Diktyo: Network-Aware Scheduling in Container-based Clouds

José Santos*, *Member, IEEE,* Chen Wang†, *Member, IEEE,* Tim Wauters*, Filip De Turck*, *Fellow, IEEE*
* Ghent University - imec, IDLab, Department of Information Technology, Gent, Belgium
Email: josepedro.pereiradossantos@UGent.be
† IBM Thomas J. Watson Research Center, New York, USA.

***Abstract*—Containers have revolutionized application deployment and life-cycle management in current cloud platforms. Applications have evolved from single monoliths to complex graphs of loosely-coupled microservices. However, the efficient allocation of microservice-based applications is challenging due to their complex inter-dependencies. Further, recent applications are becoming even more delay-sensitive, demanding lower latency between dependent microservices. Scheduling policies in popular container orchestration platforms mainly aim to increase the resource efficiency of the infrastructure, insufficient for latency-sensitive applications. Application domains such as the Internet of Things and multi-tier web services would benefit from network-aware policies that consider network latency and bandwidth in the scheduling process. Previous works have studied network-aware scheduling via theoretical formulations or heuristic-based methods evaluated via simulations or small testbeds, making their full applicability in popular platforms difficult. This paper proposes a novel network-aware framework for the popular Kubernetes (K8s) platform named Diktyo that determines the placement of dependent microservices in long-running applications focused on reducing the application's end-to-end latency and guaranteeing bandwidth reservations. Simulations show that Diktyo can significantly reduce the network latency for various applications across different infrastructure topologies compared to default K8s scheduling plugins. Also, experiments in a K8s cluster with microservice benchmark applications show that Diktyo can increase database throughput by 22% and reduce application response time by 45%.**

***Index Terms*—Microservices, Container Scheduling, Kubernetes, Network-Aware**

I. Introduction

Microservice architectures have gradually become the *de-facto* paradigm for application deployment in modern cloud platforms [1]. The traditional large monolith is decomposed into multiple loosely-coupled microservices, implemented and deployed independently. The increasing adoption of containers calls for efficient orchestration strategies for microservice-based applications in current cloud platforms (e.g., Kubernetes (K8s) [2], Amazon AWS [3]). However, next-generation applications are pushing cloud infrastructures further by demanding even lower latency between dependent microservices. Application domains such as multi-tier web services [4], the Internet of Things (IoT) [5], video streaming services [6], and databases [7] are latency-sensitive, requiring sub-millisecond end-to-end (E2E) latency for their proper operation. Current scheduling methods in popular orchestration platforms focus mostly on optimizing resource utilization in the infrastructure (e.g., CPU and Memory), insufficient to satisfy the stringent requirements of these applications, especially concerning latency and bandwidth. K8s is currently the most popular container orchestration platform. It automates several processes through the containerized applications' life-cycle, including deployment and scaling [8]. However, network-aware scheduling policies are missing in K8s to enable the network-aware placement of dependent microservices of long-running applications in container clouds.

Latency reduction is a primary objective since users usually face latency issues when using multi-tier applications, affecting the overall application performance [9]. These applications typically include tens to hundreds of microservices with complex inter-dependencies. Network latency is usually the primary culprit since these microservices are scheduled in the infrastructure without latency awareness, resulting in large distances between servers allocating dependent microservices. Also, **Bandwidth** optimization plays a key role, especially for those applications with high volumes of data transfers among microservices. For example, multiple replicas in a database application may require frequent copies to ensure data consistency. Spark jobs [10] may have regular data transfers between mappers and reducers. Insufficient network capacity on links between nodes would lead to increasing delay or packet drops, which would further degrade the application's Quality of Service (QoS). These applications would benefit from network-aware scheduling policies that determine the service placement based on latency and bandwidth metrics in addition to computing resources (e.g., CPU and Memory). Network latency and available bandwidth on links between cluster nodes can vary according to their locations in the underlying infrastructure. Deploying dependent microservices on different nodes without latency and bandwidth awareness can drastically impact the application's response time and overall QoS. For example, in the Redis cluster application [11] (Fig. 1a), master nodes need to synchronize data with slave nodes regularly. Dependencies between the masters and the slaves need to be considered in the scheduling process. Otherwise, high latency or low bandwidth between masters and slaves can lead to slow Create, Read, Update, and Delete (CRUD) operations [12]. Similarly, several dependencies exist in typical multi-tier web applications as the Online Boutique e-commerce application [13] (Fig. 1b). Deploying a microservice instance far away from dependent microservices can impact the Online

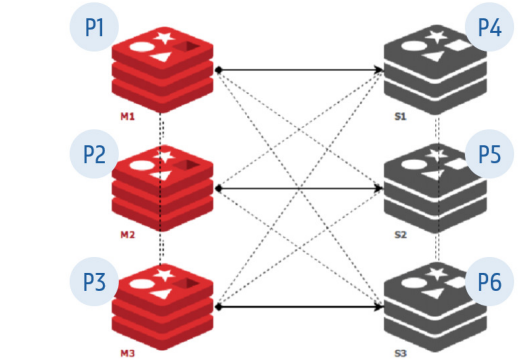
Manuscript received April 26, 2023

Boutique's E2E latency.

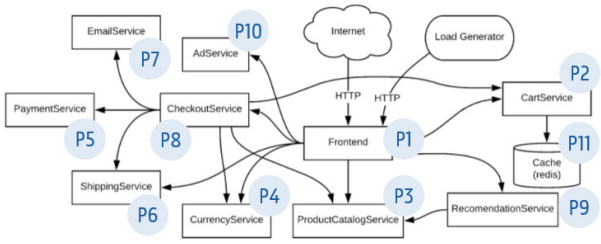
Previous works on network-aware scheduling focus mostly on theoretical formulations (e.g., [14]–[16]) or heuristic-based algorithms (e.g., [17], [18]) that usually are assessed via simulations or small testbeds, limiting their applicability in production systems. This paper presents a novel network-aware scheduling framework, named Diktyo [1], for the K8s platform inspired by its recent scheduling plugins architecture [19]. Diktyo proposes additional scheduling plugins aiming to determine low-latency deployment schemes for applications scheduled on K8s clusters. The framework minimizes the application's E2E latency by selecting nodes with low network costs for dependent microservices and with sufficient bandwidth capacity based on previous microservice allocations. Diktyo provides near-optimal container placement, considering the application's microservice dependencies and the underlying infrastructure topology. Section V presents further details about the Diktyo framework. To the best of our knowledge, Diktyo goes beyond the current state-of-the-art since it is the first attempt toward scalable network-aware placement of dependent microservices in future cloud-native architectures. The main contributions of the paper are the following:

- **Diktyo framework**: The design and implementation of a network-aware scheduling framework that separates the control plane (the scheduling logic) from the data plane (i.e., the application microservice dependencies and the cluster network topology). Two asynchronous controllers manage two Custom Resources (CRs) defined as Custom Resource Definitions (CRDs) [20] in K8s: **AppGroup** CRD establishes application's microservice dependencies; **NetworkTopology** CRD caches and updates network costs between regions and zones for the underlying K8s cluster topology. The proposed framework has already been accepted in the K8s scheduling community open-source repository [21] as an alternative scheduler. Researchers can use our framework to deploy their applications with network awareness in a K8s cluster and evaluate their performance against current mechanisms.
- **Mixed-Integer Linear Programming (MILP)**: The formulation of a MILP model for the container scheduling problem in K8s, including the specification of application dependency graphs and the underlying cluster topology with varying links in terms of network latency and bandwidth capacity. K8s allows the dynamic configuration of distinct plugins to achieve different goals concerning application scheduling. Prior works have not studied the performance of plugin combinations compared to an optimal solution. We answer this important question by evaluating an optimal scheme given by a MILP formulation, serving as an ideal baseline to assess the performance of the Diktyo framework and existing K8s scheduling plugins. Simulations show that Diktyo significantly outperforms current scheduling plugins concerning the application's E2E latency (Sec. VI-A).

[1]Diktyo means "network" in Greek, as an analogy of Kubernetes (K8s), which means "pilot" in Greek.



(a) Redis Cluster application.



(b) Online Boutique application.

Fig. 1: Illustration of microservice dependencies [11], [13].

- **Plugin implementation**: Three scheduling plugins have been designed for the Diktyo framework. The combination of these plugins aims to approximate the optimal solution given by the MILP model focused on minimizing the application's E2E latency in a scalable manner. A **TopologicalSort** plugin sorts microservices based on topological information, a **NodeNetworkCostFit** plugin filters out nodes based on microservice dependencies, and a **NetworkMinCost** plugin scores nodes based on network weights ensuring low latency between dependent microservices. All plugins achieve logarithmic performance over the number of nodes and microservices (Sec. VI-B).
- **Evaluations with microservice applications**: The evaluation considers real-world applications typically used as microservice benchmarks: a multi-tier web application named Online Boutique [13], and a database application named Redis cluster [11]). Experiments in a distributed K8s cluster show that Diktyo increases throughput by 22% for Redis and reduces the application's response time on average by 45% for Online Boutique (Sec. VI-C).

II. Related Work

This section addresses current literature on network-aware scheduling. Section II-A presents prior works related to network-aware placement or topology-aware scheduling. Section II-B reviews research on current open-source projects related to container scheduling, including production-ready plugins focused on microservice dependencies.

A. Network-aware and Topology-aware scheduling

Theoretical formulations have been the most applied method to solve network-aware allocation in the last few years [14]–[18], [22], [23]. These proposals typically focus on Integer Linear Programming (ILP) models to find the optimal allocation scheme 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. However, the modeling can always provide an optimal benchmark for heuristic-based algorithms. **Containerized applications** have been recently studied [24]–[30]. These works focus on optimizing container placement by addressing different challenges, including the efficient resizing of containers [29], multi-tenant fair scheduling [26], reducing E2E tail latency [30], and reducing the network latency [25]. **Microservice dependencies** have also been addressed mainly in the field of batch job scheduling systems [31]–[36]. Most efforts focus on improving resource efficiency [32], [35], characterizing inter-job dependencies [34], reducing the makespan of jobs [33], or improving the system's throughput [31]. The considered dependencies usually are temporal [34], [35], meaning later running jobs depend on the successful completion of earlier ones. However, the microservice dependencies considered in this paper are spatial, meaning all containers need to communicate with others and run as a whole for the application. Neglecting the impact of dependent containers during scheduling can impact the performance of the whole application.

Topology-aware scheduling has also been studied lately [37]–[41]. These efforts focus on placing microservices based on cluster topology information or application characteristics. Authors aim to improve performance by focusing on the heterogeneity of resources [40], the cluster energy efficiency [38], or the node Non-Uniform Memory Access (NUMA) topology [42]. However, most works do not address both the application characteristics and the infrastructure network topology for long-running applications with several microservices as typical K8s workloads. **Data Centers (DCs)** are also an important scenario where network-aware scheduling has been studied in recent years [18], [43]–[48]. These works focus on optimizing network bandwidth to reduce traffic congestion [44], reducing the average task completion time [43], improving performance focused on resource dependencies [47], or priority-based flow-aware scheduling [48]. However, practical implementations of these methods are missing since most network-aware algorithms are evaluated via simulations. In addition, most efforts focus on Virtual Machine (VM) placement and only a few address container allocation [18]. Nevertheless, network latency is usually overlooked in current network-aware approaches for DC topologies since network bandwidth is their primary goal. **Geo-distributed clouds** also impose several challenges toward network-aware deployment schemes [22], [27], [28], [49]–[51]. Recent works [27], [28] model application dependencies as service chains to optimize service scheduling in edge-fog computing scenarios [52]. High latency is a major concern for many applications (e.g., IoT and video streaming) in these distributed topologies. These efforts have shown the benefits of network-aware placement methods concerning the application's response time and throughput.

