Cost-aware service placement and scheduling in the Edge-Cloud Continuum

SAMUEL RAC and MATS BRORSSON, University of Luxembourg, Luxembourg

The edge to data center computing continuum is the aggregation of computing resources located anywhere between the network edge (e.g., close to 5G antennas), and servers in traditional data centers. Kubernetes is the de facto standard for the orchestration of services in data center environments, where it is very efficient. It, however, fails to give the same performance when including edge resources. At the edge, resources are more limited, and networking conditions are changing over time.

In this paper, we present a methodology that lowers the costs of running applications in the edge-to-cloud computing continuum. This methodology can adapt to changing environments, e.g., moving end-users. We are also monitoring some Key Performance Indicators of the applications to ensure that cost optimizations do not negatively impact their Quality of Service. In addition, to ensure that performances are optimal even when users are moving, we introduce a background process that periodically checks if a better location is available for the service and, if so, moves the service. To demonstrate the performance of our scheduling approach, we evaluate it using a vehicle cooperative perception use case, a representative 5G application. With this use case, we can demonstrate that our scheduling approach can robustly lower the cost in different scenarios, while other approaches that are already available fail in either being adaptive to changing environments or will have poor cost-effectiveness in some scenarios.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Theory of computation** → **Scheduling algorithms**; • **General and reference** → Performance; • **Computing methodologies** → Model development and analysis.

Additional Key Words and Phrases: Cloud computing, Edge computing, Scheduling, Container Orchestration, Resource allocation, 5G, Kubernetes

1 INTRODUCTION

Edge computing is a paradigm that brings computing capabilities closer to the sources of data. Edge computing helps to reduce network delays and bandwidth usage, and to improve security and privacy by keeping the data local. Therefore, to use edge resources, application developers need tools to deploy software effortlessly at the edge. The current practice is mostly to statically predetermine which services in a distributed application should execute at the edge and which elsewhere.

The *data center to the edge computing continuum* has recently become a concept meaning the aggregation of resources located in both traditional data centers, at the edge of the network, and at any node in between. A service may be deployed to any node in this computing continuum. However, the geographical location of that node affects the latency with which the service communicates with end-users and/or with other services. Thus, an application scheduler for the edge-to-cloud continuum should consider more parameters than what is needed

Extension of Conference Paper: This article extends our IEEE IC2E 2023 [21] conference paper. This journal version presents a new scheduling approach based on communication patterns. This new approach improves the results of the approach initially presented. In addition to this new approach, we propose the following contributions as new content: a simulation to motivate our study better and additional experiments and implementation details.

Authors' address: Samuel Rac, samuel.rac@uni.lu; Mats Brorsson, mats.brorsson@uni.lu, University of Luxembourg, Luxembourg.

ACM 1544-3566/2024/1-ART
https://doi.org/10.1145/3640823

for a traditional data center. In a data center, the network topology is horizontal, every node has direct access to the other nodes with high bandwidth and low latency. Networking conditions with edge nodes are different. The latencies and the available bandwidth vary from node to node. Furthermore, network conditions can evolve over time; the network is more likely to be congested due to more limited bandwidth, or the latencies can change if an end-user is moving or the communication pattern changes. Edge nodes should not be chosen only for their processing capabilities but also for their geographical location; the location of a node modifies the latency with end users.

Deploying applications in the cloud-to-edge computing continuum should be easy for developers [20]. We believe that developers should focus on the business logic, and define requirements (e.g., CPU, memory, latencies, and other Service Level Objectives) but not need to think about the physical locations of their services. Furthermore, manual and static placement of services makes it more difficult to scale up the application. Deploying at the edge should be seamless, as easy as it is when deploying applications in traditional data centers.

Edge computing can help reduce the costs of deploying an application. Cloud providers offer their resources as *Everything as a Service (XaaS)* [3]. Therefore, the customers pay only for the resources they use. E.g. customers pay an hourly rate for the server they use. In addition, cloud providers also charge for the traffic that goes outside of their data centers. Deploying network-intensive applications at the edge may save the outgoing network costs; data stays local and traffic between the edge and the data center is not charged.

We propose two schedulers to lower the costs of deploying distributed applications in the cloud-to-edge continuum. We consider applications that consist of several services that need to interoperate. To make application deployment easy and as close as possible to industrial standards, we implement our methodology as a Kubernetes scheduler plugin [9] based on network requirements and costs. Any containerized application can be deployed and make use of our schedulers. To reduce the costs of running applications, we want to leverage local data processing to save bandwidth costs.

When end-users (e.g. a phone, a vehicle, or a camera connected to an application hosted in the computing continuum) are moving, the delay to connect or communicate with a service of the application may become longer. This means that the optimal location of a service may vary over time. We need to be able to move services automatically when users are moving, otherwise the Quality of Service (QoS) will deteriorate. Therefore, we also propose a *rescheduler* to monitor the application and trigger service migration when needed. Our rescheduler is a background process that periodically checks costs and KPIs to identify better nodes on which a service can be deployed. If the rescheduler detects an improvement, it will automatically migrate the service pod. Also, when a cheaper resource becomes available, the rescheduler can detect it and move the associated service to reduce the costs (provided the QoS is not negatively affected).

We make the following key contributions in this work:

(1) Design and implementation of two cost-aware Kubernetes scheduling plugins. One based on latency between a service and an end-user (or another service), and the second based on communication patterns.
(2) Design and implementation of a network-aware rescheduler to keep application placement decisions optimal when end users are moving.
(3) A performance evaluation of our scheduling plugins and the rescheduler methodology using a realistic 5G use-case of cooperative perception for autonomous vehicles.

From our experiments, we have verified that our custom schedulers can achieve in most cases about 10% to 25% lower costs than the default Kubernetes scheduler. It is worth to note that this is not a simulation study. We have made real scheduler plugins demonstrated in a real Kubernetes cluster.

We use the Kubernetes terminology here as they are strictly not schedulers but rather *placement algorithms*.
A *pod* is the smallest schedulable unit in Kubernetes, containing at least one container

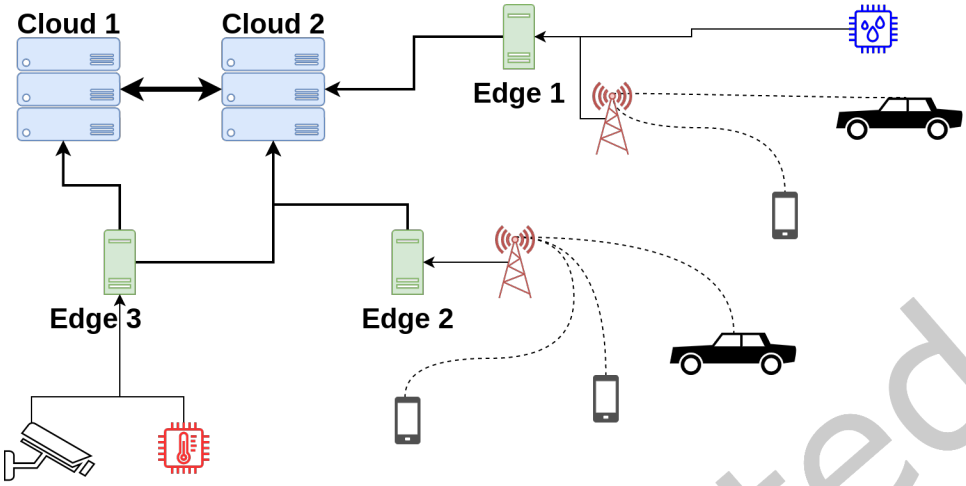

Fig. 1. The Edge-to-Cloud Computing Continuum - Computing resources are in blue and green.

