Performance Impact of Queue Sorting in Container-based Application Scheduling

José Santos*†, Miel Verkerken*†, Laurens D'hooge*, Tim Wauters*, Bruno Volckaert*, Filip De Turck*
* IDLab, Department of Information Technology at Ghent University - imec, 9000 Ghent, Belgium
Email: {josepedro.pereiradossantos, miel.verkerken, laurens.dhooge, tim.wauters; bruno.volckaert, filip.deturck}@UGent.be

***Abstract*—Containerization has revolutionized application deployments in current cloud platforms, enabling the flexible instantiation of loosely-coupled microservices and enhancing operational efficacy. However, optimizing the performance of container-based applications remains a challenge and a major topic in cloud research. This paper studies the impact of queue sorting in application scheduling, focused on complex inter-dependencies among microservices. Queue sorting determines the deployment order of containers in the infrastructure, typically based on container priorities and resource requests. Optimizing these algorithms directly influences scheduling efficiency and overall application performance. This paper compares several schedulers and sorting algorithms, leveraging extensive benchmark tests conducted on the widely-used Kubernetes (K8s) platform. The evaluation includes a novel sorting algorithm named *TopologicalSort*, designed to prioritize containers for application scheduling focused on microservice inter-dependencies. Results show the significant impact of queue sorting on application performance, with *TopologicalSort* algorithms outperforming default mechanisms, yielding an average increase of 20% in throughput and reducing response time by at least 15%. These results highlight the importance of considering microservice inter-dependencies for effective application deployment in modern container-based environments.**

***Index Terms*—Cloud, Orchestration, Containers, Kubernetes, Performance, Queue Sorting**

I. Introduction

In recent years, containerization technology has revolutionized how modern applications are deployed and managed [1], [2]. Microservice-based architectures have gradually become the widely accepted standard for application deployment in today's cloud platforms such as Amazon ECS [3], Kubernetes (K8s) [4], and Red Hat OpenShift [5]. Traditional monoliths have been decomposed into multiple loosely-coupled microservices, developed and deployed independently. This paradigm shift improves deployment flexibility and scalability, service portability, and operational efficiency, making containers the most popular choice for cloud-based infrastructures [6].

An essential part of the life-cycle management of containerized applications involves their scheduling on the infrastructure, considering various factors such as resource availability (e.g., CPU, memory, and storage) and priority levels. Scheduling systems determine a suitable node to host each container, ensuring efficient execution to meet multiple service-level objectives (e.g., latency requirements). However, efficiently orchestrating microservice-based architectures in cloud platforms poses significant challenges due to microservice inter-dependencies. Containers communicate and rely on one another, forming complex dependencies essential for the application's proper operation. The existing literature often overlooks these dependencies, resulting in suboptimal container deployments and subsequent performance degradation.

This paper studies the impact of queue sorting in application scheduling, which relates to the process of determining the order in which containers are allocated in the infrastructure. These algorithms typically optimize resource allocation and reduce the overall execution time for scheduling, focused on container priorities and resource requests. Only a few works address container co-location or microservice inter-dependencies [7]–[9]. This study investigates the impact of different schedulers and sorting algorithms on application deployment by performing extensive benchmark tests in different infrastructure topologies with multiple microservice benchmark applications in the popular K8s platform.

Recently, in our previous work, a network-aware framework named Diktyo [10], [11] has been proposed for the K8s platform to enable network-aware placement of containerized applications. A novel sorting algorithm named *TopologicalSort* has been presented as part of the Diktyo framework, developed to sort containers for application scheduling focused on microservice inter-dependencies. This paper aims to provide insights regarding the performance of multiple sorting algorithms in container scheduling and to identify the most efficient algorithms for different workloads. Results show that the applied sorting algorithm significantly impacts application performance, with Diktyo *TopologicalSort* outperforming default sorting algorithms regarding the application's response time and throughput for several evaluated applications (Sec. V). This study helps application developers and cloud providers to select appropriate sorting algorithms based on the microservice inter-dependencies of their applications. The main contributions of the paper are twofold:

- **Sorting approach for microservice inter-dependencies**: Further insights are provided on why queue sorting and microservice inter-dependencies play a major role in application performance focused on the recent *TopologicalSort* plugin (Sec III).
- **Evaluation with microservice-based applications**: The evaluation considers real-world microservice applications

† José Santos and Miel Verkerken contributed equally to this work.

978-3-903176-59-1 ©2023 IFIP

TABLE I: Comparison of existing works related to application scheduling.

Authors	Year	Virtualization	Dimension	Main Focus	Microservice Dependencies	Queue Sorting	Evaluation Method
Huang, K. C., et al. [15]	2015	VMs	P	P & R	✕	✓	S
K8s *PrioritySort* [16]	2020	C	P	P	✕	✓	K8s
Tang, B., et al. [17]	2022	C	P & D	P & R	✕	✓	S
Hu, Y., et al. [18]	2017	C	D	D & QoS & T	✕	✓	YARN
Narayanan, D., et al. [19]	2020	VMs	D	T & M	✕	✓	S
Gao, Y., et al. [20]	2022	C	D	D & R	✕	✓	YARN
Chung, A., et al. [21]	2018	VMs	C	R	✕	✓	S
Liu, B., et al. [22]	2018	C	C	R & NB	✕	✕	D
Ranjan, R., et al. [23]	2020	VMs & C	C	E & M	✓	✓	S
K8s *QoSSort* [24]	2020	C	C	QoS & R	✕	✓	K8s
Venkataraman, S., et al. [7]	2014	VMs	L	AP & L	✓	✕	S + CT
Zhao, D., et al. [8]	2018	C	L	L & NB & R	✓	✓	CT
Blöcher, M., et al. [9]	2021	C	L & C & N	L & NB & R	✓	✓	S
Larumbe, F., et al. [25]	2017	VMs	N & T	NB & Top	✕	✕	S
Rodrigues, L., et al. [26]	2019	C	N	NB & QoS & E	✕	✓	S
Santos, J., et al. [27]	2019	C	N & T	NB & NL	✕	✕	K8s
Ryu, B, et al. [28]	2020	VMs	T	Top & NB	✕	✓	S
Muhammad, A., et al. [29]	2021	VMs	T & N & C	Top & NB & R	✕	✕	CT
Volcano's *TaskTopology* [30]	2021	C	T & C	T & R	✓	✓	K8s
Diktyo's *TopologicalSort* [31]	2022	C	T	AP	✓	✓	K8s

