-intensive in their interactions [19]. These systems have seen widespread adoption—by 2024, over 89% of organizations had adopted cloud-native technologies, with Kubernetes remaining the dominant orchestration platform at over 93% usage [5]. At the same time, frameworks like Apache Spark are increasingly deployed in such environments to support large-scale data processing.

We select Kubernetes as the orchestration platform and Apache Spark as the representative data-intensive application framework to ensure our solution is relevant to modern cloud-native big data deployments.

2.3 Supervised Learning for Scheduling

Prior efforts like heuristic systems often require extensive tuning to specific workload characteristics, and their one-size-fits-all logic struggles when faced with dynamic or unforeseen scenarios. Another issue is that heuristics are inherently brittle and myopic as they lack the ability to learn and improve. The effectiveness of such systems is often limited in real-world use due to dynamic workload variations and changing system configurations [21].

RL-based systems typically require a huge number of trial runs to converge on a good scheduling policy, because they learn only from incremental rewards. This is computationally expensive and slow [10]. Another concern is stability and convergence guarantees. RL algorithms can be unstable i.e., small changes in hyperparameters or reward design can lead to very different outcomes. They often converge to local optima. This is RL's poor generalization, where the policy overfits to the training scenario unless a very broad range of environments is used in training [25].

Supervised models generalize patterns from historical data and are easier to update when workload characteristics change. They can capture complex nonlinear relationships (e.g., between job attributes and optimal scheduling decisions) that are generally hard to encode in static heuristics [7]. Unlike RL, supervised learning does not require an active simulation loop to improve and can utilize existing logs and off-policy data to train offline, so there's no risk of disrupting live systems during learning.

We would like to make a note that the choice is not always binary. We use supervised learning because it is data-efficient, easy to retrain, and avoids the instability and high overhead of RL.

2.4 Considerations for Deployability and Adaptability

Production schedulers must balance performance with operational simplicity. RL-based methods often require complex infrastructure for online training and are difficult to integrate into existing schedulers [23]. They also risk performance degradation during exploration phases. Heuristic systems are easier to deploy but require manual tuning and frequent intervention to remain effective under changing workloads [2].

Supervised learning offers a middle ground. Models can be trained offline on real telemetry data, validated against historical outcomes, and integrated into production with minimal runtime overhead. Retraining does not require system downtime or large-scale infrastructure changes.

3 SYSTEM DESIGN

We build a supervised learning–based scheduling framework for running network-intensive Spark jobs. The system is trained offline using historical job completion times, corresponding telemetry, and application configurations. Upon a job submission request, the framework ranks all candidate nodes using real-time network and node-level metrics, and selects the one predicted to offer the lowest job completion time. The framework runs externally in user space and integrates seamlessly with Kubernetes. Thus, it requires no modifications to Spark or Kubernetes control plane components.

3.1 Background: Default Scheduling Behavior and Telemetry-Aware Decisions

In Kubernetes, pod placement decisions are handled by the default scheduler (kube-scheduler), which assigns pods to nodes based on resource availability and policy constraints. This process involves two stages: filtering, where nodes that do not satisfy basic requirements (e.g., insufficient CPU/memory) are eliminated, and scoring, where remaining nodes are ranked using a set of scoring functions (e.g., least requested resources, affinity, taints/tolerations). The node with the highest score is then selected for placement [21].

However, the default scheduler is blind to runtime factors such as network variability, CPU pressure, or memory contention. It relies on static resource requests declared in pod specifications, rather than real-time system telemetry or application-specific performance considerations. As a result, even if a node satisfies the requested resources, actual job performance may degrade due to

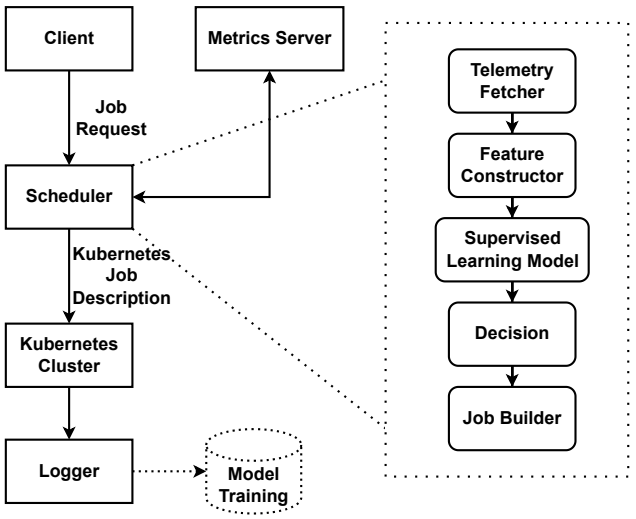

Figure 1: Overview of the system architecture.

background load or interference from co-located pods. This is particularly problematic for data-intensive workloads, where performance is highly sensitive to such runtime variables.