B. Open-Source Projects

The Volcano project [53] provides several plugins for K8s focused on microservice dependencies. The Gang [54] 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. The Task Topology [55] 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. Latency is not addressed since tasks are only placed according to the created buckets. Additional plugins in the Volcano project such as the Binpack [56] and the DRF [57] focus on improving resource utilization and preventing small job starvation [58]. **The Scheduler Plugins repository** [19] also provides additional plugins for the K8s platform. It is worth mentioning the CoScheduling [59] and the Topology-aware [60] plugins focused on microservice dependencies. CoScheduling operates similarly to the Gang plugin provided by Volcano. It ensures that a minimum number of microservices belonging to the same *PodGroup* are scheduled as a whole. The main difference between Diktyo plugins and CoScheduling is that Diktyo supports complex dependencies between heterogeneous containers. The Topology-aware [60] plugin focuses on performance issues concerning memory, and CPU accesses in NUMA nodes [61] based on node topologies specified in *NodeResourceTopology CRs*. In contrast, Diktyo considers both microservice dependencies and the underlying cluster network topology.

In summary, Table I shows a comparison of all plugins listed above with the proposed Diktyo framework concerning the plugin's extension point, scheduling goals, and network awareness. Diktyo goes beyond the current literature since it considers the applications' microservice dependencies and the underlying network topology to determine low-latency deployment schemes in K8s clusters. Prior works focus mostly on theoretical models and heuristics evaluated via simulations or small testbeds, limiting their applicability in large-scale production clusters.

III. Microservice Scheduling in Kubernetes (K8s)

Microservices in K8s are often tightly coupled into a group of containers called a *pod*. A pod is the smallest working unit in K8s representing the collection of containers and volumes (storage) running in the same execution environment [2]. K8s schedules a given pod on a node based on the pod deployment requirements and the cluster's available resources. The component responsible for scheduling operations is called Kube-Scheduler (KS), the default scheduler in K8s. The KS chooses a node for the pod deployment based on a two-step operation. Firstly, the KS checks if nodes can run the pod based on a set of filters, also known as predicates. These filters focus mostly on the pod's resource requirements (e.g., CPU and Memory) and check if the node has enough capacity to

TABLE I: Comparison among different scheduling plugins.

Plugin / Framework	Project	Extension Point	Scheduling Goal	Service Topology	Microservice Dependencies	Latency	Bandwidth
Gang	V	PF	AP & R	✕	✕	✕	✕
Task Topology	V	QS & PF & S	T & R	✓	✕	✕	✕
Binpack	V	S	R	✕	✕	✕	✕
Dom. Res. Fairn. (DRF)	V	QS & PF	R	✕	✕	✕	✕
CoScheduling	S	QS & PF	AP	✓	✕	✕	✕
Topology-aware	S	F	R & T	✓	✕	✕	✕
Diktyo	D	QS & F & S	L & B	✓	✓	✓	✓

Project: V = Volcano, S = Scheduler Plugins, D = Diktyo framework.
Extension Point: QS= QueueSort, PF = PreFilter, F = Filter, S = Score.
Scheduling Goal: R = Resources, AP = App. Dependencies, T = Topology-aware, L = Latency, B = Bandwidth.
Service Topology, Microservice Dependencies, Latency, Bandwidth: ✓= addressed, ✕= not considered.

meet those. Secondly, the KS applies node priority calculation by ranking each remaining node based on a set of scoring algorithms, also called priorities. Then, KS selects the highest-scoring node for the pod deployment. To ease the development of further filter and scoring functions, K8s released a scheduling framework [62] so that developers can implement their algorithms and contribute to the K8s project. The K8s scheduling framework implements several extension points for the KS. The framework allows developers to implement their algorithms as plugins without interfering with the main scheduling components. The framework currently exposes the following main extension points, responsible for:

- **QueueSort**: sort pods in the scheduling waiting queue.
- **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 network awareness in application scheduling. We tackle this challenge by proposing the Diktyo framework that applies these extension points to include bandwidth and latency in the K8s scheduling process, which will significantly impact most industries. Instead of exploring novel designs that can take years to impact the current systems, we focus on solving this issue by designing missing components based on readily available features in K8s since network awareness is an urgent need [63]. Sec. V details the proposed framework, while the next section presents a MILP model for the K8s deployment scheme, where pods are placed on nodes based on resource constraints and focused on minimizing the network latency between dependent microservices. The model provides a benchmark for the Diktyo framework and existing plugins to compare their sub-optimal allocation schemes with an optimal solution.

IV. Mixed-Integer Linear Programming (MILP) Approach

This section presents the MILP formulation for the network-aware approach in K8s scheduling, which places pods on nodes to both maximize the total number of applications admitted and minimize the applications' network latency.

TABLE II: Input variables of the MILP model.

Symbol	Description
N	The set of nodes on which pod instances are executed.
A	The set of applications $a \varepsilon A$. Each application a consists in a group of different pods with established dependencies.
P	The set of pods $p \varepsilon P$.
Z	The set of cluster zones where applications can be deployed.
R_p	The requested number of replicas for each pod $p \varepsilon P$.
R_{max}	The maximum number of replicas for each pod p.
$I_{a,p}$	The Instance matrix (binary). If $I_{a,p} = 1$, the pod p belongs to application a.
$\Omega_n[c]$	The capacity vector of node n. c denotes resources as CPU (in vcpu), memory (in Mi), and bandwidth (in Mbps).
$\omega_p[d]$	The demand vector of pod p. d denotes resources as CPU (in vcpu), memory (in Mi), and bandwidth (in Mbps).
B_{n_1,n_2}	The network bandwidth matrix. B_{n_1,n_2} indicates the bandwidth capacity (in Mbps) between node n_1 and node n_2.
τ_{n_1,n_2}	The network latency matrix. τ_{n_1,n_2} indicates the latency (in ms) between the node n_1 and the node n_2.
C_{p_i,p_j}	The pod communication matrix. It indicates the minimum bandwidth (in Mbps) demand of the network flow between the pod p_i (source) and p_j (sink).
$E_{n,z}$	The zone matrix (binary). If $E_{n,z} = 1$, the node n is at the cluster zone z.
α_{p_i,p_j}	The application matrix (binary). If $\alpha_{p_i,p_j} = 1$, the flow bandwidth between the pods p_i and p_j must be guaranteed.

TABLE III: Decision variables of the MILP model.

Symbol	Description
G_a	The acceptance matrix (binary). If $G_a = 1$, the application a is deployed.
$G_{a,p}$	The pod acceptance matrix (binary). If $G_{a,p} = 1$, the pod p for the application a is allocated.
$P_r^{a,p}(n)$	The placement matrix (binary). If $P_r^{a,p}(n) = 1$, the replica r of pod p from application a is executed on node n.
$F_{p_j,r_j}^{a,p_i,r_i}(n_1,n_2)$	The flow matrix (binary). It indicates that the rep. r_i of pod p_i is allocated on node n_1 and the rep. r_j of pod p_j is deployed on node n_2 for app. a.
Λ_a	The App. latency matrix. It indicates the total network latency (in ms) expected for app. a.

A. Model Description & Variables

The main advantage of MILP is the flexibility to analyze NP-hard problems [64] as the K8s deployment model and provide a benchmark for developed heuristics [65]. The MILP model formulates the pod placement problem in K8s as an optimization problem subject to several constraints. Table II lists input variables, and Table III presents decision variables. The model decomposes an application $a \varepsilon A$ in a set of different pods $p \varepsilon P$. The maximum number of allowed instances per

pod is given by R_{max}. These pods have a specific number of requested instances (R_p) that need to be deployed in the cluster, subject to multiple constraints. Nodes have limited capacities in terms of CPU, memory, and bandwidth resources ($\Omega_n[c]$); communication links have limited bandwidth capacity (B_{n_1,n_2}), and pod dependencies directly affect the application latency (Λ_a).

The MILP model determines a node for the placement of each pod replica depending on the considered objective. The objectives considered in the model are the following:

1) **Maximization of Application Deployments** (MAX AD).
2) **Minimization of the Network Latency** (MIN NL).

These objectives are executed iteratively, meaning that a different optimization is applied in each iteration. First, the acceptance of application deployments is maximized, and then, in the second iteration, the network latency is minimized. An additional constraint is added to the model to retain the objective value obtained in the first iteration to ensure the number of admitted applications remains after applying the second objective. Thus, the second iteration refines the previously obtained allocation scheme by considering an additional objective. A multi-objective function in a single iteration is a viable alternative that can be considered. However, it has been favored the consideration of two objectives in different iterations since the multi-objective function would have become a highly complex objective, making it difficult to interpret the model results. Further explanations concerning all variables are given in the next section, where constraints are detailed.

B. Constraints

The placement of an application needs to satisfy multiple constraints. First, all pods of the application need to be deployed as a whole. An indicator constraint (1) states that an application a is deployed (i.e., $G_a = 1$) if all pod instances belonging to the application have been deployed. Also, a given pod p is scheduled (i.e., $G_{a,p} = 1$) if all pod replicas have been instantiated as shown in (2). Further, pods are only deployed on nodes with enough computing resources (3).

$$\forall a \,\varepsilon\, A : \sum_{p \,\varepsilon\, P} G_{a,p} = \sum_{p \,\varepsilon\, P} I_{a,p} \quad \textbf{if } G_a = 1 \tag{1}$$

$$\forall a \,\varepsilon\, A, p \,\varepsilon\, P : \sum_{r \,\varepsilon\, R_{max}} \sum_{n \,\varepsilon\, N} P_r^{a,p}(n) = I_{a,p} \times R_p \quad \textbf{if } G_{a,p} = 1 \tag{2}$$

$$\forall n \,\varepsilon\, N : \sum_{a \,\varepsilon\, A} \sum_{p \,\varepsilon\, P} \sum_{r \,\varepsilon\, R_{max}} P_r^{a,p}(n) \times \omega_p[d] \leq \Omega_n[c] \tag{3}$$

The application flow matrix F is subjected to various constraints to accurately represent network flows. Pod dependencies are expressed by the Flow Factor $\Upsilon p_i, p_j$ as shown in (4) by using the App. matrix α_{p_i,p_j}. Bandwidth limitations (5) ensure the infrastructure capacity is respected, while flow conservation constraints (6) and (7) ensure no flow is lost within the network. The total network latency (in ms) of each application (Λ_a) can be derived from the application flow matrix F as shown in (8).

