Sakkara: Intelligent Topology-Aware Scheduling for Kubernetes in the Age of AI

José Santos*, *Member, IEEE,* Asser Tantawi†, Pavlos Maniotis†, *Member, IEEE,* Chen Wang†, *Member, IEEE,* Olivier Tardieu†, Tim Wauters*, *Senior Member, IEEE,* Filip De Turck*, *Fellow, IEEE*
* Ghent University - imec, IDLab, Department of Information Technology, Gent, Belgium
† IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA.
Correspondence Email: {josepedro.pereiradossantos}@UGent.be

***Abstract*—The rapid growth of Artificial Intelligence (AI) workloads has introduced unprecedented challenges to modern cloud-native systems, particularly in Kubernetes (K8s)-based environments. These workloads often demand low-latency communication, high resource locality, and efficient utilization of heterogeneous hardware devices such as Graphics Processing Units (GPUs) and specialized accelerators. However, the existing scheduling mechanisms in K8s are typically unaware of the underlying physical topology, leading to performance degradation and inefficient resource usage. This paper presents *Sakkara*, a novel topology-aware scheduling framework designed to optimize the placement of AI workloads in K8s clusters. *Sakkara* incorporates a hierarchical model of the Data Center (DC), including nodes and racks, enabling flexible scheduling strategies that account for resource availability and risk-aware metrics that mitigate performance interference and constraint violations caused by topology-unaware placement. *Sakkara* extends existing scheduling logic in K8s with placement strategies that guide pod allocation using configurable topology constraints, aiming to minimize communication costs and maximize workload performance. We evaluated *Sakkara* on a representative AI workload, a distributed training application under different cluster configurations. Experimental results show that *Sakkara* improves job completion time, throughput, and memory utilization compared to available K8s schedulers, achieving improvements of up to 10%. *Sakkara*, available as open-source, offers a promising pathway toward topology-conscious orchestration of AI workloads in next-generation cloud environments.**

***Index Terms*—Artificial Intelligence, Topology-aware, Scheduling, Orchestration, Kubernetes**

I. Introduction

Kubernetes (K8s) has emerged as the de facto orchestration platform for containerized applications, providing robust abstractions for resource management, scaling, and automation [1]. Its widespread adoption spans a variety of workloads, ranging from microservices to large-scale data processing pipelines [2]. However, the recent surge in Artificial Intelligence (AI), particularly those involving distributed training and inference, has exposed fundamental limitations in the available K8s scheduling mechanisms [3], [4].

AI workloads exhibit stringent requirements for resource locality, inter-node bandwidth, and accelerator availability [5]. Training large-scale models often involves frequent communication between nodes, where minimizing latency and maximizing throughput are essential to reduce job completion time and improve system efficiency [6]. Similarly, inference tasks benefit from optimal placement near specialized hardware such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs) to meet performance targets [7]. Existing K8s schedulers, however, typically operate with limited visibility into the physical topology of the infrastructure, leading to suboptimal scheduling decisions that degrade performance.

Many Data Centers (DCs) today follow hierarchical physical layouts, where nodes are grouped under various levels of aggregation, such as servers, racks, rows, and zones [8]. This hierarchical tree topology significantly influences workload performance. In traditional cloud environments, major providers (e.g., AWS [9], Azure [10], IBM [11]) already offer placement groups for Virtual Machines (VMs), allowing users to specify policies such as co-location or distribution of resources across the DC topology to optimize application performance and availability. Despite their prevalence in High Performance Computing (HPC) and cloud platforms, K8s still lacks robust topology-aware scheduling mechanisms.

To address the limitations of existing K8s schedulers, this paper introduces *Sakkara*[1], a topology-aware scheduling framework for K8s, designed specifically to address the ongoing deployment challenges introduced by AI workloads. *Sakkara* is designed to efficiently deploy sets of homogeneous pod replicas[2], by leveraging hierarchical topological information. It models the cluster as a tree structure: leaves represent individual nodes, while internal nodes capture higher-level logical groupings such as racks and zones. This structure enables Sakkara to support a vast set of fine-grained deployment policies, including *packing* and *spreading* at any level of the hierarchy. These policies allow application-specific placement strategies that closely align with both the physical and logical

Manuscript received May 22, 2026

[1] *Sakkara* refers to the ancient Egyptian step pyramid at Saqqara, considered the earliest large-scale cut stone construction. The name reflects the scheduler's hierarchical approach to workload placement, where pods are methodically arranged according to the cluster's tree topology.

[2] In K8s, a pod is the smallest deployable unit, typically representing one or more tightly coupled containers sharing the same network namespace and storage volumes.

layout of the cluster.

Inspired by the step pyramid structure of its namesake, *Sakkara* uses a solver-based approach to find optimal placements that satisfy all constraints of the group that leverages the *Chic-sched*[3] algorithm [12]. *Chic-sched* operates with loosely defined constraints, such as *packing* and *spreading*, and avoids retries by offering suboptimal orchestration strategies that reduce placement failures. In addition, a notable feature of Sakkara is its support for gang scheduling with priorities, allowing entire groups to be preempted to favor higher-priority jobs [13], [14]. This work builds upon our prior publication [15], where we evaluated the impact of *Chic-sched* strategies on the communication overhead of common collective operations used in modern AI workloads through experiments with the *Venus* discrete-event network simulator [16].

The key novelty of this work lies in introducing a topology-aware scheduling framework for K8s that enables fine-grained, hierarchy-aware placement decisions across multiple levels of the cluster (e.g., node, rack, and zone). In contrast to existing schedulers that rely on coarse-grained placement constraints or flat scheduling policies, *Sakkara* explicitly integrates hierarchical topology information into the scheduling process through a solver-based approach, enabling expressive and workload-aware placement strategies. The main contributions of this work are the following:

- **Design and Implementation of *Sakkara***: a topology-aware scheduling framework that enhances K8s scheduling features with fine-grained topology constraints and priority-based gang scheduling. In contrast to prior work, which primarily considers coarse-grained placement policies, *Sakkara* enables expressive placement policies (e.g., rack- and node-level spreading/packing) directly aligned with the physical cluster topology. The idea of topology awareness in K8s behind *Sakkara* was first introduced in our previous work [17]. Its design and implementation are detailed in Section III, including the integration of the *Chic-sched* solver to compute fine-grained placements.
- **Extensive Evaluation of *Sakkara* under AI workloads**: The experiments consist of representative distributed training and inference workloads under varying network latency conditions. Results show that *Sakkara* consistently improves throughput and reduces job completion time compared to state-of-the-art K8s schedulers, while maintaining competitive resource efficiency (Section V).

The remainder of the paper is organized as follows: Section II reviews related work on K8s scheduling and pod placement, with a particular focus on recent advances in topology-aware strategies. Section III introduces the design of *Sakkara*, detailing its core features, including topology specification, supported placement constraints, and the integration of *Chic-sched* algorithms. This section also analyzes the scalability of *Sakkara* and the impact of its different algorithmic variants. Section IV describes the experimental setup, including the testbed configuration, evaluated schedulers, and selected performance metrics. Then, Section V presents and discusses the evaluation results. Finally, Section VI summarizes the paper and outlines directions for future work.

[3]*Chic-sched* is openly available here: https://github.com/IBM/chic-sched

II. Background & Related Work

K8s Scheduling and Pod Placement. K8s provides a flexible yet general-purpose scheduling framework that supports various policies such as binpacking, resource-aware placement, and affinity/anti-affinity constraints [25]. The default scheduler, known as KS, mainly focuses on individual pod placement based on available resources (e.g., CPU and memory), and when requested, it considers node affinity and topology spread constraints to control distribution across predefined failure domains (e.g., zones or regions). However, these mechanisms are relatively simple and primarily target independent pods rather than coordinated groups of pods with complex placement needs.

The Volcano project [26] provides additional scheduling plugins for K8s:

1) The Gang [20] plugin considers tasks running in different containers as a group. It ensures that a minimum number of containers in the group can be deployed as a whole based on the cluster resource availability.
2) The Task Topology [21] plugin groups containers into buckets based on task affinities and anti-affinities. It minimizes data transmissions between tasks in the same bucket, thus decreasing transmission delay in the overall job execution time.
3) Binpack [22] and the DRF [27] focus on improving resource utilization and preventing small job starvation.

While these plugins improve support for additional constraints for pod placement, they lack fine-grained topology awareness beyond basic resource locality.

The Scheduler Plugins repository [28] provides a collection of extensible plugins for the K8s platform, enabling advanced scheduling strategies beyond the default KS. Notable examples include CoScheduler [19] and the Topology-aware scheduler [23], both of which address microservice dependencies. CoScheduler operates similarly to the Gang plugin provided by Volcano. It ensures that a minimum number of pods belonging to the same *PodGroup* are scheduled as a whole. The Topology-aware plugin focuses on performance issues concerning memory and CPU accesses in Non-Uniform Memory Access (NUMA) nodes, leveraging node-level information defined via *NodeResourceTopology* Custom Resources (CRs) to make placement decisions that respect hardware locality [23]. In addition, the Diktyo plugin [24] further extends K8s scheduling capabilities by supporting complex dependency graphs among heterogeneous pods. Diktyo takes into account both the logical dependencies between pods and the network topology of the cluster, enabling more network-aware placement decisions for microservice applications.

Topology-aware scheduling has been explored in various contexts in recent years [29]–[32]. These efforts focus on

TABLE I: Comparison among different scheduling plugins for the K8s platform.