To address the gap, telemetry-aware scheduling introduces live system signals into placement decisions. These signals may include network telemetry (e.g., inter-node RTTs, bandwidth usage), host-level metrics (e.g., CPU and memory pressure), or application-level traces. By leveraging such telemetry, a scheduler can better anticipate the performance impact of placement decisions and optimize for job-level objectives such as completion time.

3.2 Architecture Overview

The overall architecture of the scheduling system is shown in Figure 1. The system consists of five main components: (1) a client that issues job requests, (2) a metrics server that aggregates telemetry data across nodes, (3) a scheduler module that makes predictive placement decisions based on telemetry and a trained model, (4) a Kubernetes cluster that executes the jobs, and (5) a logger that stores outcomes for future retraining. In the following subsections, we describe each of these components in detail.

3.2.1 Client. The client is the entry point to the scheduling pipeline. It is responsible for initiating a job submission request, which includes application-specific parameters such as job type (e.g., sort, join), input data size, and resource configuration (e.g., executor count, memory). The request from the client is handled by the scheduler component (Section 3.2.3) before it reaches the cluster. The client is agnostic to scheduling logic and only handles job initiation.

3.2.2 Metrics Server. The metrics server is responsible for aggregating and exposing system-level telemetry used by the scheduler for decision-making. In our system, this role is fulfilled by a Prometheus instance [18], which is configured to scrape telemetry from multiple sources, including node-exporter for host-level statistics and custom ping mesh exporters for inter-node network latency.

The collected metrics provide a view of each node's runtime state. These include: (a) Network-level telemetry: inter-node RTTs, transmit and receive rates, (b) Node-level telemetry: CPU load, and memory availability. The node-level metrics help capture transient runtime conditions that may not be visible from network telemetry alone.

3.2.3 Scheduler. The scheduler is the decision-making component of the system. It receives job requests from the client, queries current telemetry from the metrics server, and selects the optimal node for job placement using a trained supervised learning model. The scheduler is implemented as an external user-space script that runs outside the Kubernetes control plane. To maintain modularity, the scheduler is internally composed of five submodules:

Telemetry Fetcher. This component queries the Prometheus metrics server at scheduling time to retrieve the most recent telemetry snapshot. It fetches inter-node RTTs from the ping mesh, as well as per-node metrics such as CPU and memory load. These raw values are passed to the Feature Constructor for transformation.

Feature Constructor. The feature constructor transforms raw telemetry and job configuration into structured input vectors for model inference. For each candidate node, it computes a fixed-size feature vector by combining telemetry data from the fetcher with static job-level attributes (e.g., input size, application type, memory requirement). These job-specific fields are critical because different jobs react differently to the same system state; for example, large shuffles are more sensitive to network load than CPU-bound tasks. Including this context helps the model make more accurate predictions across diverse workloads. The complete set of features is summarized in Table 1.

Table 1: Input features used by the scheduling model.

Feature	Description	Type
RTT	Mean, max, and standard deviation of RTT to all peers	Network
Tx/Rx Rate	Transmit and receive throughput (bytes/sec)	Network
CPU	CPU load average (runnable processes)	Node
Memory	Available memory (bytes)	Node
Application Type	Categorical job type (e.g., sort, join)	Job
Input Size	Size of input data (e.g., number of records)	Job
Other Configuration	Total executions, requested memory, etc.	Job

Supervised Learning Model. The scheduler uses a supervised regression model to predict the expected job completion time for a given job on each candidate node. The model is trained offline using historical data collected from real job executions. Each training sample consists of a feature vector constructed from both node-level telemetry and job configuration features (Table 1), along with

the corresponding job completion time, which serves as the prediction target. We describe our application workloads and data characteristics in Section 4.

To evaluate modeling strategies, we experiment with several supervised learning approaches, including linear regression, random forests, and gradient-boosted decision trees (XGBoost) [20, 28]. These models are well-suited to structured telemetry data due to their ability to capture nonlinear interactions and tolerate noisy or missing inputs, while maintaining fast inference and low deployment complexity. Random forests and XGBoost provide strong predictive accuracy and interpretable feature importance, while linear regression serves as a simple, interpretable baseline. We build and evaluate all three models.