$$\Upsilon_{p_i,p_j} = I_{a,p_i} \times I_{a,p_j} \times \alpha_{p_i,p_j} \tag{4}$$

$$\forall n_1, n_2 \,\varepsilon\, N : \sum_{a \,\varepsilon\, A} \sum_{p_i,p_j \,\varepsilon\, P} \sum_{r_i,r_j \,\varepsilon\, R_{max}} \Upsilon p_i, p_j \times C_{p_i,p_j} \times F_{p_j,r_j}^{a,p_i,r_i}(n_1, n_2) \leq B_{n_1,n_2} \tag{5}$$

$$\forall a \,\varepsilon\, A, p_i, p_j \,\varepsilon\, P, r_i, r_j \,\varepsilon\, R_{max}, \forall n_1, n_2 \,\varepsilon\, N : F_{p_j,r_j}^{a,p_i,r_i}(n_1, n_2) = 0 \quad \text{if } P_{r_i}^{a,p_1}(n) = 0 \vee P_{r_j}^{a,p_j}(n) = 0 \tag{6}$$

$$\sum_{a \,\varepsilon\, A} \sum_{p_i,p_j \,\varepsilon\, P} \sum_{r_i,r_j \,\varepsilon\, R_{max}} \sum_{n_1,n_2 \,\varepsilon\, N} F_{p_j,r_j}^{a,p_i,r_i}(n_1, n_2) = = \sum_{p_i,p_j \,\varepsilon\, P} \alpha_{p_i,p_j} \times R_{p_i} \times R_{p_j} \tag{7}$$

$$\forall a \,\varepsilon\, A : \Lambda_a = \sum_{p_i,p_j \,\varepsilon\, P} \sum_{r_i,r_j \,\varepsilon\, R_{max}} \sum_{n_1,n_2 \,\varepsilon\, N} \tau_{n_1,n_2} \times F_{p_j,r_j}^{a,p_i,r_i}(n_1, n_2) \qquad \text{(in ms)} \tag{8}$$

In addition, two constraints have been added to represent topology preferences [66]. Pod anti-affinity rules (9) ensure replicas of the same pod are allocated on different nodes, and zone anti-affinity rules (10) spread pod replicas across different zones in the cluster.

$$\forall n \,\varepsilon\, N, \forall a \,\varepsilon\, A, \forall p \,\varepsilon\, P : \sum_{r \,\varepsilon\, R_{max}} P_r^{a,p}(n) \leq 1 \tag{9}$$

$$\forall a \varepsilon A, \forall p \varepsilon P, \forall z \varepsilon Z : \sum_{n \,\varepsilon\, N} \sum_{r \,\varepsilon\, R_{max}} P_r^{a,p}(n) \times E_{n,z} \leq 1 \tag{10}$$

C. Objectives

The first objective (i.e., **MAX AD**) is expressed in (11) by using the acceptance matrix G_a. The second objective (i.e., **MIN NL**) is expressed as shown in (12), where the model determines an allocation scheme based on the applications' dependency graph. The application latency matrix (i.e., Λ_a) is an auxiliary decision variable since its value depends on which nodes dependent pods are deployed. The exact location of these replicas directly affects the application latency formulation, as shown in (8). Dependent pods are allocated close to each other to reduce the expected network latency while respecting all constraints. As mentioned, an additional constraint (13) ensures the objective value of the first iteration is kept while executing the second iteration.

$$max \sum_{a \,\varepsilon\, A} G_a \tag{11}$$

$$min \sum_{a \,\varepsilon\, A} \Lambda_a \tag{12}$$

$$\sum_{a \,\varepsilon\, A} G_a = obj_1 \tag{13}$$

obj_1 = Objective value obtained in the first iteration

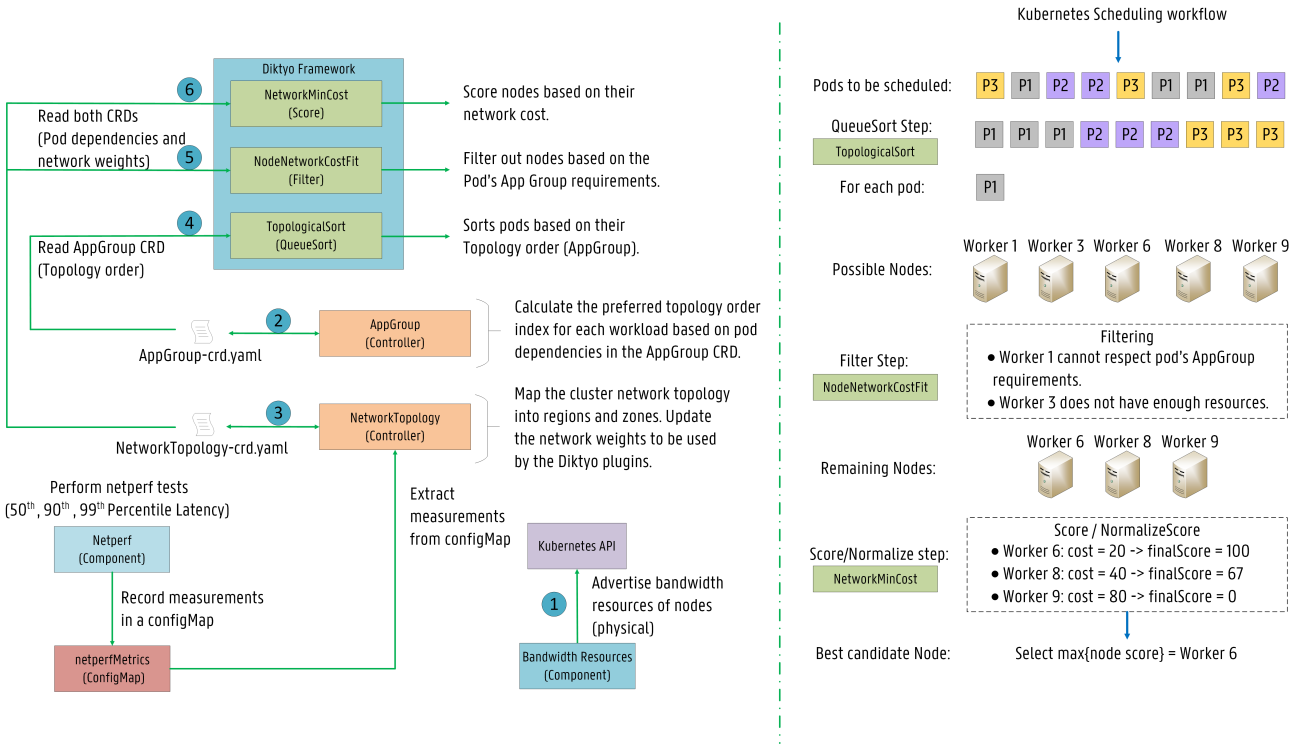


Fig. 2: Illustration of the Diktyo framework and the Kubernetes scheduling workflow.

V. Diktyo Framework: System Design

A. System Overview

Fig. 2 shows an overview of the Diktyo framework and the typical K8s scheduling workflow, including Diktyo scheduling plugins. Bandwidth resources are advertised to the K8s Application Program Interface (API) to consider the node available bandwidth in the scheduling process ①. The framework introduces two CRs: **AppGroup** and **NetworkTopology** to maintain both the application dependency information ② and the infrastructure network topology ③. Diktyo considers both application dependencies and the cluster network topology when scheduling pods in K8s. The **NetworkTopology** controller updates network weights between cluster nodes across regions and zones based on a netperf component. Diktyo provides network-aware algorithms implemented as three scheduling plugins based on the K8s scheduler framework [19]: **TopologicalSort**, **NodeNetworkCostFit** and **NetworkMinCost**. First, pods are sorted based on their established dependencies ④. Then, nodes are filtered out based on the pod's AppGroup requirements ⑤, and finally, nodes are scored based on network weights ensuring low network costs between dependent pods ⑥. Further explanations are given below on how these plugins interact with both CRs.

B. Bandwidth Enforcement

In K8s, the node's available bandwidth can be defined via extended resources [67]. A bandwidth component [68] has been developed to send HTTP requests to the K8s API to advertise the node's (physical) bandwidth based on its network interface speed. The label *network.aware.com/bandwidth* has been created to specify the node's available bandwidth and to request bandwidth resources in pod deployments. This allows performing default filtering and scoring algorithms based on bandwidth resources (e.g., *MostRequested*, *BalancedAllocation*) and not only on CPU and memory resources. We acknowledge the difficult task of specifying accurate bandwidth resources since it relies on the knowledge of the application developer to provide accurate numbers. This is a known concern today for CPU and memory resources since demand changes over time, thus the number of resources needed. A potential research direction is the development of softwarized components that keep track of the resource usage of containers. These components could provide historical reports to the system and ultimately recommend accurate numbers for resource requirements based on the container's usage report. However, this is out of the scope of this paper.

In addition, the bandwidth CNI plugin [69] can enforce network bandwidth allocations for pods. It supports ingress and egress traffic shaping to limit the pod bandwidth. Pods share the host network bandwidth when deployed on the same node. Limiting Pod bandwidth can prevent mutual interference and improve network stability [70]. The addition of *kubernetes.io/ingress-bandwidth* and *kubernetes.io/egress-bandwidth* annotations to the pod deployment file ensures bandwidth limitations are respected.

C. Application Group CRD & Controller

An AppGroup CRD [71] has been created to describe the application's pod dependencies. Fig. 3 shows an example of an application composed of three pods in K8s and the corresponding dependency graph. Developers need to specify

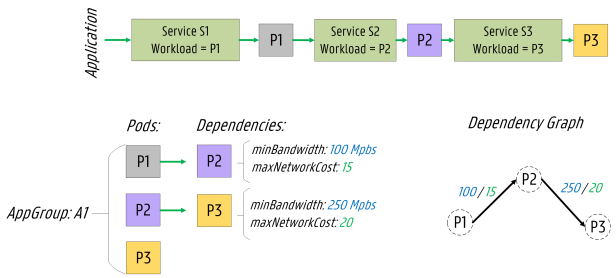


Fig. 3: Example of application's pod dependencies in K8s.

TABLE IV: Topological sorting for Online Boutique.

Algorithm	Topological order
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]

all pod dependencies (i.e., which pods communicate with another pod) so that the Diktyo framework can find the optimal allocation scheme focused on low latency. Also, two additional requirements can be added to the specified dependencies helping to fine-tune the behavior of the Diktyo framework. First, the *minbandwidth* requirement defines the minimum bandwidth between two pods belonging to the same AppGroup. Thus, nodes with insufficient bandwidth that cannot respect the specified bandwidth requirements cannot be selected for the pod deployment. Second, the *maxNetworkCost* requirement determines the maximum network cost between two pods. If the network cost between two nodes is higher than the specified *maxNetworkCost*, Diktyo does not place these two pods on these nodes. Another important concept is a K8s *Service* [72], an abstract way to define a logical set of pods and expose the applications running on them. K8s services make load-balancing a straightforward process since pods have their own IP address, and a single Domain Name System (DNS) name exists for a set of pods. The rationale behind this abstraction is that pods are not permanent resources, being terminated and redeployed constantly in the cluster. Thus, certain pods are terminated, and new ones are deployed without users' awareness. The proposed AppGroup selects pods based on their corresponding K8s service.

An application might consist of several pods with dependencies. The tighter constraints a pod has, the more likely the pod cannot find a node that satisfies all constraints. Scheduling the pod with tighter constraints earlier would be preferred, so it would not be blocked later, leading to starvation. However, it is not straightforward to determine which pod has tighter constraints. Thus, the developed **AppGroup controller** [73] calculates the preferred scheduling order for the AppGroup via six heuristic topological sorting algorithms [74]: *Kahn* [75], *Tarjan* [76], *AlternateKahn*, *AlternateTarjan*, *ReverseKahn*, *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 and modifies the order of Tarjan. ReverseKahn and ReverseTarjan essentially reverse the preferred order given by Kahn and Tarjan, respectively. For the previous AppGroup example, the topology order for Kahn would be **P1, P2, P3**, and **P3, P2, P1** for ReverseKahn. Also, a highly-complex AppGroup is the Online Boutique application previously shown in Fig. 1b. It consists of eleven pods, named from P1 to P11. Table IV presents the preferred order for all sorting algorithms. As shown, significant differences in the preferred order are obtained depending on which topology algorithm is selected.