Plugin / Scheduler	Extension Point	Goals	Cluster Topology	Network Topology	Gang Scheduling	Preemption
Kube-Scheduler (KS) [18]	All	R	✗	✗	✗	✗
CoScheduler [19]	QS & PF & P & R	P	✗	✗	✓	✓
Volcano Gang [20]	PF	P & R	✗	✗	✓	✓
Volcano Task Topology [21]	QS & PF & S	T & R	✗	✗	✗	✗
Volcano Binpack [22]	S	R	✗	✗	✗	✗
Topology-aware [23]	F	T & R	✓	✗	✗	✗
Diktyo [24]	QS & F & S	L & B & P	✓	✓	✗	✗
***Sakkara* (This paper)**	All	T & R	✓	✓	✓	✓

Extension Points: PE = PreEnqueue, QS = QueueSort, PF = PreFilter/PostFilter, F = Filter, PS = PreScore, S = Score, R = Reserve, P = Permit.
(Scheduling) Goals: R = Resources, P = Pod Dependencies, T = Topology, L = Latency, B = Bandwidth.
Cluster Topology, Network Topology, Gang Scheduling, Preemption: ✓= considered; ✗= not considered.

placing microservices based on the infrastructure or application characteristics. Authors aim to improve performance by focusing on the heterogeneity of resources [30], the cluster energy efficiency [29], the underlying processor topology [31], and the node NUMA topology [32]. However, these approaches generally focus on intra-node placement and often lack support for multi-level hierarchical topologies spanning racks and clusters.

Sakkara extends this line of research by introducing a comprehensive framework for topology-aware scheduling in K8s. It supports hierarchical placement policies at arbitrary levels of the cluster topology and enables flexible placement group semantics. Furthermore, its solver-based approach allows it to handle complex topological constraints and gang preemption efficiently. Table I shows a comparison of all K8s plugins mentioned above with *Sakkara* concerning its extension points, scheduling goals, and topology awareness. *Sakkara* goes beyond the current literature by unifying topology-aware and application-aware scheduling into a single, extensible framework that supports a wide range of placement constraints, including packing and spreading trade-offs, locality-based grouping, and dynamic reconfiguration. In summary, *Sakkara*, by combining advanced topological placement policies with native K8s components, offers a practical solution for modern AI workloads in containerized environments.

III. Methodology

This section introduces *Sakkara*, a topology-aware scheduler designed for the K8s platform that aims to optimize the placement of distributed AI workloads by leveraging the physical and logical characteristics of the underlying infrastructure. It incorporates workload-specific requirements and cluster topology constraints into the scheduling process to minimize communication latency, reduce resource contention, and maximize overall performance.

A. System Overview

In 2023, IBM revealed the architectural design and core principles behind *Vela*, the first cloud-native AI supercomputer fully integrated within the IBM Cloud ecosystem [33]. Engineered with flexibility, scalability, and fault tolerance in mind, *Vela* is capable of efficiently training today's most demanding

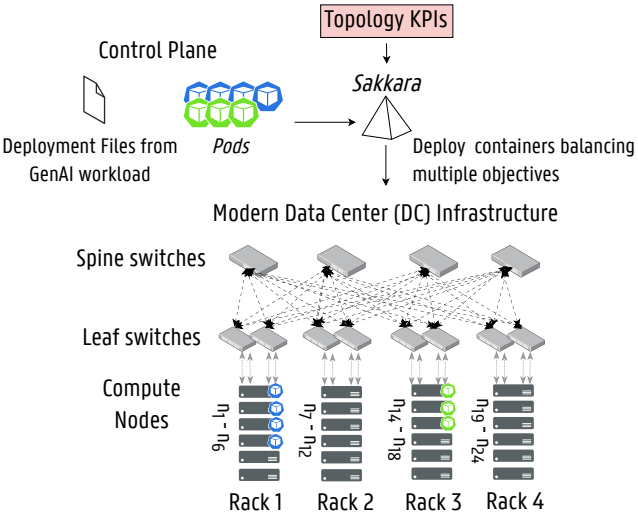


Fig. 1: High-level view of the deployment of AI workloads with *Sakkara* within K8s-based DC infrastructures.

large-scale Generative Artificial Intelligence (GenAI) models, while being readily adaptable to evolving future workloads and innovations in AI research [33]. The networking fabric of *Vela* employs a two-level Clos topology and supports Equal-Cost Multi-Path routing (ECMP). Each leaf switch connects to the spine layer with two 100G links and maintains a 2:3 oversubscription ratio, resulting in an aggregate rack-level bandwidth of approximately 1.6 Tbps per direction, as each rack includes two leaf switches. This topology not only maximizes throughput but also ensures high availability and resilience since the network can sustain failures at the Network Interface Card (NIC), leaf switch, or spine switch levels without impacting ongoing workloads, thereby guaranteeing uninterrupted operation and data flow.

This robust and high-performance infrastructure serves as the backbone for deploying GenAI workloads within containerized environments orchestrated via K8s. Fig. 1 provides a high-level architectural overview, illustrating the interplay between distributed AI training components, the topology-aware scheduler *Sakkara*, and the underlying physical DC topology. The containerized approach enables modular, portable, and scalable deployment of complex GenAI applications across

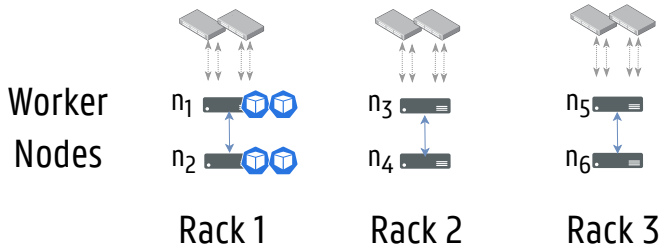


Fig. 2: Hierarchical placement strategy: pods are packed at the rack level and spread across nodes within each rack, corresponding to the constraints in Listings 1 and 2.

distributed cloud environments. Each workload is encapsulated in multiple pods, which can be orchestrated independently while maintaining full interoperability. Distributed AI training components—such as master and worker pods—are deployed across multiple compute nodes within a DC infrastructure, which may span one or more K8s clusters. These components communicate over high-bandwidth, low-latency network links to ensure efficient synchronization of gradients and model updates during training.

Sakkara enhances the orchestration layer by integrating detailed knowledge of the DC hierarchical topology—such as the number of racks and compute nodes—directly into the K8s scheduling decisions. This topology-awareness enables *Sakkara* to optimize pod placement and communication paths, reducing network congestion and improving overall training efficiency. In addition, by leveraging weights that represent real-time network conditions, *Sakkara* dynamically adapts its scheduling policies to balance workloads and avoid hotspots, which is critical for the performance-sensitive nature of distributed GenAI training. *Sakkara* bridges the gap between cloud-native container management and hardware-level network topology, delivering significant improvements in resource utilization and training throughput for GenAI workloads. In addition, *Sakkara* is being open-sourced to the K8s scheduling community[4], allowing practitioners and researchers to explore its design, contribute enhancements, and integrate it into diverse orchestration environments.

B. Cluster Topology Specification in Sakkara

Sakkara models the K8s cluster topology as a hierarchical tree structure to accurately represent the underlying physical or logical organization of the DC. In this model, the leaf nodes correspond to the actual worker nodes where pods are scheduled and run, while each intermediate node represents aggregation layers such as servers, racks, rows, or availability zones. The root of the tree represents the entire cluster as a unified resource pool.

This hierarchical representation allows *Sakkara* to reason about locality, resource distribution, and network characteristics at multiple levels, enabling topology-aware scheduling decisions that improve communication efficiency and fault tolerance. The cluster topology is specified through a Kubernetes ConfigMap labeled with `sakkara.topology`, as shown in Listing 1. This declarative representation exposes the hierarchical structure assumed by the scheduler and enables *Sakkara* to reason about placement decisions across multiple topology levels. Note that the ConfigMap can be updated dynamically at runtime, allowing *Sakkara* to adapt to changes in cluster topology (e.g., adding or removing nodes or racks).

[4]A PR has been opened here: https://github.com/kubernetes-sigs/scheduler-plugins/pull/837

Listing 1 Example of the topology ConfigMap for *Sakkara*.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-topology-configmap
  namespace: sakkara
  labels: {
    "sakkara.topology": # filtering label
  }
data:
  name: "k8s-cluster-tree"
  resource-names: | # resource names
    ["cpu", "memory", "nvidia.com/gpu"]
  level-names: | # level names top down
    ["rack", "node"]
  tree: | # hierarchical topology
    {
      rack-1": {
        "node-1": {},
        "node-2": {}
      },
      "rack-2": {
        "node-3": {},
        "node-4": {}
      },
      "rack-3": {
        "node-5": {},
        "node-6": {}
      }
    }
```

Listing 2 Example of the workload ConfigMap for *Sakkara*.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: workload-test
  namespace: test
  labels:
    "sakkara.group": "" # filtering label
data:
  "name": "workload-test" # group name
  "size": "4" # group size
  "priority": "0" # group's priority
  "constraints": | # constraints to consider
    {
      "rack": {
        "type": "pack"
      },
      "node": {
        "type": "spread"
      }
    }
```

Updates take effect from the subsequent scheduling cycle to maintain consistency during pod group placements.

Alongside the cluster topology, *Sakkara* supports workload-specific placement constraints specified through another ConfigMap labeled with `"sakkara.group"`. This allows users to express how pods within a workload group should be distributed or packed across different topology levels to optimize resource utilization or network locality. Listing 2 shows an example workload ConfigMap defining constraints to pack pods tightly within racks but spread them across individual nodes to reduce single-node dependency, thereby improving resilience and load balancing. In this example, the workload contains 4 pods. Following the specified constraints, all pods are packed into *rack-1*, and then spread evenly across its two nodes, resulting in 2 pods on *node-1* and 2 pods on *node-2*, as illustrated in Fig. 2.

C. Placement Constraints and Chic-sched integration

Sakkara supports a versatile and expressive set of placement constraints that can be applied independently at different levels of the cluster topology hierarchy. These constraints enable fine-grained control over how pods are distributed across resources, helping optimize for locality, resilience, load balancing, and resource efficiency. The supported constraint types include the following:

- **Pack**: Attempts to maximize pod co-location by placing as many pods as possible on the same unit (e.g., rack or node), enhancing data locality and minimizing inter-node communication overhead.
- **Spread**: Ensures pods are distributed evenly across units at the specified hierarchy level to improve resilience by avoiding concentration of pods and reduce resource contention hotspots.
- **Partition**: Divides pods into a specified number of partitions, each an approximately equal-sized group, to achieve controlled grouping or isolation among pods.
- **Range**: Specifies the minimum and maximum allowable number of pods per unit, providing flexible bounds for placement density that can balance packing and spreading goals.
- **Factor**: Requires that the number of pods placed on each unit be a multiple of a given factor, useful for aligning workloads with hardware affinities or multi-instance requirements.

