Latency and bandwidth-aware orchestrator for QoS-sensitive applications using a reinforcement learning-based scheduler with kubernetes

Wei Huang, Andrea Araldo, Hind Castel-Taleb, Badii Jouaber

▶ To cite this version:

Wei Huang, Andrea Araldo, Hind Castel-Taleb, Badii Jouaber. Latency and bandwidth-aware orchestrator for QoS-sensitive applications using a reinforcement learning-based scheduler with kubernetes. 30th IEEE Symposium on Computers and Communications (ISCC), Jul 2025, Bologna Italy, France. pp.107824. ⟨hal-05176713⟩

HAL Id: hal-05176713
https://hal.science/hal-05176713v1

Submitted on 22 Jul 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Latency and Bandwidth-Aware Orchestrator for QoS-Sensitive Applications Using a Reinforcement Learning-Based Scheduler with Kubernetes

Massinissa Ait Aba
Davidson Consulting
Paris, France
Massinissa.ait-aba@davidson.fr

Abdenour Yasser BRAHMI
SAMOVAR, Telecom SudParis,
Institut Polytechnique de Paris
abdenour-yasser.brahmi@telecom-sudparis.eu

Hadil Bouasker
SAMOVAR, Telecom SudParis,
Institut Polytechnique de Paris
hadil.bouasker@telecom-sudparis.eu

Hind Castel-Taleb
SAMOVAR, Telecom SudParis,
Institut Polytechnique de Paris
hind.castel@telecom-sudparis.eu

Badii Jouaber
SAMOVAR, Telecom SudParis,
Institut Polytechnique de Paris
badii.jouaber@telecom-sudparis.eu

Abstract

In the realm of Fifth Generation (5G) and the upcoming Sixth Generation (6G) networks, the efficient management of network resources becomes increasingly critical, particularly for applications that have strict Quality of Service (QoS) requirements. This paper addresses the complexities associated with Virtual Network Embedding (VNE), a vital process for establishing multiple virtual networks on shared physical infrastructure within the context of network slicing. We introduce the SetpodNet scheduler, a novel orchestration solution that leverages reinforcement learning to enhance the optimization of latency and bandwidth allocation specifically in Kubernetes environments. The SetpodNet scheduler is designed to dynamically adapt to fluctuating slice arrivals and varying resource demands, ensuring that network performance remains consistent and reliable. Through comprehensive experimental evaluations, we demonstrate improvements in slice acceptance ratios and optimizing QoS.

Keywords— 5G, QoS, bandwidth, latency, Kubernetes scheduling, Virtualization, VNE.

1 Introduction

The rise of cloud-native architectures has fundamentally transformed how applications are designed and deployed, emphasizing scalability, flexibility, and resilience. As organizations increasingly adopt containerization and microservices, the orchestration of these containers has become pivotal for optimal application performance. However, the rapid evolution of cloud-native applications has necessitated a shift in resource management strategies within Kubernetes (K8s) environments. Traditional scheduling mechanisms often struggle to meet the dynamic requirements of modern workloads, particularly in the context of network slicing, which is critical for 5G and emerging 6G networks. The ability to deploy multiple virtual networks on shared physical infrastructure highlights the need for enhanced scheduling solutions that can effectively manage the complexities and resource demands of current applications [1]. While the default K8s scheduler provides basic pod placement capabilities, the increasing complexity of modern applications has led to the development of custom schedulers tailored to specific use cases [2, 3]. In earlier work, we proposed Setpod-scheduler [4], a custom K8s scheduler designed to optimize the deployment of multi-pod applications. It addresses the limitations of K8s' default pod-by-pod scheduling approach by considering all pods of an application holistically. This scheduler aims to improve resource utilization, reduce node usage, and prevent unnecessary pod deployments. Setpod-scheduler first checks for a feasible assignment of all application pods before initiating deployment, ensuring optimal resource allocation and avoiding partial deployments. It

offers features like application-aware scheduling, resource optimization, and customizable deployment sequencing, making it particularly useful for complex, multi-pod applications in K8s environments. Building upon this foundation, we propose SetpodNet scheduler [5], an enhanced version that incorporates network performance considerations. This new scheduler represents a comprehensive solution that not only optimizes computing and storage resource allocation but also accounts for latency and bandwidth requirements. SetpodNet scheduler aims to provide a more efficient scheduling solution for modern applications in K8s environments, where network performance is increasingly critical. In this context, our goal is to create a scheduler that adapts to the complexities of modern networked environments while maintaining scalability and performance.