Virtualization: VMs = Virtual Machines, C = Containers.
Dimension: P = Priority-aware, D = Deadline-aware, C = Cost-aware, L = Location-aware, N = Network-aware, T = Topology-aware.
Main Focus: P = Priorities, R = Resources, AP = App. Dependencies, D = Deadlines, QoS = Quality of Service, T = Throughput, Top = Topology-aware, NB = Network Bandwidth, NL = Network Latency, M = Makespan, E = Energy Consumption, L = Location-aware.
Microservice Dependencies, Queue Sorting: ✓= addressed, ✕= not considered.
Evaluation Method: K8s = Kubernetes, S = Simulation, D = Docker, YARN = Apache YARN, CT = Custom-made Testbed.

typically used for benchmarking: a Machine Learning (ML)-based Multi-Stage Intrusion Detection System (IDS) [12] capable of real-time cyberattack detection, a test application named TeaStore (TS) [13], and a multi-tier web application named Online Boutique (OB) [14]. Experiments in different K8s clusters show that Diktyo sorting can increase throughput on average by 20% and reduce the application's response time by 15% (Sec. V).

The remainder of the paper is organized as follows: the state-of-the-art on queue sorting is discussed in the next section. Sec. III highlights the impact of microservice inter-dependencies in application deployment, describing the Diktyo approach focused on topological sorting. Sec. IV describes the evaluation setup, followed by the results in Sec. V. Sec. VI concludes this paper.

II. Related Work

Container orchestration has been an active research topic in recent years. Numerous studies have proposed various scheduling policies to optimize container allocation in popular cloud platforms. This section provides a review of the most relevant works on application scheduling, mainly focusing on approaches that investigate efficient queue-sorting algorithms.

Priority-aware scheduling has been extensively studied in the last few years [15]–[17]. These algorithms typically schedule workloads based on their priority or resource requests. In [17], the authors aim to meet the deadline constraints of high-priority tasks. However, practical implementations of these methods are missing since most scheduling algorithms are evaluated only by simulations. **Deadline-aware** methods [18]–[20] focus mainly on meeting the deployment time restrictions of applications while scheduling them in the infrastructure. Apache YARN [32] has been vastly applied as an evaluation platform, especially for works related to batch job scheduling. In addition, several proposals focus on theoretical modeling, such as Integer Linear Programming (ILP) models, to find the optimal scheduling based on a particular objective. The main drawback of these modeling approaches is that they cannot find a feasible solution within an acceptable time, thus limiting their applicability in operational environments. Nonetheless, the modeling can serve as an optimal benchmark for heuristic-based algorithms.

Cost-aware algorithms [21]–[24] have been widely studied in recent years. These works focus mainly on optimizing resources, such as CPU and memory usage, based on the workloads that need scheduling. These proposals relate to task assignment or job scheduling problems rather than deploying long-running applications with several microservices as typical K8s workloads. **Location-aware** scheduling has been investigated recently [7]–[9]. These works optimize application performance focused on resource dependencies [9] or data locality [8]. The performance of data-intensive applications such as Apache Hadoop or Spark jobs depends on the data awareness of the scheduler since jobs share data and communicate with each other [33]. Thus, dependent microservices are typically co-located in the same compute node to improve performance. However, practical implementations of these methods are still scarce since most algorithms are mainly evaluated via simulations or custom-made testbeds.

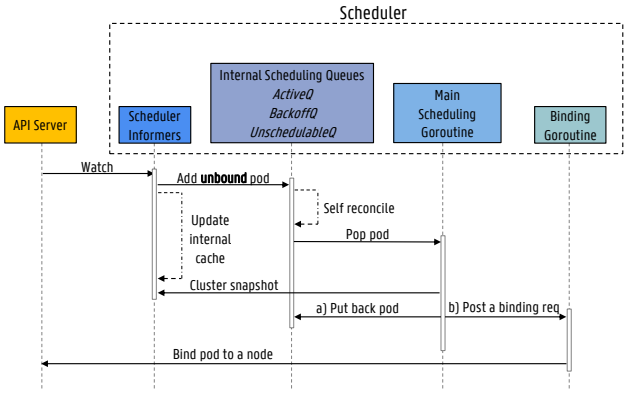


Fig. 1: The Kubernetes (K8s) scheduling workflow [34].

Network-aware scheduling [25]–[27] has recently gained growing consideration with the goal of optimizing network bandwidth to reduce traffic congestion and improve application performance. Despite this attention, practical implementations of these methods are lacking since most network-aware algorithms are evaluated only through simulations. Existing works focus mainly on Virtual Machine (VM) placement, and only a few studies address container allocation [26], [27]. **Topological-aware** scheduling has also been addressed lately [28]–[31]. The focus of these efforts is to optimize network bandwidth [28] and resources [29] by placing VMs or microservices according to cluster topology information or application characteristics. The authors of these studies strive to improve performance by prioritizing these factors. Also, the *Volcano project* provides several plugins for K8s focused on task scheduling. In particular, the *TaskTopology* plugin [30] groups containers into buckets based on task affinities to minimize data transmissions between dependent tasks. The proposed *TopologicalSort* plugin [31] is designed to incorporate application dependencies and determine precise sorting strategies for long-running applications with numerous microservices as typical K8s workloads.

To summarize, Table I compares all works referenced above, including the proposed *TopologicalSort* algorithm, which is integrated into the Diktyo framework. *TopologicalSort* goes beyond the current literature since it considers the microservice dependencies of the applications to determine the optimal order for deploying complex microservice-based applications in K8s clusters. Prior works mainly concentrate on theoretical models and heuristics evaluated via simulations or small testbeds, which limits their practical applicability in large-scale production clusters.