By default, these constraints are *soft*, meaning that *Sakkara* tries to satisfy them to the greatest extent possible but will relax constraints if strict enforcement conflicts with resource availability or cluster state. This balance allows the scheduler to adapt gracefully under resource pressure or topology changes. Though, these constraints can be set as *hard* as well.

At each scheduling cycle, *Sakkara* gathers the current cluster state, topology information, and workload group constraints to formulate the placement of the group as a constrained optimization problem. This formulation captures the multi-level hierarchical constraints alongside resource availability, priority, and affinity considerations. To solve this deployment challenge efficiently, *Sakkara* integrates with an open-source solver implementing the *Chic-sched* algorithm [12]. The solver uses combinatorial optimization techniques to explore feasible placements and outputs an optimal or near-optimal pod-to-node assignment for the entire group in a single run. This solver-based approach offers several advantages:

1) It naturally handles multi-level and multi-type constraints simultaneously.
2) It provides a flexible framework to incorporate new constraint types or policies in the future.
3) It enables *Sakkara* to maintain high scheduling efficiency even for large-scale clusters and workloads.

The main procedures of the *Chic-sched* algorithm are detailed in Alg. 1. It is a heuristics-based method for plac-

Algorithm 1 *Chic-sched*: Group Placement Strategy [12]

procedure PLACEGROUP($pTree, g$) ▷ Compute slots
  for all $n \in pTree$ **do**
    calculate $A(n)$ ▷ Available slots in subtree, root n
  end for
  $M(pTree.root) \leftarrow g.size$ ▷ Remaining members
  $N(pTree.root) \leftarrow 1$ ▷ Remaining siblings
  $root \leftarrow$ PLACEATNODE($pTree.root$)
  return $lTree(root)$ ▷ Log. placement tree for group
end procedure
procedure PLACEATNODE(n)
  $n \leftarrow newLNode(n)$
  $l \leftarrow n.level$
  $R(n) \leftarrow$ RANGE($n, g.const(l), M(n), N(n)$)
  $Z^*(n) \leftarrow \min(\min(A(n), R(n).b), R(n).d)$
  if $Z^*(n) = 0$ **then**
    return n ▷ skip node if cannot place any member
  end if
  if $l = 0$ **then** ▷ leaf node
    $Z(n) \leftarrow \min(A(n), Z^*(n))$
    return n
  end if
  $Z(n) \leftarrow 0$
  $C \leftarrow Order(n.children, g.const(l-1))$
  $siblings \leftarrow |C|$
  for all $c \in C$ **and** $Z(n) < Z^*(n)$ **do**
    $M(c) \leftarrow Z^*(n) - Z(n)$
    $N(c) \leftarrow siblings$
    $c \leftarrow$ PLACEATNODE(c) ▷ recursive depth-first
    if $Z(c) > 0$ **then**
      $Z(n) \leftarrow Z(n) + Z(c)$
      $n.addChild(c)$
    end if
    $siblings \leftarrow siblings - 1$
  end for
  $n.count \leftarrow Z(n)$
  return n
end procedure
function RANGE($n, constraint, M(n), N(n)$)
  if constraint is Hard Pack **then**
    $a, b, d \leftarrow$ all remaining members
  else if constraint is Hard Spread **then**
    $a, b, d \leftarrow$ exactly one remaining member
  else if constraint is Soft Pack **then**
    $a \leftarrow 1$, $b \leftarrow M(n)$, $d \leftarrow$ all remaining
  else if constraint is Soft Spread **then**
    $a \leftarrow 1$, $b \leftarrow M(n)$, $d \leftarrow \lceil M(n)/N(n) \rceil$
  end if
  return $\{a, d, b\}$ ▷ min a, desired d, and max b
end function

ing a group of members g onto a physical tree of nodes $pTree$ while respecting hierarchical constraints. The goal is to produce a logical placement tree $lTree$ that satisfies the group's placement constraints (i.e., Pack or Spread, Hard or Soft) and maximizes the use of available resources. *Chic-sched* operates recursively, starting from the root of the physical tree $pTree.root$, and traversing the nodes in a depth-first order. At each node n, the algorithm computes the number of available slots $A(n)$ and determines the number of members $M(n)$ to attempt to place. It then decides the actual number of members $Z(n)$ to place, creating a corresponding logical node $lNode(n)$ in the resulting logical tree $lTree$. The process continues until all members are placed or no more nodes can accommodate additional members. To avoid repetition, a detailed description of the *Chic-sched* algorithm can be found in our previous work [12].

Regarding the complexity of the *Chic-sched* algorithm, let N_{total} denote the total number of nodes in the $pTree$, N the number of leaf nodes (i.e., physical entities), and M the group size. Since each node is visited at most once and the number of operations performed per node is $O(1)$, the overall complexity is $O(N_{\text{total}})$, except for the node ordering step [12]. For a node with c children, the ordering operation has a complexity of $O(c \log c)$. Consequently, the worst-case scenario occurs in a single-level tree where the root node has N children, yielding an overall time complexity of $O(N \log N)$ [12].

In addition, *Sakkara* supports gang scheduling [13] at the group level. Pods within the same group are scheduled and deployed atomically to ensure consistency and performance guarantees during distributed training or batch workloads. *Sakkara* implements preemption policies [14] that allow lower-priority groups to be preempted as a whole to make room for higher-priority groups under resource contention scenarios. This group-level preemption ensures fairness and priority adherence while minimizing disruption and fragmentation in cluster resource allocation.

D. Weight parameter to guide placement decisions

To further refine placement decisions beyond traditional resource availability and topology constraints, *Sakkara* introduces a flexible *weight* parameter. This concept allows *Sakkara* to incorporate environmental and risk-related metrics (e.g., memory pressure, node temperature) into the decision process, thus optimizing system stability and energy efficiency. While many K8s schedulers prioritize CPU and memory fit or balance resource pressure across nodes, modern AI and HPC workloads introduce new scheduling challenges. Sustained high utilization on certain nodes can lead to thermal hotspots, hardware degradation, or unpredictable performance. For example, average temperature across nodes or racks can serve as a proxy for hardware risk. Placing pods on cooler nodes reduces thermal wear and thermodynamically balances the cluster. Other possible metrics include power consumption trends, thermal inertia, fan speed, or reliability scores derived from monitoring data.

Inspired by Trimaran's Load Variation Risk Balancing plugin [34], *Sakkara* generalizes the concept to support a wide range of customizable *risk* metrics, represented as the weights associated with nodes or racks. By default, *risk* is computed as a composite score that captures both the average resource utilization and its variability across available nodes, currently supporting CPU and memory metrics. Weights are treated as a tunable modifier during placement scoring. Each node or rack is assigned a dynamic weight, reflecting its current operating *risk* level or desirability. This weight influences pod placement in the following ways:

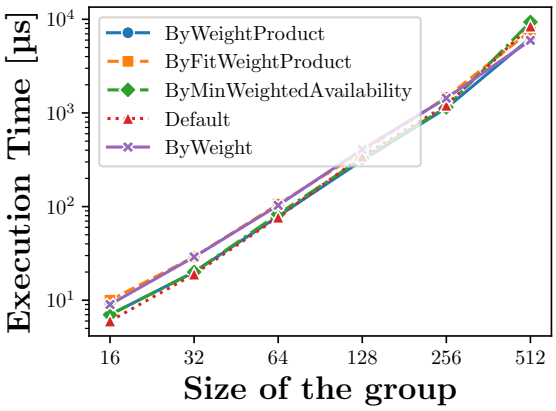
- ***ByWeight***: If two candidate units are otherwise equivalent in terms of resource availability and topology constraints, the scheduler compares their weights and selects the unit with the lower risk.
- ***ByWeightProduct***: Compares candidates based on the product of weight and claimed resources, prioritizing units that can accommodate the workload with minimal risk.
- ***ByFitWeightProduct***: Uses the product of weight and fit resources to favor units where the placement is both feasible and risk-efficient.
- ***ByMinWeightedAvailability***: Selects nodes that minimize the weighted inverse of available resources, effectively balancing risk and utilization in the long term.

These strategies allow *Sakkara* to fine-tune scheduling for heterogeneous clusters, where some nodes may be more suitable than others due to thermal zones, age, or hardware characteristics. Note that *Sakkara* also enables a *Default* strategy that disregards weights entirely for deployment decisions.

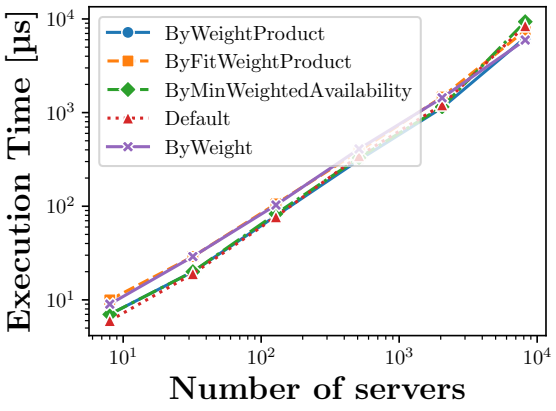
E. Scalability Analysis

The experiment shown in Fig. 3 considers a single-zone deployment with variable numbers of racks, each containing 2 servers, and the group size varies over a factor of 8. Each server has a CPU capacity of 16 units, with a demand of 2 units per placement request and an overall cluster load of 40%. Resource allocation follows a skewed distribution with $p_0 = \frac{1}{4}$, $p_1 = \frac{1}{8}$, and a coefficient of variation (CoV) of 1.5. Soft constraints enforce rack-level spreading and server-level packing. Each data point represents the average over 10000 execution runs on a laptop with 14-core Intel i7-12700H CPU @ 4.7 GHz processor with 16 GB of memory.