The remainder of this paper is organized as follows. Sec.II provides an overview of the virtualization environment and the proposed model. Sec.III reviews related work on K8s scheduling, comparing various custom schedulers and their performance metrics. Sec.IV discusses the specifics of the SetpodNet scheduler. Sec.V analyzes the numerical results from experiments with the SetpodNet scheduler, comparing it to two other schedulers. Finally, Sec.VI concludes with potential future research directions.

2 Virtualization environment and model

While alternatives like Docker Swarm and Amazon ECS exist, K8s is often preferred for its large community and multi-cloud support. However, our approach could be applied to other platforms as well. In K8s, a node is a physical or virtual machine that runs containerized applications as part of a cluster managed by the K8s control plane. Nodes have resources such as CPU, memory, and storage. Additionally, SetpodNet scheduler measures latency and bandwidth between nodes and assigns separate values to each, ranging from 1 to 10, where 1 represents poor performance and 10 is optimal. Each node can run multiple pods, the smallest K8s unit, which may contain multiple containers sharing the same network and storage. Pods are created, updated, and scaled using YAML or JSON configuration files. K8s handles pod scheduling, health monitoring, scaling, networking, updates, and cleanup tasks.

In a network slicing scenario, where a single physical network infrastructure is divided into multiple virtual networks (slices) to accommodate various services or applications, pods can be utilized to deploy and manage containerized applications within these slices. In this work, we assume that a slice consists of a set of pods that host network functions. The pods can be configured with the necessary resources, such as CPU and memory requests, based on the service requirements of the network slice. These requests are used by K8s schedulers to ensure that pods are deployed on nodes with sufficient reserved resources to run them. Additionally, some pairs of pods may have QoS requests. The requests on a link between two pods are defined by two values ranging from 1 to 10 for both latency and bandwidth, where 1 indicates low requirements and 10 indicates high requirements. These requests ensure that specific service levels are maintained between interconnected pods, reflecting their varying needs for latency and bandwidth within the network slice. When two pods are connected via a physical link, the utilization of that link reduces its capacity in terms of latency and bandwidth. To account for this utilization, we subtract the values of the requests from the available capacity of the link. If two pods are on the same node, their latency and bandwidth requirements are considered satisfied. SetpodNet scheduler can treat these QoS requirements as either constraints that must be respected or best-effort requests that can be satisfied if possible, depending on the configuration of the pods. Furthermore, pods may have dependencies that require a specific order of deployment to ensure proper communication between them.

We assume that we have several slices to deploy on a physical network infrastructure managed by a network operator. A slice is deployed if all the pods of the slice have been deployed in the physical network infrastructure, respecting the appropriate order of deployment and the CPU and memory requests. If latency and bandwidth are treated as requests, our objective is to maximize the acceptance ratio by deploying as many slices as possible while also maximizing the QoS. However, if they are treated as constraints, we focus solely on maximizing the acceptance ratio. It is worth noting that modeling network slicing depends on specific requirements, constraints, and goals, and may need customization based on the infrastructure and service context.

3 State of the art

Kube-scheduler is a key component of the K8s control plane responsible for scheduling pods to run on specific nodes within a K8s cluster. It ensures that pods are deployed to appropriate nodes based on the defined constraints, such as resource requirements, affinity rules, and other scheduling policies. The default scheduling algorithm used by Kube-scheduler in K8s is known as the "default scheduler" or "predicate-based scheduler" [6], it operates in two stages. First, it applies predefined filtering rules to identify nodes eligible for pod placement based on resource requirements and constraints. Then, it assigns priority scores to these nodes, selecting the one

Table 1: Comparison of Custom K8s Schedulers.

Scheduler	**Key Features**	**Use Cases**	**Latency Aware**	**Bandwidth Aware**	**Resource Optimization**	**Open Source**
kube-scheduler [6]	General-purpose	General workloads	✗	✗	✓	✓
Setpod-scheduler [4]	Batch scheduling	Complex microservices	✗	✗	✓	✓
Kube-batch [7]	Batch scheduling	HPC, AI workloads	✗	✗	✓	✓
Volcano [8]	Batch scheduling	Big Data, AI workloads	✗	✗	✓	✓
Koordinator [9]	Coordinated scheduling	Big Data, AI workloads	✗	✗	✓	✓
K8s Descheduler [10]	Workload rebalancing	Long-running clusters	✗	✗	✓	✓
Tensile-kube [11]	Multi-cluster integration	GPU-intensive apps	✗	✗	✓	✓
YuniKorn [12]	Batch scheduling	Big data, IA workloads	✗	✗	✓	✓
Poseidon [13]	Flow network scheduling	large-scale workloads	✓	✓	✓	✓
Stork [14]	Storage-aware scheduling	Storage-intensive apps	✓	✗	✓	✓
SetpodNet [5]	Batch scheduling	Complex microservices	✓	✓	✓	✓

