ChronosGuard: A Hierarchical Machine Learning Intrusion Detection System for Modern Clouds

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

***Abstract*—Traditional Intrusion Detection Systems (IDSs) have been a cornerstone of network security for many years. Nevertheless, with the advent of containerized applications in the last few years, there is a growing need to understand how intrusion detection can adapt to these dynamic environments. This paper presents *ChronosGuard*, a hierarchical machine learning (ML) IDS designed for containerized environments. *ChronosGuard's* adaptable architecture consists of multiple components, each optimized for deployment in varying configurations ranging from monolithic to micro-service architectures. The performance impact of various factors such as network topology, workload orchestration, and deployment strategies has been assessed through extensive experiments concerning the scalability and resource utilization of *ChronosGuard*. Results show the effective prioritization of benign traffic of up to 85% compared to malicious traffic, the negligible impact of small network delays on performance metrics, and up to 10% decrease in response times with network-aware orchestration for complex deployment configurations. This study introduces a robust, containerized IDS that can be easily adapted to meet various operational needs, ranging from a full privacy-preserving local deployment to a scalable cloud deployment but also provides foundational insights for future research into optimizing containerized security solutions.**

***Index Terms*—Security, Machine Learning, Intrusion Detection Systems, Cloud Computing, Containers, Kubernetes**

I. Introduction

In our digitized world, the extensive use of interconnected systems and digital technologies amplifies the importance of cybersecurity for individuals, institutions, and organizations [1], [2]. Critical infrastructure systems such as power grids, electronic healthcare, and financial systems are often distributed, and their potential breaches pose severe global risks. Operational, financial, and personal losses can be significant in such events. To counteract such threats, it is essential to implement robust cybersecurity measures. Various tools and techniques [3]–[5] exist for preventing, detecting, and mitigating cybersecurity risks. These complementary tools can be collectively deployed to satisfy the required security standards. Among these, Intrusion Detection Systems (IDSs) [6], [7] play a crucial role.

An IDS is designed to monitor a network or system and to identify malicious activities or potential intrusions. Typically, it serves as a second line of defense, supplementing a firewall. A firewall, which is the primary defensive layer, enhances security by managing incoming and outgoing traffic and enforcing network restrictions. However, firewalls also have inherent limitations [8], [9]. They only secure the perimeter of the system, offering no protection against internal threats. Historically, the focus was on preventing unauthorized external access while safeguarding authorized internal users, a concept contradicting the zero-trust principle, which assumes no secure zones. In addition, their capacity to examine network traffic is limited, and the adoption of encrypted traffic further undermines their deep packet inspection capabilities. This is where an IDS, commonly advertised as the next-generation firewall, proves its worth.

Over the last decade, the intersection of Machine Learning (ML) and IDS has gained the attention of cybersecurity researchers, primarily driven by breakthroughs in Artificial Inteligence (AI) [10], [11]. The ML algorithms empower IDS to learn patterns and identify anomalies from massive amounts of data, enabling the detection of both known and unknown intrusions. With the widespread use of containerized applications and cloud infrastructures, monitored systems are typically distributed across various geographical locations [12], [13]. To protect these modern complex environments, there is a need for a distributed hierarchical IDS, capable of effectively monitoring and protecting each layer from the user devices to the cloud infrastructure itself [14].

This paper introduces *ChronosGuard*, a containerized implementation of a highly adaptable, multi-tiered hierarchical IDS designed to minimize latency and bandwidth requirements while preserving privacy constraints. A comprehensive experimental analysis has been conducted to measure *ChronosGuard's* performance through multiple metrics, such as latency, throughput, and resource utilization. Also, the evaluation assesses the influence of other factors including cloud infrastructure topologies, deployment strategies, container orchestrators, queue sorting algorithms, workload scenarios, and load patterns. The main contributions of this paper are twofold:

- **Deployment-ready containerized hierarchical multi-stage IDS:** The implementation of a multi-stage hierarchical IDS, consisting of four containerized key components, deployed via four distinct deployment strategies. The artefacts [1] have been made publicly available to facilitate future research and replication of our results.

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

[1]https://github.com/idlab-discover/ChronosGuard

- **Comprehensive evaluation of a containerized IDS deployment:** Through a multitude of experiments conducted on a Kubernetes (K8s) cluster, the impact of infrastructure topologies, container orchestrators, queue sorting algorithms, deployment strategies, workload scenarios, and traffic load patterns on the performance of containerized IDSs is extensively studied.

II. Related Work

The field of IDS has evolved significantly since the adoption of rapid advancements in AI. Researchers increasingly adopt ML techniques to learn malicious patterns from data or detect suspicious activity deviating from baselines. Numerous studies focus on the design, implementation, and optimization of ML-based IDS, aiming to improve classification performance and tackle aspects such as recall, precision, and the number of false positives. However, a considerable gap remains in addressing scalability and the practical deployment of IDS in dynamic, real-world environments. Sharma et al. [15] conducted a systematic review on multi-objective optimizations for intrusion detection, analyzing 25 studies across a set of nine objectives. Yet, crucial aspects such as scalability were not considered. Similarly, Liu et al. [16] performed a literature review on IDS in the cloud, and only 7% of the selected studies addressed scalability as part of their work. According to Apruzzese et al. [17], the current ML-IDS research does not meet the practical demands of industry developers which require an increased focus on operational viability and practical requirements. This demonstrates the general oversight in current IDS research and its implications for real-world applications.

Recently, several new ML-based architectures have been proposed. In their comparison of different IDS deployment strategies in the Internet of Things (IoT) landscape, Khraisat and Alazab [18] found that the hierarchical strategy was extremely deployable across large and heterogeneous networks, with the only drawback being the complexity of the IDS. Pundir et al. [19] specified three essential requirements for an effective IDS, including the necessity to minimize system and network resource consumption. In addition, Lai et al. [20], [21] have defined a theoretical three-stage ML IDS consisting of three in-sequence tasks: pre-processing, binary detection, and multi-class detection. Using queueing theory and simulated annealing, the authors evaluated ten different task assignments across edge, fog, and cloud. The results conclude that while each task assignment presents unique benefits and drawbacks, a hierarchical approach empowers cloud offloading, cost minimization, and enhanced privacy.