Decision Module. Once the supervised model predicts expected job completion times across candidate nodes, the scheduler ranks nodes in ascending order of predicted duration. The top-ranked node is selected as the launch node for the application.

Job Builder. The module generates and submits jobs to the Kubernetes cluster based on the placement decision. It renders a declarative YAML manifest that is understood by Kubernetes for job launch. Node placement is enforced by injecting `nodeAffinity` rules into the generated specification. This ensures that the application is launched only on the selected node. Other parameters, such as job type, input size, and resource limits, are also populated dynamically at this stage.

3.2.4 Kubernetes Cluster. The system runs on a Kubernetes cluster deployed across geographically distributed sites on FABRIC. The nature of workloads is described in Section 4, and the cluster configuration is detailed in Section 5.

3.2.5 Logger. The module captures telemetry and performance data at two stages of each job's lifecycle. Before we submit a job, it records network and node-level telemetry (e.g., latency, throughput, CPU usage) to snapshot the system state. After the job completes, it collects application-level metrics such as job duration, stage execution times, and output size from the application logs. The collected data is used to support offline model training.

4 WORKLOADS

To evaluate the effectiveness of our scheduler, we select a set of data-intensive applications representative of common patterns in distributed computing. Specifically, they differ in their reliance on shuffle-heavy operations (which involve large-scale data exchanges between executors during stages like grouping or sorting) and have varying degrees of inter-node communication and CPU utilization.

We evaluate our system using three Spark-based workloads [1] with differing communication characteristics (Table 2). Each application launches a driver pod on a scheduler-selected node, while executor pods are placed independently by the default Kubernetes scheduler.

The network telemetry of the `Sort` workload is shown in Figures 2 and 3. These plots show differences in network latency and bandwidth across nodes. For example, `Node3` and `Node4` have higher latency, while `Node1` sees more transmit activity. The variation is captured during the learning phase and is fed to the scheduler.

Table 2: Characteristics of the selected workloads.

Application	Rationale
Sort	High network and CPU usage from large shuffles; moderate memory load
PageRank	High network and CPU usage from iterative data exchange; moderate memory load
Join	Skewed network, CPU, and memory usage due to imbalanced joins

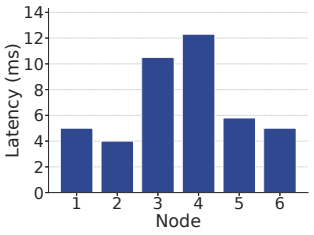

Figure 2: Average latency per node across five runs of `Sort`.

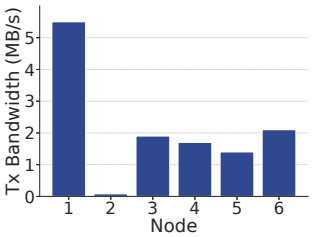

Figure 3: Average transmit bandwidth per node across five runs of `Sort`.

5 EXPERIMENTAL SETUP

We evaluate our scheduler on a multi-site Kubernetes cluster deployed across geographically distributed nodes on the FABRIC testbed. This section describes the cluster configuration, and experiment workflow.

5.1 Cluster Configuration

We deploy a 6-node Kubernetes cluster across three FABRIC [3] sites: UC San Diego (UCSD), Florida International University (FIU), and SRI International (SRI), with two nodes provisioned at each site. The nodes are configured with 6 CPUs, 8 GB RAM, and 25 GB of disk space. Nodes are connected to isolated L3 IPv4 subnets using dedicated 100 Gbps SR-IOV virtual NICs (Mellanox ConnectX-6). To enable inter-node communication, we configure static routes that forward traffic through the FABNetv4 data plane. FABNetv4 spans the FABRIC testbed and provides routing between these isolated subnets. This setup avoids using the management network for application traffic.

To run Spark applications, we use the Spark Operator. The operator allows submission and management of jobs through custom Kubernetes resources. For monitoring, we install the Prometheus stack to collect node-level telemetry for network, CPU, and memory usage. For inter-node RTT, we deploy `ping_exporter` [6] as a DaemonSet. This enables full mesh probing across all nodes.