with the highest score for placement. In the landscape of K8s schedulers, several other notable solutions have been developed to address specific challenges. For instance, Kube-Batch scheduler [7] is an open-source custom scheduler for K8s, designed to optimize batch job scheduling. It enhances K8s' capabilities by introducing fair sharing, queue-based scheduling, and job prioritization features. This scheduler is particularly suited for AI/ML workloads and big data processing tasks that require sophisticated resource allocation and job management. Another notable batch scheduler is Volcano [8] which designed to optimize complex workloads such as AI, machine learning, and big data processing. It extends K8s' capabilities with features like queue management, job prioritization, and fine-grained resource sharing policies. Volcano introduces advanced concepts like PodGroups for co-scheduling and job dependencies, enabling more sophisticated job management. This scheduler is particularly well-suited for high-performance computing environments and multi-tenant clusters. Additionally, Koordinator [9] is a QoS-based scheduler that focuses on the efficient orchestration of microservices, AI, and big data workloads. It aims to improve runtime efficiency and reliability for both latency-sensitive workloads and batch jobs by providing fine-grained resource orchestration and isolation mechanisms. In the realm of QoS-based scheduling, K8s Descheduler [10] is a tool designed to improve cluster resource utilization by evicting pods that no longer satisfy scheduling requirements. It runs as a background process, periodically identifying pods that can be relocated to achieve better balance across nodes. The Descheduler implements various strategies such as removing duplicate pods, balancing resource usage, and removing pods violating node affinity rules. By using the default K8s scheduler to reschedule these pods, potentially onto more suitable nodes, the Descheduler helps maintain an optimized and efficient cluster state over time. For multiple K8s clusters, Tensile-kube [11] scheduler enables resource sharing. It supports various scheduling policies and can be customized for multi-cluster management, potentially including GPU resources. Developed by Apache, YuniKorn [12] scheduler enhances K8s' native scheduling capabilities by introducing features like hierarchical queues, resource fairness across queues, and gang scheduling. YuniKorn is particularly well-suited for managing big data and machine learning workloads, offering improved resource utilization and application performance.

The authors in reference [15] introduce Firmament, a flow network optimization model for cluster scheduling, offering global optimization decisions that consider the entire cluster state. This approach, based on solving min-cost max-flow problems, aims to enhance cluster utilization and reduce scheduling latency across various cluster sizes. Firmament's implementation demonstrates significant improvements in scheduling quality and speed, particularly in large-scale deployments with diverse workloads. Firmament can handle bandwidth and latency constraints between tasks, dynamically adapting to observed network utilization, thus ensuring optimal placement decisions that consider both computational and network requirements of applications. Poseidon [13] integrates the Firmament scheduler into K8s, leveraging flow network graph-based scheduling for globally optimal placement decisions.The scheduler demonstrates superior scalability compared to the default K8s scheduler, particularly as the number of nodes increases, making it ideal for large-scale deployments with diverse and dynamic workloads. During our preliminary testing of the Poseidon/Firmament scheduler, we encountered some issues related to K8s volume management, which may have stemmed from our specific setup. As a result, we chose not to include it in our comparative analysis to ensure a fair evaluation. Finally, Stork [14] is a custom scheduler designed to enhance storage management and scheduling in K8s clusters. It provides storage-aware scheduling, ensuring that pods are placed optimally in relation to their data dependencies, which improves performance for storage-intensive applications. Stork integrates closely with various storage solutions and K8s, making it an ideal choice for managing data-intensive workloads. It schedules pods on nodes where their associated data replicas are located, minimizing network hops and ensuring faster access times. The Stork scheduler indeed appears very interesting and shows similarities with our work, particularly in its approach to latency management.

Table 1 summarizes the key characteristics of various custom K8s schedulers. SetpodNet scheduler enhances its predecessor by incorporating network performance metrics into its decision-making process while retaining features like application-aware scheduling and resource optimization. This holistic approach optimizes both resource utilization and network efficiency. Additionally, it allows for defining latency and bandwidth constraints between pods, specifying the criticality of each link, which further refines pod placement based on communication requirements. Further details on SetpodNet scheduler are provided in the following section.