In summary, *ChronosGuard* is the first study, to the best of our knowledge, to extensively evaluate how various factors impact the performance of IDSs in containerized environments. These factors include network topologies, deployment strategies, container orchestrators, and traffic load patterns. The outcomes of this study, combined with an in-depth analysis of these results, offer unique insights that contribute towards the development of more scalable and efficient security-oriented applications.

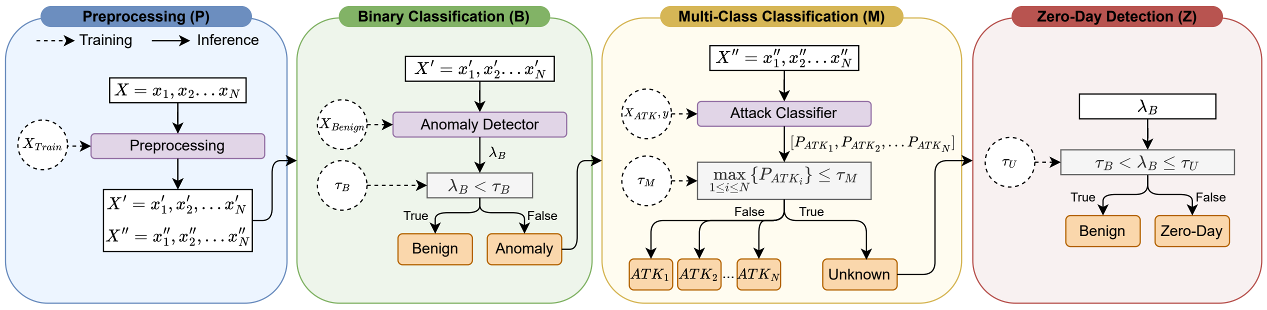
III. System Design

This section introduces *Chronosguard*, a containerized multi-stage ML-IDS specifically designed for hierarchical deployments. Building on our previous research [22], which optimized and analyzed the ML pipeline, *ChronosGuard* is evaluated through extensive real-world experiments using the K8s orchestration platform to assess its performance.

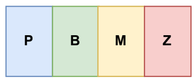
A. Components

ChronosGuard leverages a state-of-the-art multi-stage hierarchical IDS methodology consisting of four main components: preprocessing, binary classification, multi-class classification, and zero-day detection. Figure 1a illustrates the connections and internals of the key components within the ML-based IDS pipeline. This pipeline creates a cascading processing sequence where each network flow is sequentially analyzed through successive components until a final classification is achieved. An unsupervised one-class SVM and supervised random forest are used respectively for the anomaly detector and attack classifier with a combined performance of 0.9875 f1-score, see our prior work [22] for more details on the hyperparameters and optimal thresholds of the ML pipeline.

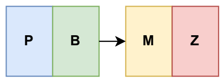
Preprocessing (P) The first component receives a vector, $X = x_1, x_2, ...x_N$, encapsulating network flow characteristics such as packet count, duration, and packet inter-arrival times. This vector is transformed into a suitable format for subsequent analysis by the ML-based anomaly detector and multi-class classifier. **Binary Classification (B)** The second component performs a lightweight anomaly detection, assigning an anomaly score, λ_B, to each network flow based on the network flow characteristics. Flows with a corresponding anomaly score below a predefined threshold, τ_B, are classified as benign and are excluded from further analysis. Contrarily flows whose scores exceed this threshold are flagged as suspicious and are forwarded to the subsequent component. This filtering process forwards only a subset of all network flows to subsequent stages. By doing so, it significantly reduces bandwidth requirements and allows for the use of more computationally intensive techniques in the stages that follow. **Multi-class Classification (M)** The third component classifies flows identified as suspicious into known attack types based on confidence levels. A flow is assigned to the attack category with the highest calculated probability, unless this prediction confidence, P_{ATK_i}, falls below the threshold τ_M. Flows failing to meet the minimum confidence threshold are considered unrelated to any attack classes seen during training and are consequently forwarded to the last component for final analysis. **Zero-day Detection (Z)** The fourth and final component utilizes the anomaly score, λ_B, previously calculated by the Binary Classifier to distinguish between zero-day or unseen attack types and benign flows erroneously flagged as suspicious. Flows with scores exceeding a more rigorous threshold, τ_Z, are categorized as zero-day attacks, whereas those below are reclassified as benign.



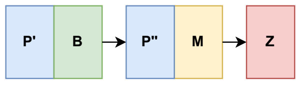
(a) Overview of the multi-stage hierarchical IDS four key components: preprocessing (P), binary classification (B), multi-class classification (M), zero-day detection (Z)



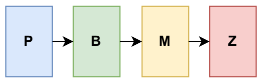
(b) Monolith



(c) 2-pod



(d) 3-pod



(e) 4-pod

Fig. 1: (a) Illustration of *ChronosGuard*, detailing the interactions of the four main components. (b-e) Illustration of the multiple deployment schemes, organized into numerous pods: (a) Monolith, (b) 2-pod, (c) 3-pod, and (d) 4-pod

B. Containerized Deployments

ChronosGuard's architecture is highly configurable within a containerized environment, impacting overall system performance significantly. This research evaluates four distinct deployment configurations, depicted in Figures 1b through 1e, ranging from a monolithic to a fully micro-serviced architecture. A "pod" in K8s refers to the smallest deployable unit, typically containing one or more containers that share resources. *ChronosGuard*'s key components are preprocessing (P), binary classification (B), multi-class classification (M), and zero-day detection (Z).

Monolithic Deployment (PBMZ) integrates all four components within a single pod. This configuration allows the least flexibility in terms of scaling due to its non-granular scaling.

2-Pod Deployment (PB-MZ) separates the components into two groups: preprocessing with binary classification and multi-class classification combined with zero-day detection.

3-Pod Deployment (P'B-P"M-Z) divides the preprocessing component into two parts: preprocessing for the binary classifier (P'), which computes X', and preprocessing for the multi-class classifier (P''), which computes X''. These are paired with the binary (B) and multi-class (M) components, respectively. The zero-day detection remains by itself.