5.2 Experiment Workflow

We structure each experiment as a batch run and execute Spark jobs sequentially. Our setup includes 60 distinct job configurations across three Spark applications (Section 4) and covers a range of input sizes, executor counts, memory allocations, and shuffle patterns (e.g., group-by, join). To capture diverse runtime conditions,

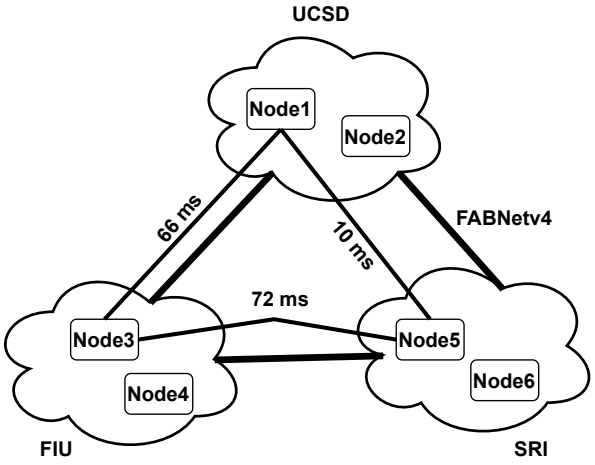

Figure 4: Geographical layout of the cluster across three FABRIC sites with RTT measurements shown along the connecting lines.

each configuration is executed across different `target_node` values (e.g., node-1, node-6), and repeated 10 times. We introduced these variations because they affect both scheduling placement and telemetry input, and thus produce a broad range of system states. In total, our training dataset includes 3600 samples, where each sample represents a unique combination of job configuration and collected telemetry.

Background load (a pod that repeatedly downloads a 10MB file over HTTP using `curl`) is placed randomly on selected nodes during job execution. This simulates network and CPU contention, and leads to observable differences in metrics such as transmit/receive throughput and RTT. While simplified, this setup introduces variability closer to the dynamics observed in multi-tenant environments and helps the model learn from a broader range of system behaviors.

We train three supervised regression models using the full dataset: linear regression, random forest, and gradient-boosted decision trees (XGBoost). Each training sample is formed by joining node-level telemetry collected at scheduling time with static job configuration fields such as application type, input size, and resource requests (Table 1). Telemetry is recorded before the job is launched because the model is designed to make proactive scheduling decisions. The job completion time obtained from application-level logs serves as the prediction target. A representative example of a training row is shown in Table 3.

Table 3: Training sample: Input feature set (subset of the full feature set).

RTT (s)	Rx (MB)	Tx (MB)	CPU (%)	Mem (%)	Input Size	Dur. (s)
0.011	0.024	0.056	2.14	5.8	100000	18.18

6 PRELIMINARY RESULTS

We compared our scheduler against the Kubernetes default scheduling behavior and noted how accurately the scheduler's choice matched the actual fastest node (Top-1) or included it among its two highest-ranked choices (Top-2). The actual fastest node was determined after the run based on the node that yielded the shortest job completion time. Random Forest achieved the highest Top-1 and Top-2 accuracies, followed by XGBoost and Linear Regression. Compared to the default scheduler, supervised learning approaches improved Top-1 accuracy by 34–54% and Top-2 accuracy by 34–62%, depending on the model.

Table 4: Top-1 and Top-2 accuracy of different scheduling approaches in selecting the fastest execution node.

Method	Top-1	Top-2
Kubernetes Default	0.160	0.260
Linear Regression	0.500	0.600
XGBoost	0.560	0.720
Random Forest	**0.700**	**0.880**

While promising, these results are preliminary and drawn from a limited dataset in a controlled environment. They indicate the potential of supervised learning for network-aware scheduling, but further experimentation is needed to validate the findings under a wider range of workloads and cluster conditions (Section 8).