4 SetpodNet scheduler

SetPodNet scheduler is based on DL-ViNE (Direct Link Virtual Network Embedding), a reinforcement learning algorithm designed to solve the VNE problem. In the following, we first describe DL-ViNE and then outline the key steps of the SetpodNet scheduling process.

4.1 DL-ViNE Algorithm

VNE is a fundamental mechanism for embedding multiple dynamic virtual networks onto a physical network infrastructure. Each virtual network consists of virtual nodes and links that need to be mapped onto physical infrastructure while respecting resource constraints. Given that the VNE problem is $\mathcal{NP}$-hard [16], heuristic and approximation methods are necessary to derive feasible solutions. DL-ViNE is based on the Nested Rollout Policy Adaptation (NRPA) algorithm [17]. It efficiently maps connected virtual nodes onto the same physical node whenever possible. Otherwise, it places them closer together to minimize mapping costs. This approach minimizes resource consumption, enhances slice acceptance, and reduces latency, thereby improving overall QoS. In **Algorithm 1**, DL-ViNE takes as input the search level l and the number of iterations N, then explores the search tree by making recursive calls, structured as follows:

- At level $l = 0$, the VNE procedure is executed to embed virtual nodes and links. It returns the obtained reward ($Reward$), along with the set of physical nodes ($hNodes$) and links ($hLinks$) that host the virtual nodes and links.
- At higher levels ($l \neq 0$), multiple rollouts are performed to refine the policy matrix P and improve future embeddings. The final iteration yields the best reward ($Reward^{best}$), along with the best set of physical nodes ($hNodes^{best}$) and links ($hLinks^{best}$), marking the end of the algorithm.

The embedding process starts at level $l = 0$ with the VNE procedure, which proceeds in two stages: first, mapping virtual nodes, then mapping virtual links. In the node mapping stage, virtual nodes are selected one by one, and candidate physical nodes that meet resource constraints are identified. To optimize placement, these candidates are further filtered to prioritize mapping connected virtual nodes onto the same physical node or as close as possible. From the remaining candidates, one is probabilistically selected based on the policy matrix P. Once all virtual nodes have been successfully placed, the second stage begins. Given the set of physical nodes ($hNodes$), the procedure attempts to map virtual links one by one onto physical links with sufficient resources. If all virtual links are successfully mapped, the procedure returns the $Reward$, along with $hNodes$ and $hLinks$. However, if at least one virtual node or link cannot be mapped, the procedure terminates, returning a reward of 0 and an empty set of physical nodes and links. For more details on the algorithm, its implementation, and experimental results, refer to [18].

Algorithm 1 DL-ViNE Algorithm

Require: l: Search level , N: number of iterations, P : Policy matrix, V: Virtual network, G: Physical network
Ensure: $Reward^{best}$: Best reward, $hNodes^{best}$: Best set of physical nodes hosting the virtual nodes, $hLinks^{best}$: Best set of physical links hosting the virtual links

0: **if** $l = 0$ **then**
0: $Reward, hNodes, hLinks \leftarrow$ **VNE**(V, P, G)
0: **return** $Reward, hNodes, hLinks$
0: **end if**
0: $Reward^{best} \leftarrow -\infty$
0: $hNodes^{best} \leftarrow \emptyset$
0: $hLinks^{best} \leftarrow \emptyset$
0: **for** N iterations **do**
0: $Reward, hNodes, hLinks \leftarrow$ **DL-ViNE**($l - 1, N, P, V, G$)
0: **if** $Reward^{best} \leq Reward$ **then**
0: $Reward^{best} \leftarrow Reward$
0: $hNodes^{best} \leftarrow hNodes$
0: $hLinks^{best} \leftarrow hLinks$
0: **end if**
0: $P \leftarrow$ **ADAPT**($P, hNodes^{best}$)
0: **end for**
0: **return** $Reward^{best}, hNodes^{best}, hLinks^{best}$ =0

4.2 Scheduling process

The key steps in the scheduling process of the SetpodNet scheduler, from receiving pod specifications on a slice to the final deployment phase, are as follows:

1. **Receiving Pods:** The process begins when SetpodNet scheduler receives a new scheduling request for a pod or a group of pods associated with a slice. The K8s API server alerts the scheduler of any pending pods that need to be scheduled. Each pod request includes metadata, resource requirements, links and QoS requirements, and any other constraint specified by the slice.
2. **Resource Analysis:** Once all requests from the slice's pods are received, a directed graph is created for the slice, where the vertices represent pods with their CPU and memory requests, and the links indicate the corresponding latency and bandwidth requirements from 1 to 10, where 1 indicates low requirements and 10 indicates high requirement. Additionally, SetpodNet scheduler constructs a complete directed graph representing the K8s cluster, where each node has its own CPU and memory capacity, and each node is connected to other nodes by links with bandwidth and latency values measured by SetpodNet scheduler based on the network characteristics and conditions, measuring the values in both direc-

tions between each pair of nodes. It assigns separate values to latency and bandwidth between each pair of nodes, ranging from 1 (poor) to 10 (optimal). This setup defines an instance of the VNE problem, which will then be solved using DL-ViNE algorithm.

