Towards Intent-based Scheduling for Performance and Security in Edge-to-Cloud Networks

José Santos*, Eddy Truyen†, Christoph Baumann‡, Filip De Turck*, Gerald Budigiri†, Wouter Joosen †
* IDLab, Department of Information Technology at Ghent University - imec, 9000 Ghent, Belgium
Email: {josepedro.pereiradossantos, filip.deturck}@UGent.be
† KU Leuven, imec-DistriNet, Leuven, Belgium - Email: {eddy.truyen, gerald.budigiri, wouter.joosen}@kuleuven.be
‡ Ericsson Security Research Stockholm, Sweden - Email: christoph.baumann@ericsson.com

***Abstract*—Telecom vendors have adopted containerization to improve their software for heterogeneous edge-to-cloud networks. Container orchestration platforms face however challenges when adapting to the dynamic demands of multi-tenancy, often leading to performance and security conflicts between Cluster Administrators (CAs) and applications administrators. In this paper, we propose that network security and performance constraints should be directly integrated into the scheduler. By dynamically determining preferred network segments for each application, this approach will reduce the network attack surface, enhance resource utilization, and ensure improved performance. The presented prototype and evaluation is just a first step. A comprehensive solution must also support an intent-based interface to the scheduler that simplifies expression of cluster node segmentation constraints and thus frees the CA from node-level management. We outline the vision for such an intent-based interface and emulate it by means of traditional security groups.**

***Index Terms*—Security, Intent-based Scheduling, Kubernetes, Containers, Cloud Computing**

I. Introduction

5G networks support ultra-reliable low latency communication (URLLC) applications through coupling with the technologies of edge computing and containerization, provided that plugged-in container network solutions are properly configured with eBPF [1] and direct routing of container network traffic within L3 subnets [2]. Telecom vendors have largely adopted Kubernetes™ (K8s), the de-facto standard in container orchestration, to increase the portability of their software across heterogeneous edge-to-cloud infrastructures. In comparison to more traditional cluster installations, telco-managed clusters, require a very strict separation of duties between Cluster Administrators (CAs), who manage the cluster nodes – bare-metal machines or Virtual Machines (VMs) in the cloud, and application administrators who manage Pods, which is the unit of deployment in K8s encapsulating one or more containers.

A challenge with K8s is that non-functional requirements such as security and performance are implemented by multiple system layers in the stack (i.e., cloud, clusters, Pods, and application code). As such, performance and security misconfigurations arise due to inconsistencies between different system layers. This problem manifests itself most profoundly at the network-level where the attained network bandwidth and latency between two Pods is not only controlled by the application code and networking stack of the Pods, but also depends on the bandwidth and latency between the cluster nodes, which in turn may depend on the configuration and physical equipment of the underlying service provider. Similarly, the network firewalls configured at Pod level depend also on a consistent configuration of the firewall rules for the cluster nodes. For example, Fig. 1 illustrates the relation between the deployment units of a K8s cluster on top of Amazon Web Services. If a network policy is added to the Pod layer, security groups on the VM layer may conflict with that network policy.

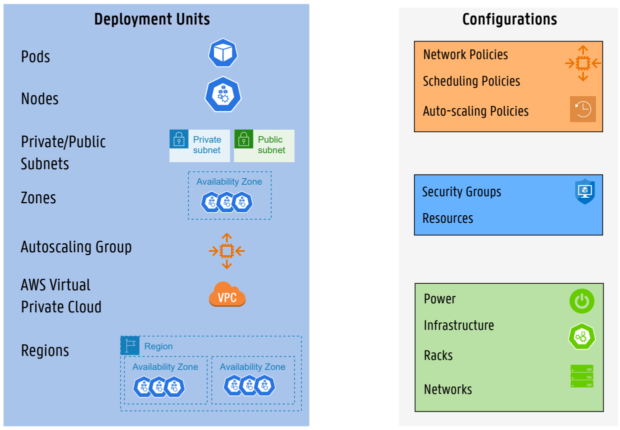

Fig. 1: Typical relations between deployment units and network configurations in cloud-native cloud stacks.