The rest of the article is organized as follows. Section 2 presents the relevant background and outlines our proposal from a simulation study. Then, we present the relevant related work in section 3. Our scheduling methodology is exposed in section 4, and in section 5 we present our evaluation methodology which is based on a vehicular cooperative perception workload running on a Kubernetes cluster. Finally, we present our conclusions and future work directions in section 6.

2 BACKGROUND AND MOTIVATION

2.1 Background

We define the *edge-to-cloud computing continuum* as the aggregation of servers located at the edge of the network and those in traditional data centers (cloud). The edge nodes are servers located at the edge of the network, outside of the data centers, e.g., near 5G base stations. Fig. 1 represents such an edge-to-cloud-computing continuum. The blue servers are the cloud nodes, and the green ones are the edge nodes.

End-users (e.g., vehicles, phones, cameras, or smart sensors) are connected to services hosted in the computing continuum. The end-users may be connected directly to cloud centers or to an edge node. Some end-users can be mobile, and in this case, their closest node changes over time.

We are considering distributed *applications*, made of *services*, to be deployed on this cloud-to-edge computing continuum. A service can run on an edge node as well as on a cloud node. A *scheduler* maps the services to the nodes. The *rescheduler* is a background process that can update a service allocation. It can move a service to another node in the cluster.

There are two kinds of costs in our edge to data center computing continuum: instance cost (price paid by hours and number of CPU used), and communication costs (per GB). It is possible to define additional costs like storage (per GB), but these are outside of the scope of this study.

Communication costs represent the price to pay to transfer data from the edge to a data center. We are not counting the transfer price between an edge node and its end-user (e.g., through 5G). This parameter can be optimized by the choice of an internet provider or a telco operator and is outside of the scope of this study. Also, if two data-center nodes are part of one data-center, we are not counting communication costs. But, communication costs apply for transfers between two edge nodes.

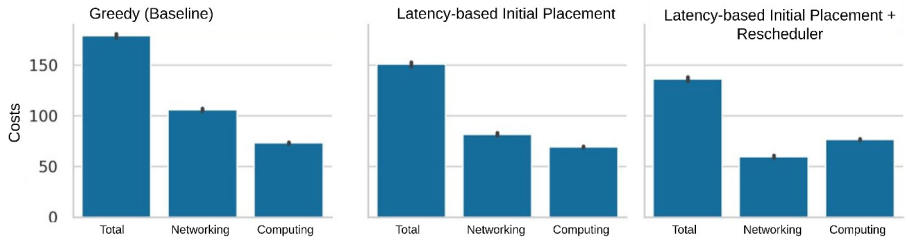

Fig. 2. Results of the simulation - Costs are detailed in two parts: computing and networking

Different kinds of nodes could be added to this computing continuum. The scheduling decisions are based on node characteristics such as available resources, bandwidth, latency, or costs. We use static node types like edge and data center nodes to understand the computing continuum behaviors better.

2.2 Motivation

Our service initial placement method and our rescheduler can help reduce the total costs of running an application in the edge to data center computing continuum. We can illustrate how to perform cost optimization with the following example. We have a car connected to a video processing service that detects other vehicles. There are three main configurations for placing this service: on a data center node (paying the data center instance price and the communication), on the edge node close to the car (paying only the instance price), or on a different edge node (paying the edge node instance price and the communication). Video processing is a bandwidth-intensive application, it is cheaper to choose the edge node where there are no communication costs (because the data stays local). We need the service initial placement method to find a tradeoff between paying instance costs and communication costs. However, when the car is moving, the previously selected edge node is not optimal anymore; it stops processing local data. We need a background process (rescheduler) to watch the cluster state and migrate the services when necessary; e.g., when a car is moving, we need to move the service to the edge node close to the car.

In order to understand the room for improvement of distributed applications in a cloud-to-edge computing continuum, we designed a simulation experiment using our scheduling methodology. This simulation demonstrates the benefits of our scheduling methodology on a larger cluster (around 50 nodes).

The simulator is a C++ program that can generate random graphs with data center and edge nodes. We use a synthetic workload which is a simple application with an end-user client located at the edge (e.g., a car) communicating with a server (e.g., a video processing service). The server service can be deployed anywhere in the simulated edge-to-cloud computing continuum. The clients are randomly moving between different edge nodes, and new clients are randomly arriving following a Poisson distribution.

We ran an experiment with graphs containing 20 data center nodes and 30 edge nodes on which we deployed 10 to 20 services. In this scenario, we are considering the edge nodes to be more expensive (i.e., higher price per CPU and per hour) due to their limited resources. We are comparing three different approaches in this simulation: i) a greedy approach (choosing the cheapest node at each time step), ii) a latency-based initial placement (LIP) described in section 4.2 iii) the latency-based initial placement, and the rescheduler (LIP+RS) described in section 4.3. The LIP approach minimizes the latency between the client and the server.

Given that the simulation is a randomized process, we ran each experiment ten times and reported the average. Fig. 2 presents the results showing the cost of running the workload for different approaches. The lower the cost, the better. Using the latency-based initial placement without or with the rescheduler outperforms the greedy strategy. Greedy is taking the best decision at each time step; however, this does not lead to a globally optimal

solution. Using our methodology, we can find a better global solution. The rescheduler is choosing more expensive nodes to save data transfer costs, this explains why it gets better results than the initial placement approach only.

This simulation shows that our methodology outperforms a greedy strategy on a large graph (50 nodes) with a simple workload. It is possible to lower the costs of running applications by using servers at the edge to save networking costs. This encouraged us to make a real implementation suitable for Kubernetes integration.

2.3 Proposed solution

We think that Kubernetes is one of the best candidates for orchestrating applications over the whole cloud-to-edge computing continuum. Kubernetes is the de facto industrial standard for container orchestration. It is not only limited to Docker containers [2], but it can manage any artifact following the Open Container Initiative (OCI) specification [5]. Using standard containers is important to ensure compatibility with most of the different hardware (e.g., with different CPU architectures) we can find in this computing continuum. Also, it is easier for the industry to adopt this technology if they can continue using the same containers they already have.

However, Kubernetes is not yet ready to orchestrate resources over the whole cloud-to-edge continuum. The default scheduling approach of Kubernetes is to spread the containers over the cloud, choosing the least allocated server. This is good practice in a traditional data center as it avoids server overload. But, this is not applicable to edge nodes. It is important to consider the geographical location of the servers in order to achieve low latency to access the services. Therefore, it is not possible to place services in the whole computing continuum efficiently without considering the networking resources.

We propose to use and extend Kubernetes to orchestrate applications deployed in the cloud-to-edge computing continuum. With a network-aware scheduling approach, we think it is a good candidate for orchestrating heterogeneous resources. Using the Kubernetes self-healing mechanism also helps improve the reliability of the system. If an edge node fails or if it has networking issues, its pods can be redeployed on another available node.

In this heterogeneous Kubernetes cluster, we define a cost policy based on two values: CPU and network usage. Instance price is based on the time and the number of CPUs used. Networking costs are charged based on the amount of data transferred. If data stays in the same geographical location, networking costs do not apply. They only apply when data is transferred from one region to another, e.g., from one edge location to a central data center. This cost policy is in line with what public cloud providers use.