The *Default* strategy, which ignores node weights, successfully handles all placement requests for cluster sizes higher than 32 servers. At the smallest scale (8 servers), it fails to allocate 1.62% of requests, indicating constraint violations due to limited capacity and strict placement preferences. As expected, runtime scales approximately linearly, increasing from 6 μs at 8 servers to 8.5 ms at 8192 servers. All weight-based strategies eliminate placement failures at scales higher than 32 servers, confirming that integrating weights does not add additional complexity to the *Chic-sched* algorithm. At the smallest scale (8 servers), *ByWeight*, *ByMinWeightedAvailability*, and *ByFitWeightProduct* strategies incur marginally higher

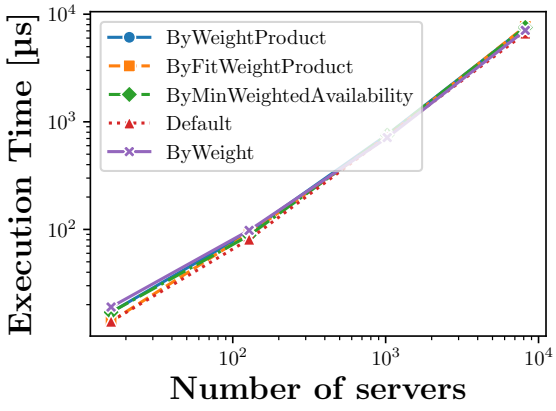


(a) Size of the group.

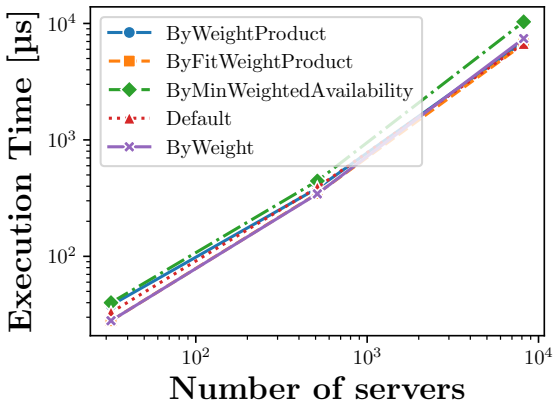


(b) Number of servers.

Fig. 3: The scalability of the various placement algorithms for a 2−level tree. Both axes are shown on a logarithmic scale.



(a) 3−level tree.



(b) 4−level tree.

Fig. 4: The scalability of the various placement algorithms for higher tree depths. Both axes are in logarithmic scale.

failure rates (1.6–3.7%) compared to *Default*. This is likely due to additional tie-breaking logic based on weight scores that interferes slightly with packing/spreading soft constraints under extreme capacity limits. In terms of execution time, *ByWeightProduct* and *ByMinWeightedAvailability* achieve the lowest latency at 8192 servers (6.3 ms and 9.3 ms, respectively), on par with *Default*. *ByFitWeightProduct* shows slightly higher runtime (7.5 ms at 8192 servers) due to additional scoring logic that combines fit and weight evaluation.

Overall, all heuristics scale well to large cluster sizes while preserving low latencies and avoiding placement failures. *ByWeightProduct* and *ByMinWeightedAvailability* emerge as strong candidates for practical deployment, offering a trade-off between computational cost and placement robustness. These results highlight the effectiveness of weight-guided placement in managing heterogeneity and improving placement flexibility under soft constraints.

To further evaluate the effect of hierarchical depth on *Sakkara*, an additional experiment was conducted with different hierarchical configurations: a 3-level tree (zones → racks → servers) and a 4-level tree (clusters → zones → racks → servers). These configurations allow us to investigate how increasing the number of hierarchical levels impacts the scheduling latency of *Sakkara* under varying cluster sizes. In addition, to characterize the computational cost of each configuration, we model the $pTree$ using a branching factor vector $[1, b_1, b_2, \ldots, b_d]$, where b_l denotes the number of children of each node at level l. The total number of nodes is then:

$$N_{\text{total}} = \sum_{l=0}^{d} \prod_{i=1}^{l} b_i \tag{1}$$

The dominant source of computational overhead is the ordering of children at each internal node, which incurs a cost of $O(c \log c)$ for a node with c children, yielding a total ordering cost proportional to $c_v \log_2 c_v$ summed over all internal nodes.

Table II reports these complexity metrics alongside the measured execution times for all three configurations at 8192 servers. N_{total} increases by only ∼6% from the 2-level to the 4-level configuration (8,257 → 8,777 nodes), as leaf nodes dominate the total count. Furthermore, deeper trees with lower per-level branching factors yield a *reduced* total ordering cost (57,344 → 32,768 operations), owing to the *faster-than-*

TABLE II: Sakkara scales well with hierarchy depth: while deeper hierarchies naturally introduce additional traversal overhead, the increase in execution time remains modest. Despite increasing depth, N_{total} grows by only 6% and the ordering cost decreases, explaining the near-constant execution time across all configurations.

Depth	Structure	Group Size	N_{total}	Cost	Exec. Time
2-level	[1, 64, 128]	512	8,257	57,344 ops	7.52 ms
3-level	[1, 16, 16, 32]	128	8,465	40,960 ops	7.31 ms
4-level	[1, 8, 8, 8, 16]	64	8,777	32,768 ops	7.63 ms

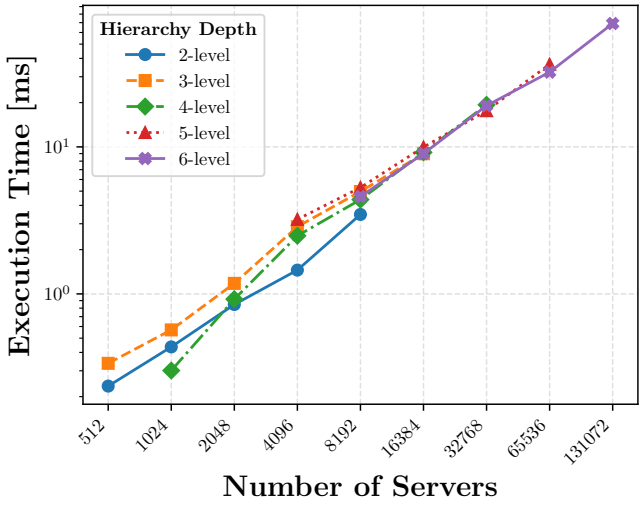


Fig. 5: Average latency of *Sakkara* across all placement strategies as a function of cluster size, for hierarchy depths ranging from 2 to 6 levels. Both axes are in logarithmic scale. Note that not all depth configurations share the same server range, as each hierarchy depth imposes structural constraints on achievable cluster sizes via its branching factor vector.

linear growth of $c \log c$ with respect to c. These theoretical observations are consistent with the measured execution times of 7.52 ms, 7.31 ms, and 7.63 ms for the 2-, 3-, and 4-level configurations, respectively (Fig. 4). This demonstrates that *Sakkara*'s algorithms effectively handle more complex hierarchical structures without significant performance degradation.

Additionally, Fig. 5 shows the average latency across all placement strategies as a function of cluster size, for hierarchy depths ranging from 2 to 6 levels. The results confirm that *Sakkara* scales consistently across all depths: the latency curves are nearly parallel on a log-log scale, indicating near-linear scaling with respect to cluster size regardless of hierarchy depth. Moreover, at equivalent cluster sizes (e.g., 8192 servers), all depth configurations yield comparable latencies (4–5 ms), demonstrating that increasing the number of hierarchy levels introduces only marginal overhead.

F. Placement Strategy Evaluation

To evaluate the behavior of different placement strategies, we designed a controlled scenario consisting of six physical nodes with identical resource capacities but varying initial loads and assigned weights. This setup allows us to analyze how each strategy responds to resource constraints and hierarchical topology. The nodes are organized into a two-level tree structure rooted at *root*, with two child racks (r_1, r_2):

$$root \rightarrow \{r_1 \rightarrow [n_1, n_2, n_3],\ r_2 \rightarrow [n_4, n_5, n_6]\}$$

Each node has a total capacity of $[10, 10]$ and begins with a predefined resource allocation. To simulate heterogeneity, weights are assigned to nodes after the initial load is applied, and average weights are calculated for the corresponding racks, as detailed below:

- n_1: allocated $[3, 3]$, weight $w = 46$
- n_2: allocated $[2, 2]$, weight $w = 76$
- n_3: allocated $[1, 1]$, weight $w = 12$
- n_4: allocated $[2, 2]$, weight $w = 36$
- n_5: allocated $[3, 3]$, weight $w = 19$
- n_6: allocated $[1, 1]$, weight $w = 57$
- r_1: $[n_1, n_2, n_3]$ - $w = 46$
- r_2: $[n_4, n_5, n_6]$ - $w = 38$

Soft constraints were applied at different levels of the placement hierarchy. At Level 1 (rack), an affinity of type *Pack* was used, encouraging workloads to be placed within the same rack when possible. In contrast, at Level 0 (node), an affinity of type *Spread* was applied to promote distribution across individual servers within a rack.

Fig. 6 compares five placement strategies available in *Sakkara* for placing a group of 20 pods in these nodes. The gray bars indicate the existing resource allocation on each node before scheduling the group. The *Default* strategy ignores node weights, and as a result, the placement heavily utilizes nodes n_4, n_5, and n_6, leading to a heavy load on rack r_2. Neglecting weights can lead to hotspots and inefficient resource usage. In contrast, all weight-aware strategies prioritize levels with higher weights, consequently favoring racks with higher weights and consequently nodes, concentrating the load on n_1, n_2, and n_3, part of rack r_1. These weight-aware strategies are especially suitable for heterogeneous environments, where leveraging node weights enables more balanced placements, improved resource efficiency, and reduced risk of bottlenecks.

TABLE III: Software versions of the testbed infrastructure.