Such configuration inconsistencies are especially difficult to solve in K8s clusters where application administrators and CAs have separate duties, as is the case for telco service providers. For instance, while the former want the best (network) performance for their workloads and also potentially exclusive access to certain compute infrastructure and accelerators, the latter are mainly concerned with optimizing the usage of hardware resources and ensuring different tenants are properly isolated. Naturally, CAs strive to honor service level agreements with their customers, but they also need to protect the clusters from potentially malicious tenants and thus may want to restrict accumulation of privileged workloads on a single node [3] or restrict the possibility for lateral movement of attackers by least privilege network firewall rules [1], [4]. Any kind of network segmentation solution must take both parties' requirements into account and attempt to reconcile them optimally. Given that network segmentation and Pod-to-node requirements may also change dynamically, e.g., by scaling existing or deploying new applications, as well as adding new tenants to a cluster, this becomes a non-trivial problem for container orchestrators like K8s.

This paper argues that the management of such configuration and requirement conflicts is best left in the hands of the *scheduler*. As a central component controlling the scheduling of deployment units (e.g. Pods) within the underlying layer's deployment units (e.g. nodes), the scheduler has the power to dynamically define network segments according to CA security intentions, while also trying to align performance metrics with the needs of the application being scheduled, deploying inter-dependent Pods on nodes that belong to the same network segment. However, the scheduler alone cannot prevent inter-layer misconfigurations or network security vulnerabilities. We propose to add a VM-level security policy manager that automatically configures the VM network segments in a consistent, least-privilege fashion, reacting to the deployment of K8s application Pods and their network policies, but respecting the CA segmentation requirements. Remaining conflicts on this level may then be escalated back to the scheduler, e.g., to reschedule any offending Pods.

Currently the CA perspective is supported by a limited number of low-level mechanisms that do not scale for many cluster nodes (see Section II for an essential critique). Existing state-of-the-art platforms for micro-segmentation offer Application Program Interfaces (APIs) for detecting conflicts between policies of different layers. However, such API-driven policy conflict management is tedious and error-prone and impedes Pod readiness.

This work aims to take first steps into improving the state of the art in this respect. In particular, we:

- highlight the problem of performance and security conflicts that arise in multi-tenant cloud platforms between application and CAs.
- propose an intent-based scheduling solution that enables CAs to describe segmentation requirements in a simple and versatile manner.
- present a prototype integrating the Diktyo [5] scheduler and the GrassHopper (GH) [2] security policy manager, and provide obtained evaluation results validating the viability of the approach.

The remainder of the paper is organized as follows: Sec. II details the motives behind the proposed intent-based framework. Sec. III discusses the state-of-the-art on container security techniques. Sec. IV presents the intent-based framework, and its main components. Sec. V describes the evaluation methodology, followed by the results in Sec. VI. Lastly, Sec. VII concludes this paper and addresses future directions.

II. Background & Motivation

It is a common notion that application developers may want to fine-tune the scheduling of their Pods on a K8s cluster. Requirements may include scheduling a given Pod only on nodes that have certain resources, e.g., Graphics Processing Units (GPUs) and other accelerators, or avoiding certain nodes, e.g., due to their geographic location and data protection concerns. Similarly, developers may want to schedule Pods on the same node to reduce latency, or different replicas of the same Pod on different nodes to strengthen availability. K8s offers several mechanisms for fine-tuning its scheduler in multiple ways:

- *Node selectors* allow to schedule Pods only on nodes with certain node labels;
- *Node (anti-)affinity* rules allow to attract or repel Pods to or from nodes based on rules expressed as logical or arithmetic conditions on node labels;
- *Taints* can be added to nodes to disallow scheduling Pods on them unless the latter have been marked with a corresponding *toleration*;
- *Pod (anti-)affinity* rules allow to specify that Pods with certain Pod labels must either always or never be scheduled on the same node.