3. **Node Filtering:** After assessing resources, the scheduler filters out nodes that do not meet the minimum requirements (CPU and memory) of the pods. This filtering phase narrows the list of candidate nodes to those that are viable for the deployment.
4. **Solving the VNE Instance:** Using the instance of the VNE problem and the list of possible node assignments for each pod, SetpodNet scheduler employs DL-ViNE algorithm to determine the optimal assignment of each pod to the appropriate node.
5. **Binding and Deployment:** If the DL-ViNE algorithm fails to find a valid solution, no scheduling will occur. Conversely, if a solution is found, the scheduler assigns each pod to a specific node and communicates the decision to the K8s API server. This triggers the deployment phase, during which the server pulls container images, configures networking, and prepares each pod for execution on its designated node.

SetpodNet scheduler overcomes issues of the default pod-by-pod deployment, offering key advantages:

- **Holistic Resource Management:** Unlike the default scheduler, which makes decisions for each pod independently, SetpodNet evaluates the entire slice as a whole. This approach prevents inappropriate resource exploitation by ensuring that the placement of each pod considers the future scheduling of all pods within the slice. Additionally, SetpodNet optimizes node utilization by minimizing the number of nodes needed and preventing unnecessary saturation of individual nodes.
- **Reduced Waste of Resources:** With SetpodNet, the risk of deploying unnecessary pods is significantly lowered. By planning the deployment of all pods within a slice, the scheduler can ensure that resources are allocated effectively. This prevents scenarios where some pods are deployed but cannot be used due to insufficient resources for the remaining pods, thereby eliminating wasted time and energy.
- **Improved Performance and Scalability:** The SetpodNet's capability to consider bandwidth and latency values in both directions between nodes enhances overall network performance. This allows for more efficient communication between pods and contributes to improved application performance and scalability within the K8s cluster.

5 numerical results

To evaluate our scheduler, we compare SetpodNet, Stork and Kube-scheduler in several scenarios using a K8s cluster. Figure 1 shows the experimental scheme, we detail each part in the following.

5.1 Description of physical network infrastructure

The testbed is built on the Grid'5000 bare-metal platform [19], Our K8s cluster comprises nine nodes distributed across three sites in France: one master node in Grenoble and eight worker nodes spread between Grenoble, Nancy, and Rennes. The master node is equipped with 2 Intel Xeon Gold 6130 processors (16 cores each) and 192 GB of RAM. The worker nodes in Grenoble also feature Xeon Gold 6130 processors with 192 GB of RAM, while those in Nancy utilize Xeon Gold 5220 processors (18 cores) with 96 GB of RAM. The Rennes nodes are powered by Xeon E5-2630 v3 processors (16 cores total) with 128 GB of memory. Each site benefits from high-speed network connectivity, facilitating effective communication within the cluster.

5.2 Description of the emulation application

Our testing application is designed to evaluate the schedulers using a set of eight interconnected pods (pod-a through pod-h), each representing a component of a distributed application. In this setup, four pods (pod-a, pod-b, pod-c, and pod-d) are configured to send data to four other pods (pod-e, pod-f, pod-g, and pod-h), with varying communication patterns between them. The size of the data being sent from each source pod is approximately 0.5 MB. This data transfer simulates various communication patterns and workloads between the pods. We use the same application to test the three schedulers, allowing us to compare their performance in managing resource allocation and inter-pod communication with this consistent data transfer size. The use of simulated pods instead of real network slices provides flexibility and reproducibility while avoiding the complexities and costs associated with deploying a full 5G network infrastructure.