We use the Kubernetes scheduler framework [9] to implement our scheduling methodology for the edge-to-cloud computing continuum. The Kubernetes scheduling framework is easy to extend. Every scheduling stage is defined as a plugin; we can write new plugins to replace the default ones.

There are four main scheduling stages in the *kube-scheduler*: *filter*, *score*, *normalize score*, and *reserve*. The scheduling pipeline first selects an unallocated pod (i.e., a set of containers, the atomic schedulable unit in Kubernetes) from the scheduling queue. Then, un-schedulable nodes (e.g., reserved nodes, control plane only, out of resources, etc.) are filtered and removed from the possible nodes. After the filtering stage, the remaining nodes are scored, and the scores are normalized to between 0 and 100. Normalization is necessary to get an average score when using many scoring plugins. The node with the highest score is selected and reserved for the pod. If two nodes get the same score, one of them is randomly selected.

In order to extend the Kubernetes scheduler for the edge-to-cloud computing continuum, we propose a twofold solution: a service *initial placement* method and a *rescheduler* (RS). The service initial placement assigns a node to every service. The rescheduler watches the existing services and migrates those that are not in an optimal location. When the optimal solution changes over time, a rescheduling process is necessary to adapt the service locations. E.g., when end-users are moving, the optimal solution changes over time.

We propose two methods for the service initial placement: One based on latency and the other based on communication patterns.

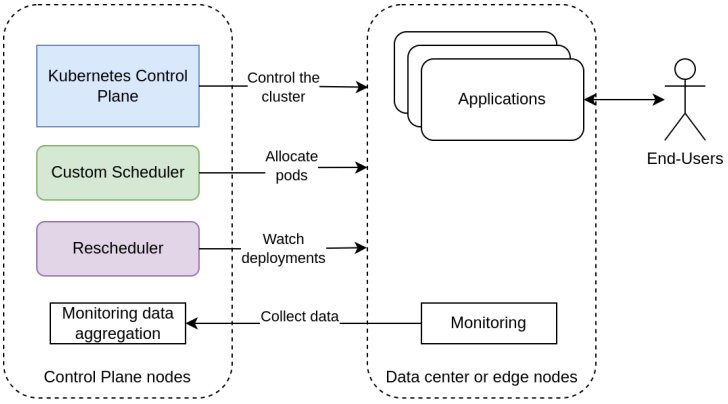

Fig. 3. Architecture overview of the system. Our scheduling components are running on control plane nodes. Applications can run anywhere in the cloud-to-edge computing continuum.

The latency-based initial placement method assigns unscheduled services to nodes. To select which node will host a service, we rank the nodes using both their costs and their latencies to other nodes. In the scope of this study, the cost is the money paid to a cloud provider to run a service. Other quantities, such as energy (to power the servers and the network equipment), can also be considered and minimized using this methodology, which we will do in future work. Checking the delays with other nodes helps to reduce the end-to-end latency of an application and ensure the quality of service.

The communication-based initial placement method also assigns unscheduled services to nodes. However, it chooses to host services using a different approach. Instead of using latency, this scheduler uses the communication patterns between services (inter-service traffic) and their end-users. The scheduler uses the communication patterns of previously deployed services to predict the behavior of the new services. It assumes that new services with the same code will have similar behavior.

The rescheduler is a background process that re-evaluates the initial placement decision of the services and moves the services to different nodes if it finds a better solution. The optimal solution changes over time, users can be moving, or new resources may become available. Using the rescheduler ensures keeping the costs low by taking new placement decisions; moving the services to a cheaper node if it is possible.

We include a monitoring system in our placement methodology to support the scheduler and the rescheduler. The monitoring system collects metrics about the cluster state and the applications running in the cluster. Also, it stores the metrics and serves them to the scheduling components. The monitoring system can serve raw data or aggregated data, e.g., using basic statistical functions such as an average or a median. The implementation of the monitoring system relies on open-source components. It is described in section 4.4.4.

Fig. 3 presents a high-level view of our system. We run a custom scheduler to deploy applications in the edge to cloud computing continuum. There are two versions of this custom scheduler: one implementing the latency-based approach and the other implementing the communication-based approach. We use the custom scheduler to orchestrate workloads, but the default scheduler is still usable to deploy the control plane and the monitoring functions. In the cluster, we reserve nodes for the control plane (in a traditional data center) and nodes for deploying workloads (at the edge or in traditional data centers). The custom scheduler and the rescheduler are running on the control plane nodes of the cluster. These nodes are located in a traditional data center. Applications can run on nodes located either at the edge of the network or in traditional data centers.

3 RELATED WORK

In this section, we present some research related to the scheduling of services in the edge-to-cloud continuum. In this study, we define the scheduling of services as the mapping of a service to a server where it can run. We organize the different scheduling methodologies into three categories: cost-aware, network-aware, and QoS-aware scheduling.

Cost-aware scheduling. Lai et al. present a cost-aware scheduler in [11]. They use a heuristic approach (most capacity first) to maximize the number of allocated edge users while minimizing the number of necessary servers at the edge. This work has no mechanism to move applications when end-users are moving. In addition, scheduling on edge nodes is outside of the scope of this study.

The authors of articles: [12, 26] present approaches to reduce costs by improving the Kubernetes scheduler. However, their main interest is in cloud computing and cannot be extended over the whole continuum without additional work. Li et al. [12] present a meta-heuristic-based scheduler that minimizes the energy costs of CPU, RAM, and network usage in addition to the networking costs of offsite nodes. They also propose a rescheduler to monitor changes in business requirements. This study only focuses on the cloud environment, where computing resources and resources are mostly homogeneous (e.g., same kind of server hardware, same latency between the nodes). Also, the scheduling decision does not consider any latency requirements. Zhong et al. [26] propose a scheduling methodology that reduces the number of allocated Virtual Machines when using the Kubernetes Autoscaler to lower the instance costs. To save costs, they propose to use a background process that checks if it is possible to shut down a server and migrate its pods to another node. They try to maximize resource utilization to save costs. This approach would have a limited impact on the scope of edge computing, node geographical location is an important parameter to consider in order to lower latency or reduce backhaul networking costs. In addition, this work does not consider the costs of the traffic going outside of data centers. Outgoing traffic is expensive when a network-intensive application is not deployed in the same location as its end users.

Network-aware scheduling. Kaur et al. [7] present a scheduling algorithm that minimizes inter-service communication delays. It relies on two heuristic approaches: a greedy and a genetic algorithm. This study makes the assumption that data traffic between the services of the application is known. It is not the case in practice with most applications. It would require additional work to specify or learn the data-traffic pattern. This limitation makes their approach difficult to use with real applications. Also, this study does not use any background process to monitor the movements of the end-users at the edge. Marchese et al. have proposed using a rescheduling mechanism [13, 14]. In [14], they present a network-aware scheduler plugin and a descheduler that checks if a node with a better score can be found. The proposed scoring method is not effective for initial placement. It relies on the input from previous data traffic that is null at the initialization. It is worth mentioning that our scheduling approach is not based on this work; we independently built similar experimental setups (based on common open-source software).