4-Pod Deployment (P-B-M-Z): Represents a true micro-service deployment, with each component operating independently within its own pod. This configuration allows the most scaling flexibility due to its granularity.

IV. Methodology

This section outlines the methodology employed to evaluate *ChronosGuard*. The evaluation framework is described, including experiment automation, data collection, and load generation, followed by the specification of the used software and hardware in the experimental testbed.

A. Evaluation Framework

To support a comprehensive analysis of *ChronosGuard*'s performance across various configurations and scenarios within cloud environments, a custom testbed has been designed and implemented on top of K8s. This testbed is designed to automate the execution of a multitude of experiments, each exploring a different permutation of factors that potentially impact the IDS's performance. Cumulatively, several thousand experiments were executed over the course of one month. The automation of this testing methodology is achieved through custom bash and Python scripts. These scripts are responsible for critical functions, such as experiment initialization, load generation, and data collection.

Key to the evaluation framework is the capability to provide a controlled and consistent environment for each test iteration. This is achieved by initiating each experiment with a fresh deployment within K8s, ensuring that no residual data or configurations from previous tests influence the results. This approach guarantees that each set of tests is conducted under uniform conditions, producing reliable and reproducible results. To facilitate this, a Bash script is used to orchestrate the experimental setup. The script interfaces directly with the K8s API using Kubectl, the command-line tool that allows for efficient management and operation of Kubernetes clusters.

Load Generation for *ChronosGuard* is generated using Locust [23], a popular open-source load testing framework. This framework enables the simulation of both benign and malicious user interactions with the IDS. Benign users submit benign network flows to the hierarchical IDS, simulating a normal baseline network operation, while malicious users send malicious network flows, emulating potential cybersecurity threats. To reflect a realistic scenario, the ratio of benign to malicious users is set at 4:1, similar to popular benchmark datasets [24]. Furthermore, the probability of a malicious user selecting a known attack type is intentionally set at twice the likelihood of selecting an unknown attack type, further mirroring potential real-world attack distributions.

At the beginning of each experiment, the load generator loads a collection of 59,435 network flows extracted from the CIC-IDS-2017 benchmark dataset [25], comprising benign flows, five distinct known attacks, and unknown attacks. After initialization, each user submits only one simultaneous request to *ChronosGuard*, ensuring a controlled load simulation. The simulated users randomly sample a network flow from the appropriate category from the initially loaded collection.

Data Collection within the *ChronosGuard* evaluation framework is divided into two primary categories: resource and performance metrics, each essential for a comprehensive analysis of system performance. This dual approach to data collection not only improves the understanding of *ChronosGuard*'s operational efficiency but also enables performance evaluation by correlating resource utilization with system responsiveness.

Resource Metrics are continuously monitored by K8s kubelets, which track CPU usage, memory consumption, and network bandwidth every two seconds. This data is aggregated by Prometheus at the same interval. After each experimental run, the data is queried from the Prometheus metric server and saved in CSV format. This systematic collection and aggregation process ensures detailed resource usage statistics are available for subsequent analysis.

Performance Metrics are the primary focus here is on the collection of response times for each network request processed by *ChronosGuard*. To achieve this, custom event handlers within Locust are implemented to record the latency from when a network flow is transmitted to when its associated prediction is received. These response times are then exported into a single CSV file on the host running Locust, along with experiment and flow properties, providing a granular view of the system's responsiveness under various load conditions.

The Testbed Infrastructure utilizes the imec Virtual Wall (VWall) infrastructure located at IDLab, Belgium. The setup comprises a single K8s cluster with ten nodes (Ubuntu 20.04.2 LTS), each equipped with dual hexacore Intel E5645 CPUs, 24GB RAM, and dedicated Gigabit NICs. The K8s cluster has been set up with Kubeadm and Kubectl version v1.22.4, and Docker version 20.10.12. Network delays are emulated using TC [26] to simulate different network topologies.

The Experimental Setup systematically evaluates each permutation of variables through a series of repeated independent experiments, each comprising eleven distinct steps characterized by a progressively increasing load. In each step, network load is generated by a predefined number of users over a 30-second interval. This period is followed by a 10-second user spawning phase, during which new users are gradually introduced at a consistent rate of one user per second. The procedural flow of these operations is illustrated in Figure 2. The experiment begins with a single user, gradually increasing the user count linearly in each subsequent step, ending with 100 users in the final step. To reflect a realistic scenario, the ratio of benign to malicious users is set at 4:1. At each step,

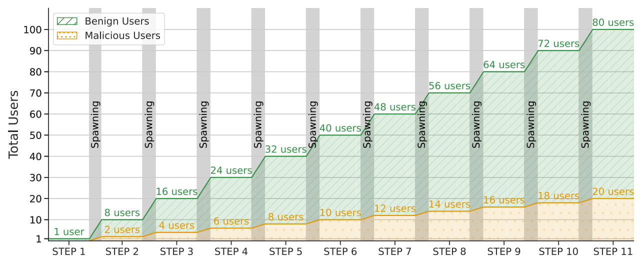


Fig. 2: Diagram showing a single experimental run: traffic load incrementally increases from one to a maximum of 100 users.

TABLE I: Overview of the experimental parameters.

Variable	#	Values
Topology	2	Cluster, Edge-Cloud
Deployment	4	Monolith, 2-pod, 3-pod, 4-pod
Scheduler	2	Kube-Scheduler (KS), Diktyo
Sorting Alg.	9	Priority, QoS, Cycle, (Alt. / Rev.) Kahn / Tarjan
Scenarios	3	Initial, Scale-up, Scale-down
Users	11	1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Repetition	8/10	1-8 (Edge-fog-cloud), 1-10 (Cluster)

key performance and resource utilization metrics are collected. These detailed results at every stage enable a thorough assessment of the system's scalability and responsiveness, providing insights into its performance under varying operational loads and experiment configurations.