D. Network Topology CRD & Controller

A networkTopology CRD [77] has been created to define the K8s infrastructure topology. A networkTopology CR stores network costs between all pair-wise nodes in the cluster based on their zones and regions. As an initial design, network weights can be manually defined in a networkTopology CR where network costs between zones and between regions are specified. In addition, to accurately measure the latency in the cluster, a **netperf component** [78] has been developed for the Diktyo framework. Netperf tests [79] are executed based on the infrastructure, allowing the estimation of the latency between cluster nodes, especially different latency percentiles (i.e., 50th, 90th, and 99th percentile). These measurements are recorded in a configmap [80] as key-value pairs with *origin* and *destination* as labels. Then, the developed **networkTopology controller** [81] accesses the configmap to extract the netperf measurements and calculates accurate network costs across regions and zones in the cluster. Then, the networkTopology controller dynamically updates the CR accordingly, so Diktyo plugins can apply updated network weights instead of the one-time manually configured weights. The periodical probing of the network latency via the netperf component is only necessary for one pair of nodes between zones/regions. One-time probing between a single pair of nodes is sufficient if nodes within a zone have similar connections. Thus, the probing is limited, avoiding significant overhead for large-scale clusters. The netperf component currently supports two execution modes. The *basic* mode runs a netperf test from a node to another one (i.e., in total N tests, being N the number of cluster nodes), while the *full* mode runs a test from all nodes to every other node in the cluster (i.e., $N \times (N-1)$ tests). Bandwidth measurements (Table V) based on the testbed infrastructure (Fig. 15) shown in Sec. VI-C demonstrate that the developed netperf component is still lightweight for small to medium-sized clusters. Also, the networkTopology controller can work with any customized software components that update the configmap. Cloud administrators can apply various methods to update the network costs according to their preferences.

The networkTopology controller also maintains the available bandwidth (i.e., *bandwidthAllocatable*) between regions and zones in the cluster. The networkTopology controller keeps a record of the allocatable bandwidth in the cluster so that pods are not scheduled on nodes where the available bandwidth between zones or regions is minimal. The goal is to avoid

TABLE V: The traffic generated by the netperf component in a 16-node K8s cluster.

Mode	Number of measurements	Avg. received traffic (in Mbps)	Avg. transmitted traffic (in Mbps)
Basic	16	12.96 Mbps	13.44 Mbps
Full	240	33.52 Mbps	35.20 Mbps

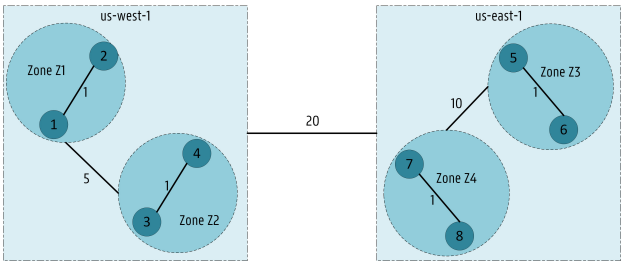


Fig. 4: Illustration of a Network Topology CR.

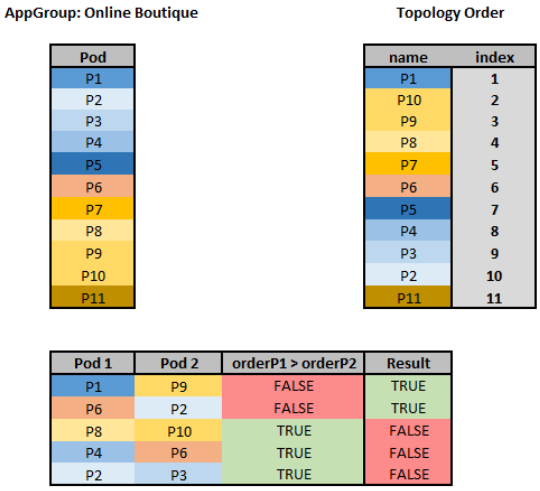


Fig. 5: Example of the TopologicalSort plugin operation.

network congestion in the K8s cluster infrastructure by placing pods on nodes with enough bandwidth for dependent pods and creating pod eviction events [82] in case bandwidth-constrained zones or regions are detected. Pod allocations and corresponding dependencies are read from an AppGroup lister [83], and bandwidth reservations are saved in the networkTopology CR based on current pod deployments. Fig. 4 shows a graphical representation of a networkTopology CR with two regions (*us-west-1 and us-east-1*) and four zones (*z1, z2, z3, z4*) detailed in [84].

E. Diktyo Scheduling Plugins

This section presents further details on the scheduling plugins designed and implemented for the Diktyo framework [85]. First, the **TopologicalSort Plugin (QueueSort)** sorts pods belonging to an AppGroup based on the topology order calculated by the AppGroup controller. It prioritizes pods based on the preferred allocation order. If pods do not belong to an AppGroup or belong to different AppGroups, the plugin follows the strategy provided by the QoS plugin [86]. For

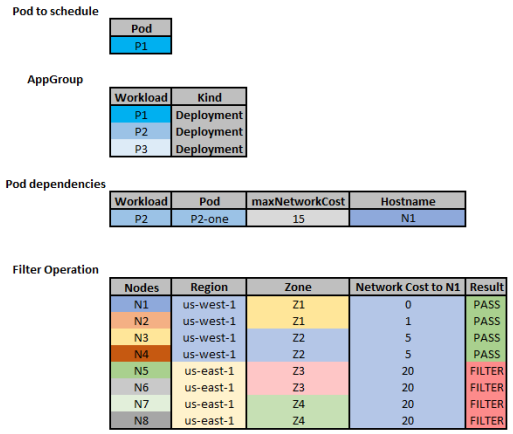


Fig. 6: Example of the NodeNetworkCostFit plugin operation.

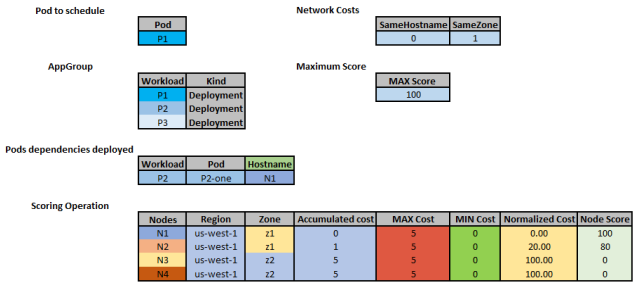


Fig. 7: Example of the NetworkMinCost plugin operation.

instance, consider the Online Boutique application and the correspondent topology order based on the KahnSort algorithm shown previously in Table IV. Depending on the two pods from Online Boutique considered for deployment in the scheduling queue, the result of the TopologicalSort plugin can be significantly different, though it favors low indexes (Fig. 5).

Second, the **NodeNetworkCostFit Plugin (Filter)** respects pod dependencies established in the AppGroup CR. The implementation currently focuses on *maxNetworkCost* requirements, filtering out nodes that would generate higher network costs than the specified threshold. Since applications usually include multiple pods with inter-dependencies, the pods deployed later may have constraints not fully respected. Thus, the plugin filters out nodes that cannot support most dependencies of already deployed pods. The goal is to reduce the number of nodes to score, avoiding a large set of candidate nodes in the scheduling workflow that would cause high network costs. Fig. 6 shows an example of the NodeNetworkCostFit plugin operation by deploying a pod from the AppGroup *A1* shown previously in Fig. 3 and the network topology infrastructure shown in Fig. 4.

Lastly, the **NetworkMinCost Plugin (Score)** favors the node with the lowest aggregated network cost to nodes allocating dependent pods. The aggregated network cost is calculated based on pod dependencies maintained by the AppGroup CR. The cost per dependency is the network weight between zones or regions available in the networkTopology CR of the nodes allocating dependent pods. Scheduled pods of a certain

TABLE VI: The hardware configuration of each node based on Amazon EC2 On-Demand Pricing [88].

Topology	Node	Amazon Image	CPU (cpu)	RAM (Mi)	Band. (Gbps)
Multi-Region	Cloud	a1.4xlarge	16.0	32.0	40.0
	Fog Tier 2	a1.2xlarge	8.0	16.0	10.0
	Fog Tier 1	a1.xlarge	4.0	8.0	5.0
	Edge	a1.large	2.0	4.0	1.0
Cluster	Fog Tier 1	a1.xlarge	4.0	8.0	1.0
Data Center	Fog Tier 1	a1.xlarge	4.0	8.0	1.0

AppGroup are retrieved via a pod lister [87] from the K8s API. Thus, the plugin calculates the aggregated cost to schedule a pod on a particular node based on previous pod placements. For the first pod deployment in the AppGroup, the plugin favors all candidate nodes equally. Also, this plugin normalizes scores between 0 and 100 based on all candidate nodes' maximum and minimum scores. Nodes with lower costs are favored since it also corresponds to lower latency. The plugin can be combined with other scoring functions (e.g., *BalancedAllocation*, *LeastRequestedPriority*), but a higher weight should be attributed to the *NetworkMinCost* plugin to prefer latency-aware scheduling schemes. Fig. 7 shows an example of the NetworkMinCost plugin operation by sequentially running the scoring plugin after the previous filter example.

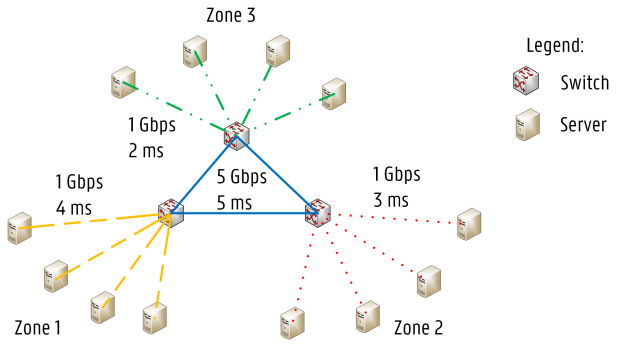
VI. Evaluations

This section presents the experiments conducted to evaluate the performance of the Diktyo framework. Sec. VI-A presents a simulation environment used to validate the proposed approach, and Sec. VI-B shows the scalability benchmarks of the implemented Diktyo plugins. Lastly, Sec. VI-C presents the testbed experiments performed in a distributed K8s cluster to assess the implemented framework.

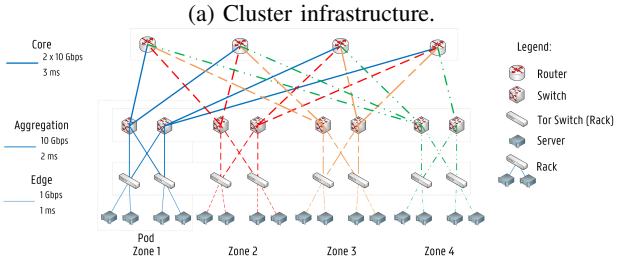
A. Simulation Environment

1) Infrastructure Topologies & Input Variables: Fig. 8 shows the three evaluated infrastructure topologies. First, Fig. 8a represents a highly available cluster with nodes deployed across zones. Second, Fig. 8b illustrates a DC with nodes connected via a fat-tree network topology [89]. Lastly, Fig. 8c shows a Multi-Region (MR) edge cluster with nodes distributed across several zones and regions. Nodes provide computing resources to allocate pods in the infrastructure. Table VI presents the nodes' hardware configurations for each topology. The bandwidth matrix B_{n_1,n_2} is based on the link's available bandwidth and the latency matrix τ_{n_1,n_2} is calculated based on the shown latency values.