III. Toward Improved Queue Sorting in Kubernetes

A. The Kubernetes (K8s) scheduling workflow

Microservices in K8s are commonly deployed via a *pod*, which is the smallest working unit in K8s capable of hosting one or more containers running within the same execution environment [4]. Moreover, a *K8s deployment* can be employed to efficiently manage multiple instances of pods running in the cluster. Therefore, an entire microservice-based application is composed of several K8s deployments. The deployment of pods is typically determined by specific deployment requirements and the available resources of the K8s cluster (Fig. 1). The component responsible for scheduling operations is named Kube-Scheduler (KS), the default scheduler in K8s. All pods submitted for scheduling are inserted into a queue (i.e., *activeQ*) through an informer. The scheduling queue consists of three main queues: *activeQ*, *backoffQ*, and *unschedulableQ*. Pods awaiting scheduling in *activeQ* are processed individually, meaning a single pod is fetched for scheduling based on the applied queue sort plugin. Once a pod is selected from *activeQ*, it is removed and passed into the scheduler pipeline, along with the relevant information of the cluster node obtained from the internal cache. In the event of a scheduling failure, the pod is placed in either *unschedulableQ* or *backoffQ*, depending on why the operation failed. Typically, the pod is placed into the *backoffQ* if the node and pod caches are modified during the pod's scheduling. The wait and binding processes occur asynchronously and in parallel. The wait process ensures that pod-associated resources are ready, such as the successful creation of volumes. Meanwhile, the binding routine persistently stores the associations between pods and nodes in K8s.

To facilitate the development of new scheduling algorithms, K8s released a scheduling framework [35] that enables developers to implement their own methods and contribute to the K8s project. Thus, developers can create plugins that integrate with the existing scheduling components without interfering with the main scheduling components. The framework provides various extension points, serving as entry points for custom algorithm implementation. These extension points are primarily responsible for the following functionalities:

- **QueueSort**: sort pods in the scheduling queue.
- **PreFilter**: pre-process information about the pod.
- **Filter**: filter out nodes that cannot run the pod.
- **Score**: rank nodes that have passed the filtering phase.
- **NormalizeScore**: modify scores before final ranking.

K8s is currently the *de facto* standard for deploying applications in the cloud, widely used by most companies, and currently lacks context and network awareness in application scheduling. This challenge was tackled by developing the Diktyo framework that applies several extension points to handle microservice dependencies and topology awareness in the K8s scheduling process, which will significantly impact most industries. Instead of exploring novel designs that can take years to influence the current systems, we focus on solving this issue by designing missing components based on readily available features in K8s since network and topology awareness are urgent needs [36]. The proposed *TopologicalSort* plugin is detailed next, where pods in the scheduling queue are sorted based on microservice inter-dependencies aiming to improve the performance of container-based applications in K8s clusters.

B. The influence of microservice inter-dependencies

Network latency is a primary concern when deploying multi-tier applications since it affects the overall application performance [37]. These applications typically include tens to hundreds of microservices with complex inter-dependencies. Current allocation strategies do not schedule dependent microservices with latency awareness, possibly resulting in large distances between compute nodes hosting dependent microservices. In addition, **network bandwidth** plays a key role, especially for applications with high volumes of data transfers among microservices. Spark jobs have regular data transfers between mappers and reducers, and if the available network bandwidth is insufficient to handle all these transfers, then the performance degrades. These applications can benefit from topology-aware scheduling strategies that determine the service placement based on application dependencies and their specific requirements in terms of latency and bandwidth. The network latency and bandwidth may vary according to the compute nodes hosting dependent microservices within the underlying cloud infrastructure.

The Diktyo framework proposes two Custom Resources (CRs) (i.e., **AppGroup** and **NetworkTopology**) to consider both the application dependencies and the cluster network topology when scheduling pods in K8s. Diktyo provides network-aware algorithms implemented as three scheduling plugins based on the K8s scheduler framework [34]: **TopologicalSort**, **NodeNetworkCostFit** and **NetworkMinCost**. This paper focuses on the impact of queue sorting in application performance and how Diktyo optimizes performance by considering application dependencies to sort pods needing allocation. Further details about the Diktyo framework are available in [10], [11], [38] since the framework has been accepted in the open-source repository of the K8s scheduling community as an alternative scheduler for K8s workloads.

C. Diktyo's Topological Sorting

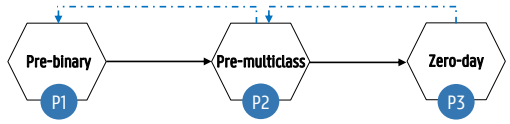
To determine Diktyo's optimal allocation order, developers need to specify all pod dependencies, indicating which pods communicate with each other. An application might consist of several interdependent pods, ranging from two to hundreds. The preference is to schedule pods with tighter constraints first, meaning those with a higher number of dependencies to avoid potential blocking and starvation issues. However, it is not a straightforward task to determine which pod has tighter constraints. Diktyo addresses this challenge by using six heuristic topological sorting algorithms [39] to calculate the preferred scheduling order of an application, based on the specified dependencies in the AppGroup CR. These algorithms include *Kahn*, *Tarjan*, *AlternateKahn*, *AlternateTarjan*, *ReverseKahn*, and *ReverseTarjan*.

Alternate Kahn modifies the order given by Kahn by selecting the first element of Kahn as its first element, the last of Kahn as its second, the second of Kahn as its third, and so on. AlternateTarjan follows the same pattern as AlternateKahn but modifies the order of Tarjan. ReverseKahn and ReverseTarjan essentially reverse the preferred order given by Kahn and Tarjan, respectively. These algorithms have been developed for Diktyo since it is unclear which algorithm might be preferred for a particular use case. While deploying pods with tighter constraints might be preferred in most cases, it might be undesirable for specific applications. The alternate sorting algorithms allow the sequential deployment of tighter pods and more flexible pods alternately, which can be beneficial when tighter pods need allocation and dependent pods are not yet deployed in the system. Table II presents the determined order for all sorting algorithms based on the microservice-based applications shown in Fig 2. Also, a default algorithm has been developed for Diktyo to handle cyclic applications since the aforementioned algorithms do not support cyclic dependencies. The aim is to specify bidirectional dependencies even if there is only one-way communication because certain pods can be placed close to pods that communicate with them. Depending on the selected sorting algorithm, significant differences are observed in the deployment order. The K8s *PrioritySort* and *QoSSort* are also shown to highlight the differences between the proposed *TopologicalSort* and available sorting algorithms. However, both provide an identical order in our evaluation, as all pods have the same priority level, and resource limitations are higher than resource requests (Sec. IV - Table III).