In [13], the authors present a network-aware scheduler plugin to extend the InterPodAffinity module from their previous work. Their approach automatically updates the static Kubernetes manifest with real-time data collected from the cluster. In addition, they are using a Kubernetes controller that updates the manifests and triggers rolling updates if an improvement can be found. However, initial placement is not as good as the default approach in some cases. They need a few iterations or a larger workload to be better than the default approach.

Wojciechowski et al. [24] present a data traffic-aware scheduler that minimizes inter-node communications. This study does not handle the case of moving user equipment. Also, it does not consider latencies between the nodes.

In [23], Toka presents a latency-aware scheduler that maximizes resource utilization at the edge. He also introduces a rescheduler that can improve application placement over time. However, the inter-service data traffic is not considered in this work.

QoS-aware scheduling. Mattia and Beraldi [15] present a reinforcement learning based scheduling approach that improves the stability of the frame rate of AR/VR applications. The experimental results are limited to a simulation; applying this methodology on a real Kubernetes would require a large dataset for the training stage.

The Polaris scheduler is presented in [19]. It is an SLO-aware (Service Level Objective) scheduler that considers many network metrics. The authors extend many Kubernetes scheduler plugins (pre-filter, filter, and score) to consider the topology of the cluster, the dependencies of the services, and the SLOs. However, no rescheduling mechanism is presented in this study. Also, the long computing time for placement is a problem in a dynamic environment where application placement needs to be often reevaluated. In [16], Orive et al. present a scheduling approach to minimize the application end-to-end (E2E) latency and maximize E2E reliability. They propose an architecture to define the application requirements. Their Kubernetes scheduler plugin uses these requirements to score the nodes. Nautilus [6] is a run-time system that maps micro-services to nodes based on communication overhead, resource utilization, and IO pressure.

Table 1 summarizes the different scheduling methodologies described in this section.

4 SCHEDULING METHODOLOGY

In this section, we are presenting an overview of the optimization problem we are solving. Then, we present the algorithms for i) the scheduler scoring plugins, and ii) the rescheduler. Finally, we expose how we implement this scheduling methodology in a Kubernetes cluster.

4.1 Optimization problem overview

This study presents a scheduling methodology that minimizes the cost of deploying applications in the cloud-to-edge computing continuum. We are considering two different kinds of costs: i) computing costs and ii) networking costs.

Computing cost is the price to pay to use a server from a cloud provider. It is usually charged by hour or month and depends on the characteristics of the machine. In this study, we make the assumption that every node in the computing continuum is using the same pricing policy: an hourly rate that depends only on the characteristics of the instance. Note that we are not labeling the edge nodes as special nodes; all the nodes are the same for the scheduler. Only the instance characteristics (e.g., processors, memory, geographical location) and its networking capabilities (e.g., latencies with other machines, available bandwidth) matter for the scheduling decision.

Table 1. Related work summary

Approach	Cost-aware	Network-aware	QoS-aware	Rescheduling mechanism	Using Kubernetes
Kubernetes [9]	-	-	-	-	✓
[11]	✓	-	-	-	-
[12, 26]	✓	-	-	✓	✓
[7]	-	✓	-	-	-
[13, 14, 23, 24]	-	✓	-	✓	✓
[15]	-	-	✓	-	-
[6, 16, 19]	-	-	✓	-	✓
Our approach	✓	✓	-	✓	✓

Networking cost is the price to pay to send data over the network. This price depends on the quantity of data sent and its destination. Data traffic is not charged for machines in the same data center. However, if data goes outside of the data center to the edge, then, traffic is charged. Also, no additional network cost is charged if edge data is processed locally. However, if data traffic is sent to a data center, then network traffic is charged.

Our optimization objective is to find a tradeoff between paying computing capabilities and networking costs. Depending on the workload characteristic it can be cheaper to deploy the application in a data center or at the edge. For instance, it is cheaper to deploy a network-intensive workload at the edge where data is produced rather than in a data center since data stays local.

To find an approximate solution to the problem defined above, we are using heuristic optimization as shown in Fig. 4 which presents a high-level view of the scheduling workflow. The first part details the initial placement stage where the scheduler assigns an unallocated pod to a node. Then, the rescheduler periodically checks in the background if a better node can be found for this pod.

4.2 Service initial placement

The objective of service initial placement is to associate the unallocated pods from the scheduling queue with a node in the edge-to-cloud computing continuum. The first step is to build a list of nodes where the pod can run. We remove from that list the nodes that cannot host the pod (e.g., reserved for the control plane, not enough resources). Then, the nodes are scored, and the best one is selected to host the pod.

We propose two Service Initial Placement algorithms that share the objective of minimizing the total costs. To reach the objective, the methods are finding an initial location for deploying a service.

4.2.1 Latency-based Initial Placement. Algorithm 1 describes the node scoring method of the Latency-based Initial Placement (LIP) approach. The LIP scheduler runs the corresponding scoring method for each schedulable node. The scoring method sorts the nodes using the networking delays and their price (i.e., the hourly price paid to use a server). We choose a node close to the connected services to improve the end-to-end latency and lower the networking costs (saving bandwidth).

Nodes that are close to connected services get a higher score. To reduce the distance between a pod and the connected services, we defined a list of these services for each pod: *pod.dependencies*. Then, we get a list of the nodes where these services are deployed. We do not add the server to the list if the service is not deployed yet.

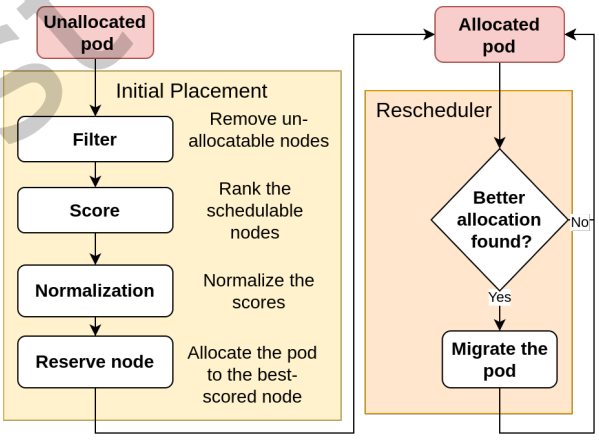

Fig. 4. Scheduling workflow: Initial Placement (yellow), Rescheduler (orange)

Algorithm 1 Latency-based Initial Placement (LIP) Scheduler: Scoring

Require: $\alpha > 0$, $\beta > 0$, Pod, Node
  $n_s \leftarrow 0$ ▷ Node score
  for service in Pod.dependencies **do**
    $\lambda \leftarrow GetLatency(Node, service)$
    if $\lambda < \alpha$ **then**
      $n_s \leftarrow n_s + \beta$
    end if
  end for
  $n_s \leftarrow n_s + \frac{1}{Node.Price}$
  return n_s

Once the server list is built, we evaluate the latencies between these servers and the evaluated node. If the latency is lower than α, we add β to the final score (α is a latency distance in ms, and β is a score modifier.). The more connected services in a radius of α, the higher the score.

To lower the deployment costs, we add the inverse of the node price to the final score. The lower the node price, the higher the score.

The values of α and β should be chosen regarding the cluster characteristics. α should be chosen relative to the values of the latencies between the nodes. β should be chosen relative to the values of the inverse of the node price.

In every case, this algorithm selects a node for the pod to deploy. If two nodes get the same score, one is randomly selected by the scheduler.