Thus an app developer can, for example, require scheduling of Pods on nodes with GPUs by adding node selectors for the corresponding GPU support node label to the Pod configuration. Moreover, different Pods can be forced on the same node by adding Pod affinity rules to both Pod's configurations. Nevertheless, app developers are not the only actors with requirements for the scheduler. CAs may have additional business or security-related scheduling constraints such as:

- constraining tenant Pods to certain regions or node types.
- scheduling Pods from different tenants on different nodes.
- scheduling Pods with different privilege levels on different nodes.
- scheduling Pods from different tenants on different isolated network segments.

Unfortunately, such constraints are not natively supported in K8s. In some cases, these constraints may be implemented by existing means, e.g., using a static labeling of nodes and an admission controller that adds Pod labels and scheduling constraints to the configurations of scheduled Pods. Nevertheless, such bespoke solutions have a number of drawbacks.

Most prominently, segmentation approaches based on node labels are static. For certain use cases, e.g., restricting Pod placement to certain geographies or node types, this may be sufficient, but a static segmentation of nodes for multi-tenancy may not adapt easily to the dynamic scaling of tenants. Moreover, restricting the scheduler in this way may undermine its other objectives w.r.t. cost, performance, etc., leading to sub-optimal Pod placements. Any bespoke solution will necessarily add complexity and be a source for vulnerabilities, not to mention additional overheads of the actual scheduling process. Finally, using scheduling constraints alone, there is no way to enforce network isolation, e.g., between tenants: malicious application admins may attempt to counteract any attempts of the CA to implement multi-tenancy in a cluster, e.g., by manipulating labels and affinity rules in Pod configurations or adding compromising network policies.

We propose that Pod network segmentation constraints should be supported by the scheduler via an intent-based interface to the CA. Ideally, the CA should not worry, for instance in a multi-tenancy scenario, which exact nodes are occupied by Pods of one tenant, as long as the Pods from all other tenants are scheduled on different nodes. In the same way, for network segmentation, the exact set of nodes within each segment should be determined dynamically by the scheduler, taking into account all other application requirements, and the CA should only specify which segments are needed, e.g., by identifying associated node names, generic Pod labels, container privileges with respect to access to the operating system or K8s API, or tenant namespaces. Finally, while network policies may allow the communication between Pods, there should also be a way to restrict the communication between network segments in a way that cannot be overruled by application admins.

III. Related Work

Microsegmentation involves the use of fine-grained and distributed firewall rules to restrict the lateral movement of an attacker within cloud-based applications, similar to K8s network policies. The major microsegmentation platforms today, such as Cisco ACI [6] and VMware NSX [4], support integration with K8s through Container Networking Interface (CNI) network plugins. Illumio's Core [7] installs a separate agent on each K8s node, yet not integrated with CNI. Overall these platforms have certain limitations:

- Microsegmentation platforms segregate VM-based and K8s-based firewall rules in distinct administrative domains. Lack of intent-based and automated coordination between these layers results in security/performance inconsistencies, detectable only by combining APIs for these domains. Performing accurate and comprehensive consistency checks is a challenging and error-prone task.
- Chaining multiple consistency checks can significantly delay Pod-readiness. For instance, Illumio's policy convergence controller has a configurable delay (i.e., 0 to 300 seconds) with a default of 15 seconds [7].
- The agent-based approach of Illumio enforces VM-level security policies within VMs rather than at the VM-network level with security groups.

Security-oriented scheduling aims to enhance the security posture of containerized applications within a cloud environment by selecting secured nodes for container placement. Containers offer high degrees of flexibility and scalability, but they also introduce unique security challenges. K8s policies [8] such as *ResourceQuotas* and *LimitRanges* are typically applied to restrict the amount of resources that containers can consume. Recently, in [9], the authors implemented a syscall-aware scheduler to improve container security. Their scheduler reduces extraneous system calls by up to 2x compared to the default scheduler, while also reducing overall host attack surface by 20%. However, the solution focuses only on one aspect of platform security, leaving other concerns of the CA out of scope.