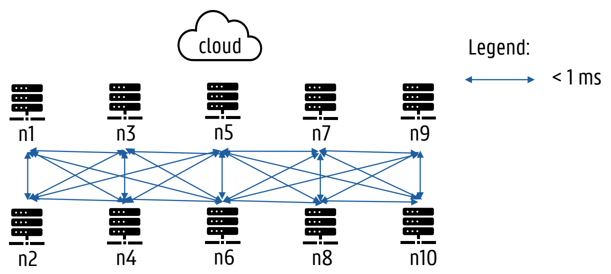
B. *Evaluated Variables*

Table I presents a detailed overview of the evaluated variables and their respective values.

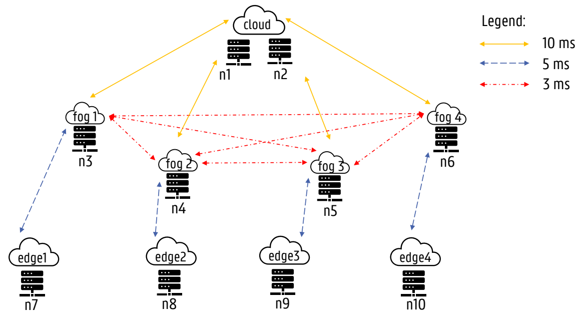
Network Topology Figure 3 shows the two distinct cloud topologies evaluated in this study: the cluster and edge-fog-cloud. On one hand, Figure 3a presents a highly available cluster configuration, where nodes are provisioned within a single region with similar network connections. Consequently, this produces negligible inter-node delays in the order of microseconds. On the other hand, Figure 3b presents a multi-region cluster, where network delays vary considerably. Delays range from negligible among the cloud nodes to as much as 15ms between cloud and edge.

Deployment Strategy In this study, four distinct deployment configurations are evaluated, ranging from a combined monolithic deployment of all four stages to a true micro-service deployment, where each stage is deployed individually. The CPU and memory requests and limits for each configuration are displayed in Table II. The resource allocation is designed such that the cumulative total remains consistent across all deployment strategies. Additionally, the first components (P, B) are allocated a relatively larger share of resources compared to the subsequent components (M, Z), as they handle the full load before passing only a fraction of the requests to the subsequent ML components in the pipeline.

Orchestration Schedulers and Sorting Algorithms The K8s component responsible for scheduling operations is known as KS, the default scheduler in K8s. Our experiments considered two distinct schedulers: KS and Diktyo [27]. While



(a) Cluster topology with similar network connections.



(b) Edge-fog-cloud topology with different network delays.

Fig. 3: Illustration of the evaluated cloud infrastructures.

TABLE II: Deployment properties of *ChronosGuard* components: CPU and memory requests (R) and limits (L).

App.	Deployment	CPU R/L (mCPU)	MEM R/L (MiB)
Monolith	PBMZ	1000/2000	512/1024
2-pod	PB	700/1400	384/768
	MZ	300/600	128/256
3-pod	P'B	500/1000	256/512
	P"M	400/800	192/384
	Z	100/200	64/128
4-pod	P	450/900	192/384
	B	250/500	128/256
	M	200/400	128/256
	Z	100/200	64/128

KS primarily focuses on optimizing resource usage across all cluster nodes, Diktyo considers both application dependencies and infrastructure topology during pod scheduling within K8s to find network-aware placement schemes for containerized applications. To determine the optimal allocation order, Diktyo also considers pod dependencies specified by developers to sort pods based on topological sorting information [28]. Our evaluation assessed several topological sorting algorithms, which define the preferred deployment order for pods in a multi-pod application with interdependencies, as detailed in Table III. In contrast, KS applies the default priority sorting algorithm available in K8s, yielding an identical order to *Kahn* since all our pods have the same priority level.

Scenario Each experiment is conducted across three sequential scenarios: the initial phase, with one instance per pod in the deployment strategy; the scale-up phase, where each pod is increased to five replicas; and the scale-down phase, where three instances of each pod are terminated, leaving two instances per pod. This increases the total available resources by 5-fold and 2-fold compared to the initial phase and enables studying the impact of application scheduling.

TABLE III: Orchestration order for the different deployment schemes of *ChronosGuard*.

Deployment	Algorithm	Topological order
Monolith	*KS*	[PBMZ]
2-pod	*KS*	[PB, MZ]
	Kahn Tarjan & Cycle	[PB, MZ]
	Alt. Kahn & Alt. Tarj.	[PB, MZ]
	Rev. Kahn & Rev. Tarj.	[MZ, PB]
3-pod	*KS*	[P'B, P"M, Z]
	Kahn Tarjan & Cycle	[P'B, P"M, Z]
	Alt. Kahn & Alt. Tarj.	[P'B, Z, P"M]
	Rev. Kahn & Rev. Tarj.	[Z, P'M, P"B]
4-pod	*KS*	[P, B, M, Z]
	Kahn Tarjan & Cycle	[P, B, M, Z]
	Alt. Kahn & Alt. Tarj.	[P, Z, B, M]
	Rev. Kahn & Rev. Tarj.	[Z, M, B, P]

Simulated Users The variable *users* quantifies the number of concurrent users spawned by the Locust load generator. Each user is configured to only send a single concurrent request to *ChronosGuard*; therefore, the number of users directly determines the load intensity directed at the system. Thus, an increase in the number of users directly corresponds to an increased load.

Repetition To ensure statistical significance and reliability of the experimental results, each experiment is replicated multiple times. Specifically, for the cluster topology, each experiment is replicated ten times, with the load generator, simulating an ingress point, systematically rotated across each node in the cluster to ensure a balanced evaluation. Similarly, for the edge-fog-cloud topology, each experiment is repeated eight times, with the load generator cycled twice over each edge node.

V. Results

This section presents the obtained results through the evaluation of *ChronosGuard*. The results analyze the effectiveness of different architectural approaches across several performance metrics including response times, throughput, and resource utilization.