Software	Version
Kubeadm / Kubectl	v1.28.15
Go	go1.22.2
NVIDIA device plugin for K8s	v0.17.1
Containerd	containerd://1.7.24
Linux Kernel	6.8.0-54-generic
Operating System	Ubuntu 24.04.2 LTS

IV. Evaluation Setup

Testbed Infrastructure. A K8s cluster is set up using kubeadm[5] on bare metal machines created on the Virtual

[5]https://kubernetes.io/docs/reference/setup-tools/kubeadm/

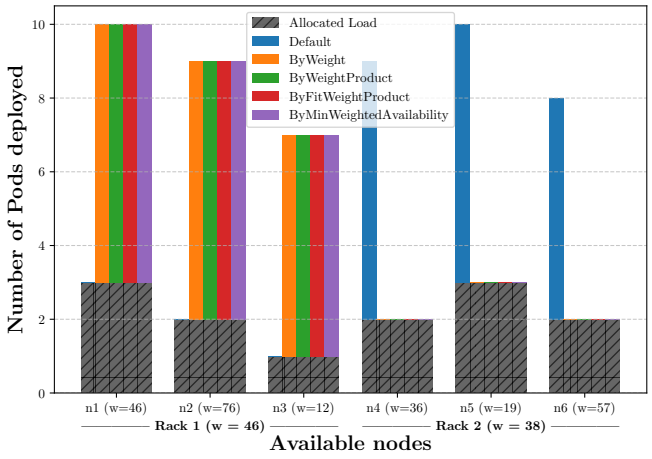


Fig. 6: Comparison between the placement strategies available in *Sakkara*. The *Default* strategy ignores node weights, while weight-aware strategies clearly favor racks and consequently nodes with higher weights.

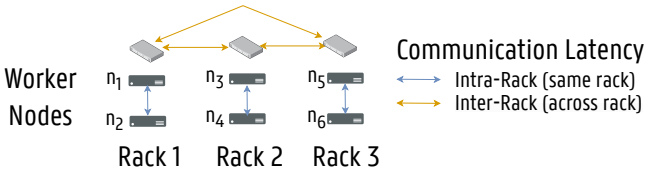
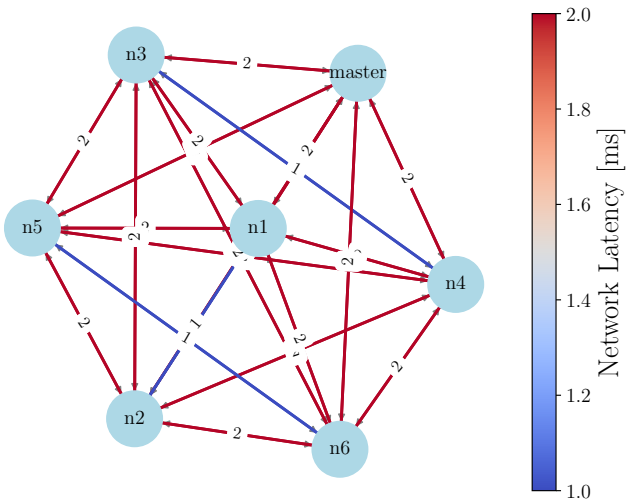


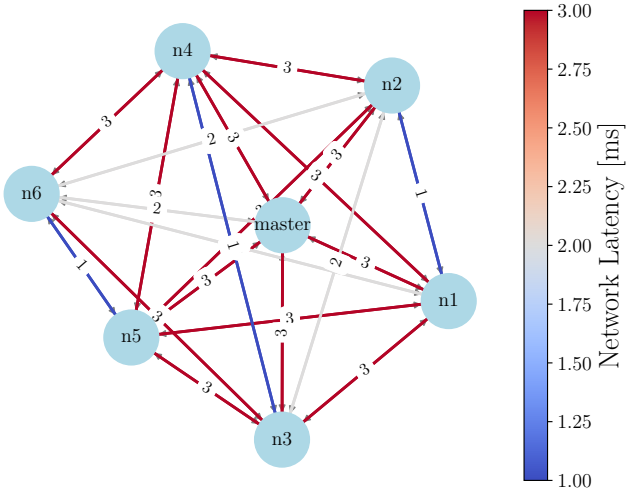
Fig. 7: Illustration of the testbed infrastructure.

TABLE IV: Latency configurations for both scenarios.

Communication	Moderate (2×)	High (5×)
Intra-Rack (same rack)	0.25 ± 0.1 ms	0.20 ± 0.1 ms
Inter-Rack (across racks)	0.50 ± 0.2 ms	1.00 ± 0.2 ms



(a) Moderate scenario.



(b) High scenario.

Fig. 8: The network graph of the K8s cluster when setting up different Traffic Control (TC) rules for both scenarios.

Wall (VWall) testbed, typically used for large networking and cloud experiments [35], [36]. The network connections between servers are illustrated in Fig. 7, and Table III lists the software versions of all the components used to set up the K8s cluster. The K8s cluster consists of 7 nodes (1 master and 6 workers), with servers representing three racks: r_1: $[n_1, n_2]$, r_2: $[n_3, n_4]$, and r_3: $[n_5, n_6]$. Each worker node is equipped with an NVIDIA GeForce GTX 1080 Ti GPU, a 12-core Intel Core i7-8700 CPU (3.20 GHz), and 32 GB of RAM. Two scenarios have been evaluated to assess the impact of intra-rack versus inter-rack delay on application performance:

- **Moderate**: This scenario models a 2× latency difference between intra-rack and inter-rack communication. It reflects moderate network heterogeneity within a DC or K8s cluster.
- **High**: This setup introduces a 5× latency difference between intra-rack and inter-rack communication, representing more pronounced network asymmetry, such as that found in poorly optimized topologies or geographically stretched racks.

The authors in [37] showed that the differences between intra-rack and inter-rack communication, with inter-rack latency typically 2–5× higher due to topology and bandwidth allocation constraints. Varying link delays are emulated on network connections via Traffic Control (TC) [38]. Table IV summarizes the network interface delays introduced at each node for both scenarios. Meanwhile, Fig. 8 illustrates the network latency graph of the K8s cluster, generated using a developed netperf component[6]. Note that the delays have been originally measured in microseconds and divided by 1000 to present them in milliseconds for improved readability. As shown, on one hand, the moderate scenario results in latency differences of approximately 1 ms for intra-rack communication and at least 2 ms for inter-rack communication. On the other hand, the high scenario exhibits larger differences of around 1 ms for intra-rack and at least 3 ms for inter-rack.

[6]https://github.com/jpedro1992/pushing-netperf-metrics-to-prometheus/

Evaluated Schedulers. This paper evaluates the performance of three distinct K8s scheduling strategies:

- ***Sakkara***: the topology-aware scheduler proposed in this paper that prioritizes minimizing communication delays across distributed workloads by leveraging the physical topology information.
- **KS** [18]: representing the default scheduling strategy used in K8s, offering baseline performance based mostly on resource availability.
- ***CoScheduler*** [19]: a co-location-aware scheduling plugin for K8s that implements gang scheduling to improve the performance of tightly coupled distributed tasks. It coordinates the placement of homogeneous pods across multiple nodes, ensuring that all required resources are available before scheduling the group as a whole.

Note that the *Default* strategy of *Sakkara*, which does not consider weights, has been considered during the experiments. While we recognize the importance of assessing weight-aware strategies, this paper focuses specifically on quantifying the benefits of topology-awareness alone on application performance in K8s containerized environments. A comprehensive evaluation of the weight-aware strategies of *Sakkara* is planned as part of future work.

AI workload. The evaluation consists of a distributed PyTorch job orchestrated via a K8s ConfigMap[7], which executes a multi-GPU script using the CIFAR-10 dataset[8]. During the evaluation, both training and inference tasks were assessed for this workload. For *training*, each pod is part of a PyTorch Distributed Data Parallel (DDP) group, where processes synchronize using the NCCL backend. The script initializes a distributed process group based on environment variables, sets GPU affinity based on rank, and distributes the data using PyTorch's *DistributedSampler*. The training model is MobileNetV2 [39], and each pod performs training over 10 epochs on a reduced CIFAR-10 subset (2,500 samples) with a batch size of 128. The workload tracks key running metrics, such as the average epoch time, the gradient synchronization time, and the data synchronization time, using *barrier()* calls. Performance metrics such as throughput (images/sec) and memory usage (via psutil) are also logged per run. The script includes timing logic to analyze inter-node communication performance, making it suitable for evaluating scheduling decisions for tightly coupled, latency-sensitive AI workloads.

For *inference*, the same model (MobileNetV2) is deployed in a multi-GPU PyTorch DDP setup. The same subset of the CIFAR-10 test dataset (2,500 samples) is used to simulate realistic inference requests across multiple nodes. Each pod performs batched forward passes without gradient computation and tracks per-batch latency, throughput, and memory usage. This workload captures the performance impact of scheduling and placement decisions on inference jobs, which have different synchronization patterns and latency sensitivity compared to training.

[7]https://kubernetes.io/docs/concepts/configuration/configmap/

[8]https://www.cs.toronto.edu/~kriz/cifar.html

Performance Metrics. To evaluate the impact of scheduling decisions and network conditions on distributed AI training and inference, we assess the following metrics:

- **Gradient Time [ms]**: Measures the average time spent during gradient computation and synchronization, meaning the duration of the *loss.backward()* call. This metric is measured only for the training workload.
- **Synchronization Time [ms]**: Captures the time spent on explicit synchronization barriers (i.e., *torch.distributed.barrier()*). High synchronization times might indicate poor scheduling deployments, where some pods are significantly slower. This metric is measured only for the training workload.
- **Episode Time [s]**: Represents the average time per epoch, including the gradient and synchronization phases for the training workload.
- **Memory Usage [MB]**: Represents the memory consumed by the AI job.
- **Throughput [img/s]**: Computed as the number of images processed per second over the entire job duration.
- **Total Time [s]**: Measures the total wall-clock time taken to complete the training or inference job.

All these performance metrics allow us to assess the impact of various scheduling strategies on compute performance, communication efficiency, and resource utilization in distributed AI workloads.