IV. Intent-based scheduler architecture

This section presents the vision for a scheduling extension in K8s designed to prevent security conflicts by strategically placing Pods on nodes that align with all CA preferences. Specifically, Pods from different applications are scheduled across node segments based on the preferences set by the CA. Moreover, the objective is to consistently deploy Pods of the same application in a segment that meets the performance requirements of that application. Fig. 2 provides an overview of the envisioned architecture and its main components.

The Diktyo network-aware scheduler [5] determines the efficient placement of dependent microservices in long-running applications focused on minimizing end-to-end latency and ensuring bandwidth reservations. Diktyo proposes AppGroup (AG) and NetworkTopology (NT) Custom Resource Definitions (CRDs) to record desired network performance between microservices, and actual network performance between nodes, respectively. In addition, a **Security policy manager** based on GrassHopper (GH) [2] verifies that network policies within microservice-based applications are free from internal conflicts or excessive permissions and enforces the resulting network segmentation on the VM-level via dynamic security group configuration. In cases where such network policies are absent, they are automatically generated from the dependencies between microservices managed by AG Custom Resources (CRs), ensuring adherence to the principle of the *least-privilege* - allowing only the minimal permissions necessary for a functioning system.

Lastly, the envisioned architecture comprises a **Segmentation CRD** to enable the declarative specification of cluster segments across edge-to-cloud regions and zones, identified by name or label. This allows CAs to specify high-level segmentation constraints via a Segmentation CR instance.

The subsequent subsections elaborate further on these three components of the intent-based scheduling architecture. Note that only the first two components have been implemented.

A. *Extending Kubernetes (K8s) scheduling through Diktyo*

Diktyo manages both application dependencies (i.e., AG CR) and infrastructure network topology (i.e., NT CR) when scheduling Pods in K8s. It provides network-aware scheduling plugins to *filter* out nodes based on AG requirements, and *scores* nodes using network weights to ensure appropriate network latency and bandwidth between dependent Pods. This work extends Diktyo toward security-oriented scheduling by introducing the following functionalities:

- Monitoring the bandwidth/latency and security posture (i.e., existing security group rules) of the K8s cluster network topology.
- AGs specify microservice inter-dependencies as well as latency and bandwidth requirements of each dependency, including additional overall application requirements.
- Diktyo matches AGs to segments, considering segmentation constraints and desired network performance in novel filtering and scoring plugins. Pod-to-node scheduling respects these constraints and the network topology status.