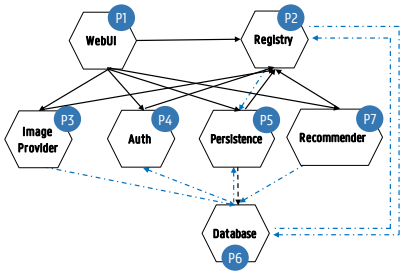
TABLE II: Topological sorting for the different applications.

Multi-Stage IDS application (3-deployments)	
Algorithm	**Topological order**
Kahn & Tarjan	[P1, P2, P3]
Alt. Kahn & Alt. Tarj.	[P1, P3, P2]
Rev. Kahn & Rev. Tarj.	[P3, P2, P1]
Default (Cycle)	[P1, P2, P3]
K8s *Priority & QoS*	[P1, P2, P3]
TeaStore (TS) application (7-deployments)	
Kahn	[P1, P4, P3, P5, P7, P2, P6]
Alt. Kahn	[P1, P6, P4, P2, P3, P7, P5]
Rev. Kahn	[P6, P2, P7, P5, P3, P4, P1]
Tarjan	[P1, P4, P3, P5, P7, P6, P2]
Alt. Tarj.	[P1, P2, P4, P7, P3, P6, P5]
Rev. Tarj.	[P2, P6, P7, P5, P3, P4, P1]
Default (Cycle)	[P6, P2, P5, P4, P3, P7, P1]
K8s *Priority & QoS*	[P6, P2, P5, P4, P3, P7, P1]
Online Boutique (OB) application (11-deployments)	
Kahn	[P1, P10, P9, P8, P7, P6, P5, P4, P3, P2, P11]
Alt. Kahn	[P1, P11, P10, P2, P9, P3, P8, P4, P7, P5, P6]
Rev. Kahn	[P11, P2, P3, P4, P5, P6, P7, P8, P9, P10, P1]
Tarjan	[P1, P8, P7, P5, P4, P2, P11, P9, P10, P6, P3]
Alt. Tarj.	[P1, P3, P8, P6, P7, P10, P5, P9, P4, P11, P2]
Rev. Tarj.	[P3, P6, P10, P9, P11, P2, P4, P5, P7, P8, P1]
Default (Cycle)	[P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11]
K8s *Priority & QoS*	[P10, P2, P8, P4, P7, P1, P5, P3, P9, P11, P6]

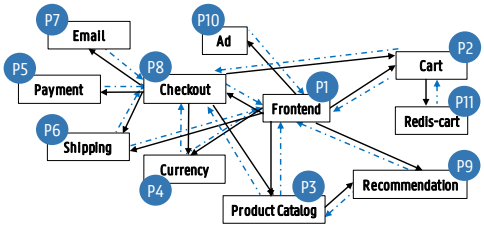
The *TopologicalSort* plugin reorders pods in the scheduling queue based on the calculated order. Each microservice receives an index based on the preferred allocation order, and *TopologicalSort* prioritizes pods with lower indexes. For example, consider the TS application and the correspondent order based on the *KahnSort* algorithm shown previously in Table II. Depending on the selected two pods from TS by the plugin, the result of *TopologicalSort* can be significantly different while favoring low indexes (Fig. 3).



(a) Multi-Stage IDS application [12].



(b) TeaStore (TS) application [40].



(c) Online Boutique (OB) application [14].

Fig. 2: Illustration of microservice inter-dependencies in typical container-based applications. Blue lines represent additional dependencies included in the cycle version.

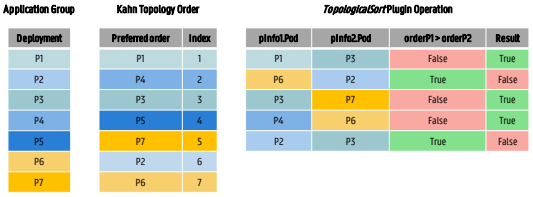


Fig. 3: Example of the *TopologicalSort* plugin operation [11].

IV. Evaluation Setup

This section presents an overview of the implemented testbed used to evaluate the performance of the container-based applications. Sec. IV-A shows deployment requirements for the respective applications, and Sec. IV-B presents the K8s cluster topologies.

A. Container-based Applications

Table III shows the deployment requirements for the three evaluated applications. The first application (Fig. 2a) is a **Multi-Stage IDS** [12] designed to detect both known and unknown cyberattacks in real-time utilizing ML models. The multi-stage detection process is split into three containerized K8s deployments. Identified anomalies are forwarded to a multi-class classifier to determine the attack class. Otherwise, attacks are classified as unknown. The second application (Fig. 2b) is **TS** [13], a microservice benchmark framework. TS consists of seven K8s workloads with distinct performance characteristics allowing the evaluation of scheduling and auto-scaling techniques. Lastly, the third application (Fig. 2c) is the **OB** application [14], an e-commerce application consisting of eleven K8s deployments. It is a web-based marketplace where users can browse and purchase items. Recent studies have utilized OB to demonstrate novel advancements in the microservice research domain.

TABLE III: Deployment properties of the evaluated container-based applications.

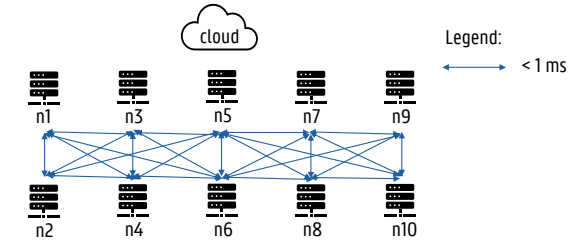
Application	Deployment	CPU R/L (in m)	MEM R/L (in Mi)
Multi-Stage IDS	pre-binary (P1)	500/1000	256/512
	pre-multiclass (P2)	400/800	192/384
	zero-day (P3)	100/200	64/128
TeaStore (TS)	webui (P1)	500/1000	512/1024
	registry (P2)	150/500	384/768
	image (P3)	500/1000	512/1024
	auth (P4)	500/1000	512/1024
	persistence (P5)	500/1000	512/1024
	db (P6)	150/500	128/256
	recommender (P7)	150/500	384/768
Online Boutique (OB)	frontend (P1)	100/200	64/128
	cart (P2)	200/300	180/300
	product (P3)	100/200	64/128
	currency (P4)	100/200	64/128
	payment (P5)	100/200	64/128
	shipping (P6)	100/200	64/128
	email (P7)	100/200	64/128
	checkout (P8)	100/200	64/128
	recommend. (P9)	100/200	64/128
	ad (P10)	200/300	180/300
	redis-cart (P11)	70/125	200/256

TABLE IV: Expected latency (in ms) in Edge-cloud.