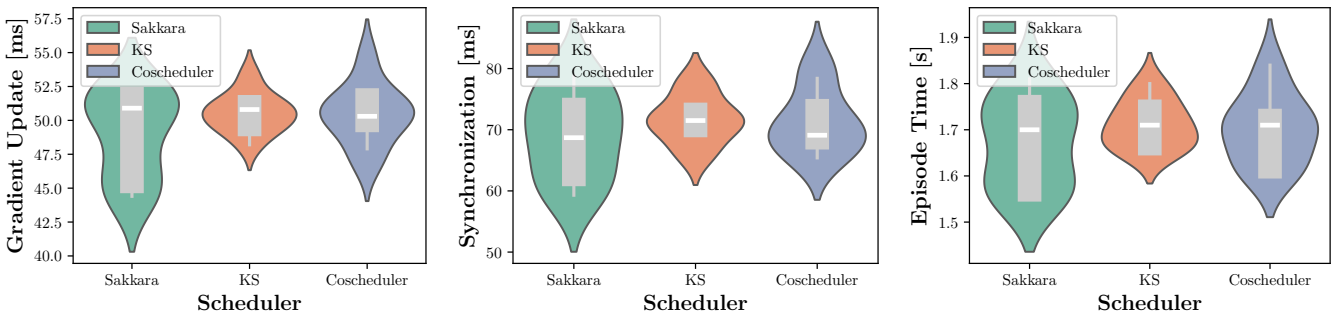
V. Evaluation Results

This section presents the experimental evaluation of *Sakkara* and compares its performance against baseline K8s schedulers under representative distributed AI workloads. We assess the performance of both a training and an inference workload to capture different communication and synchronization characteristics, and we consider two infrastructure scenarios with increasing network latency (moderate and high).

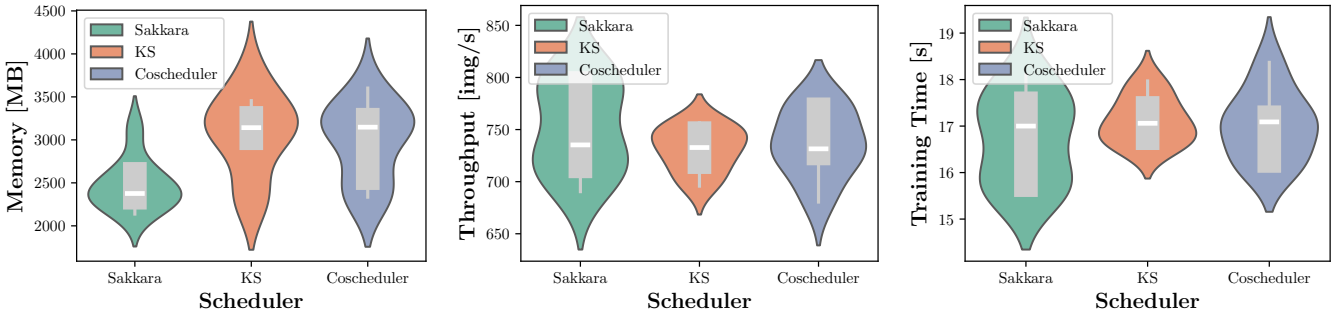
A. Training Workload

The experimental evaluation provides insights into the performance of the three schedulers for both moderate (Fig. 9) and high (Fig. 10) scenarios during training The results consistently demonstrate that *Sakkara* outperforms existing schedulers in key performance metrics, particularly under the higher scenario, where topology-aware placement becomes more critical.

The analysis of training performance into gradient computation and synchronization phases reveals the benefits of topology-aware placement (Figures 9a, 10a, 9b, and 10b). Gradient computation times remain consistently lower for *Sakkara* (48.9 ± 2.0ms moderate, 48.3 ± 1.4ms high) compared to KS and *CoScheduler* (both around 50–52ms). The most notable improvements are observed in synchronization times: in the high scenario, *Sakkara* achieves $62.8 \pm 1.5ms$, significantly outperforming KS (71.7 ± 3.7ms) and *CoScheduler* (69.6 ± 4.3ms). This result indicates that *Sakkara's* topology-aware placement schemes reduce inter-node communication

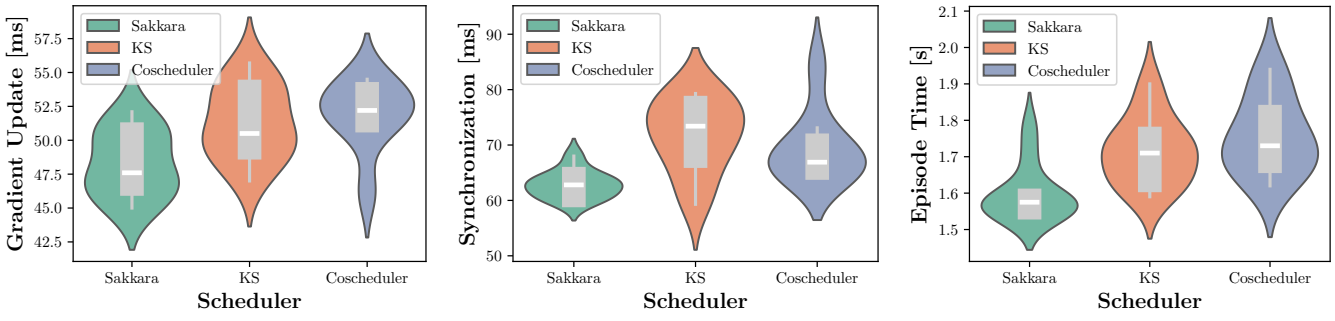


(a) Gradient Time. (b) Synchronization Time. (c) Episode Time.

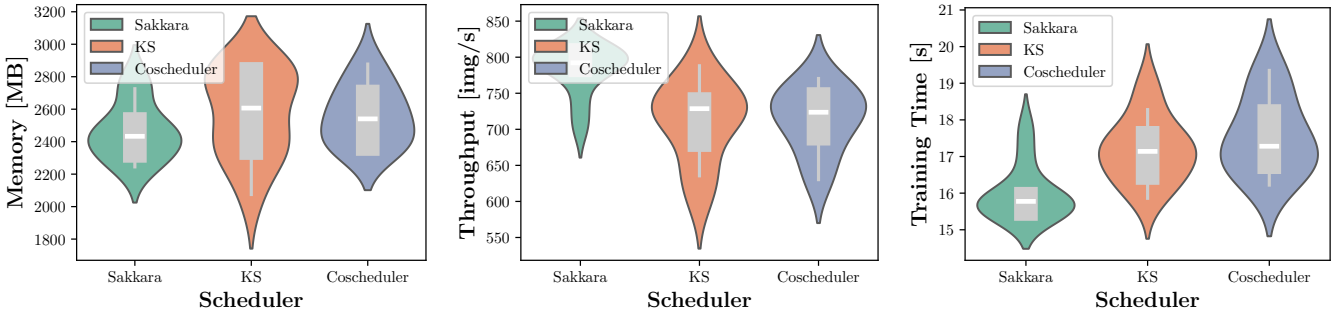


(d) Memory Usage. (e) Throughput. (f) Total Time.

Fig. 9: Performance evaluation of the training workload for the moderate scenario. *Sakkara* consistently achieves lower synchronization overhead and improved throughput compared to the baseline schedulers.

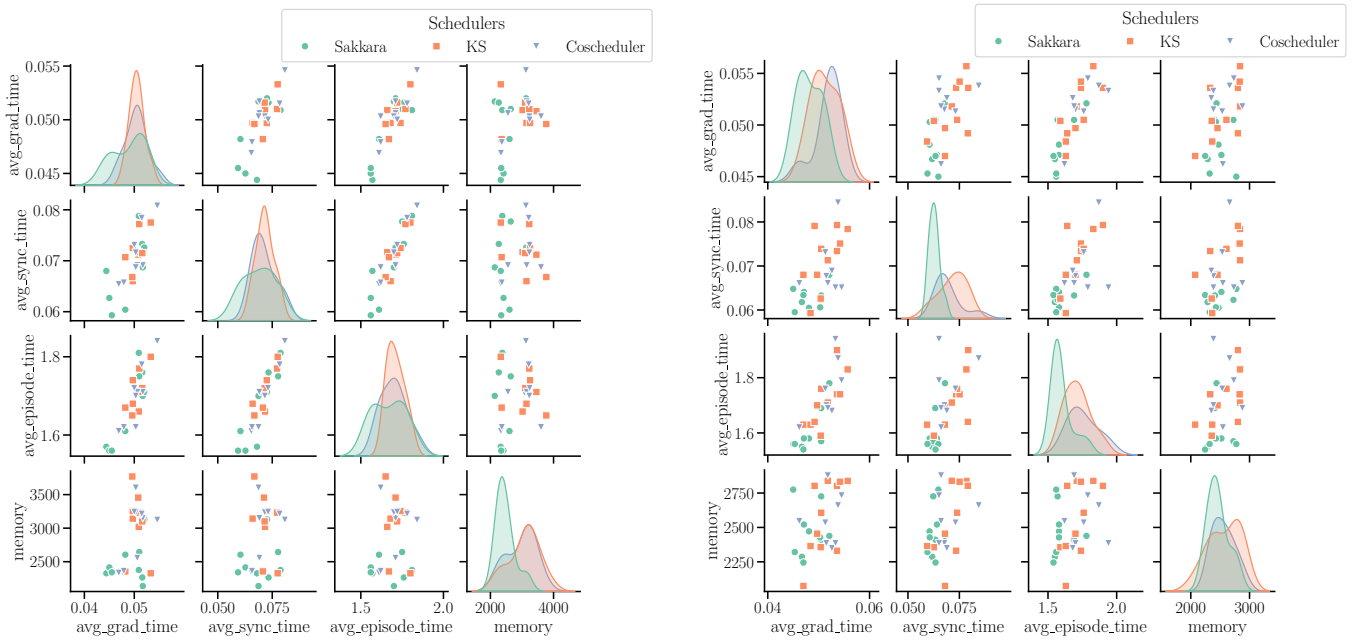


(a) Gradient Time. (b) Synchronization Time. (c) Episode Time.



(d) Memory Usage. (e) Throughput. (f) Total Time.

Fig. 10: Performance evaluation of the training workload for the high scenario. The advantages of topology-aware scheduling become more pronounced under higher network latency, with *Sakkara* achieving lower execution time and higher throughput.



(a) Moderate scenario.

(b) High scenario.

Fig. 11: The pairplot for both the moderate and high scenarios, illustrating the pairwise relationships between various metrics. These graphs highlight the reduced synchronization overhead and improved execution efficiency achieved by *Sakkara*, particularly for the high latency scenario. The color code of each scheduler is similar to Figures 9 and 10.

delays, which directly benefits one of the most time-critical components of distributed AI training workloads, as the reduction in synchronization time shows that such communication bottlenecks become increasingly dominant under higher network latency conditions and in communication-intensive distributed training workloads.