5.3 Description of the emulation scenario

We compare the three schedulers across 50 distinct scenarios involving arrivals and departures of independent 12 applications. To generate the arrivals and departures of the applications, a uniform random variable x is sampled from the range $[0, 1]$. We introduce a parameter P to serve as a threshold for distinguishing between new arrivals and

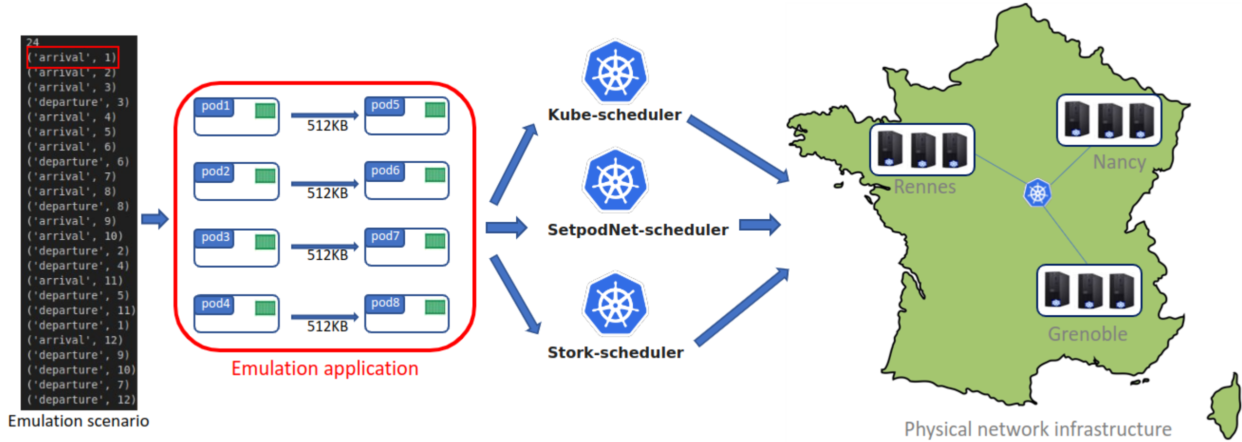

Figure 1: Experimental scheme used to compare the three schedulers.

departures. Specifically, if $x > P$, we generate an 'arrival of an application' event; otherwise, we generate a 'departure of an application' event. The parameter P can be interpreted as the departures/arrival ratio. We vary P across different scenarios to demonstrate its impact on the acceptance rate. Figure 1 shows a scenario containing the arrivals and departures of 12 applications. For each application, pods are assigned different CPU and memory requests using a uniform probability distribution. We conduct two sets of tests to evaluate the performance of three different schedulers. In the first test, pods are assigned CPU requests ranging from $800m$ to $2400m$ and memory requests from $1024Mi$ to $3072Mi$, using a uniform probability distribution. In the second test, we increase the resource demands, assigning CPU requests from $3200m$ to $9600m$ and memory requests from $4096Mi$ to $12288Mi$, again using a uniform probability distribution. This approach allows us to compare scheduler performance under different resource demand scenarios, simulating both moderate and high-resource workloads. For QoS constraints, we randomly generate a value between 5 and 10 for both latency and bandwidth between each pair of connected pods using a uniform distribution. This range represents average to high QoS requests, ensuring that the network slice accommodates services with moderate to stringent performance requirements. By setting the values within this range, we prioritize maintaining a reasonable level of service quality while still allowing flexibility in resource allocation. This information is considered only by SetpodNet scheduler, prompting it to adhere to the specified QoS constraints. The objective is to maximize the acceptance ratio by deploying as many applications as possible while also maximizing the QoS. We employ three criteria for conducting comparisons: the acceptance ratio, which measures the percentage of applications successfully deployed on the cluster; the average transfer time (latency), representing the average transfer time per pair of connected pods; and the average deployment time, which compares the average time taken for deployment per application. Figure 2 and Figure 4 show the comparison results for the first and second tests, respectively, among the three schedulers for $P \in \{1, 3, 5\}$. Similarly, Figure 3 and Figure 5 display the percentage of connected pods deployed on the same node or site for the first and second tests, respectively. Furthermore, we compare two versions of SetpodNet scheduler. In the first version, we measure latency and bandwidth before each application deployment. In the second version, we only measure latency and bandwidth once, at the initial deployment of SetpodNet scheduler, and then remeasure only if a node is added or removed from the cluster, which does not occur in these tests. The data used in this study are available at [5].

5.4 Experimental results

From Figure 2, we can see that the acceptance ratio is 100% for all three schedulers due to low resource requests. In terms of average latency, a topology aware scheduler, like SetPodNet, can reduce latency by 75% compared to kube-scheduler and by 85% compared to Stork, with a slight advantage observed for SetpodNet with update. This difference is expected, as neither kube-scheduler nor Stork considers network topology when placing pods. Since these schedulers do not optimize for network-aware placement, their higher latency results are a natural outcome of their design. In contrast, SetPodNet minimizes latency by prioritizing the placement of connected pods on the same site and node, as illustrated in Figure 3. For average deployment time, SetpodNet with updates takes approximately 82 seconds, while SetpodNet without updates, kube-scheduler, and Stork scheduler each takes around 10 seconds.