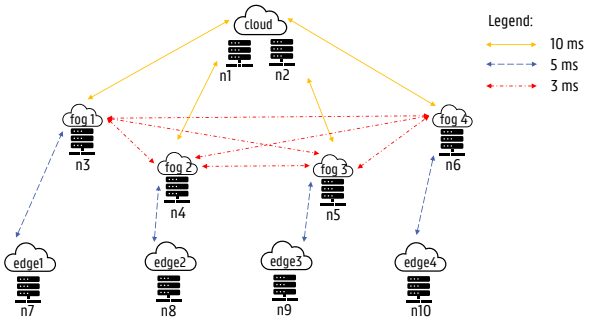
	n1	n2	n3	n4	n5	n6	n7	n8	n9	n10
n1	–	–	10	10	10	10	15	15	15	15
n2	–	–	10	10	10	10	15	15	15	15
n3	10	10	–	3	3	3	5	8	8	8
n4	10	10	3	–	3	3	8	5	8	8
n5	10	10	3	3	–	3	8	8	5	8
n6	10	10	3	3	3	–	8	8	8	5
n7	15	15	5	8	8	8	–	13	13	13
n8	15	15	8	5	8	8	13	–	13	13
n9	15	15	8	8	5	8	13	13	–	13
n10	15	15	8	8	8	5	13	13	13	–

B. Cloud Infrastructure

Fig. 4 shows the two evaluated infrastructure topologies. Firstly, Fig. 4a represents a highly available cluster where nodes are deployed across a single region. These nodes have similar network connections, resulting in negligible delays in the order of microseconds. Secondly, Fig. 4b shows a multi-region cluster with different network connections. Table IV shows the expected network delays for this topology. Both topologies have been set up by using the imec Virtual Wall (VWall) infrastructure [41] at IDLab, Belgium. Network delays are emulated using TC [42]. Table V lists the software versions applied to set up the K8s cluster.



(a) Cluster topology with similar network connections.



(b) Edge-cloud continuum with different network connections.

Fig. 4: Illustration of the evaluated infrastructures.

TABLE V: Software Versions of the Testbed.

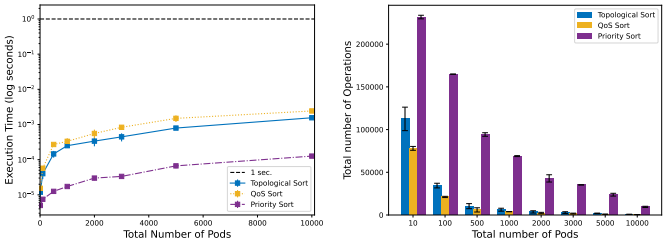
Software	Version
Kubeadm & Kubectl	v1.22.4
Operating System	Ubuntu 20.04.2 LTS
Docker	docker://20.10.12
Linux Kernel	5.4.0-67-generic

TABLE VI: The testbed methodology: evaluated topologies, applications, schedulers, and sorting algorithms.

Methodology	Measurements
Infrastructure	**2x types**: Cluster, Edge-cloud
Application	**3x types**: Multi-Stage IDS, TS, OB
Scheduler	**2x types**: KS, Diktyo
Sorting Alg.	**9x types**: Priority, Cycle, Kahn, ... , Tarjan.
Scenarios	**3x types**: *Initial*, *ScaleUp*, *ScaleDown*

C. Testbed Methodology

An extensive set of experiments evaluates the performance of the three applications when deployed with different schedulers and with various sorting plugins, as outlined in Table VI. Each experiment runs for three scenarios: the *Initial* phase, where one instance of each workload was deployed; the *ScaleUp* scenario, where all workloads were scaled up to five replicas; and the *ScaleDown* phase, where three instances per workload were terminated. To assess the performance in terms of response time and throughput, a load generator based on the locust load tool [43] was utilized. Emulated users generated a mix of *GET* and *POST* requests to simulate realistic workload conditions. Sec. V shows results with a 95% confidence interval, with each configuration running at least ten times to ensure statistical significance.



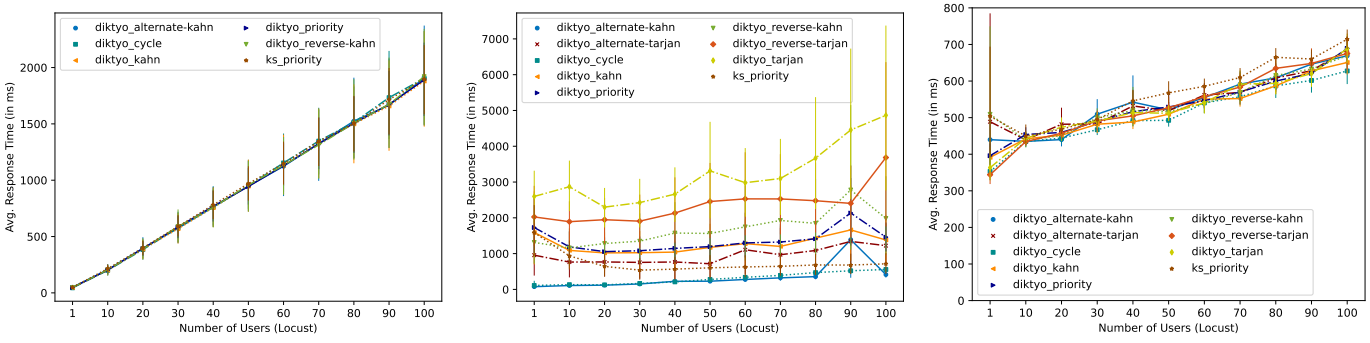
(a) Execution Time. (b) Number of Operations.

Fig. 5: Benchmark of the QueueSort plugins for the OB application. The execution time increases logarithmically over the number of pods.