A. Prioritization of benign traffic

In an optimal scenario, an IDS should minimally disrupt the normal operations of the networks it protects, adding as little latency as possible to benign traffic while detecting attacks within seconds. Our analysis of the experimental data collected reveals a significant difference in processing times between benign and malicious network flows. Notably, benign traffic benefits from substantially lower median response times than attack and zero-day traffic across all evaluated deployment strategies, as illustrated in Figure 4. The most pronounced differences in response times are observed in the configurations utilizing 2-pod and 3-pod deployments, where the median response time for benign flows is reduced by 85% and 75%, respectively, in comparison to malicious traffic. Conversely,

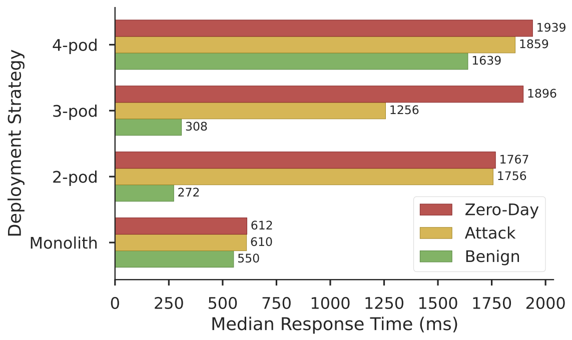


Fig. 4: Median response times by flow type: a significant reduction for benign versus malicious traffic across all deployment strategies, minimizing latency for normal operations.

the difference in response times is less pronounced within the monolith and 4-pod deployment, with benign traffic showing a smaller reduction in median response times of respectively 10% and 12% relative to malicious flows. These results for the 4-pod deployment deviate from the other microservice deployments, which can be attributed to suboptimal static resource allocation across the pods (see Section V-D). The lowest median response times for benign traffic were observed for the 3-pod deployment during the scale-up scenario with 1 and 10 users, respectively, yielding 27 and 46 ms of latency.

In conclusion, the multi-stage hierarchical IDS consistently prioritizes benign traffic over malicious by achieving lower average processing times, regardless of the deployment strategy selected. However, it is the adoption of a micro-service architecture that significantly enhances this effect, demonstrating the capability to reduce benign response times by up to eight-fold compared to a monolithic deployment through strategically offloading the processing of suspicious network flows and the ability to allocate resources for each IDS component independently. Moreover, a micro-service architecture can reduce the absolute median response time for benign flows up to 51% (for 2-pod) compared to a monolith architecture.

B. Cluster vs Edge-Fog-Cloud architecture

The comparative analysis of the results reveals no significant differences between the cluster and edge-fog-cloud topologies in terms of throughput, latency, or resource utilization. Figure 5 presents the throughput averaged over the repeated experiment runs, relative to the load generated by the locust users for the default KS across the three scenarios and two topologies. The results were nearly identical for the initial, scale-down, and scale-up scenarios in both topologies. The detailed performance metrics in Table IV reveal only minor variations between the cluster and edge-fog-cloud topology in terms of throughput and response time. The minimal introduced delay in node-to-node communication within the edge-fog-cloud topology does not significantly affect performance. Specifically, the small increase in communication delay, in the order of a few milliseconds, is considered negligible in contrast to the more substantial inference times associated with the employed ML models.

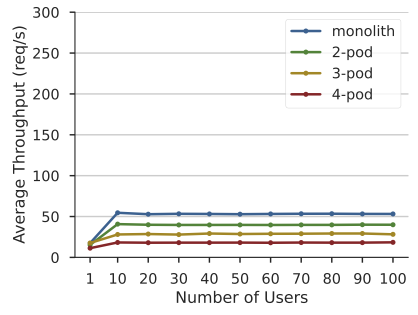
TABLE IV: Impact of the topology on throughput and response times for KS, computed over all users and scenarios.

Scenario	Topology	Throughput (req/s)	Response Time (ms)		
			Median	Mean	95th Pctl
Initial	Cluster	33	1235	1725	3364
	Edge-fog-cloud	33	1240	1730	3380
Scale-down	Cluster	65	637	882	1856
	Edge-fog-cloud	64	640	896	1845
Scale-up	Cluster	157	267	349	796
	Edge-fog-cloud	154	266	351	817

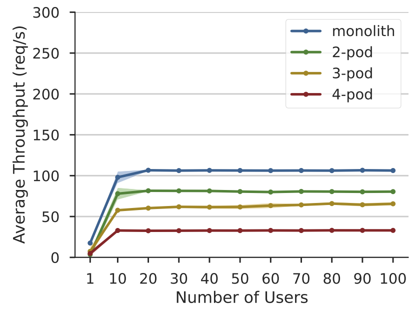
C. Topological-aware orchestration

Topological-aware orchestration represents a paradigm within application deployment that takes into account pod dependencies and inter-node network latencies, in contrast to the default KS, which prioritizes optimal resource utilization across all the cluster nodes. This methodology ensures that the deployment order is optimized for network efficiency, potentially reducing latency and enhancing overall application performance. The selected deployment strategy is crucial as it directly correlates with the number of pods to be scheduled. For instance, a monolithic application, characterized by a singular pod, presents only one deployment permutation and lacks communication inter-dependencies, thus simplifying the orchestration process. Conversely, as the complexity and pod count of an application increase, so do the permutations for network-aware sorting and inter-pod dependencies, introducing a multitude of potential deployment sequences. The load scenario also significantly impacts the orchestration process, with the required number of pods varying—specifically, one, two, and five pods, respectively, for each scenario, multiplied by the number of pods of the selected deployment strategy.

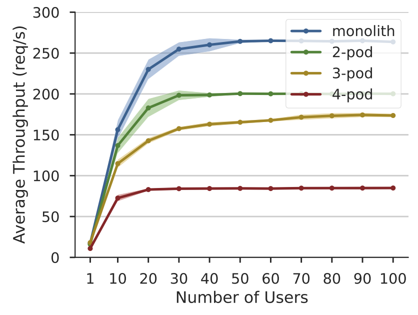
Empirical evidence supporting the efficacy of network-aware sorting algorithms within topological-aware orchestration is presented in Figure 6. It illustrates the distribution of response times across different deployments within a cluster topology during scale-up. Results reveal that the influence of network-aware orchestration on application deployment demonstrates a correlation to the number of pods to be scheduled. For simpler deployments, such as those involving monolithic, 2-pod, and 3-pod, the impact of network-aware sorting on response times is negligible. This observation is consistent with insights from earlier studies [29], indicating a limited scope for optimization in less complex applications. However, noticeable variations are observed as the deployment complexity increases to four pods. In this case, network-aware orchestration regardless of the sorting algorithm selected, demonstrated the capability to enhance performance, as demonstrated by slightly lower response times. Particularly, changing from the default KS to the Reverse Kahn sorting algorithm decreased the average response times by 10%, significantly lowering the response time for the first quartile, median, and last quartile from 316 to 284, 628 to 566, and 955 to 927 ms, respectively. This signifies a tangible, albeit limited, improvement in deployment efficiency through topological-aware orchestration and sorting.