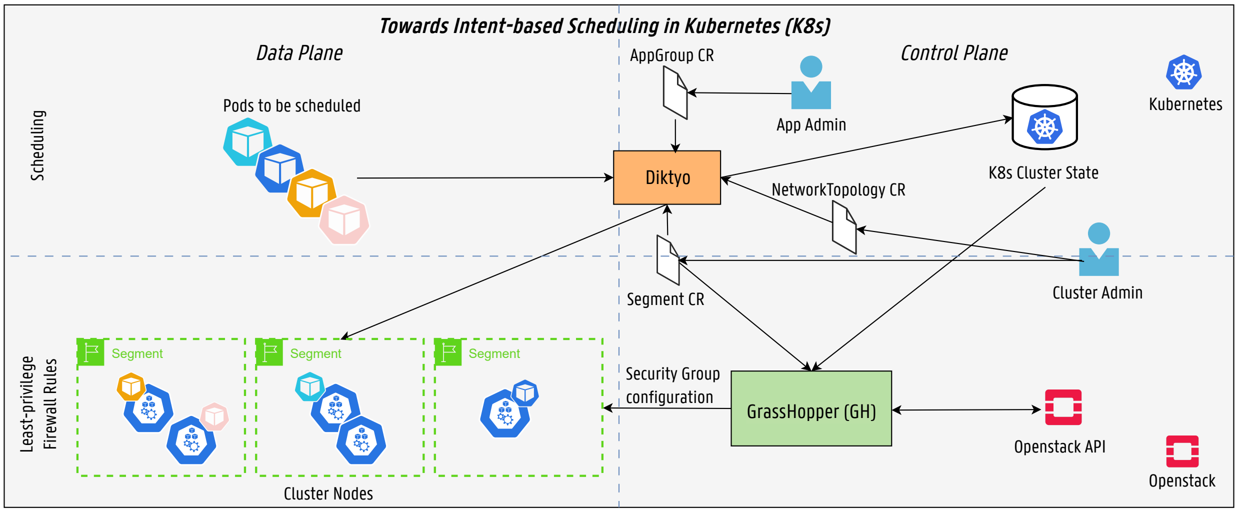

Fig. 2: Schematic representation of the Intent-based Scheduling framework for Kubernetes (K8s).

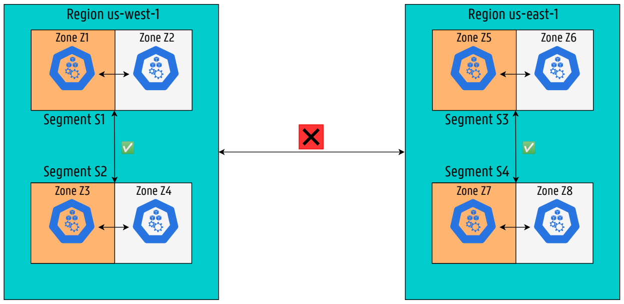

Fig. 3: Illustration of the extended Network Topology (NT).

- Unmet scheduling constraints due to unavailable nodes trigger default node auto-scaling strategies, while other unmet constraints lead to pending Pod notifications.

AppGroup The existing design has been enhanced by adding AG statistics for each microservice and the entire AG to determine the application's overall deployment cost. Diktyo applies these statistics to identify an appropriate segment for Pods of a given application by comparing these with network cost statistics available in the extended NT CR.

NetworkTopology Diktyo monitors available bandwidth and network latency between nodes of a K8s cluster, maintaining this information in the NT CR by using topology keys for identifying zones and regions. All cluster nodes are labeled with these keys by default. The NT CRD has been extended with an additional topology key for node segments (i.e., *topology.kubernetes.io/segment*). The enhanced design supports intra- and inter-origin statistics for the entire network topology, determining the overall capacity of a location (i.e., region, zone, or segment) in the K8s cluster and whether segment communication is allowed as shown in Fig. 3. Diktyo compares inter- and intra- origin statistics with AG statistics to find the optimal node segment for a certain application. Considering security group rules set by the CA, network latency for inter-node connections blocked by these rules is recorded as infinitely high in the NT. From this, segment topology key can be automatically inferred. This extended design addresses various scenarios, such as (1) identifying the optimal segment based on overall AG requirements and current segment capacity, and (2) implementing automatic scaling by adding nodes to a segment when all existing nodes are at full capacity.

B. *GrassHopper (GH): efficient Security policy manager*

GH is designed for URLLC applications, where Pod-level network traffic is routed as-is on the VM-level network to avoid network encapsulation overhead. To allow this Pod traffic on the VM network, GH ensures security group rules at the VM level are added/removed dynamically after Pod scheduling/deletion, but only if verified K8s network policies allow this Pod traffic.

In this work, we have integrated GH with Diktyo so that when an application is deployed across segments, its connection requests are all automatically blocked. As such, there is a second line of defense against tenants' applications that violate segmentation constraints.