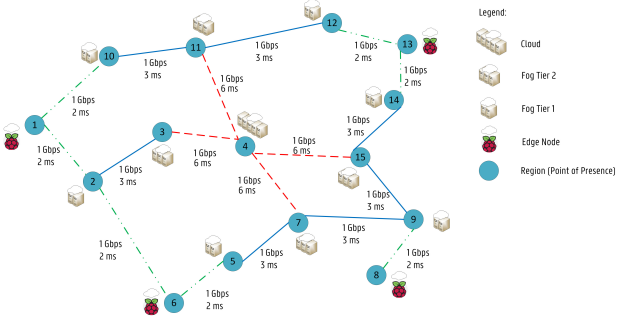
The described MILP model has been implemented in Python using the IBM ILOG CPLEX ILP solver [90]. As previously stated, the model considers two consecutive objective functions: the maximization of application deployments (i.e., MAX AD) and then the minimization of the network latency (i.e., MIN NL). The model has been executed on a 6-core Intel i7-9850H CPU @ 2.6 GHz processor with 16 GB of memory. The model and the scheduling algorithms are executed 50 times, and the results are shown with a 95% confidence interval.



(a) Cluster infrastructure.



(b) Data Center (DC) fat tree topology.



(c) Multi-Region (MR) scenario.

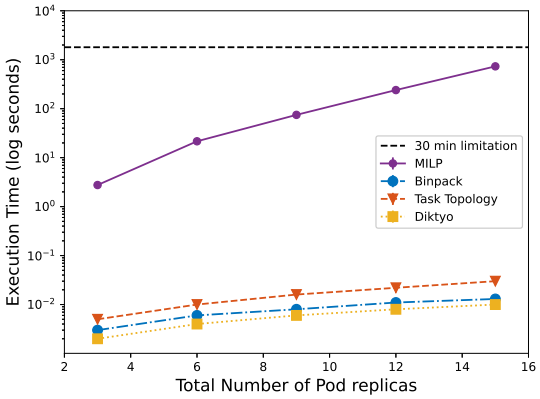
Fig. 8: Illustration of the three evaluated topologies.

2) Applications & Scheduling algorithms: Table VII shows the experimental settings for the three evaluated applications. The **Basic** (a_1) application provides a naive application composed of three pods previously shown in Fig. 3 to demonstrate the viability of the MILP model. Then, two use cases based on real-world applications are assessed: the **Redis Cluster** (a_2) database and the **Online Boutique** (a_3) application as illustrated previously in Fig. 1. The simulation compares four scheduling algorithms focused on their network awareness: the MILP's optimal allocation scheme, the Volcano Binpack plugin, the Volcano Task Topology plugin, and the Diktyo framework. The simulation does not include a scenario in which applications cannot be all deployed in the cluster. As a result, all algorithms are able to maximize the number of accepted applications (i.e., MAX AD). Thus, results on the acceptance rate are not shown. Though, we acknowledge the importance of analyzing how these algorithms would behave when available resources are not enough to handle all requests. These constrained scenarios are planned as part of future work.

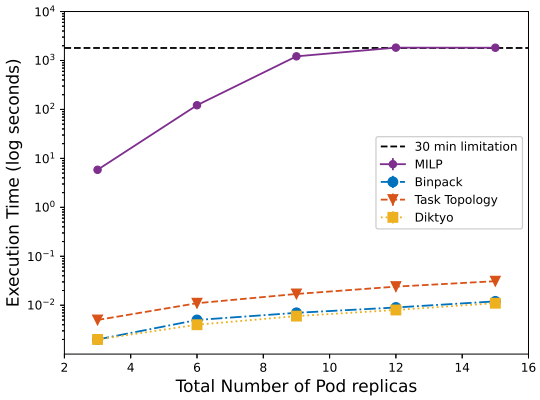
3) Simulation Results: Fig. 9a and Fig. 9b compare the execution time of the four scheduling algorithms to obtain the pod placement scheme for the Basic application on both

TABLE VII: Deployment properties of the evaluated applications.

App.	Pods & Number of Pod Replicas	CPU (cpu)	RAM (Mi)	Band. (Mbps)
Basic (a_1)	P1 (p_1): [1 to 5] P2 (p_2): [1 to 5] P3 (p_3): [1 to 5]	1.0	1.0	500
Redis Cluster (a_2)	Master 1 (p_1): [1] Master 2 (p_2): [1] Master 3 (p_3): [1] Slave 1 (p_4): [1 to 5] Slave 2 (p_5): [1 to 5] Slave 3 (p_6): [1 to 5]	1.0	1.0	250
Online Bout. (a_3)	Frontend (p_1): [1,2] Cart (p_2): [1,2] Product (p_3): [1,2] Currency (p_4): [1,2] Payment (p_5): [1,2] Shipping (p_6): [1,2] Email (p_7): [1,2] Checkout (p_8): [1,2] Recom. (p_9): [1,2] Ad (p_{10}): [1,2] Redis (p_{11}): [1,2]	1.0	1.0	250



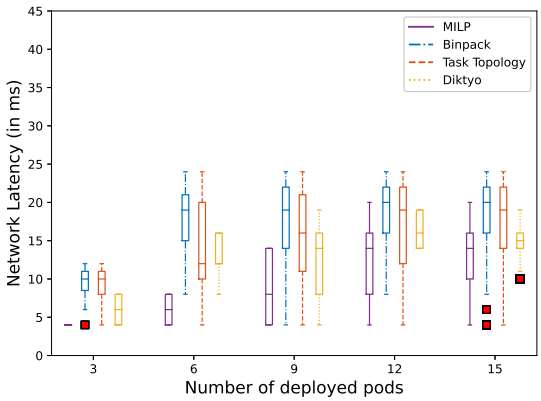
(a) Execution time (Cluster).



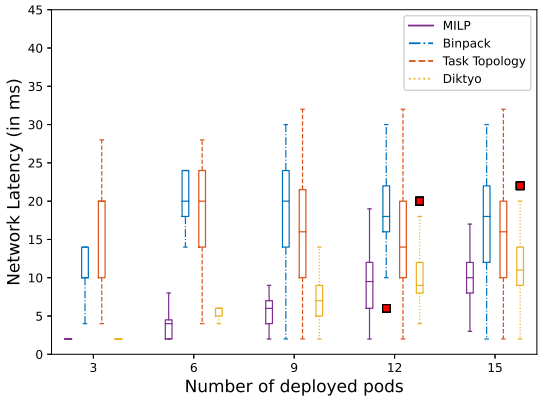
(b) Execution time (MR).

Fig. 9: The MILP model requires 12 and 30 minutes for 15 pods in the cluster and MR infrastructures for the basic application.

the cluster topology and the MR scenario. The MILP model provides an optimal solution that typically cannot be applied in practice, as its execution time increases significantly as



(a) Network latency (Cluster).

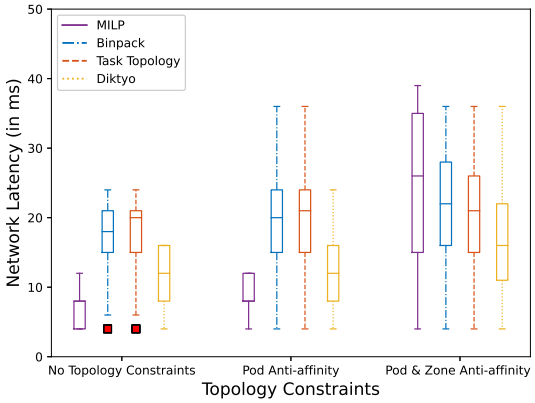


(b) Network latency (MR).

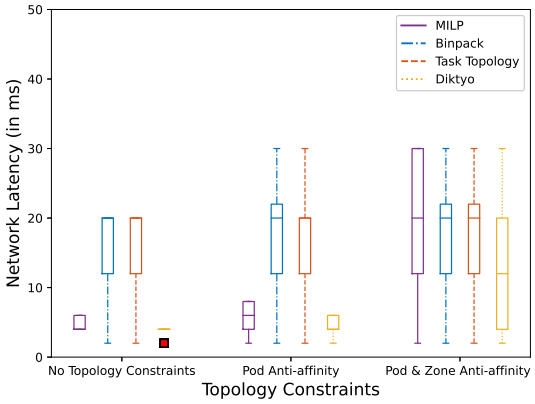
Fig. 10: The Diktyo framework reduces the expected network latency by up to 34% compared to existing plugin algorithms.

the number of pods in the application increase. As it is not acceptable to run scheduling methods for a long time in production systems, a 30-minute limitation has been added to the MILP model. Though the execution time of all algorithms increases due to the increasing number of pods, the evaluated heuristic-based algorithms only increase the execution time slightly compared to the MILP model. The MILP model requires 12 minutes and over 30 minutes to place 15 pods for the Basic application on the cluster and MR topologies, respectively. In contrast, Diktyo requires only 0.01 seconds for both topologies, similar to the execution times of the Binpack and Task Topology algorithms. This occurs because all network weights are pre-calculated beforehand, and no recalculation occurs in the Diktyo plugins. In addition, Diktyo considerably reduces the expected network latency compared to the Binpack and Task Topology algorithms, as shown in (Fig. 10a and Fig. 10b). It achieves reductions of up to 34% and 26% for the MR scenario with 15 pods compared to Binpack and Task Topology, respectively. Compared to the optimal solution given by the MILP model, Diktyo only increases the average network latency by 20% and 15% for 9 and 15 pods, respectively. However, it provides a scalable solution due to its lower execution time.

Fig. 11 evaluates the network latency in box plots for the Redis Cluster application under different topology constraints.

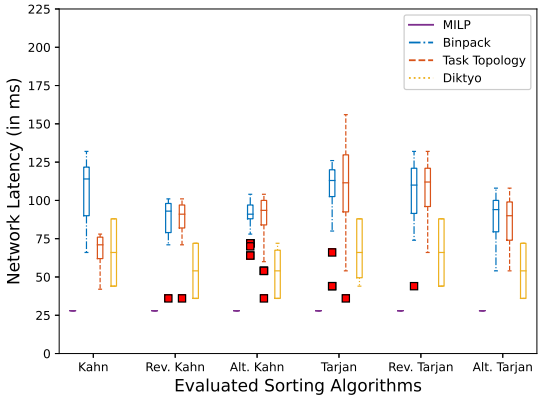


(a) Network Latency (Cluster).

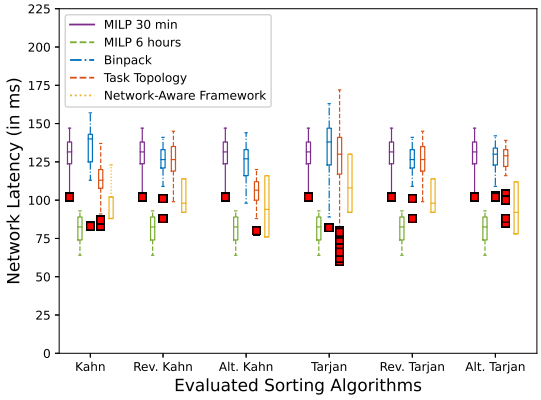


(b) Network Latency (DC).

Fig. 11: Deployment of the Redis cluster application (12 pods) with different topology constraints.



(a) Network Latency (11 pods).