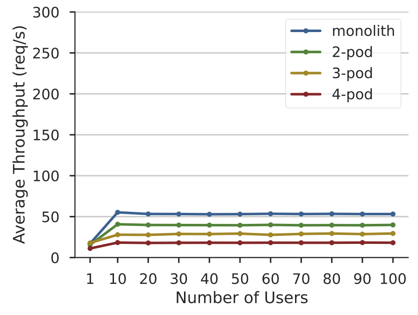
(a) Cluster - initial



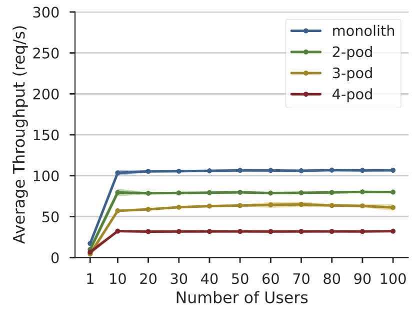
(b) Cluster - scale-down



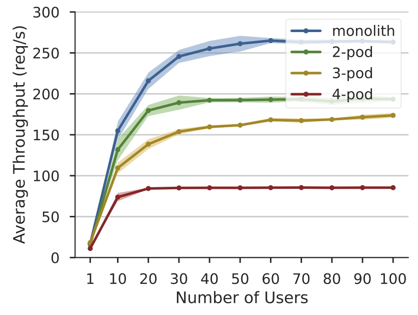
(c) Cluster - scale-up



(d) Edge-Cloud - initial



(e) Edge-Cloud - scale-down



(f) Edge-Cloud - scale-up

Fig. 5: The impact of the deployment strategy on the average obtained throughput: a higher number of K8s pods per deployment has a significantly negative influence on the throughput. Once resources are saturated, increasing the users has no further effect.

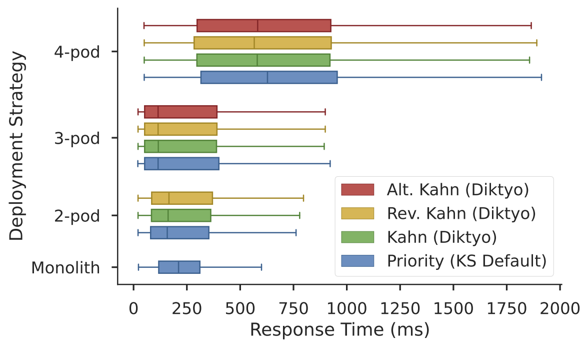


Fig. 6: Distribution of response times across deployment strategies in cluster topology during scale-up scenario.

TABLE V: Throughput and response times across the different scenarios for the KS and cluster topology.

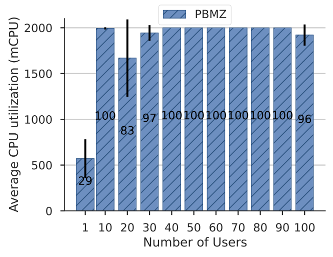
Deployment	**Scenario**	**Throughput (req/s)**	**Response Time (ms)**		
			Median	**Mean**	**95th Pctl**
Monolith	Initial	50	971	974	1114
	Scale-down	98	486	490	627
	Scale-up	228	199	203	321
2 pods	Initial	38	440	1300	3313
	Scale-down	73	278	645	1809
	Scale-up	176	164	265	751
3 pods	Initial	28	680	1765	5567
	Scale-down	58	261	814	2941
	Scale-up	147	107	311	1161
4 pods	Initial	17	2847	2861	3463
	Scale-down	30	1523	1579	2045
	Scale-up	77	599	619	950

D. Joint task assignment in container clouds

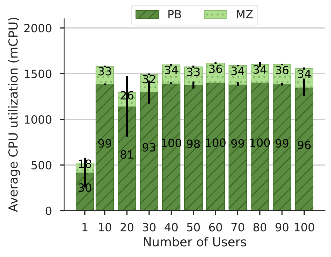
This analysis evaluates four unique deployment strategies for container-based cloud environments (previously presented Figure 1), comparing their performance to determine the most effective configuration and the trade-offs to consider. Among the configurations tested, the monolithic deployment outperforms the other strategies on the performance metrics, such as throughput and upper-bound response times. While it exhibits lower upper-bound response times, it is important to note that higher median response times were observed than most micro-service deployments. The greater performance of the monolithic architecture can be attributed to the absence of inter-pod communication, thereby reducing overhead when compared to micro-service architectures and eliminating the need for complex resource allocation strategies within *ChronosGuard*. However, this strategy does present drawbacks in terms of non-granular scaling and deployment flexibility:

- **Local Deployment**: Preserves data privacy but fails to leverage broader cloud computing benefits.
- **Cloud Deployment**: Compromises privacy and requires higher bandwidth, mitigating the primary advantages of localized computation.