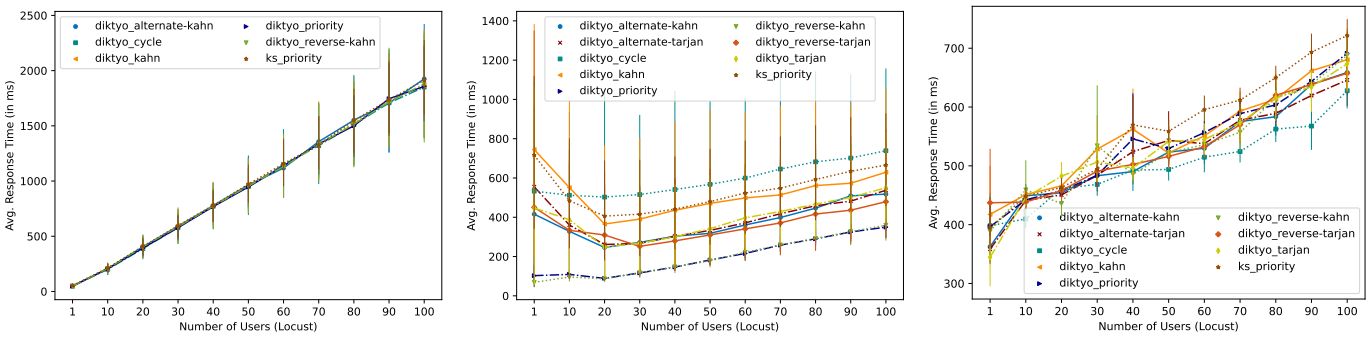
V. Results

Time Complexity has been accessed via the Go testing package that provides an integration testing utility that can benchmark the performance of the queue sorting plugins. Implemented integration tests evaluated the execution time and scalability of all plugins: *PrioritySort*, *QoSSort*, and *TopologicalSort*. Fig. 5 shows the plugins' execution time based on the number of pods in the scheduling waiting queue for the OB application. Each test must run the code N times. During its execution, N is adjusted until the benchmark function lasts long enough to be timed reliably. The following output: 36378 - 31.52 ns/op means that the operation (i.e., plugin algorithm) executed 36378 times at a speed of 31.52 ns per operation. The execution time of all plugins increases logarithmically with the number of nodes while remaining below 1 second for 10000 nodes. The *PrioritySort* plugin is significantly faster than both *QoSSort* and *TopologicalSort*, while *TopologicalSort* is slightly more scalable than *QoSSort*. *TopologicalSort* can handle 10000 pods in about 1.57 ms while *QoSSort* and *PrioritySort* need 2.5 ms and 0.13 ms, respectively. These results highlight that the proposed *TopologicalSort* algorithm introduces minimal overhead in terms of execution time to the K8s scheduling process. Furthermore, it demonstrates high scalability even with a substantial number of pods (10,000).

Latency results concerning the application's response time in Fig. 6 demonstrate that the higher the number of K8s deployments, the higher the impact queue sorting has on application performance. Relatively small differences are observed for the Multi-Stage IDS application, while discrepancies are more noticeable for the TS and OB applications. Large fluctuations exist between the evaluated schedulers and sorting plugins. Also, the stable edge-cloud continuum topology created with TC offers slightly higher performance than the cluster topology, especially for the TS application. Although delays are smaller in the cluster topology, these tend to fluctuate, resulting in greater instability and unpredictability in the response time of the TS application. Regarding the OB application, Diktyo cycle achieves slightly higher performance compared to the other schedulers (Fig. 7). On average, Diktyo cycle can reduce the expected response time by at least 15% compared to KS for the *Initial* and *ScaleDown* scenarios, achieving higher reductions for a greater number of users.

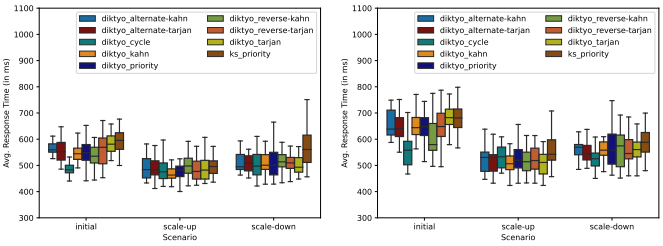


(a) Cluster - Multi-Stage IDS. (b) Cluster - TeaStore (TS). (c) Cluster - Online Boutique (OB).



(d) Edge-Cloud - Multi-Stage IDS. (e) Edge-Cloud - TeaStore (TS). (f) Edge-Cloud - Online Boutique (OB).

Fig. 6: The average response time (in ms) obtained during the evaluation. The higher the number of K8s deployments, higher influence of sorting on application performance.



(a) OB for 50 users. (b) OB for 70 users.

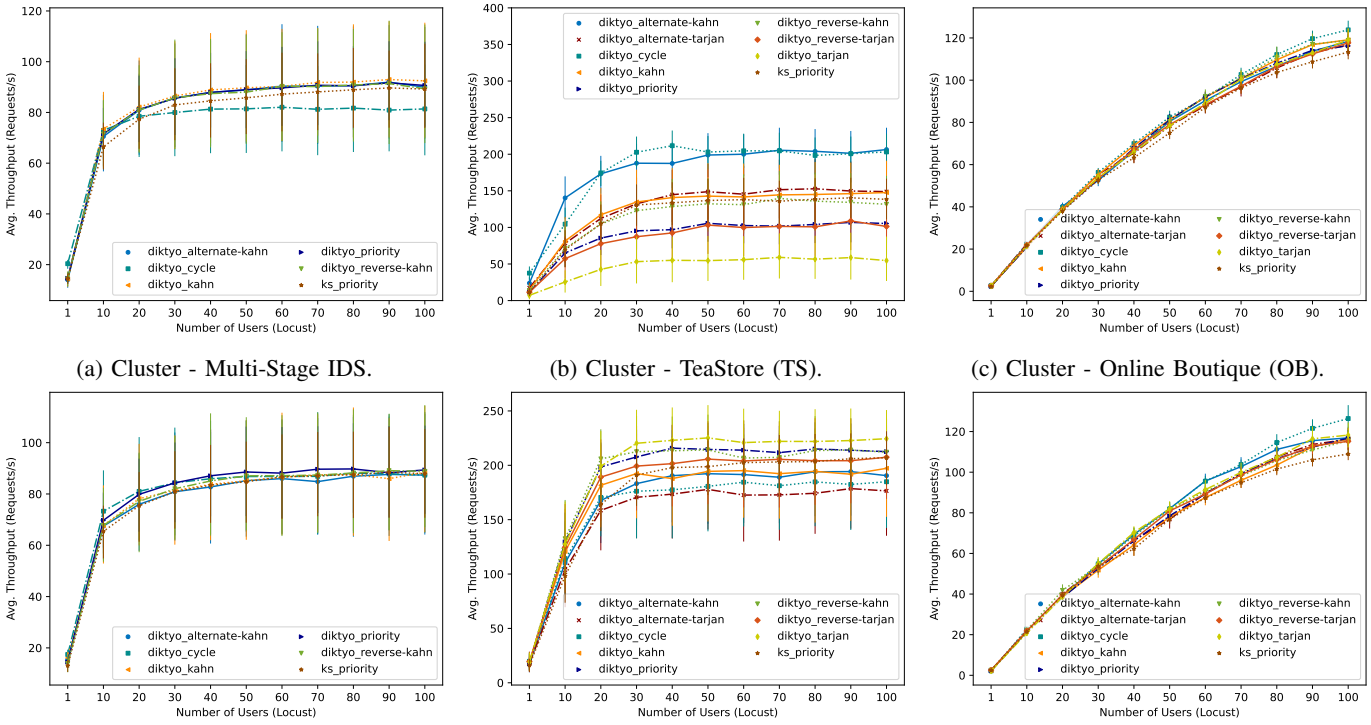
Fig. 7: The application response time (in ms) for the edge-cloud continuum based on a certain number of users.