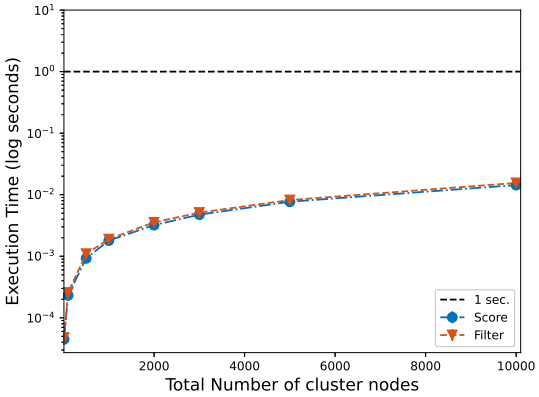


(b) Network Latency (22 pods).

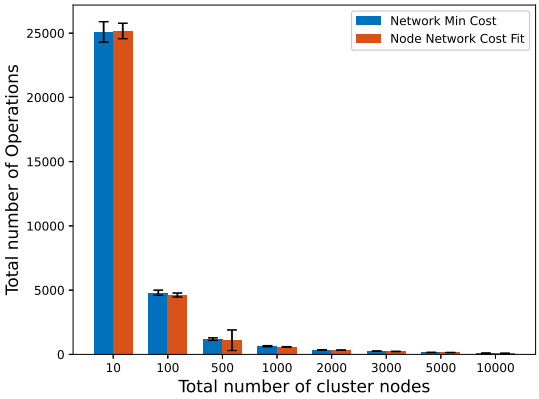
Fig. 12: Deployment of the Online Boutique application (cluster topology) for different sorting algorithms.

The zone/region anti-affinity constraints are usually applied to provide high availability for Redis clusters. No particular topology sorting algorithm is applied in this scenario since all pods depend on all others in the Redis Cluster application. As shown, latency reductions by the Diktyo framework are more noticeable when pod and zone anti-affinity rules are not considered. Diktyo reduces the latency up to 34% in the cluster (Fig. 11a) and up to 77% in the DC topology (Fig. 11b) compared to Binpack and Task Topology. Compared to the MILP model, Diktyo increases the average network latency by 37% in the cluster topology and reduces it by 22% in the DC topology as the MILP model fails to find the optimal solution within 30 minutes. Furthermore, even with additional topology constraints, Diktyo can still reduce the latency by up to 20% in the cluster topology and by up to 38% in the DC topology compared to Binpack and Task Topology algorithms. The MILP model cannot find the optimal solution for both scenarios and obtains similar results to the Binpack algorithm due to the increased complexity of additional anti-affinity constraints.

Fig. 12 evaluates the impact of different topology sorting algorithms for scheduling the Online Boutique application. The first scenario (Fig. 12a) considers 11 pods, in which the application's network latency has only one possible path since there is only one replica per pod type. As expected, the MILP model obtains the lowest latency (28 ms) while Diktyo produces a placement scheme 50% worse on average when combined with Alt. Kahn and Alt. Tarjan sorting algorithms. Nevertheless, Diktyo can still reduce the latency up to 34% on average compared to Binpack and Task Topology algorithms. The second scenario (Fig. 12b) considers 22 pods, in which several paths are possible due to two replicas per pod type. Two allocation schemes are obtained from the MILP model: the first experiment runs the model for up to 30 minutes and the second for up to 6 hours to search for the optimal placement scheme. However, the MILP model cannot find the optimal solution within 6 hours due to the highly complex pod dependency graph for the Online Boutique application with 22 pods. Diktyo reduces the network latency by up to 30% compared to the MILP 30-minute model. Also, it significantly reduces the expected latency compared to Binpack and Task Topology algorithms. It only increases the latency by up to 13% compared to the MILP 6-hour model. Furthermore, Diktyo provides a more scalable solution as it solves the scenario on average in about 0.015 seconds. Concerning sorting, Diktyo achieves lower latency for Kahn, Alt. Kahn and Alt. Tarjan for the Online Boutique application. Based on our experiments, Diktyo plugins produce deployment schemes with lower latency for the alternate versions of the sorting algorithms since pods with several dependencies are

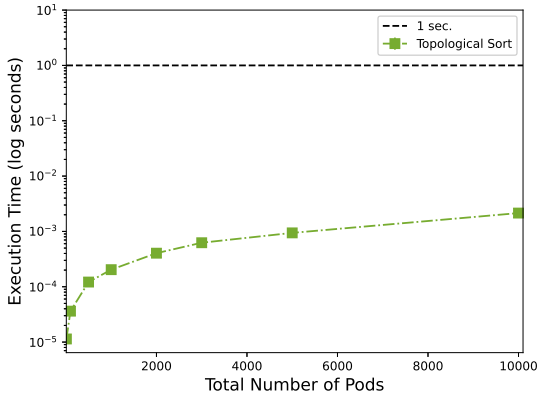


(a) Execution Time.

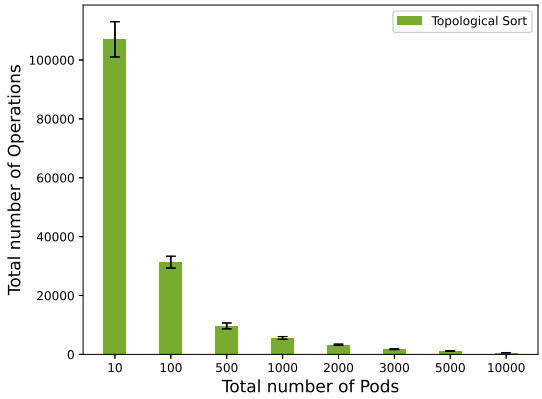


(b) Number of Operations.

Fig. 13: The benchmark shows that the execution time of both plugins (Filter and Score) increases logarithmically over the number of nodes.



(a) Execution Time.



(b) Number of Operations.

Fig. 14: Benchmark of the QueueSort plugin. The execution time increases logarithmically over the number of pods.

intercalated with pods with few dependencies.

In conclusion, the results show that Diktyo can significantly reduce the expected network latency in K8s clusters. The performance of Diktyo has been compared against two available scheduling algorithms, Binpack and Task Topology, showing that these plugins are not latency-aware. The MILP model shows that finding the optimal network-aware placement scheme is a highly complex optimization problem, requiring an unacceptable execution time. Thus, MILP methods are useful benchmarks but cannot be deployable in production systems. The Diktyo framework achieves similar execution times to Binpack and Task Topology by pre-calculating all network weights beforehand. Diktyo obtains sub-optimal solutions on average 10% to 30% worse than the MILP model but provides a significantly scalable method.

B. Plugin Evaluation

The Go [91] testing package provides an integration testing utility that can benchmark the performance of the Diktyo scheduling plugins. Integration tests have been implemented to assess the scalability of all plugins, including *NodeNetworkCostFit* and *NetworkMinCost*. Fig. 13 shows the plugins' execution time over the number of cluster nodes. 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. Thus, the following output: 25093 - 44.8 ns/op means that the operation (i.e., plugin function) executed for 25093 times at a speed of 44.8 ns per operation. Though the execution time of both plugins increases logarithmically over the number of nodes, the execution time for 10000 nodes is still below 1 second. It confirms that accessing network weights via CRs does not add significant overhead in terms of execution time to the K8s scheduling process. Also, the *TopologicalSort* plugin has been evaluated based on the number of pods in the sort queue as shown in Fig. 14. Results indicate a similar pattern (i.e., logarithmic time) as the other two plugins. This benchmark highlights that the plugins designed for the Diktyo framework do not introduce significant overhead over the scheduling process and are scalable for clusters with 10000 nodes/pods.

C. Testbed Evaluation

1) Testbed Infrastructure: A K8s cluster is set up using Kubeadm [92] on VMs created with IBM Cloud [93]. It consists of 16 nodes (1 master and 15 workers), each labelled with region (i.e., *topology.kubernetes.io/region*) and zone (i.e., *topology.kubernetes.io/zone*) labels. Since the cluster belongs to a single region in the IBM cloud, varying delays are emulated on network connections via Traffic Control (TC)

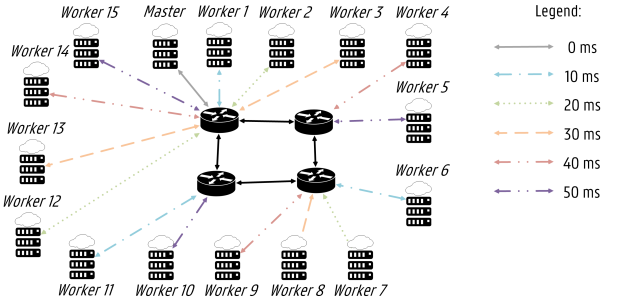


Fig. 15: Illustration of the testbed infrastructure.

TABLE VIII: Software Versions of the Testbed Infrastructure.

Software	Version
Kubeadm / Kubectl	v1.22.4
Go	go1.16.10
Docker	docker://20.10.10
Linux Kernel	5.4.0-80-generic
Operating System	Ubuntu 20.04.2 LTS

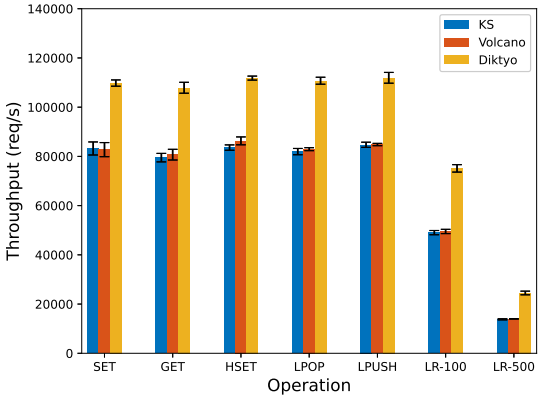
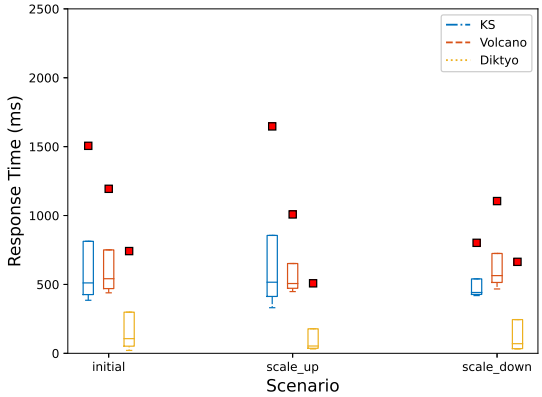


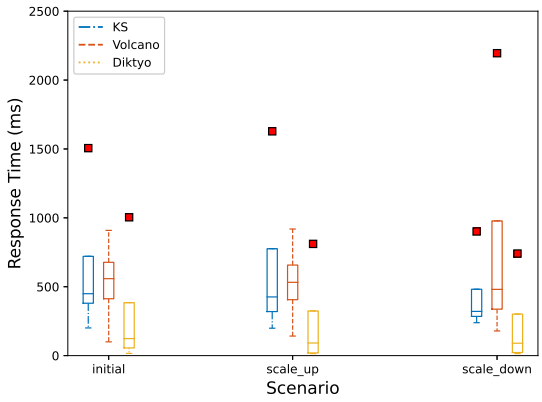
Fig. 16: Diktyo increases the throughput of the Redis Cluster application by up to 22% for most operations.