4.2.2 Communication-based Initial Placement. This approach uses the knowledge from the communication patterns of already deployed pods to get an accurate estimation of the networking costs. Using this information, the Communication-based Initial Placement (CIP) scheduler can choose the cheapest node. I.e., with the lowest sum of instance and networking costs.

The details about how to collect the communication patterns and how to implement them are detailed in section 4.4.2. The main component of this implementation is the *traffic exporter*. It provides the information necessary for the networking cost estimation to the scheduler.

The traffic exporter needs to access previous communication patterns to provide accurate estimations. The first service is placed using only the instance price. Then, the following ones can use the estimation of the traffic exporter.

4.2.3 Initial placement methods comparison. The CIP approach presents many benefits over the LIP approach. CIP does not require a manual setting of its parameters, while we need to set LIP parameters: connected services, α, and β, which are not always trivial to decide. Also, using an estimation of the networking costs (in the CIP approach) leads to better performance.

However, the CIP approach needs to collect some data before it can be effective. The CIP scheduler starts collecting data when the first pod is deployed. In practice, this startup time is not an issue, according to experimental results presented in section 5.3.2. Future work may however investigate a hybrid approach: using the LIP scheduler while the CIP is collecting the initial data, and then using the CIP scheduler. The LIP scheduler could be used to deploy the first instance of each pod, and the CIP scheduler could be used to deploy the following instances.

4.3 Service rescheduling

The rescheduler is a background process that periodically checks if a better node is available to host a pod. We build the service rescheduler for two main reasons: i) the best location for a service varies over time (e.g., node availability changes depending on the current load, end-users are moving), ii) we can use data collected when the service is running to improve the scheduling decision. If the rescheduler finds a better node for a pod, it will migrate the pod to the better node.

Algorithm 2 describes the rescheduling process. This function is called periodically. The algorithm iterates over every workload pod in the cluster. The first step of this algorithm is to estimate the current cost of the evaluated pod. This evaluation includes the computing and networking costs. Then, all of the other schedulable nodes are evaluated in the same way, we keep the node with the best score. If it is the same as the original one there is nothing to do, if it is a different node, the rescheduler will migrate the pod toward this node.

Estimation of networking costs is key in this algorithm. Computing costs are easy to evaluate, they are static and known. However, networking costs depend on each service behavior and are not known a priori. Using the monitoring setup described in 4.4, we can consult how much data was sent to which destination. Using this information, we can estimate future data usage and networking costs.

Algorithm 2 Rescheduler: Background routine

for pod in WorkloadPods **do**
  $pod_{c\_cost} \leftarrow pod_{CPU} \times$ pod.node.Price
  $pod_{n\_cost} \leftarrow GetNetwCostEstimation(pod.node)$
  $best_{score} \leftarrow pod_{c\_cost} + pod_{n\_cost}$
  $best_{node} \leftarrow$ pod.node
  for node in SchedulableNodes **do**
    $pod_{c_cost} \leftarrow pod_{CPU} \times$node.Price
    $pod_{n\_cost} \leftarrow GetNetwCostEstimation(node)$
    $node_{score} \leftarrow pod_{c\_cost} + pod_{n\_cost}$
    if $node_{score} < best_{score}$ **then**
      $best_{score} \leftarrow node_{score}$
      $best_{node} \leftarrow$ node
    end if
  end for
  if pod.node $\neq best_{node}$ **then**
    pod.MigrateTo($best_{node}$)
  end if
end for

4.4 Implementation on Kubernetes

In this section, we describe how we implemented the above-described methodology to make it usable on a Kubernetes cluster with any containerized workload.

We are using the default Kubernetes plugins for the Filter and Reserve Node phases. The default Filter plugin removes nodes with a *taint* that the pod does not tolerate (e. g. the master mode has the *taint* "master" to prevent workload pods from running with the control plane node) from the list of the schedulable nodes. It also removes nodes with not enough resources to host the pod. The default Reserve Node phase updates ETCD [4],

the Kubernetes internal database. Then, the selected node will start the pod deployment based on the information in this database.

4.4.1 Latency-based initial placement. For the latency-based initial placement (LIP) we use the scheduler plugin framework, [9], which we have extended with our scoring plugin for latency-based initial placement. We collect the necessary latencies using the open source tools described in section 4.4.4.

4.4.2 Communication-based Initial Placement. The Communication-based Initial Placement method scores nodes based on an estimation of the cost of deploying a pod on that node. This estimation includes both networking and instance costs. Knowing how much data will be transmitted is essential to estimate the networking costs. The *traffic exporter* is a component that provides an estimation of the network traffic that the pod will generate and send it to the scoring plugin. The network traffic estimation includes communication with the other pods and communication with the end-users.

In order to estimate the communication patterns, we are looking at the previous communication. Based on the previous data, we try to predict the next ones, assuming a similar communication pattern in the future. The *traffic exporter* relies on two graphs: an abstract graph and a real communication graph. The abstract graph contains the communication pattern prediction. The choice of the estimator depends on the nature of the workload and the characteristics of its communication patterns. The real communication graph represents all the volume of data exchanged between two pods or between a pod and its user. The vertices are the pods, and the edges contain the volume of data exchanged. We use the real communication graph to build the abstract graph. Once the abstract graph is generated, we can use it to get estimations of data traffic. That estimation is important to estimate the networking costs and therefore the total cost of using a node. Using the cost estimation of each node, we can rank the nodes and choose the cheapest one. If two nodes have the same cost, we choose one randomly.

The traffic exporter is doing three independent tasks:

1) Record the traffic. the traffic exporter gets the traffic between the pods and stores it as the *Real Communication Graph*.

Each pod is a vertice in this graph. The edges represent the amount of data exchanged between two pods or between a pod and its end user. On each vertice, it stores the hash of the container images of the pod. It uses this hash as an identifier (a pod *type*). Two pods running the same software have the same type. The traffic exporter also attributes a type for each end-user.

2) Generate communication patterns. The traffic exporter builds an abstract representation of the traffic to summarize the information in the measured communication graph.

The traffic exporter constructs an abstract graph using the type of pods. The objective is to extract a common data traffic pattern from pods of the same type. The abstract graph uses the types of nodes as vertices and the traffic as edges. To build this graph the traffic exporter aggregates the traffic using an aggregation function (e.g., the average).

The CIP training process corresponds to the steps of recording the real traffic and building the abstract. The training starts as soon as one pod is deployed. Then, the following pods with the same type can benefit from that learning.

3) Serve the communication patterns. The traffic exporter provides communication patterns for a given type when the CIP scoring plugin requests it. The scoring plugin requests communication patterns based on the type of the evaluated pod.

Fig. 5 illustrates the CIP mechanisms. The traffic exporter extracts communication patterns from running pods and serves them to the CIP scheduler. We store the graphs of the communication patterns in a graph database. Then, we expose them using Prometheus API. The Prometheus server collects them periodically, and the

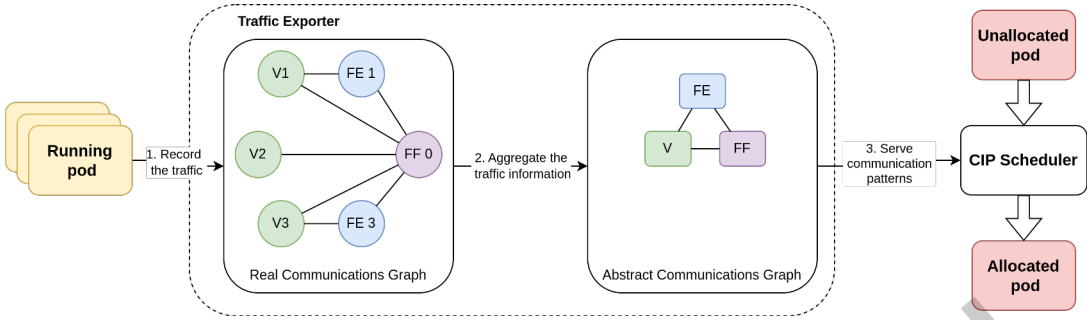

Fig. 5. Illustration of the CIP scheduler mechanism. The node colors represent the types.