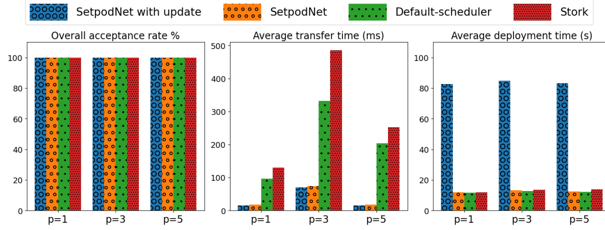

Figure 2: Comparison of the three schedulers for low resource requests.

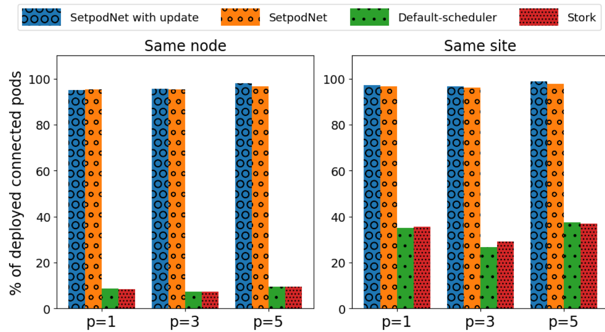

Figure 3: Percentage of connected pods deployed on the same node or site for low resource requests.

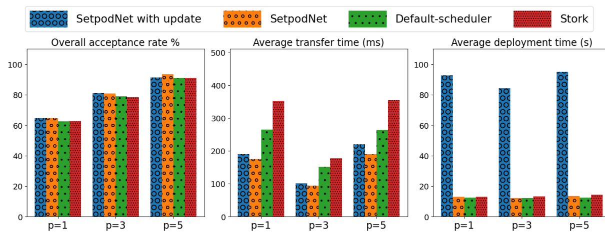

Figure 4: Comparison of the three schedulers for high resource requests.

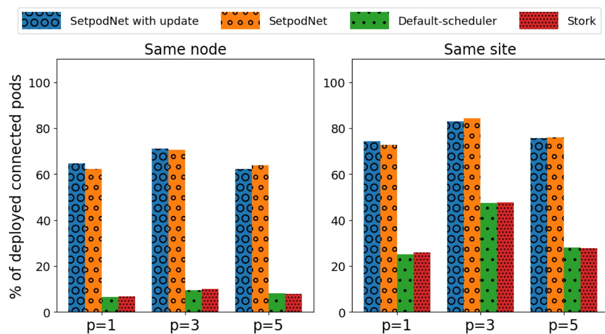

Figure 5: Percentage of connected pods deployed on the same node or site for high resource requests.

This difference is due to the tests performed by SetpodNet to measure latency and bandwidth between each pair of nodes (in both directions) within the cluster.

From Figure 4, we notice that by decreasing the value of P, the acceptance ratio decreases for the three schedulers due to high resource requests, with a slight advantage observed for SetpodNet scheduler. This is quite logical, because the smaller the value of P, the higher the number of simultaneous application arrivals. Deploying several applications without releasing them saturates the resources of the physical infrastructure, which implies the rejection of several applications. For average latency, SetpodNet performs the best, reducing latency by $\sim$ 30% compared to kube-scheduler and by $\sim$ 40% compared to Stork, with a slight advantage observed for SetpodNet without updates. The difference between SetpodNet and other schedulers is less significant in this second test because SetpodNet was unable to place many pairs of connected nodes on the same site, and even fewer on the same node, as shown in Figure 5. This is due to the increasing resource consumption, which reduces the availability of sufficient resources on a single node, limiting the flexibility to group connected pods together while still meeting their individual resource demands. For average deployment time, SetpodNet with updates takes approximately 90 seconds, while SetpodNet without updates, kube-scheduler, and Stork scheduler each around 10 seconds. This extended duration is attributable to the additional network measurements performed by SetPodNet to assess latency and bandwidth between nodes before making placement decisions.