7 RELATED WORK

Learning-based schedulers span several domains. DA-DRLS applies drift-adaptive deep reinforcement learning to IoT resource management, aiming to minimize energy consumption and response time under dynamic workloads [4]. Marchese and Tomarchio extend the Kubernetes scheduler with a communication-aware plugin that models microservice dependencies, traffic history, and network latency to guide pod placement [14]. Intel's Telemetry-Aware Scheduling (TAS) [13] is a scheduler extender that uses platform telemetry from the Custom Metrics API to apply policy-defined filtering, prioritization, and descheduling at placement time and during runtime. Zeus [29] is a Kubernetes-compatible scheduler that colocates latency-sensitive services with best-effort jobs by monitoring server utilization, coordinating isolation features, and opportunistically placing workloads. It reports an increase in CPU utilization from 15% to 60% without SLO violations. In the Spark ecosystem, Shen et al. propose a multi-task prediction model that integrates convolutional layers and multi-head attention, applies dimensionality reduction to selected parameters, and estimates execution time across multiple applications [22]. For ML inference clusters, Eddine et al. propose a tail-latency-aware scheduler for real-time workloads on resource-constrained devices [8]. Their framework combines AI model features and hardware profiles, using offline-trained regression models and an online scoring function to guide task placement. Beyond scheduling and job prediction, prior systems have explored network-centric mechanisms to improve performance and flexibility in distributed environments. N-DISE [27] achieved sustained >31 Gbps WAN throughput in CMS

workflows by exploiting in-network caching, while LIDC [24] introduced a location-independent framework that routes compute requests by name via in-network routers using Kubernetes.

8 CONCLUSION AND FUTURE WORK

This paper presented a network-aware scheduler that uses supervised learning to predict job completion time based on real-time telemetry. Through initial experiments on shuffle-heavy Spark workloads across a geographically distributed Kubernetes cluster, we showed that the scheduler can outperform the Kubernetes default scheduling behavior. In particular, the supervised learning model enabled more informed placement decisions, and placed jobs on better-performing nodes with accuracy gains of 34-54% over the default scheduler. We outline several technical directions to expand and refine our system:

Evaluation at a larger scale with real workloads. While our current evaluation focused on a limited dataset and representative jobs, we plan to deploy the scheduler on larger topologies with real-world, data-intensive applications such as distributed ML pipelines, iterative graph algorithms, and multi-stage streaming jobs. This will allow us to validate the scheduler's performance and scalability under production-like conditions and a wider range of workload characteristics.

Quantifying deployability and retraining costs. Our system is designed to be modular, lightweight and easy to integrate across diverse environments, but further evaluation is needed to measure the overhead of telemetry collection, model retraining frequency, and the trade-offs between model accuracy and scheduling latency. Understanding these costs will help assess how practical the system is for dynamic or production workloads.

Richer network telemetry integration. We currently rely on node-level metrics such as RTT and aggregate bandwidth usage. We aim to incorporate fine-grained telemetry such as link-level utilization (e.g., per-interface throughput and congestion), queuing delay (e.g., estimated buffer occupancy), and passive flow-level statistics (e.g., flow count, duration). These metrics can help capture transient congestion, path-level variability, and network bottlenecks that are not visible at the node level.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grants 2126148 and 2430341.