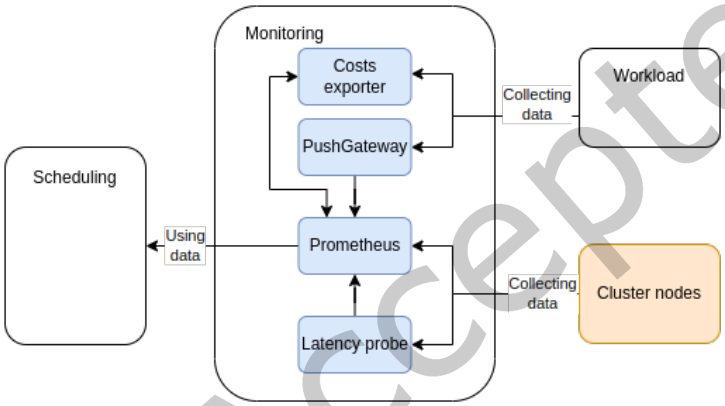

Fig. 6. Monitoring architecture

scheduler can request the Prometheus server. We provide more details on the monitoring setup and Prometheus in section 4.4.4.

4.4.3 Rescheduler. The Rescheduler is a Go application running in a dedicated pod. We use the Kubernetes and Prometheus libraries to collect all the necessary information. The traffic estimation is using linear regression. Future work can investigate more complex prediction methods to address different workloads.

To migrate a pod, the rescheduler updates its deployment manifest. This automatically triggers a rolling update with no downtime to the service. A new pod is deployed on the best node, and the old pod is deleted when it is up.

The duration of the rescheduling routine depends on the time to evaluate each pod configuration and the time to complete all pod migrations. Although we can manually set the rescheduling frequency, how often the rescheduler calls the rescheduling routine. This frequency can be low if the cluster state does not evolve quickly or higher if necessary. However, the rescheduling period cannot be lower than the duration of the rescheduling routine.

4.4.4 Monitoring setup. We are using open-source tools to collect data about the cluster and the workload state, and we make it available for scheduling decisions.

Fig. 6 presents the monitoring workflow, how data are collected and made available for taking scheduling decisions. This methodology can support any containerized workload.

Prometheus [17] is the main monitoring component. It periodically pulls many metrics about the cluster state (about the nodes, the pods, and the containers). Then, this data can be requested from the scheduling modules or a monitoring dashboard.

In addition, to the standard Prometheus metrics, we collect specific metrics about our workload and the network state. We can expose Key Performance Indicators such as the End-to-End latency. To ensure that the metrics are collected even if the pod is killed, we use the Prometheus PushGateway [18]. Data is pushed from the pods of the workload to this component. Then, the PushGateway can be periodically pulled by Prometheus like any other module.

We have built a module: *CostExporter* to compute and export the costs. It is a Go application that exposes metrics that are accessible through Prometheus. It exports the costs of using a server and networking traffic.

The cluster latency matrix (i.e., a matrix that summarizes the latency between every node in the cluster) is a key metric in our methodology. To collect this metric, we are using a DaemonSet (i.e., a Kubernetes tool that ensures that every node in the cluster runs a copy of a pod) that deploys a pod that pings all other nodes in the cluster. These latency probes [1] make the latency matrix directly available through Prometheus.

This monitoring system is designed to have limited overhead on the workload execution. Most of the monitoring services are running outside of the workload application. Cluster state sensors run in dedicated pods. Communication probes run in side-car containers. A side-car container has access to the same network as the containers of the workload application. It can export networking metrics to Prometheus. However, the QoS probe runs inside the workload application. A dedicated thread (to avoid interruptions) pushes the QoS to the Prometheus push gateway.

5 EVALUATION

In order to demonstrate the potential and effectiveness of our approach, we have devised a workload that mimics the computational need and communication of a real vehicular cooperative perception application. This workload is a distributed application consisting of multiple services that are deployed in a real Kubernetes cluster in a public cloud. We control the latencies between nodes to emulate a cloud-edge continuum infrastructure. We have then done some experiments to evaluate the latency-based initial placement (LIP), communication-based initial placement (CIP), and rescheduling (RS) algorithms in a realistic environment.

5.1 Experimental Methodology

We build an experimental cluster using public cloud resources. We use Virtual Machines (VMs) to have cluster nodes with different characteristics (e.g., different number of processors and amount of memory), and we add artificial delays between them to simulate the physical distances between the nodes. The details on how this is done can be found in our previous work [22]. We deploy a Kubernetes cluster using all these nodes. In this infrastructure, only the latencies are emulated, we deploy the Kubernetes cluster on real nodes.

Fig. 7 shows the infrastructure graph of the experimental cluster with all nodes and latencies between them. Our experimental cluster includes one node for the control plane and one for the monitoring tools. A workload cannot be deployed on these two nodes. The edge and data center (DC) nodes are hosting the services of the workload. End-user nodes host the end-user application workload. In a real-life environment, end-user nodes would be replaced by dedicated equipment such as a smartphone or a car.

The physical distances between the edge nodes induce networking delays. These delays are not represented as direct links on the graph to improve readability. The graph summarizes the delays as they are experienced by the different nodes. E.g., there is a delay of 25ms between the nodes E1 and E3.

a container running in the same pod as an application

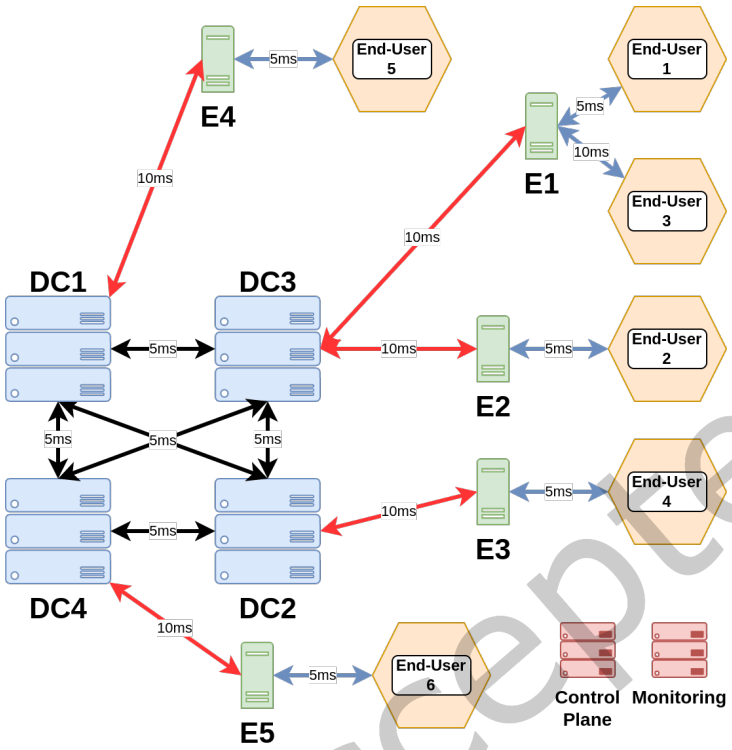

Fig. 7. Experimental cluster infrastructure graph (not all physical links between nodes are represented)

The difference between edge and DC nodes is the geographical location and the available resources. The nodes are AWS instances. The edge nodes have 4 CPUs and 16 GiB of memory. Other nodes have 8 CPUs and 32 GiB of memory.