In both scenarios, *Sakkara* achieves the lowest average episode times, reducing execution latency compared to KS and *CoScheduler* (Figures 9c and 10c). Under the moderate scenario, *Sakkara* yields an average episode time of 1.67 ± 0.06s, slightly below KS (1.71 ± 0.03s) and *CoScheduler* (1.70 ± 0.05s). The gains are more pronounced under the high scenario, where Sakkara reduces average episode time to 1.60 ± 0.04s, while KS and *CoScheduler* exhibit slower execution at 1.72 ± 0.05s and 1.76 ± 0.07s, respectively. These improvements indicate that *Sakkara's* topology-aware policies reduce intra-job communication overhead, leading to faster task completion as network latencies increase. A similar trend emerges when examining total execution time (Figures 9f and 10f). In the moderate scenario, *Sakkara* reduces job completion time to 16.7 ± 0.6s, outperforming KS (17.1 ± 0.3s) and *CoScheduler* (17.0 ± 0.5s). In the high scenario, *Sakkara* completes jobs in 15.99 ± 0.44s, whereas KS and *CoScheduler* require 17.19 ± 0.5s and 17.55 ± 0.69s, respectively. These reductions are important in multi-tenant clusters, where shorter job durations translate to improved overall system responsiveness and higher effective utilization.

Interestingly, *Sakkara* also shows more efficient memory utilization (Figures 9d and 10d). In the moderate scenario, it requires 2471 ± 180 MB, compared to 3073 ± 290 MB for KS and 2979 ± 276 MB for *CoScheduler*. A similar trend holds in the high scenario, where *Sakkara* consumes 2462 ± 103 MB, compared to 2573 ± 152 MB for KS and 2559 ± 122 MB for *CoScheduler*. The lower memory footprint indicates that better workload placement reduces memory fragmentation, further contributing to efficient cluster resource utilization. Throughput aligns with episode time reductions, highlighting the benefits of topology-aware scheduling for sustained workload efficiency (Figures 9e and 10e). In the moderate scenario, *Sakkara* reaches 750.7 ± 26.6 images/s, outperforming KS (730.9 ± 13.1 images/s) and CoScheduler (736.3 ± 19.9 images/s). Under the high scenario, the difference becomes more significant: *Sakkara* achieves 783.0 ± 20.1 images/s, a 10% increase compared to KS (709.8 ± 31.3 images/s) and *CoScheduler* (713.0 ± 29.6 images/s). These results emphasize that topology awareness not only reduces latency but also enhances overall cluster throughput under resource contention. Evaluating the impact of topology-aware scheduling on GPU utilization under higher job concurrency and multi-tenant workloads is left for future work.

To further analyze the performance of the three schedulers, we visualized the relationships among the main metrics using pairplots for both the moderate and high scenarios (Figures 11a and 11b). These plots provide additional insights into the trade-offs across schedulers and highlight the factors driving performance differences. For the moderate scenario, the distributions for the three schedulers overlap in most metrics, yet *Sakkara* shows a tendency toward lower average synchronization times

and reduced memory consumption. These differences translate into modest reductions in average episode time and overall execution time, confirming that topology-awareness already provides benefits under moderate latency. In the high scenario, a much clearer separation among schedulers is shown. *Sakkara* consistently achieves lower average synchronization times (63ms) while KS and *CoScheduler* achieve above 70ms. Gradient computation times also follow this trend, with *Sakkara* reducing delays relative to both baselines. Lower synchronization and computation overheads lead directly to shorter average episode times for *Sakkara*, compared to KS and *CoScheduler*.

The observed performance improvements are primarily driven by *Sakkara*'s topology-aware placement strategy, which reduces cross-rack and cross-node communication during distributed execution. By co-locating communication-intensive tasks closer in the network hierarchy, *Sakkara* decreases synchronization overhead and improves resource locality, which directly translates into lower execution time and higher throughput.

B. Inference Workload

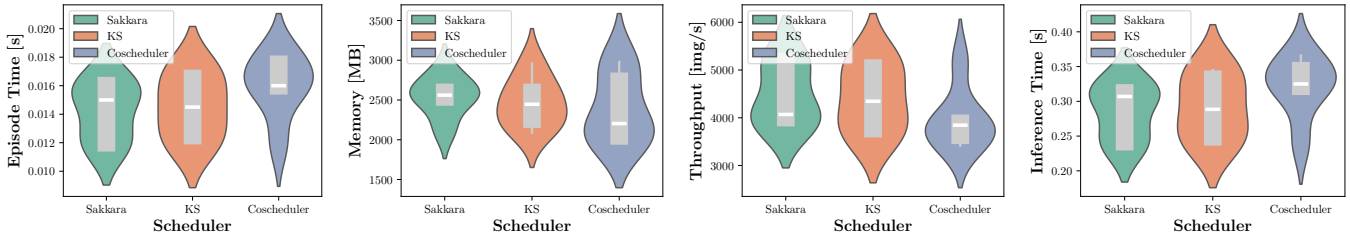
This section evaluates the performance of the inference workload for the considered schedulers for both moderate (Fig. 12) and high (Fig. 13) scenarios. In the moderate scenario, the three schedulers exhibit comparable performance. Both KS and *Sakkara* achieve nearly identical average episode times as shown in Fig. 12a (0.014 ± 0.002s), indicating stable and consistent execution, while *CoScheduler* incurs a slightly higher latency (0.016 ± 0.002s). Memory consumption (Fig. 12b) remains comparable across all schedulers, with *CoScheduler* exhibiting the lowest average usage (2383.2 ± 317.9 MB), followed by KS (2464.1 ± 229.3 MB) and *Sakkara* (2532.8 ± 199.5 MB). While these differences are modest, they suggest that *CoScheduler* trades reduced memory footprint for lower throughput and higher latency by making suboptimal placement decisions. Throughput results (Fig. 12c) highlight a more pronounced separation. *Sakkara* achieves the highest throughput (4468.4 ± 513.9), closely followed by KS (4394.1 ± 517.7), whereas *CoScheduler* trails behind at 3957.9 ± 459.7. This corresponds to an approximate 10% throughput advantage for KS and *Sakkara* over *CoScheduler*, indicating more effective resource utilization for this workload under the moderate scenario. Finally, total execution time (Fig. 12d) reinforces the observed trends. *Sakkara* achieves the shortest overall completion time (0.284 ± 0.03s), closely followed by KS (0.291 ± 0.03s), whereas *CoScheduler* requires noticeably more time to complete the workload (0.321 ± 0.03s). Overall, these results indicate that under moderate load, KS and *Sakkara* provide a better balance among latency, throughput, and completion time, whereas *CoScheduler* exhibits increased overhead that impacts performance despite slightly lower memory usage.

In the high scenario, the average episode time (Fig. 13a) remains low and comparable for all schedulers, with KS and *CoScheduler* achieving an average of approximately 0.02 ± 0.004s, while *Sakkara* slightly improves this metric to 0.01 ± 0.003s. In terms of memory usage (Fig. 13b), all schedulers exhibit comparable behavior, with *CoScheduler* showing the lowest average memory consumption at 2119.6 ± 142 MB, followed by *Sakkara* and KS. Although the differences are modest, they indicate that *Sakkara* and *CoScheduler* result in more memory-efficient operation of the inference workload compared to KS. More pronounced differences emerge when analyzing throughput (Fig. 13c). *Sakkara* achieves the highest throughput, reaching 4516.1 ± 494.7 images/s on average, outperforming KS and *CoScheduler* by a noticeable margin. *CoScheduler* follows with an average throughput of 3948.5 ± 281.5 images/s, while KS records the lowest throughput at 3779.2 ± 342.6 images/s. These results indicate that *Sakkara* is more effective at sustaining a higher inference processing rate under the evaluated workload. Finally, the total execution time (Fig. 13d) further reflects the throughput trends. *Sakkara* achieves the shortest inference time (0.28 ± 0.03s), followed by *CoScheduler* (0.32 ± 0.03s) and KS (0.34 ± 0.03s). This confirms that the higher throughput observed for *Sakkara* translates into faster overall inference execution. These improvements stem from more efficient workload placement, which reduces cross-node communication overhead and improves resource locality during request processing, resulting in more stable and efficient execution behavior.

Overall, these results suggest that even in inference workloads, where synchronization is less dominant than in training, topology-aware placement improves end-to-end efficiency by reducing communication overhead and sustaining higher processing rates. *Sakkara* consistently achieves better throughput and lower total execution time, while *CoScheduler* exhibits slightly lower memory usage.

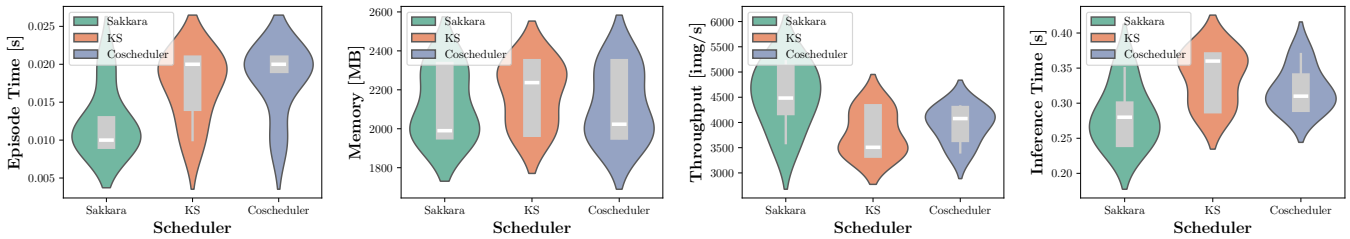
C. Summary

As discussed in our previous work [15], [17], the choice between *packing* and *spreading* strategies depends on both workload characteristics and cluster topology. *Packing* collectives within a single rack generally improves performance by minimizing cross-rack communication, reducing execution time, communication overhead, and network latency. In contrast, *spreading* workloads across multiple racks introduces additional communication overhead, but can be beneficial in certain scenarios. For instance, when a workload cannot fit within a single rack, spreading pods across racks balances traffic toward the network spine, reduces congestion, and improves execution times. Moreover, spreading workloads across racks can isolate applications, mitigating contention for shared network resources. These observations highlight the importance of tailoring placement strategies to the size of the workload and the underlying infrastructure to achieve optimal performance. Although *Sakkara* indirectly improves resilience through topology-aware spreading and packing across hierarchy levels (i.e., rack and node level), a controlled fault-tolerance evaluation, such as node and pod failures, on a



(a) Episode Time. (b) Memory Usage. (c) Throughput. (d) Total Time.