Throughput (i.e., number of requests per second) is significantly affected by queue sorting (Fig. 8). Substantial variations in attained throughput are observed across all evaluated applications, particularly when the number of emulated users exceeds 30. For the Multi-Stage IDS application, Diktyo cycle performed slightly worse than the other schedulers in the cluster topology. However, for the TS application, Diktyo cycle achieved the highest throughput, with an average increase of 30% compared to KS, similar to Diktyo *Alt. Kahn*. In addition, the Diktyo cycle slightly outperformed the other sorting algorithms for the OB application, particularly for a high number of users, resulting in a throughput increase of at least 5% for both evaluated topologies. In contrast, the default KS was the worst scheduler for the OB application, with Diktyo capable of increasing throughput by up to 16% depending on the selected sorting.

Resource Consumption is consistent across all applications for CPU and memory usage, regardless of the applied scheduler (Fig. 9). However, Diktyo Cycle achieved slightly lower resource consumption than other algorithms, specifically for the OB application, on average a reduction of approximately 10% to 20% in CPU usage, even while achieving higher performance. Thus, deploying dependent pods close to each other seems to contribute to resource efficiency while improving overall application performance.

A. *Summary*

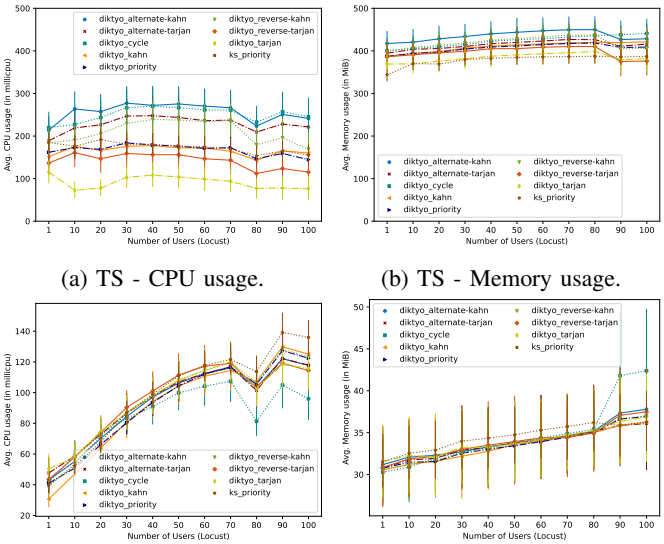
This study highlights the importance of selecting appropriate queue sorting approaches for container-based application scheduling, providing valuable insights into the performance implications associated with different sorting algorithms. The evaluation included three distinct applications, showing that the impact of an accurate sorting algorithm becomes more pronounced as the complexity of the container-based applications increases, such as the number of workloads. Significant variations in response time and throughput can be observed for the TS and OB applications when employing different sorting algorithms. These differences highlight the crucial role that sorting plays in application performance. Although the differences may be less pronounced for the Multi-Stage



(a) Cluster - Multi-Stage IDS. (b) Cluster - TeaStore (TS). (c) Cluster - Online Boutique (OB).

(d) Edge-Cloud - Multi-Stage IDS. (e) Edge-Cloud - TeaStore (TS). (f) Edge-Cloud - Online Boutique (OB).

Fig. 8: The average throughput (requests/s) obtained during the evaluation. Large differences have been obtained for the TS application while relatively small ones for the Multi-Stage IDS and OB applications.



(a) TS - CPU usage. (b) TS - Memory usage.

(c) OB - CPU usage. (d) OB - Memory usage.

Fig. 9: The resource consumption during the cluster evaluation.

IDS application, specific scenarios such as high numbers of emulated users can still unveil disparities in application performance. By understanding the performance implications of queue sorting, researchers and cloud practitioners can make informed decisions when selecting scheduling algorithms for their application deployments.

VI. Conclusions

This paper investigates the impact of queue sorting on container-based application scheduling. The evaluation considers several sorting algorithms available in the widely used K8s platform, including a novel algorithm named *Topological-Sort* introduced in our previous work. Experiments evaluated various performance criteria, including application response time, resource consumption, and throughput. Results show the significant role played by queue sorting on the application performance. The selection of a sorting algorithm substantially impacts the performance of container-based applications. The influence of queue sorting becomes more pronounced when the number of microservices within an application increases. Diktyo Cycle slightly outperformed the other sorting algorithms by improving response time and throughput on average by at least 15% and 20%, respectively. Our work contributes to the field by providing a benchmark for future research on container-based application scheduling that can guide the development of more efficient queue sorting algorithms.

Acknowledgments

José Santos is funded by the Research Foundation Flanders (FWO), grant number 1299323N. This work is supported by the Belgian Chancellery of the Prime Minister (Grant: AIDE-BOSA).