C. *Towards a Segmentation CRD*

The Segmentation CRD should have the following APIs:

- **Cluster Segmentation** A segment consists of a number of nodes across edge-to-cloud regions and zones, which can be identified by name or label. Segment identifiers are chosen by the CA, but the exact set of nodes within each segment is determined by the Diktyo scheduler.
- **High-level segmentation constraints** Application selectors (e.g., deployments, Pods) are applied in segmentation constraints. The selectors can be matched by name (e.g., AG, namespaces, Pod labels) or by a computable property (e.g., similar container privileges).
- **(Anti-)Affinity Segmentation preferences** Different kinds of constraints can be added to segment specifications, such as application-to-segment affinity, inter-application segment (anti-)affinity, as well as exceptional

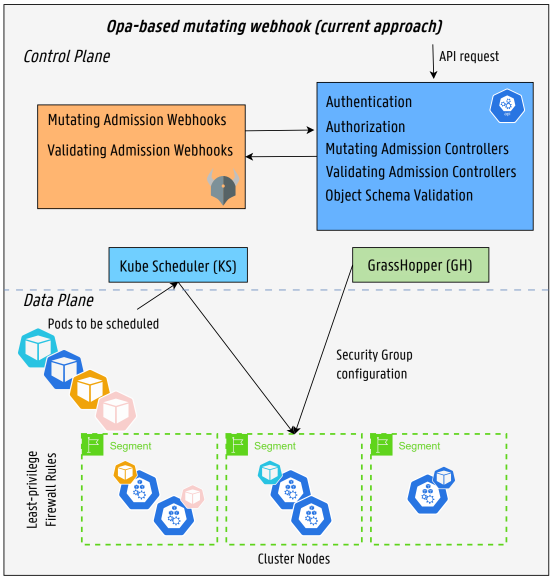
Opa-based mutating webhook (current approach)
Control Plane
API request
Mutating Admission Webhooks
Validating Admission Webhooks
Authentication
Authorization
Mutating Admission Controllers
Validating Admission Controllers
Object Schema Validation
Kube Scheduler (KS)
GrassHopper (GH)
Data Plane
Pods to be scheduled
Security Group configuration
Least-privilege Firewall Rules
Segment
Segment
Segment
Cluster Nodes

Fig. 4: Illustration of the OPA-based scheduling scheme [10].

TABLE I: Deployment properties of the TeaStore application.

Deployment	CPU Req/Lim (in mcpu)	MEM Req/Lim (in MiB)
webui	750/1000	512/1024
registry	500/500	512/768
image	750/1000	512/1024
auth	500/1000	512/1024
persistence	500/1000	512/1024
db	500/500	256/512
recommender	500/500	512/768

communication allowed between segments via designated gateway nodes that are identified by label or property.

The idea is that segments are not statically defined and assigned to applications by the CA but instead, the scheduler should collect all relevant concerns for a given Pod and decide its optimal placement, thereby creating segments dynamically. Ideally, this will allow to balance performance and security requirements, while inconsistencies between Pod and node level networking can be avoided by forwarding the resulting segmentation information to the Security policy manager. The segmentation CRD also requires to extend the Diktyo scheduler with a novel scheduling plugin that dynamically orchestrates node segmentation as driven by the segmentation CRD, deployed applications, and network topology status.

V. Evaluation and Methodology

This section presents an evaluation of a preliminary prototype that does not yet implement the segmentation CRD and associated scheduler plugins. Instead, the CA specifies traditional security groups to block all connections between certain worker nodes. As described in Section IV-A, these network blockages are automatically detected and represented in the NT CR, from which node segmentation labels can be automatically inferred. We compare this prototype (*C3*) with two other scheduling scheme candidates:

- ***C1***: The default K8s scheduler, an OPA-based Mutating Admission webhook [10] that adds a node selector for a particular segment to every Pod, and GH to restrict node-level networking as described in Section IV-B (Fig. 4).
- ***C2***: Regular Diktyo that is not segment-aware (blocked connections are monitored but nodes are not labeled with a segmentation label), and GH as described above.

To evaluate the performance of these different scheduling schemes, the TeaStore application [11] has been used as a reference application to assess scheduler performance. TeaStore is a widely used microservice benchmark framework consisting of seven workloads with distinct performance characteristics allowing the evaluation of scheduling and autoscaling techniques. It emulates a Web Store for automatically generated tea supplies and features several User Interface (UI) elements for database generation and service resetting in addition to the store itself. Table I shows the TeaStore deployment requirements (given in millicpu and mebibytes) applied in the evaluation.