[94] to assess a complex infrastructure concerning low-latency container allocation. Network connections between VMs are illustrated in Fig. 15. In the considered scenario, all nodes belong to the same region (i.e., r1), but each node is in a different zone (i.e., master, z1, .., z15). These topology labels are important to evaluate the scheduling behavior of Diktyo. The netperf component is deployed to estimate the latency between nodes in the cluster and caches the measurements in a configmap object. Then, the networkTopology controller calculates the network weights between zones based on the measurements in the configmap. Thus, the Diktyo plugins can apply accurate network weights instead of manually defined weights. Table VIII lists the software versions of all the components used to set up the K8s cluster.

2) Testbed Results: The considered applications represent typical and often used cloud-native applications requiring high bandwidth (Redis Cluster - Fig. 1a) or low latency (Online Boutique - Fig. 1b). These applications are often applied in microservice research in academic and industry settings (e.g., [95]). The first experiment consists in deploying the Redis



(a) Diktyo reduces the response time by at least 60% for *GET* requests compared to KS and Volcano.



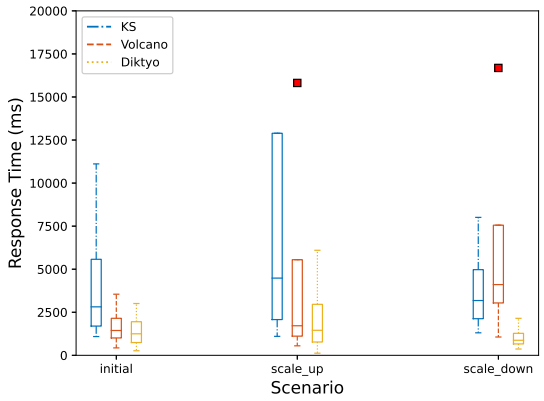
(b) Diktyo reduces the response time on average by 40% for *GET /cart* requests. A higher reduction is achieved compared to the KS.

Fig. 17: The response time of the Online Boutique application for *GET* requests deployed with different schedulers.

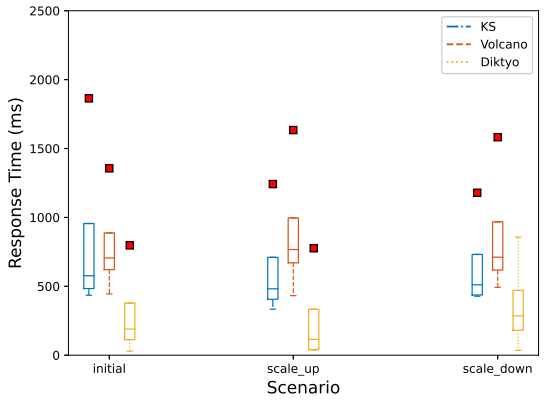
TABLE IX: The resource consumption of the different scheduling mechanisms.

Scheduler	CPU usage (in millicpu)	Memory usage (in MiB)
KS	9.41m	83.5 MiB
Volcano	76.6m	72.0 MiB
Diktyo	6.19m	82.7 MiB

Cluster application with different schedulers. The goal is to assess the performance of the Redis cluster application when scheduled with the Diktyo framework. The K8s deployment consisted of five master pods and five slave pods. The Redis-benchmark utility [96] has been applied to generate a total of 250K database queries from 50 emulated clients. Fig. 16 shows the throughput obtained with different schedulers for several database operations. The Diktyo framework achieves higher throughput on average by 22% compared to KS and Volcano since it deploys master and slave pods close to each other based on the dependencies established in the Redis AppGroup CR [97]. With the proper specification of pod dependencies, Diktyo can produce an optimized low-latency placement for typical database applications than KS and Volcano, leading to higher throughput for various database operations.



(a) Diktyo considerably reduces the response time, achieving at least 50% reductions for *POST /cart/checkout* requests in the *ScaleUp* phase.



(b) Diktyo reduces the response time on average by 45% for *POST /setCurrency* requests.

Fig. 18: The response time of the Online Boutique application for *POST* requests deployed with different schedulers.

The second experiment consists of the deployment of the Online Boutique application. The goal is to evaluate the performance of the Online Boutique application when scheduled with Diktyo concerning the application's response time. Also, this scenario considers pod terminations and rescheduling, showing how Diktyo finds near-optimal schemes even in these dynamic conditions. The declarative nature of K8s makes rescheduling a simple task. If failures happen, Diktyo deploys new pod instances as new pods requiring allocation. In addition, dynamic changes (i.e., latency and bandwidth) are detected via the networkTopology Controller since violations are observed based on CRs. Surpassing thresholds evicts pods, making Diktyo schedule new pod instances that satisfy these thresholds. An Online Boutique AppGroup [98] composed with several pod dependencies has been submitted to the K8s cluster, ensuring Diktyo has the needed pod dependency graph to find a near-optimal pod placement scheme. A load generator based on the locust load tool [99] has been used to assess the performance via different *GET* and *POST* requests. The experiment consists of three scenarios: the *Initial* phase corresponds to the deployment of one pod instance per workload, then in the *ScaleUp* scenario, all workloads are scaled up to four replicas, and lastly, in the *ScaleDown* phase, all workloads are scaled down, resulting on the termination of two instances per workload. Diktyo reduces the Online Boutique response time for most requests, as shown in Fig. 17 and Fig. 18. The red squares represent outliers present in the experimental runs. Diktyo reduces the response time by at least 45% on average. Diktyo obtains even higher reductions for the *GET* and *POST /cart/checkout* requests. Volcano achieves slightly lower latencies compared to KS, the default scheduler in K8s. This result highlights the main advantage of the Diktyo framework since it allocates pods with established dependencies close to each other, resulting in lower latency.

Regarding resource consumption, Diktyo performs similarly to KS as shown in Table IX. The components developed for the Diktyo framework are fully integrated with the K8s ecosystem since the development focuses on the available scheduling plugins framework, thus achieving similar CPU and memory usage compared to the KS. In fact, the CPU usage is slightly lower since the number of enabled plugins in Diktyo is smaller than the default configuration of KS since Diktyo consists mainly of the three presented scheduling plugins (i.e., TopologicalSort, NodeNetworkCostFit, and NetworkMinCost). In contrast, Volcano requires higher CPU resources but needs slightly lower memory usage.

In summary, the achieved results show the benefits of the proposed Diktyo framework. By specifying pod dependencies in K8s clusters, scheduling algorithms can make informed decisions regarding latency and bandwidth. Developers need to know their application dependencies to define their AppGroup properly. With proper application and network topology information, the proposed Diktyo framework produces a near-optimal placement scheme aiming to minimize the network latency concerning complex application and topology constraints in logarithmic time. The combination of the three developed plugins for the Diktyo framework approximates the optimal solution given by the MILP model. However, Diktyo does not guarantee a tolerance regarding the MILP since it depends on the status of the cluster. The experiments show that Diktyo can increase database throughput and reduce the response time for typical web-based applications in a distributed K8s cluster compared to the existing KS and Volcano schedulers.

VII. Conclusions

This paper presents a network-aware scheduling approach for the K8s platform based on its recent scheduling plugin architecture. The aim is to tackle the challenge of designing and implementing scalable and efficient network-aware scheduling algorithms capable of delivering low latency to end-users without compromising the system performance. Most efforts available in the literature require an unacceptable execution time to find proper allocation schemes. Results obtained from detailed experiments and realistic scenarios show the advantages of the Diktyo framework compared to default schedulers. Diktyo can increase the throughput by 22% for typical database applications and reduce the expected response time by 45% in web-based applications. The work is an important step toward efficient application placement in future cloud-native architectures.

ACKNOWLEDGMENTS

The authors would like to thank the Kubernetes sig-scheduling community for their valuable feedback. José Santos is funded by the Research Foundation Flanders (FWO), grant number 1299323N.

REFERENCES

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

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

[3] A. AWS, "Amazon web services," accessed on 28 March 2022. [Online]. Available: https://aws.amazon.com/.

[4] O. Adam, Y. C. Lee, and A. Y. Zomaya, "Stochastic resource provisioning for containerized multi-tier web services in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 2060–2073, 2016.

[5] K. Shafique, B. A. Khawaja, F. Sabir, S. Qazi, and M. Mustaqim, "Internet of things (iot) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5g-iot scenarios," *Ieee Access*, vol. 8, pp. 23 022–23 040, 2020.

[6] X. Li, M. Darwich, M. A. Salehi, and M. Bayoumi, "A survey on cloud-based video streaming services," in *Advances in Computers*. Elsevier, 2021, vol. 123, pp. 193–244.

[7] S. Halfpap and R. Schlosser, "A comparison of allocation algorithms for partially replicated databases," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 2008–2011.

[8] M. Luksa, *Kubernetes in action*. Simon and Schuster, 2017.

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

[10] K. Aziz, D. Zaidouni, and M. Bellafkih, "Leveraging resource management for efficient performance of apache spark," *Journal of Big Data*, vol. 6, no. 1, pp. 1–23, 2019.

[11] Redis, "Redis, an open source in-memory data structure store." accessed on 28 March 2022. [Online]. Available: https://redis.io/.

[12] Y. Mansouri, V. Prokhorenko, and M. A. Babar, "An automated implementation of hybrid cloud for performance evaluation of distributed databases," *Journal of Network and Computer Applications*, vol. 167, p. 102740, 2020.

[13] O. Boutique, "Online boutique, a cloud-native microservices demo application." accessed on 28 March 2022. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo.

[14] L. A. Rocha and F. L. Verdi, "A network-aware optimization for vm placement," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. IEEE, 2015, pp. 619–625.

[15] M. A. Abdelaal, G. A. Ebrahim, and W. R. Anis, "Network-aware resource management strategy in cloud computing environments," in *2016 11th International Conference on Computer Engineering & Systems (ICCES)*. IEEE, 2016, pp. 26–31.

[16] 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.

[17] M. Barshan, H. Moens, S. Latre, B. Volckaert, and F. De Turck, "Algorithms for network-aware application component placement for cloud resource allocation," *Journal of Communications and Networks*, vol. 19, no. 5, pp. 493–508, 2017.

[18] 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.

[19] K. 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.

[20] Kubernetes, "Custom resources," accessed on 28 March 2022. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[21] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Pr for inclusion of the networkaware plugins," accessed on 27 September 2022. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins/pull/432.

[22] A. Santoyo-González and C. Cervelló-Pastor, "Network-aware placement optimization for edge computing infrastructure under 5g," *IEEE access*, vol. 8, pp. 56 015–56 028, 2020.

[23] Y. Wu, W. Zheng, Y. Zhang, and J. Li, "Reliability-aware vnf placement using a probability-based approach," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2478–2491, 2021.

[24] 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.

[25] 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.

[26] A. Beltre, P. Saha, and M. Govindaraju, "Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters," in *2019 IEEE Cloud Summit*. IEEE, 2019, pp. 14–20.

[27] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards delay-aware container-based service function chaining in fog computing," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[28] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.

[29] A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 427–441.

[30] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov, "Inferline: latency-aware provisioning and scaling for prediction serving pipelines," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 477–491.

[31] S. Wang, X. Zhou, L. Zhang, and C. Jiang, "Network-adaptive scheduling of data-intensive parallel jobs with dependencies in clusters," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 155–160.

[32] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 18)*, 2018, pp. 595–610.

[33] Z. Hu, J. Tu, and B. Li, "Spear: Optimized dependency-aware task scheduling with deep reinforcement learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 2037–2046.