5.2 Workload: Vehicular cooperative perception

The vehicular cooperative perception workload leverages computer vision to help detect nearby vehicles. As described by Xu et al. in [25], vehicles are sharing videos they record with their cameras to improve global knowledge of the positions of all nearby vehicles. Knowing the positions of close vehicles is helpful for drivers who cannot see others in their blind spots. Also, this technology is important for self-driving vehicle implementation where perception is a major challenge. Getting accurate positions of surrounding vehicles helps to reduce the collision risk.

We implement a synthetic workload that mimics the behavior of the above-described application. The originally described application is a monolith. We break it into three services that we can deploy anywhere in the computing continuum. By splitting the application we can include vehicles with limited computing resources; the services that cannot run on these vehicles can be hosted on a server at the edge or in a data center. Also, we can place the component that aggregates the data from multiple vehicles in a central location.

Three services compose the workload application. The *vehicles* generate a video stream and send it to a *feature extractor* (FE). The FE extracts the features from the video stream and sends it to a *feature fusion* (FF). FF merges the features to get accurate positions of the vehicles. Finally, the FF broadcasts the positions to nearby vehicles.

In our experiments, we have two kinds of vehicles: the ones with embedded computing capabilities (e.g., GPU) and the ones without them. The vehicles without computing capabilities send video-stream to the FE. The vehicle with computing capabilities sends features (already extracted) directly to the FF.

Fig. 8 shows the interactions between the Vehicle, Feature-Extractor (FE), and Feature-Fusion (FF) services. There are three different kinds of nodes in our experiment: Data-Centre (DC), Edge (E), and End-User (EU). End-user nodes are hosting the vehicles only. The vehicles cannot be deployed on different nodes. Vehicles without GPU are sending a video stream to the FE. Vehicles with GPU are sending the features directly to the FF. FE and FF services can run only on Edge or DC nodes.

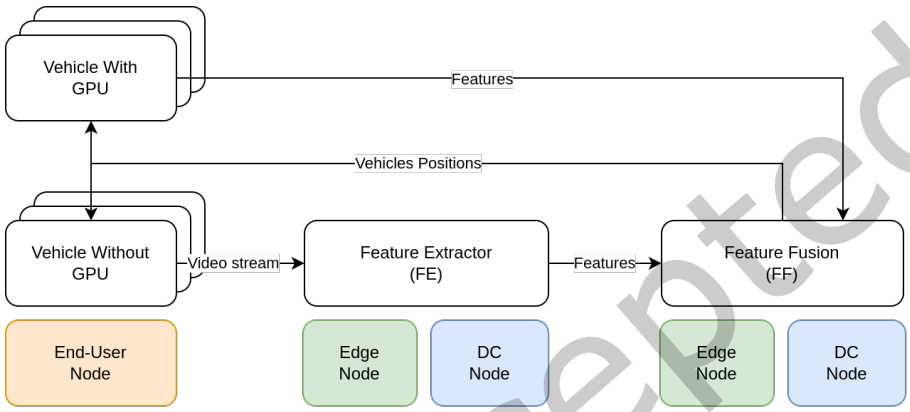

Fig. 8. Workload architecture: Workload pods and the nodes where they can run

There is only one instance of FF for the whole experiment, it aggregates the features from all of the vehicles. There are many vehicles, and each of them without a GPU is connected to one FE. We suppose that the vehicles are connected to the same network as the other services. However, the network delays depend on the geographic location of the vehicles.

For this workload, an end-user node represents a neighborhood or a 5G cell area where vehicles can go. In the experiment, vehicles are moving from one end-user node to another over time. A *json* configuration file defines all the movements of the vehicles when they are starting up or shutting down when they are moving from one area to another.

We benchmark the original application to build our synthetic workload. Table 2 summarizes the parameters we use to configure our application. This application is using CPU instead of GPU. Future work may adapt this scheduling approach to consider accelerators such as GPU or FPGA. In this synthetic application, the CPUs are running a load generated by the *stress-ng* tool [8]. Services send randomly generated data over the network using the sizes defined in Table 2.

5.3 Evaluation on a Kubernetes cluster

In this section, we evaluate our scheduling methodology on a Kubernetes cluster. We present the costs and the end-to-end (E2E) latency of the vehicular cooperative perception workload when using the Kubernetes default scheduler and our methodology to place the services. Also, we discuss the performances of the CIP approach when using initial data or not, and the CPU consumption overhead of running our scheduling components.

We define a scenario where 5 vehicles are embedding a GPU (to extract the features locally) and 5 vehicles use a feature-extractor (FE) deployed in the computing continuum. There is one feature-fusion (FF) instance for all the vehicles.

Table 2. Cooperative perception workload characteristics

Parameter	Value
Processing time (FF)	100 μs
Processing time (FE)	100 μs
Frames size	731kB
Features size	64 kB
Frame rate	35 FPS

We evaluate this scenario using seven scheduling approaches. *Baseline*: the default Kubernetes scheduler (least allocated node). *Most Allocated node* (MA). *Balanced allocation* (BA): a Kubernetes approach to balance the resource usage among the nodes [10]. *Latency-based Initial Placement* (LIP): the initial placement algorithm described in section 4.2.1. *LIP + Rescheduler* (LIP+RS): the initial placement algorithm in addition to the rescheduling methodology described in section 4.3. *Communication-based Initial Placement* (CIP): the communication-based initial placement algorithm described in section 4.2.2. *CIP + Rescheduler* (CIP+RS): the initial placement algorithm in addition to the rescheduling methodology described in section 4.3.

We compare these seven scheduling approaches to the optimal solution (using pod migration). Vehicle arrival is deterministic, it is therefore possible to find the optimal solution for a given scenario. However, we use an exhaustive search to get the optimal solution. It is not possible to use it as an online scheduling approach since the complexity is very high and the scenario should be known *a priori*. We can only search for the best solution *a posteriori* and replay it on the same cluster. Our optimization solver only works for clusters with the same instance price for each node.

We use three different cluster configurations for each approach: i) all nodes have the same instance cost, ii) data-center nodes are twice as expensive as the edge nodes, and iii) edge nodes are twice as expensive as the data-center nodes. The instance cost is somewhat arbitrarily chosen to 0.0472 per CPU.h and the communication cost to 0.01 per GB. The actual values are not so important for the performance evaluation, but they are still chosen to be in the realistic realm. Since the experiments are run on real machines, there is significant variability between each execution. Therefore, we repeat each experiment ten times and use the average value in the figures here with error bars representing 95% confidence interval.