Fig. 12: Performance evaluation of the inference workload for the moderate scenario. *Sakkara* and KS achieve comparable low-latency inference, while *Sakkara* provides the highest throughput and shortest execution time overall.



(a) Episode Time. (b) Memory Usage. (c) Throughput. (d) Total Time.

Fig. 13: Performance evaluation of the inference workload for the high scenario. *Sakkara* sustains the highest inference throughput and lowest total execution time under increased network latency.

larger-scale K8s cluster with additional worker nodes and multiple GPUs per node, is required to assess fault tolerance comprehensively and is therefore planned as future work.

In conclusion, the advantages of *Sakkara* become more pronounced as network latency increases for both workloads. While KS and *CoScheduler* provide reasonable performance under moderate latency, they fail to adapt effectively when communication and synchronization overheads dominate in high latency scenarios. *Sakkara's* holistic consideration of both cluster topology and resource utilization enables lower synchronization overhead, shorter execution times, and higher throughput, all while maintaining a lower memory footprint. These results confirm that topology-aware scheduling is essential for scaling AI workloads efficiently in K8s environments, especially in multi-tenant or resource-constrained clusters.

While the evaluation focuses on representative training and inference workloads under moderate and high latency scenarios, the underlying topology-aware scheduling principles are general and can be applied to a wider range of distributed AI workloads and Kubernetes cluster configurations. Extending the evaluation to additional workload types, larger-scale clusters, and more heterogeneous GPU configurations is part of our future work.

VI. Conclusions

This paper has introduced *Sakkara*, a topology-aware scheduler plugin for K8s, designed to address the placement needs of tightly coupled workloads in hierarchical cluster environments. By representing the cluster topology as a tree and enabling rich placement policies—such as packing, spreading, partitioning, and range-based constraints—*Sakkara* provides a flexible and efficient framework for group-based scheduling. *Sakkara* is particularly suited for AI/ML workloads, where the performance and efficiency of applications depend on the relative placement of their components within the DC hierarchy. The proposed approach allows users to define both the physical topology and application-specific placement constraints using ConfigMaps, thus seamlessly integrating with K8s declarative configuration model. A notable feature of *Sakkara* is its support for gang scheduling with preemption, where entire groups of pods can be scheduled or evicted together, respecting job priorities. The use of an open-source solver employing the *Chic-sched* algorithm further enhances the scalability and practicality of the scheduler for complex placement problems. The results show that *Sakkara* outperforms existing K8s schedulers in terms of job completion time and throughput for both training and inference workloads, achieving improvements of up to 10%. Future work includes extending *Sakkara* to support dynamic re-optimization of placements in response to cluster state changes, incorporating additional constraints such as energy efficiency, and exploring tighter integration with the K8s scheduling community through its open-source release, to simplify and increase its adoption. *Sakkara* advances the state-of-the-art in topology-aware scheduling in K8s, offering a practical and extensible solution for modern AI-driven cloud workloads.

ACKNOWLEDGMENT

José Santos is funded by the Research Foundation Flanders (FWO), grant number 1253226N.

REFERENCES

[1] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2022.

[2] G. Sayfan, *Mastering Kubernetes: Master the art of container management by using the power of Kubernetes*. Packt Publishing Ltd, 2018.

[3] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.

[4] K. Senjab, S. Abbas, N. Ahmed, and A. u. R. Khan, "A survey of kubernetes scheduling algorithms," *Journal of Cloud Computing*, vol. 12, no. 1, p. 87, 2023.

[5] Y.-D. Lin, Y.-T. Ling, Y.-C. Lai, and D. Sudyana, "Reinforcement learning for ai as a service: Cpu-gpu task scheduling for preprocessing, training, and inference tasks," *IEEE Transactions on Network and Service Management*, 2025.

[6] L. Shen, Y. Sun, Z. Yu, L. Ding, X. Tian, and D. Tao, "On efficient training of large-scale deep learning models," *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–36, 2024.

[7] X. Zou, C. Chen, P. Lin, L. Zhang, Y. Xu, and W. Zhang, "Scalable heterogeneous scheduling based model parallelism for real-time inference of large-scale deep neural networks," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 8, no. 4, pp. 2962–2973, 2024.

[8] IBM Research, "Why we built an AI supercomputer in the cloud." accessed on 16 July 2025. [Online]. Available: https://research.ibm.com/blog/AI-supercomputer-Vela-GPU-cluster.

[9] A. Wittig and M. Wittig, *Amazon Web Services in Action: An in-depth guide to AWS*. Simon and Schuster, 2023.

[10] Microsoft, "Azure. Limitless Innovation. Code. Connect. Grow." accessed on 16 July 2025. [Online]. Available: https://www.ibm.com/cloud.

[11] IBM, "IBM Cloud: AI-ready, secure, and hybrid by design." accessed on 16 July 2025. [Online]. Available: https://www.ibm.com/cloud.

[12] L. Schares, A. Tantawi, P. Maniotis, M.-H. Chen, C. Misale, S. Seelam *et al.*, "Chic-sched: a HPC Placement-Group Scheduler on Hierarchical Topologies with Constraints," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 424–434.

[13] S. Lee, N. Guan, and J. Lee, "Design and timing guarantee for non-preemptive gang scheduling," in *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 132–144.

[14] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 539–558.

[15] J. Santos, P. Maniotis, C. Wang, A. Tantawi, O. Tardieu, T. Wauters *et al.*, "Evaluating the network effects of orchestration strategies for ai collective operations in modern data center networks," *2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*, pp. 1–9, 2025.

[16] A. Varga, "Discrete event simulation system," in *Proc. of the European Simulation Multiconference (ESM'2001)*, vol. 17, 2001.

[17] J. Santos, C. Wang, A. Tantawi, O. Tardieu, P. Maniotis, T. Wauters *et al.*, "Toward efficient orchestration of genai workloads in modern container clouds," *IEEE Communications Magazine*, 2026.

[18] kubernetes, "kube-scheduler." accessed on 16 July 2025. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/.

[19] kubernetes sig-scheduling, "This folder holds the coscheduling plugin implementations based on Coscheduling based on PodGroup CRD." accessed on 16 July 2025. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/coscheduling.

[20] Volcano, "The gang plugin," accessed on 16 July 2025. [Online]. Available: https://github.com/volcano-sh/volcano/tree/master/pkg/scheduler/plugins/gang.

[21] ——, "Task topology plugin," accessed on 16 July 2025. [Online]. Available: https://github.com/volcano-sh/volcano/tree/master/pkg/scheduler/plugins/task-topology.

[22] ——, "The binpack plugin," accessed on 16 July 2025. [Online]. Available: https://github.com/volcano-sh/volcano/tree/master/pkg/scheduler/plugins/binpack.

[23] kubernetes sig-scheduling, "This folder holds the Topology-aware scheduler plugin implementations based on Topology aware scheduler plugin based on NodeResourceTopology CRD." accessed on 16 July 2025. [Online]. Available: https://scheduler-plugins.sigs.k8s.io/docs/plugins/noderesourcetopology/.

[24] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-aware scheduling in container-based clouds," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4461–4477, 2023.

[25] Kubernetes, "Scheduling Framework." accessed on 16 July 2025. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

[26] Volcano, "Volcano, a batch system built on kubernetes," accessed on 16 July 2025. [Online]. Available: https://github.com/volcano-sh/volcano.

[27] ——, "Dominant resource fairness (drf) plugin," accessed on 16 July 2025. [Online]. Available: https://github.com/volcano-sh/volcano/tree/master/pkg/scheduler/plugins/drf.

[28] Kubernetes, "Scheduler plugins, repository for out-of-tree scheduler plugins based on the scheduler framework." accessed on 16 July 2025. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins.

[29] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3:{Energy-Efficient} microservices on {SmartNIC-Accelerated} servers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 363–378.

[30] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 481–498.

[31] S. Caculo, K. Lahiri, and S. Kalambur, "Characterizing the scale-up performance of microservices using teastore," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 48–59.

[32] V. Rao, V. Singh, K. Goutham, B. U. Kempaiah, R. J. Mampilli, S. Kalambur *et al.*, "Scheduling microservice containers on large core machines through placement and coalescing," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2021, pp. 80–100.

[33] T. Gershon, S. Seelam, B. Belgodere, M. Bonilla, L. Hoang, D. Barnett *et al.*, "The infrastructure powering ibm's gen ai model development," *arXiv preprint arXiv:2407.05467*, 2024.

[34] kubernetes sig-scheduling, "Trimaran: Load-aware scheduling plugins." accessed on 16 July 2025. [Online]. Available: https://scheduler-plugins.sigs.k8s.io/docs/plugins/trimaran/.

[35] Virtual Wall, "The virtual wall emulation environment." accessed on 16 July 2025. [Online]. Available: https://doc.ilabt.imec.be/ilabt/virtualwall/index.html.

[36] J. Santos, B. V. Ninan, B. Volckaert, F. De Turck, and M. Al-Naday, "A comprehensive benchmark of flannel cni in sdn/non-sdn enabled cloud-native environments," *IEEE Transactions on Network and Service Management*, 2025.

[37] B. Ekane, A. Tchana, D. Hagimont, B. Teabe, and N. De Palma, "Networking in next generation disaggregated datacenters," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 21, p. e7702, 2023.

[38] A. N. Kuznetsov, "tc(8) — linux manual page." accessed on 16 July 2025. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html.

[39] K. Dong, C. Zhou, Y. Ruan, and Y. Li, "Mobilenetv2 model for image classification," in *2020 2nd International Conference on Information Technology and Computer Application (ITCA)*. IEEE, 2020, pp. 476–480.