The testbed used for running all the experiments is an isolated part of a private OpenStack cloud, version Yoga. The minions that run VMs have two CPU sockets with an Intel(R) Xeon(R) Silver 4316 CPU processor @ 2.30GHz (2 cores, 60 threads) and 256 GB RAM. Each minion has two 10 Gbit and one 1 Gbit network interfaces. A K8s v1.28.4 - kubeadm Cluster Infrastructure has been set up with one master node, and four worker nodes: two nodes (*w1*, *w3*) with 2 cores, 4GiB and the other two (*w2* and *w4*) with 4 cores, 4GiB. Two fixed segments have been constructed with equal resource availability: *s1*: [*w1*, *w2*], and *s2*: [*w3*, *w4*]. Network delays have been emulated using TC for segment *s2* to represent varying network conditions in the cluster. Segment *s1* has significantly lower delays than *s2*.

To evaluate the performance in terms of response time and throughput for all scheduling scenarios, a load generator based on the locust load tool [12] was utilized. Emulated users generated a mix of *GET* and *POST* requests to simulate realistic workload conditions. The performance of each scheduling approach has been evaluated based on the following metrics:

- **Throughput (requests/s)** obtained for TeaStore.
- **Response time (in ms)** for all generated requests.
- **Adequate Segment placement (in %)** according to the segmentation of the K8s infrastructure.

VI. Evaluation Results

Table II presents the performance of TeaStore across the three evaluated scenarios. In *C1*, the admission controller, coupled with the default K8s scheduler, proficiently deploys all TeaStore Pods in the designated segment (*s1*), but its correct operation relies on effort of the CA to add segment labels to nodes manually. Conversely, the existing Diktyo scheduler encounters challenges deploying Pods without dependencies in the AG, occasionally placing them in *s2*, thereby causing GH to block all connection requests of the application out of precaution. The extended version of Diktyo (*C3*) solves the placement by comparing AG statistics with overall network costs available in the NT CR for each segment.

TABLE II: TeaStore performance (response time and throughput) for the different scheduling strategies.

	Throughput (Req/s)		**Response Time (in ms)**			**Segment Placement**	**Pending Pods**
Case	Total Number	Avg. Throughput	Median	Avg.	98th Percentile	Adequate (in %)	Ratio (in %)
C1	11.34 K	47.59 ± 3.54	55.63 ± 4.43	104.9 ± 15.55	390.5 ± 109.01	100%	0%
C2	10.89 K	45.52 ± 1.96	55.40 ± 2.30	114.2 ± 9.44	482.0 ± 43.24	23.8 %	28.5%
C3	10.78 K	45.02 ± 1.15	56.43 ± 2.06	116.1 ± 5.90	500.9 ± 24.27	100%	0%

No significant differences have been obtained for the three scenarios concerning throughput and response time, with *C1* exhibiting a slightly higher throughput on average. This can be attributed to the limited size of segment *s1*, where only two nodes are available, and one cannot accommodate all dependencies between Pods, constraining the scheduler's decision. Therefore, as future work, we plan to expand the cluster with more segments and multiple nodes per segment to make the scheduling problem more difficult. Regarding the scheduling of all Pods, negligible differences were observed across the three scenarios, with all Pods being deployed within approximately 0.4 to 1 second.

In summary, *C1* demands expertise and effort from CAs for installing and configuring admission controllers, and it is unclear how dynamic segments could be supported in this way. Diktyo without the proposed extensions cannot solve the placement problem, leading to pending Pods and Pods deployed in the wrong segment. The extended Diktyo can solve the placement problem in a streamlined way by considering application dependencies and the current segment topology.