REFERENCES

[1] Apache Software Foundation. 2025. Apache Spark Documentation. https://spark.apache.org/docs/latest/. Accessed: 2025-08-09.
[2] Dariusz R Augustyn, Łukasz Wyciślik, and Mateusz Sojka. 2024. Tuning a kubernetes horizontal pod autoscaler for meeting performance and load demands in cloud deployments. *Applied Sciences* 14, 2 (2024), 646.
[3] Ilya Baldin, Anita Nikolich, James Griffioen, Indermohan Inder S Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. 2020. Fabric: A national-scale programmable experimental network infrastructure. *IEEE Internet Computing* 23, 6 (2020), 38–47.
[4] Abishi Chowdhury, Shital A Raut, and Husnu S Narman. 2019. DA-DRLS: Drift adaptive deep reinforcement learning based scheduling for IoT resource management. *Journal of Network and Computer Applications* 138 (2019), 51–65.
[5] CNCF. 2025. Cloud Native 2024: Approaching a Decade of Code, Cloud, and Change. https://www.cncf.io/reports/cncf-annual-survey-2024/
[6] czerwonk. 2025. ping_exporter. https://github.com/czerwonk/ping_exporter. Accessed: 2025-08-09.
[7] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *ACM Sigplan Notices* 49, 4 (2014), 127–144.
[8] Khelifa Saif Eddine, Miloud Bagaa, Sihem Ouahouah, Messaoud Ahmed Ouameur, and Adlen Ksentini. 2025. Tail-Latency Aware Scheduler For Inference Workloads. In *2025 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 679–684.
[9] Claire Glanois, Paul Weng, Matthieu Zimmer, Dong Li, Tianpei Yang, Jianye Hao, and Wulong Liu. 2024. A survey on interpretable reinforcement learning. *Machine Learning* 113, 8 (2024), 5847–5890.
[10] Dong Han, Beni Mulyana, Vladimir Stankovic, and Samuel Cheng. 2023. A survey on deep reinforcement learning algorithms for robotic manipulation. *Sensors* 23, 7 (2023), 3762.
[11] Xin He and Prashant Shenoy. 2016. Firebird: Network-aware task scheduling for spark using sdns. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–10.
[12] Tony Hey, Stewart Tansley, Kristin Michele Tolle, et al. 2009. *The fourth paradigm: data-intensive scientific discovery*. Vol. 1. Microsoft research Redmond, WA.
[13] Intel Corporation. 2021. Telemetry Aware Scheduling (TAS) – Automated Workload Optimization with Kubernetes (K8s*) Technology Guide. https://builders.intel.com/docs/networkbuilders/telemetry-aware-scheduling-automated-workload-optimization-with-kubernetes-k8s-technology-guide.pdf. Accessed: 2025-09-23.
[14] Angelo Marchese and Orazio Tomarchio. 2022. Communication Aware Scheduling of Microservices-based Applications on Kubernetes Clusters.. In *CLOSER*. 190–198.
[15] Angelo Marchese and Orazio Tomarchio. 2025. Enhancing the Kubernetes Platform with a Load-Aware Orchestration Strategy. *SN Computer Science* 6, 3 (2025), 1–15.
[16] Narges Mehran, Dragi Kimovski, Hermann Hellwagner, Dumitru Roman, Ahmet Soylu, and Radu Prodan. 2023. Scheduling of distributed applications on the computing continuum: A survey. In *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*. 1–6.
[17] Loizos Michael. 2010. Partial observability and learnability. *Artificial Intelligence* 174, 11 (2010), 639–669.
[18] Prometheus. 2025. Prometheus. https://github.com/prometheus/prometheus. Accessed: 2025-08-09.
[19] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. 2022. An empirical study of service mesh traffic management policies for microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 17–27.
[20] Scikit-learn. 2025. scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/. Accessed: 2025-08-09.
[21] Khaldoun Senjab, Sohail Abbas, Naveed Ahmed, and Atta Ur Rehman Khan. 2023. A survey of Kubernetes scheduling algorithms. *Journal of Cloud Computing* 12, 1 (2023), 87.
[22] Chao Shen, Chen Chen, and Guozheng Rao. 2023. A novel multi-task performance prediction model for spark. *Applied Sciences* 13, 22 (2023), 12242.
[23] Martin Straesser, Johannes Grohmann, Jóakim von Kistowski, Simon Eismann, André Bauer, and Samuel Kounev. 2022. Why is it not solved yet? challenges for production-ready autoscaling. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 105–115.
[24] Sankalpa Timilsina and Susmit Shannigrahi. 2024. LIDC: A Location Independent Multi-Cluster Computing Framework for Data Intensive Science. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 760–764.
[25] Alejandro Del Real Torres, Doru Stefan Andreiana, Álvaro Ojeda Roldán, Alfonso Hernández Bustos, and Luis Enrique Acevedo Galicia. 2023. Deep Reinforcement Learning Approaches for Smart Manufacturing. *Encyclopedia* (2023). https://encyclopedia.pub/entry/40007 Entry adapted from Appl. Sci. 2022, 12, 12377.
[26] Faheem Ullah, Shagun Dhingra, Xiaoyu Xia, and M. Ali Babar. 2024. Evaluation of distributed data processing frameworks in hybrid clouds. *Journal of Network and Computer Applications* 224 (2024), 103837. https://doi.org/10.1016/j.jnca.2024.103837
[27] Yuanhao Wu, Faruk Volkan Mutlu, Yuezhou Liu, Edmund Yeh, Ran Liu, Catalin Iordache, Justas Balcas, Harvey Newman, Raimondas Sirvinskas, Michael Lo, et al. 2022. N-DISE: NDN-based data distribution for large-scale data-intensive science. In *Proceedings of the 9th ACM Conference on Information-Centric Networking*. 103–113.
[28] XGBoost. 2025. XGBoost: Scalable and Flexible Gradient Boosting. https://xgboost.readthedocs.io/en/stable/. Accessed: 2025-08-09.
[29] Xiaolong Zhang, Lanqing Li, Yuan Wang, Enqiang Chen, and Lidan Shou. 2021. Zeus: Improving resource efficiency via workload colocation for massive kubernetes clusters. *IEEE Access* 9 (2021), 105192–105204.