REFERENCES

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[2] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Software*, vol. 35, no. 3, pp. 96–100, 2018.

[3] Amazon, "Amazon elastic container service (amazon ecs)," accessed on 22 September 2021. [Online]. Available: https://aws.amazon.com/ecs/

[4] B. Burns, J. Beda, and K. Hightower, *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media, 2019.

[5] Red Hat, "Red hat openshift container platform," accessed on 22 September 2021. [Online]. Available: https://www.redhat.com/en/technologies/cloud-computing/openshift

[6] T. Schneider and A. Wolfsmantel, "Achieving cloud scalability with microservices and devops in the connected car domain." in *Software Engineering (Workshops)*, 2016, pp. 138–141.

[7] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 14)*, 2014, pp. 301–316.

[8] D. Zhao, M. Mohamed, and H. Ludwig, "Locality-aware scheduling for containers in cloud computing," *IEEE Transactions on cloud computing*, vol. 8, no. 2, pp. 635–646, 2018.

[9] M. Blöcher, L. Wang, P. Eugster, and M. Schmidt, "Switches for hire: resource scheduling for data center in-network computing," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 268–285.

[10] 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*, 2023.

[11] Scheduler Plugins, "The diktyo scheduling plugins," accessed on 28 August 2022. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/pkg/networkaware/README.md.

[12] M. Verkerken, L. D'hooge, D. Sudyana, Y.-D. Lin, T. Wauters, B. Volckaert, and F. D. Turck, "A novel multi-stage approach for hierarchical intrusion detection," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2023.

[13] Teastore, "The teastore is a micro-service reference and test application to be used in benchmarks and tests." accessed on 2 March 2023. [Online]. Available: https://github.com/DescartesResearch/TeaStore.

[14] Online Boutique, "Online boutique, a cloud-native microservices demo application." accessed on 2 March 2023. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo.

[15] K. C. Huang, Y. L. Tsai, and H. C. Liu, "Task ranking and allocation in list-based workflow scheduling on parallel computing platform," *The Journal of Supercomputing*, vol. 71, pp. 217–240, 2015.

[16] Kubernetes, "Production-grade container scheduling and management." accessed on 28 March 2023. [Online]. Available: https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/plugins/queuesort/priority_sort.go.

[17] B. Tang, J. Luo, M. S. Obaidat, and P. Vijayakumar, "Container-based task scheduling in cloud-edge collaborative environment using priority-aware greedy strategy," *Cluster Computing*, pp. 1–17, 2022.

[18] Y. Hu, J. Wang, H. Zhou, P. Martin, A. Taal, C. De Laat, and Z. Zhao, "Deadline-aware deployment for time critical applications in clouds," in *European Conference on Parallel Processing*. Springer, 2017, pp. 345–357.

[19] 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.

[20] Y. Gao and K. Zhang, "Deadline-aware preemptive job scheduling in hadoop yarn clusters," in *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 2022, pp. 1269–1274.

[21] A. Chung, J. W. Park, and G. R. Ganger, "Stratus: Cost-aware container scheduling in the public cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 121–134.

[22] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang, "A new container scheduling algorithm based on multi-objective optimization," *Soft Computing*, vol. 22, pp. 7741–7752, 2018.

[23] R. Ranjan, I. S. Thakur, G. S. Aujla, N. Kumar, and A. Y. Zomaya, "Energy-efficient workflow scheduling using container-based virtualization in software-defined data centers," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 12, pp. 7646–7657, 2020.

[24] Scheduler Plugins, "Repository for out-of-tree scheduler plugins based on the scheduler framework." accessed on 28 March 2023. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/qos.

[25] F. Larumbe and B. Sansò, "Elastic, on-line and network aware virtual machine placement within a data center," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 28–36.

[26] L. R. Rodrigues, M. Pasin, O. C. Alves, C. C. Miers, M. A. Pillon, P. Felber, and G. P. Koslovski, "Network-aware container scheduling in multi-tenant data center," in *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019, pp. 1–6.

[27] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 351–359.

[28] B. Ryu, A. An, Z. Rashidi, J. Liu, and Y. Hu, "Towards topology aware pre-emptive job scheduling with deep reinforcement learning," in *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, 2020, pp. 83–92.

[29] A. Muhammad and M. Aleem, "A3-storm: topology-, traffic-, and resource-aware storm scheduler for heterogeneous clusters," *The Journal of Supercomputing*, vol. 77, pp. 1059–1093, 2021.

[30] Volcano, "Tasktopology," accessed on 28 May 2023. [Online]. Available: https://github.com/volcano-sh/volcano/tree/master.

[31] Scheduler Plugins, "Repository for the topologicalsort plugin based on the scheduler framework." accessed on 28 March 2023. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/networkaware/topologicalsort.

[32] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[33] N. Ahmed, A. L. Barczak, T. Susnjak, and M. A. Rashid, "A comprehensive performance analysis of apache hadoop and apache spark for large scale data sets using hibench," *Journal of Big Data*, vol. 7, no. 1, pp. 1–18, 2020.

[34] Scheduler Plugins, "Repository for out-of-tree scheduler plugins based on the scheduler framework." accessed on 28 March 2022. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins.

[35] Kubernetes, "Scheduling framework." accessed on 28 March 2022. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

[36] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2557–2589, 2021.

[37] D. Popescu, N. Zilberman, and A. Moore, "Characterizing the impact of network latency on cloud-based applications' performance," 2017.

[38] K. Scheduler Plugins, "This folder holds the networkaware plugins for kubernetes," accessed on 27 March 2023. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/networkaware.

[39] C. Pang, J. Wang, Y. Cheng, H. Zhang, and T. Li, "Topological sorts on dags," *Information Processing Letters*, vol. 115, no. 2, pp. 298–301, 2015.

[40] J. Von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 223–236.

[41] Virtual Wall, "The virtual wall emulation environment." accessed on 2 March 2023. [Online]. Available: https://doc.ilabt.imec.be/ilabt/virtualwall/index.html.

[42] A. N. Kuznetsov, "tc(8) — linux manual page." accessed on 28 May 2023. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html.

[43] Locust, "An open source load testing tool." accessed on 28 March 2023. [Online]. Available: https://locust.io/.