VII. Conclusions and Future Work

Telecom vendors have adopted K8s to improve their software for edge-to-cloud networks, leading to less dependencies between applications and infrastructure. However this shift also introduces a complex interplay between performance optimization and security configuration, which is especially hard to solve when there is a strict separation of duties between application administrators and CAs. This paper proposes a scheduling architecture that integrates container-level and node-level network segmentation constraints directly into the K8s scheduler through an intent-based interface. The presented approach dynamically determines the preferred network segments for each application. The vision towards an intent-based segmentation CRD also allows CAs to focus on broader objectives without delving into the intricacies of node-level management. Results show that admission controllers can effectively address this scheduling challenge for static segmentation, albeit demanding a substantial learning curve for CAs and manual effort in attaching segmentation label to nodes. The extended version of Diktyo emerges as a promising approach, seamlessly integrating the preferences of both application and cluster administrators without manual node labeling, and enabling efficient node-level network isolation.

This work represents a pivotal first step towards a comprehensive solution tailored for the management of container-based applications within dynamic multi-tenant environments. In future work, we plan to address the following challenges:
(1) Design an expressive and intuitive Segmentation CRD that captures all relevant intents of the CA;
(2) Develop scheduling algorithms based on the proposed Segmentation CRD to fully implement the functionalities of the envisioned intent-based scheduler architecture;
(3) Investigate efficient descheduling policies to maintain Pod relationships even after dynamic changes in security rules and node segments. It will be crucial to maintain the desired deployment patterns specified by the CA, such as keeping certain Pods together or apart.

Acknowledgment

This research has received funding under EU H2020 MSCA-ITN action 5GhOSTS, grant agreement no. 814035. José Santos is funded by the Research Foundation Flanders (FWO), grant number 1299323N.

References

[1] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen, and W. Joosen, "Network policies in kubernetes: Performance evaluation and security analysis," in *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, 2021, pp. 407–412.

[2] G. Budigiri, C. Baumann, E. Truyen, J. T. Mühlberg, and W. Joosen, "Zero-cost in-depth enforcement of network policies for low-latency cloud-native systems," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, 2023, pp. 249–261.

[3] N. Yang, W. Shen, J. Li, X. Liu, X. Guo, and J. Ma, "Take over the whole cluster: Attacking kubernetes via excessive permissions of third-party applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, New York, NY, USA, 2023, p. 3048–3062. [Online]. Available: https://doi.org/10.1145/3576915.3623121

[4] VMware, "NSX Container Plugin for Kubernetes and Tanzu Application Service - Installation and Administration Guide," https://docs.vmware.com/en/VMware-NSX-Container-Plugin/4.0/ncp_40_kubernetes.pdf, 2022, [Accessed: 2022-11-13].

[5] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-aware scheduling in container-based clouds," *IEEE Transactions on Network and Service Management*, 2023.

[6] Cisco, "Cisco aci and kubernetes integration," accessed on 13 November 2023. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/kb/b_Kubernetes_Integration_with_ACI.html.

[7] Illumio, "Illumio Core for Kubernetes and OpenShift," https://docs.illumio.com/core/22.4/Content/Resources/PDF/Illumio_Core_for_Kubernetes_and_OpenShift_21.5.18.pdf, 2022, [Accessed: 2022-11-13].

[8] Kubernetes, "Kubernetes policies," accessed on 6 December 2023. [Online]. Available: https://kubernetes.io/docs/concepts/policy/.

[9] M. V. Le, S. Ahmed, D. Williams, and H. Jamjoom, "Securing container-based clouds with syscall-aware scheduling," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 812–826.

[10] E. Truyen, "Installing mutating admission web hook in OPA," accessed on 19 December 2023. [Online]. Available: https://github.com/k8-scalar/grasshopper/tree/opa/opa.

[11] S. Eismann, J. Kistowski, J. Grohmann, A. Bauer, N. Schmitt, and S. Kounev, "Teastore-a micro-service reference application," in *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2019, pp. 263–264.

[12] Locust, "An open source load testing tool." accessed on 12 December 2023. [Online]. Available: https://locust.io/.