These observations highlight the impact of resource allocation strategies on scheduler performance. In the first test, the 100% acceptance ratio reflects the effectiveness of low resource requests, allowing SetpodNet to optimize pod placement and achieve significantly lower latency. However, in the second test, the decrease in acceptance ratio with lower P values reveals the challenges posed by high application arrival rates. Despite SetpodNet still performing best in average latency reduction, its inability to efficiently place connected pods under higher loads indicates a need for adaptive strategies to maintain performance in varied resource scenarios. Additionally, the time required for latency and bandwidth measurements can be significant, limiting their effectiveness, especially in dynamic environments where different schedulers may accept and deploy varying applications.

6 Conclusion and Future Perspectives

In this work, we compare three schedulers to effectively deploy the virtualized resources of different applications within a K8s physical infrastructure. The goal is to maximize the acceptance ratio by deploying as many appli-

cations as possible while also maximizing the QoS. The performance evaluation of the SetpodNet scheduler reveals its effectiveness in optimizing latency and acceptance ratios under low resource request conditions. Its strategy of maximizing the placement of connected pod pairs significantly enhances performance, particularly when resources are abundant. However, challenges arise as resource demands increase, indicating the need for adaptive strategies to maintain efficiency under varying loads.

For future work, incorporating real 5G network slices instead of simulated pods would provide a more accurate assessment of SetPodNet's performance in a real-world network environment. While simulated pods offer flexibility and reproducibility, real network slices would introduce more realistic constraints, including dynamic traffic variations and hardware limitations, albeit at the cost of increased complexity and deployment challenges. Another important improvement would be optimizing the average deployment time for SetPodNet with updates. Since latency and bandwidth measurements are required for each deployment, refining these processes could help reduce overhead, making scheduling faster and more responsive to dynamic network conditions. Potential optimizations could include caching previously collected network metrics, performing measurements asynchronously, or using machine learning models to estimate link performance instead of measuring it in real time. Additionally, given that kube-scheduler and Stork are not topology-aware, future evaluations should include comparisons with other network-aware orchestrators. This would allow for a more rigorous benchmarking of SetPodNet, ensuring that its advantages stem from superior scheduling strategies rather than simply the presence of topology awareness. Expanding the comparison set would help identify further improvements and validate SetPodNet's effectiveness in optimizing pod placement based on network constraints.

References

[1] K. Senjab, S. Abbas, N. Ahmed, and A. u. R. Khan, "A survey of kubernetes scheduling algorithms," *Journal of Cloud Computing*, vol. 12, p. 87, 2023.

[2] E. Gough, "Evaluation of kubernetes schedulers for a community cloud computing model," in *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*, 2024, pp. 1–7.

[3] Z. Rejiba and J. Chamanara, "Custom scheduling in kubernetes: A survey on common problems and solution approaches," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.

[4] M. Ait Aba, M. Kassis, M. Elkael, A. Araldo, A. A. Khansa, H. Castel-Taleb, and B. Jouaber, "Efficient network slicing orchestrator for 5g networks using a genetic algorithm-based scheduler with kubernetes: Experimental insights," in *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*, 2024, pp. 82–90.

[5] "Setpodnet-scheduler, https://github.com/aidy-f2n/setpodnet-scheduler," 2024.

[6] "kube-scheduler:v1.30.5, https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/," 2023.

[7] "kube-batch, https://github.com/kubernetes-retired/kube-batch," 2017.

[8] "Volcano, https://github.com/volcano-sh/volcano," 2018.

[9] "Koordinator, https://github.com/koordinator-sh/koordinator," 2022.

[10] "Descheduler, https://github.com/kubernetes-sigs/descheduler," 2017.

[11] "Tensile-kube, https://github.com/virtual-kubelet/tensile-kube," 2020.

[12] "Yunikorn, https://yunikorn.apache.org/docs/," 2022.

[13] "Poseidon, https://github.com/kubernetes-retired/poseidon," 2018.

[14] "Stork, https://github.com/libopenstorage/stork," 2017.

[15] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Assoc, Nov. 2016, pp. 99–115.

[16] M. Rost and S. Schmid, "Np-completeness and inapproximability of the virtual network embedding problem and its variants," *arXiv preprint arXiv:1801.03162*, 2018.

[17] C. D. Rosin, "Nested rollout policy adaptation for monte carlo tree search," in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[18] AIDY-F2N, "Dl-vine," https://github.com/AIDY-F2N/DL-ViNE, 2025.

[19] D. Balouek *et al.*, “Adding virtualization capabilities to the grid’5000 testbed,” in *Cloud Computing and Services Science: Second International Conference, CLOSER 2012, Porto, Portugal, April 18-21, 2012. Revised Selected Papers 2*. Springer, 2013, pp. 3–20.