[34] A. Chung, S. Krishnan, K. Karanasos, C. Curino, and G. R. Ganger, "Unearthing inter-job dependencies for better cluster scheduling," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 20)*, 2020, pp. 1205–1223.

[35] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 21)*, 2021.

[36] Y. He, W. Cai, P. Zhou, G. Sun, S. Luo, H. Yu, and M. Guizani, "Beamer: Stage-aware coflow scheduling to accelerate hyper-parameter tuning in deep learning clusters," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 1083–1097, 2021.

[37] X. Li, Z. Lian, X. Qin, and W. Jie, "Topology-aware resource allocation for iot services in clouds," *IEEE Access*, vol. 6, pp. 77 880–77 889, 2018.

[38] 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.

[39] D. Yang, D. Cheng, W. Rang, and Y. Wang, "Joint optimization of mapreduce scheduling and network policy in hierarchical data centers," *IEEE Transactions on Cloud Computing*, 2019.

[40] 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.

[41] 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.

[42] V. Rao, V. Singh, K. Goutham, B. U. Kempaiah, R. J. Mampilli, S. Kalambur, and D. Sitaram, "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.

[43] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 431–442, 2014.

[44] C. Gao, H. Wang, L. Zhai, Y. Gao, and S. Yi, "An energy-aware ant colony algorithm for network-aware virtual machine placement in cloud computing," in *2016 IEEE 22nd international conference on parallel and distributed systems (ICPADS)*. IEEE, 2016, pp. 669–676.

[45] R. Wang, J. A. Wickboldt, R. P. Esteves, L. Shi, B. Jennings, and L. Z. Granville, "Using empirical estimates of effective bandwidth in network-aware placement of virtual machines in datacenters," *IEEE Transactions on Network and Service Management*, vol. 13, no. 2, pp. 267–280, 2016.

[46] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin, "Racksched: A microsecond-scale scheduler for rack-scale computers," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 20)*, 2020, pp. 1225–1240.

[47] 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.

[48] G. Shen, Q. Li, W. Shi, F. Han, Y. Jiang, and L. Gu, "Poche: A priority-based flow-aware in-network caching scheme in data center networks," *IEEE Transactions on Network and Service Management*, 2022.

[49] E. Ahvar, S. Ahvar, N. Crespi, J. Garcia-Alfaro, and Z. A. Mann, "Nacer: a network-aware cost-efficient resource allocation method for processing-intensive tasks in distributed clouds," in *2015 IEEE 14th International Symposium on Network Computing and Applications*. IEEE, 2015, pp. 90–97.

[50] J. Darrous, S. Ibrahim, A. C. Zhou, and C. Perez, "Nitro: Network-aware virtual machine image management in geo-distributed clouds," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 553–562.

[51] D. Haja, B. Vass, and L. Toka, "Towards making big data applications network-aware in edge-cloud systems," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*. IEEE, 2019, pp. 1–6.

[52] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[53] C. N. C. F. C. Volcano, "Volcano, a batch system built on kubernetes," accessed on 28 March 2022. [Online]. Available: https://github.com/volcano-sh/volcano.

[54] ——, "The gang plugin," accessed on 28 March 2022. [Online]. Available: https://github.com/volcano-sh/volcano/tree/master/pkg/scheduler/plugins/gang.

[55] ——, "Task topology plugin," accessed on 28 March 2022. [Online]. Available: https://github.com/volcano-sh/volcano/tree/master/pkg/scheduler/plugins/task-topology.

[56] ——, "The binpack plugin," accessed on 28 March 2022. [Online]. Available: https://github.com/volcano-sh/volcano/tree/master/pkg/scheduler/plugins/binpack.

[57] ——, "Dominant resource fairness (drf) plugin," accessed on 28 March 2022. [Online]. Available: https://github.com/volcano-sh/volcano/tree/master/pkg/scheduler/plugins/drf.

[58] B. Ibryam, "Principles of container-based application design," *Redhat Consulting Whitepaper*, 2017.

[59] K. Cosheduling Plugin, "coscheduling plugin implementations based on coscheduling based on podgroup crd." accessed on 28 March 2022. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/coscheduling.

[60] K. Topology-aware Plugin, "Topology-aware scheduler plugin implementations." accessed on 28 March 2022. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/noderesourcetopology.

[61] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A hierarchical data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous multi-accelerator numa nodes," *IEEE Access*, vol. 8, pp. 7861–7876, 2019.

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

[63] 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.

[64] A. Haider, R. Potter, and A. Nakao, "Challenges in resource allocation in network virtualization," in *20th ITC specialist seminar*, vol. 18, no. 2009. ITC, 2009.

[65] C. D'Ambrosio, A. Lodi, and S. Martello, "Piecewise linear approximation of functions of two variables in milp models," *Operations Research Letters*, vol. 38, no. 1, pp. 39–46, 2010.

[66] Kubernetes, "Assigning pods to nodes," accessed on 28 March 2022. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/.

[67] ——, "Advertise extended resources for a node," accessed on 28 August 2022. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/extended-resource-node/.

[68] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Bandwidth resources advertisement via extended resources," accessed on 28 August 2022. [Online]. Available: https://anonymous.4open.science/r/bandwidth-component-3DDF/README.md.

[69] Kubernetes, "Network plugins," accessed on 28 August 2022. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.

[70] X. Fan, B. Lang, Y. Zhou, and T. Zang, "Adding network bandwidth resource management to hadoop yarn," in *2017 seventh international conference on information science and technology (ICIST)*. IEEE, 2017, pp. 444–449.

[71] J. Santos, C. Wang, T. Wauters, and F. De Turck, "App group crd yaml file," accessed on 28 August 2022. [Online]. Available: https://github.com/jpedro1992/scheduler-plugins/blob/KepDevWithNTController/manifests/appgroup/crd.yaml.

[72] Kubernetes, "Service." accessed on 28 March 2022. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/.

[73] J. Santos, C. Wang, T. Wauters, and F. De Turck, "App group controller," accessed on 28 August 2022. [Online]. Available: https://github.com/diktyo-io/appgroup-controller.

[74] 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.

[75] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[76] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[77] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Network topology crd yaml file," accessed on 28 August 2022. [Online]. Available: https://github.com/jpedro1992/scheduler-plugins/blob/KepDevWithNTController/manifests/networktopology/crd.yaml.

[78] ——, "The netperf component for the diktyo framework," accessed on 28 August 2022. [Online]. Available: https://github.com/jpedro1992/pushing-netperf-metrics-to-prometheus/tree/configmap.

[79] Netperf, "a benchmark to measure the performance of many different types of networking. it provides tests for both unidirectional throughput, and end-to-end latency." accessed on 28 March 2022. [Online]. Available: https://github.com/HewlettPackard/netperf.

[80] Kubernetes, "Configmaps," accessed on 28 August 2022. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/configmap/.

[81] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Network topology controller," accessed on 28 August 2022. [Online]. Available: https://github.com/diktyo-io/networktopology-controller.

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

[83] M. Hausenblas and S. Schimanski, *Programming Kubernetes: Developing Cloud-Native Applications*. " O'Reilly Media, Inc.", 2019.

[84] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Network topology cr example file," accessed on 28 August 2022. [Online]. Available: https://github.com/jpedro1992/scheduler-plugins/blob/KepDevWithNTController/manifests/networktopology/example.yaml.

[85] ——, "The diktyo scheduling plugins," accessed on 28 August 2022. [Online]. Available: https://github.com/jpedro1992/scheduler-plugins/tree/KepDevWithNTController/pkg/networkaware.

[86] Kubernetes, "the qos plugin sorts pods by .spec.priority and breaks ties by the quality of service class." accessed on 28 March 2022. [Online]. Available: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/qos.

[87] ——, "Core v1 pod api." accessed on 28 March 2022. [Online]. Available: https://github.com/kubernetes/client-go/blob/master/listers/core/v1/pod.go.

[88] A. EC2, "Amazon ec2 on-demand pricing," accessed on 28 March 2022. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/.

[89] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.
[90] I. ILOG, "Ibm cplex ilog optimization studio," accessed on 28 March 2022. [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio.
[91] A. A. Donovan and B. W. Kernighan, *The Go programming language*. Addison-Wesley Professional, 2015.
[92] Kubernetes, "Overview of kubeadm," accessed on 28 March 2022. [Online]. Available: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/.
[93] I. Cloud, "Hybrid. open. resilient. your platform and partner for digital transformation." accessed on 28 March 2022. [Online]. Available: https://www.ibm.com/cloud.
[94] A. N. Kuznetsov, "tc(8) — linux manual page." accessed on 28 March 2022. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html.
[95] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 805–825.
[96] Redis, "How fast is redis?" accessed on 28 March 2022. [Online]. Available: https://redis.io/topics/benchmarks.
[97] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Redis app group crd yaml file," accessed on 1 September 2022. [Online]. Available: https://github.com/jpedro1992/scheduler-plugins/blob/kepDev/manifests/appgroup/redis-appGroup-example.yaml.
[98] ——, "Online boutique app group crd yaml file," accessed on 28 March 2022. [Online]. Available: https://github.com/jpedro1992/scheduler-plugins/blob/kepDev/manifests/appgroup/onlineBoutique-appGroup-example.yaml.
[99] Locust, "An open source load testing tool." accessed on 28 March 2022. [Online]. Available: https://locust.io/.

José Santos obtained his M.Sc. degree in Electrical and Computers Engineering in July 2015 from the University of Porto, Portugal. Recently, he completed his doctoral studies at Ghent University in April 2022. He is currently a Postdoctoral Researcher in the Internet Technology and Data Science Lab (IDLab) Research Group at Ghent University - imec, Belgium. His research interests include Cloud and Fog Computing, IoT, Service Function Chaining, and Reinforcement Learning. His work has been published in more than 20 scientific publications.

Chen Wang obtained her M.S.c and Ph.D. degrees in Electrical and Computer Engineering from Carnegie Mellon University (CMU) in 2014 and 2017, respectively. Since 2017, she has worked as a Research Staff Member at IBM Thomas J. Watson Research Center. Her research interests include Cloud video streaming systems, Cloud resource management, Container Cloud platforms, Serverless Computing, Machine Learning/Data Analytics systems, and data-driven cloud system management, with a special focus on machine learning approaches. She authored and coauthored 20+ papers, and served as co-chair for ACM International Workshop on Containers, ACM Middleware Industrial Track, IEEE CloudCom, etc. She is also an open source advocate, actively contributing to projects in Linux Foundation, Cloud Native Foundation, and Kubernetes Ecosystems.

Tim Wauters received the M.Sc. and Ph.D. degrees in electro-technical engineering from Ghent University, in 2001 and 2007, respectively. He has been working as a Postdoctoral Fellow of F.W.O.-V. with the Department of Information Technology (INTEC), Ghent University. He is currently active as a Senior Researcher at imec. His work has been published in more than 150 scientific publications. His research interests include design and management of networked services, covering multimedia distribution, cybersecurity, big data, and smart cities.

Filip De Turck leads the network and service management research group at Ghent University, Belgium and imec. He (co-) authored over 700 peer reviewed papers and his research interests include design of efficient softwarized network and cloud systems. He is involved in several research projects with industry and academia, served as chair of the IEEE Technical Committee on Network Operations and Management (CNOM), and serves as a steering committee member of the IM, NOMS, CNSM and NetSoft conferences. Prof. Filip De Turck served as Editor-in-Chief of IEEE Transactions on Network and Service Management (TNSM), and was named an IEEE Fellow since 2021.