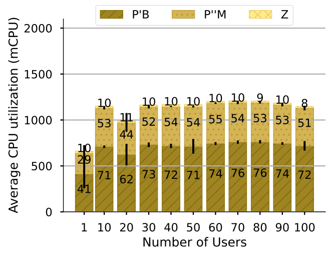
With an increase in the number of pods within the deployment, throughput tends to decrease, a trend that primarily can be attributed to the suboptimal allocation of resources among the pods. Figure 7 illustrates this trend, showing that while the monolith utilizes resources at 100% capacity, deployments with 2, 3, and 4 pods utilize only 75%, 65%, and 55%, respec-



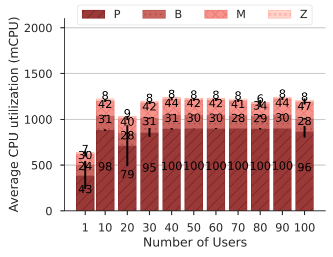
(a) Monolith - initial



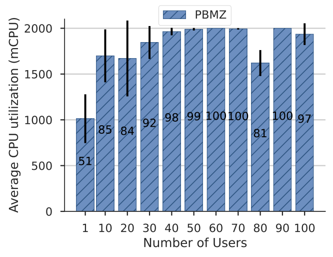
(b) 2-pod - initial



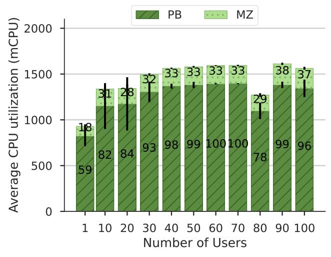
(c) 3-pod - initial



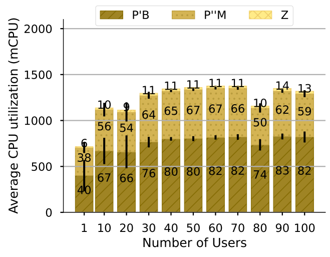
(d) 4-pod - initial



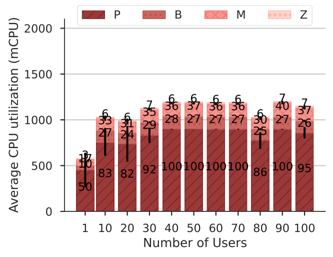
(e) Monolith - scale-down



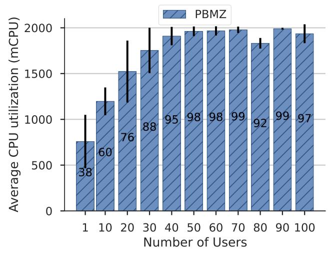
(f) 2-pod - scale-down



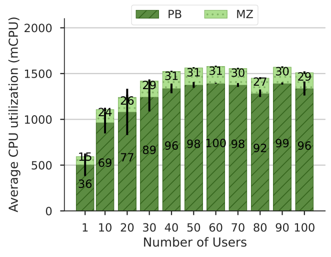
(g) 3-pod - scale-down



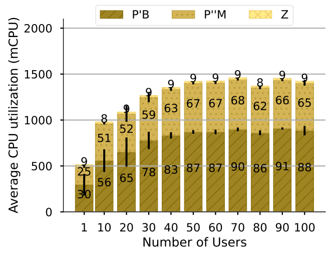
(h) 4-pod - scale-down



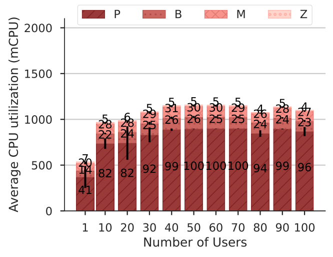
(i) Monolith - scale-up



(j) 2-pod - scale-up



(k) 3-pod - scale-up



(l) 4-pod - scale-up

Fig. 7: The average CPU utilization across deployment strategies and scenarios under varying loads: highlighting efficiency and scalability of micro-service architectures. The bars represent the usage percentage of the pod's CPU limit with 95% CI.

tively. This is caused by underestimating the resource demands for the preprocessing component, as observed in the 4-pod deployment strategy where the preprocessing pod reaches maximum CPU utilization under low loads, while the other components remain underutilized. Similar, albeit less pronounced under-utilization of pod resources is present for the 3 and 2 pod deployment strategies. When the throughput values are adjusted for resource utilization, a smaller performance gap is revealed between the micro-service architectures and the monolith application, indicating that the reduced throughput in multi-pod deployments is partly due to the inherent overhead associated with micro-services. This overhead includes the additional computational resources required for inter-service communication, such as serialization and deserialization.

Despite their computational overhead, microservice architectures can reduce overall response times, see Table V, minimizing the disruption for normal traffic and speeding up the detection of known and unknown attacks. This advantage results from the ability to deploy each component independently, allowing the system to handle the majority of traffic—which is benign—without being slowed down by the computationally intensive ML techniques required for attack detection. From a resource utilization perspective, microservice architectures consume approximately 2x to 2.5x more memory than monolithic setups. Furthermore, while they may require more CPU resources, they enable more sophisticated hierarchical deployments. For example, a deployment model that localizes the preprocessing and binary detection components can significantly reduce bandwidth requirements toward the cloud—by 25%, 33%, and 44% for 2, 3, and 4 pod deployments, respectively, compared to a monolithic deployment.

Overall, while monolithic architectures offer robust performance and lower resource demands, micro-service architectures provide enhanced flexibility and responsiveness, particularly in complex, scalable environments.

E. *Limitations*

A limitation of this study is the static resource allocation among *ChronosGuard* components. The total sum of resources assigned across all deployment strategies is equal and the initial components received a larger share, as they handle the full load and pass only a fraction to subsequent components. This allocation has proven suboptimal and could be further improved, potentially by dynamically scaling resources based on real-time demand. Additionally, the scope of this evaluation of *ChronosGuard* is limited to the containerized components of the ML pipeline. Crucial aspects such as traffic collection and flow reconstruction, which are essential to IDS, were not included in this study. Future studies could expand on this by incorporating these components to provide a more end-to-end assessment of IDS performance.

VI. Conclusion

In this work, a containerized hierarchical ML IDS, *Chronos-Guard*, consisting of four individual components, that can be deployed using four deployment strategies, is presented. Through systematic experimentation, the impact of various factors including network topology, workload orchestration, and throughput on the performance and resource utilization of each deployment strategy is evaluated. The analysis resulted in several key insights: benign traffic is effectively prioritized across all configurations with a maximum reduction of 85% for the median response times compared to malicious traffic and up to 51% lower response times for the microservice architectures compared to the monolithic deployment; minor network delays have negligible effects on throughput and response times; network-aware orchestration enhances the performance of complex security ML-based applications in cloud environments up to 10%; and there are notable trade-offs associated with different task assignment strategies in containerized settings. This research not only introduces a deployment-ready, containerized IDS to the field but also establishes a robust foundation for future studies aimed at optimizing the containerization of ML-based security components.

Acknowledgments

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

References

[1] A. Singh and K. Chatterjee, "Cloud security issues and challenges: A survey," *Journal of Network and Computer Applications*, 2017.

[2] F. Sierra-Arriaga, R. Branco, and B. Lee, "Security issues and challenges for virtualization technologies," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–37, 2020.

[3] Z. Cheng, M. Apostolaki, Z. Liu, and V. Sekar, "Trustsketch: Trustworthy sketch-based telemetry on cloud hosts," in *The Network and Distributed System Security Symposium (NDSS)*, 2024.

[4] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakcs." in *NDSS*, 2022, pp. 1–17.

[5] A. Van't Hof and J. Nieh, "{BlackBox}: a container security monitor for protecting containers on untrusted operating systems," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 683–700.

[6] F. Wei, H. Li, Z. Zhao, and H. Hu, "{xNIDS}: Explaining deep learning-based network intrusion detection systems for active intrusion responses," in *32nd USENIX Security Symposium (USENIX Security 23)*.

[7] T. Saranya, S. Sridevi, C. Deisy, T. D. Chung, and M. A. Khan, "Performance analysis of machine learning algorithms in intrusion detection system: A review," *Procedia Computer Science*, 2020.

[8] A. Voronkov, L. H. Iwaya, L. A. Martucci, and S. Lindskog, "Systematic literature review on usability of firewall configuration," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–35, 2017.

[9] L. Ceragioli, P. Degano, and L. Galletta, "Are all firewall systems equally powerful?" in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, 2019, pp. 1–17.

[10] G. Apruzzese, L. Pajola, and M. Conti, "The cross-evaluation of machine learning-based network intrusion detection systems," *IEEE Transactions on Network and Service Management*, 2022.

[11] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When malware is packin'heat; limits of machine learning classifiers based on static analysis features," in *Network and Distributed Systems Security (NDSS) Symposium 2020*.

[12] 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.

[13] A. Luckow, K. Rattan, and S. Jha, "Pilot-edge: Distributed resource management along the edge-to-cloud continuum," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 874–878.

[14] R. V. Mendonça, A. A. Teodoro, R. L. Rosa, M. Saadi, D. C. Melgarejo, P. H. Nardelli, and D. Z. Rodríguez, "Intrusion detection system based on fast hierarchical deep convolutional neural network," *IEEE Access*, vol. 9, pp. 61 024–61 034, 2021.

[15] S. Sharma, V. Kumar, and K. Dutta, "Multi-objective optimization algorithms for intrusion detection in IoT networks: A systematic review," *Internet of Things and Cyber-Physical Systems*, 2024.

[16] Z. Liu, B. Xu, B. Cheng, X. Hu, and M. Darbandi, "Intrusion detection systems in the cloud computing: A comprehensive and deep literature review," *Concurrency and Computation: Practice and Experience*, 2022.

[17] G. Apruzzese, P. Laskov, and J. Schneider, "SoK: Pragmatic Assessment of Machine Learning for Network Intrusion Detection," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, Jul. 2023.

[18] A. Khraisat and A. Alazab, "A critical review of intrusion detection systems in the internet of things: techniques, deployment strategy, validation strategy, attacks, public datasets and challenges," *Cybersecurity*, vol. 4, no. 1, p. 18, Mar. 2021.

[19] S. Pundir, M. Wazid, D. P. Singh, A. K. Das, J. J. P. C. Rodrigues, and Y. Park, "Intrusion Detection Protocols in Wireless Sensor Networks Integrated to Internet of Things Deployment: Survey and Future Challenges," *IEEE Access*, 2020, conference Name: IEEE Access.

[20] Y.-C. Lai, D. Sudyana, Y.-D. Lin, M. Verkerken, L. D'hooge, T. Wauters, B. Volckaert, and F. De Turck, "Machine learning based intrusion detection as a service: task assignment and capacity allocation in a multi-tier architecture," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, Dec. 2021.

[21] Y.-C. Lai, D. Sudyana, Y.-D. Lin, M. Verkerken, L. D'hooge, T. Wauters, B. Volckaert, and F. D. Turck, "Task Assignment and Capacity Allocation for ML-based Intrusion Detection as a Service in a Multi-tier Architecture," *IEEE Transactions on Network and Service Management*, 2022.

[22] M. Verkerken, L. D'hooge, D. Sudyana, Y.-D. Lin, T. Wauters, B. Volckaert, and F. D. Turck, "A Novel Multi-Stage Approach for Hierarchical Intrusion Detection," *IEEE Transactions on Network and Service Management*, 2023.

[23] "locustio/locust," Mar. 2024, original-date: 2011-02-17T11:08:03Z. [Online]. Available: https://github.com/locustio/locust

[24] L. Liu, G. Engelen, T. Lynar, D. Essam, and W. Joosen, "Error Prevalence in NIDS datasets: A Case Study on CIC-IDS-2017 and CSE-CIC-IDS-2018," in *2022 IEEE Conference on Communications and Network Security (CNS)*, Oct. 2022, pp. 254–262.

[25] I. Sharafaldin, A. Habibi Lashkari, and A. A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization:," in *Proceedings of the 4th International Conference on Information Systems Security and Privacy*, 2018.

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

[27] 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.

[28] T. Ahammad, M. Hasan, and M. Zahid Hassan, "A new topological sorting algorithm with reduced time complexity," in *Intelligent Computing and Optimization: Proceedings of the 3rd International Conference on Intelligent Computing and Optimization 2020 (ICO 2020)*, 2021.

[29] J. Santos, M. Verkerken, L. D'Hooge, T. Wauters, B. Volckaert, and F. De Turck, "Performance Impact of Queue Sorting in Container-Based Application Scheduling," in *2023 19th International Conference on Network and Service Management (CNSM)*, Oct. 2023.