For the LIP approach, we set $\alpha = 20$ and $\beta = 1000$. We set α in relation to the observed latencies in the cluster. It is lower than the average delay between two edge nodes and higher than the average delay between a DC and an edge node. If the α value is too small, the condition with α will never be satisfied, and the LIP will select the node with the lowest instance cost. Respectively, if α is too big, the condition will always be satisfied, and the LIP will choose the cheapest node. We set β in relation to the instance costs in the cluster. It is significantly bigger than the inverse of the average instance cost. That way, LIP first ranks the nodes by the number of α constraints satisfied and then selects the cheapest node among those that satisfy the maximum number of constraints.

5.3.1 Results. For every *experiment*, we measure the total costs of running the workload. The total costs are the sum of the computing, networking, and rescheduling costs. The rescheduling cost is the price to pay to migrate a pod from one node to a different one.

In addition to the costs, we also record the end-to-end latency (E2E latency) of each experiment. This is a key performance indicator (KPI) to check if the Quality of Service (QoS) varies when performing cost optimization. The E2E latency is the duration between the time when a frame is recorded and the time when the vehicle receives the corresponding positions of the nearby vehicles. This value aggregates the network delays between

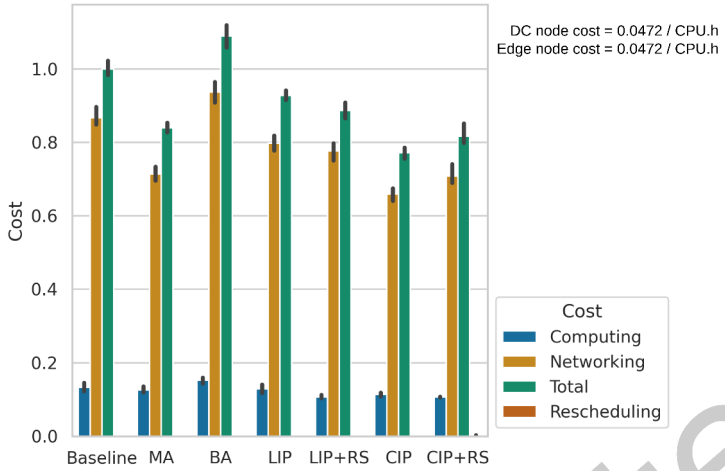

Fig. 9. Average of total costs for each approach: Baseline (default Kubernetes, Least Allocated), Most Allocated, Balanced Allocation, Latency-based Initial Placement, LIP + Rescheduler, Communication-based Initial Placement, and CIP + Rescheduler. Costs are normalized to the baseline approach. Data-center and edge nodes share the same price.

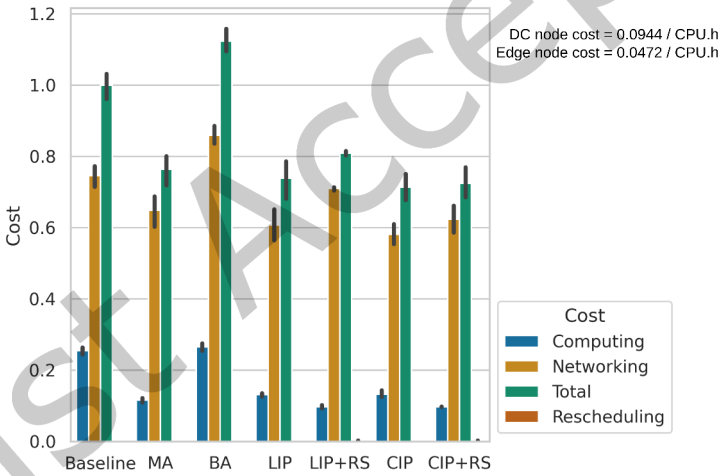

Fig. 10. Average of total costs for each approach: Baseline (default Kubernetes, Least Allocated), Most Allocated, Balanced Allocation, Latency-based Initial Placement, LIP + Rescheduler, Communication-based Initial Placement, and CIP + Rescheduler. Costs are normalized to the baseline approach. Data-center nodes are twice as expensive as edge nodes.

each service, the duration required to send the video stream and the features, the processing time for extracting the features, the time to fusion the features, and the time to broadcast the positions.

Fig. 9, 10, and 11 presents the average of the total costs for different scenarios: same instance cost for all nodes, data center nodes twice as expensive as edge nodes and edge nodes twice as expensive as data center nodes. The total costs are normalized to the optimal approach. Error bars represent the 95% confidence interval. The lower the costs, the better.

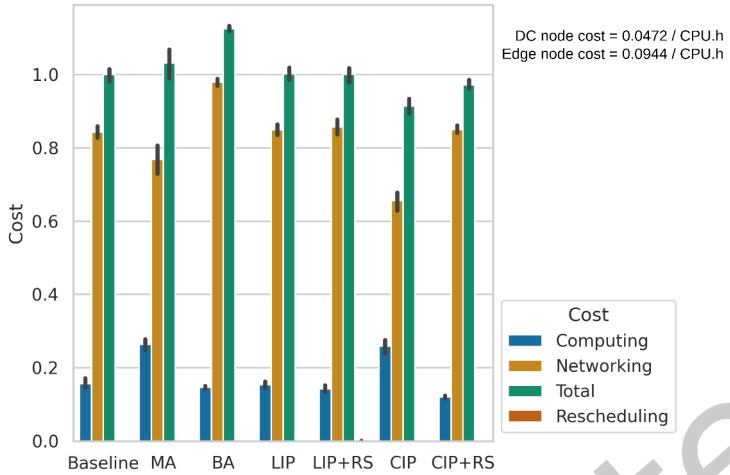

Fig. 11. Average of total costs for each approach: Baseline (default Kubernetes, Least Allocated), Most Allocated, Balanced Allocation, Latency-based Initial Placement, LIP + Rescheduler, Communication-based Initial Placement, and CIP + Rescheduler. Costs are normalized to the baseline approach. Edge nodes are twice as expensive as data-center nodes.

Fig 12. shows the 95^{th} percentile of the end-to-end latency for the different experiments. We can use this information to ensure that the Quality of Service remains at the same level.

By analyzing the node allocation of the services over time for all scenarios, we observe that the baseline approach is choosing data center nodes in most situations. Data center nodes have more computing resources, they are the least allocated nodes. The LIP, CIP, CIP+RS, and CIP+RS approaches use both edge and data center nodes (it depends on the cluster configuration).

For the first scenario where all nodes have the same instance cost, our approaches have total costs that are 7 to 23% lower than the baseline (the default Kubernetes scheduler). The optimal approach is 30% lower than the baseline approach. Our best approach is CIP, its total cost is 8% above the optimal. In this scenario, the instance costs are the same for every approach. It is the expected behavior, the node price is the same for every node. Only lower networking costs lead to lower total costs. Rescheduling costs are negligible in this experiment. They correspond to the price to pay to run the two pods during the migration, the old pod is killed only when the new one is running.

For the second scenario where the data center nodes are twice as expensive as edge nodes, our approaches have a total cost of 19 to 29% lower than the baseline. The baseline chooses the least allocated nodes, which are the data center nodes. The baseline chooses only expensive nodes in this scenario, this leaves a lot of room for improvements. CIP and CIP+RS have similar performances using two different placement strategies. In this situation, the CIP approach uses more DC nodes (that are more expensive) than the CIP+RS. The rescheduler is using more edge nodes to save both computing and networking costs. However, the un-optimal rescheduling routine duration (described in section 5.4.2) leads to higher networking costs. CIP uses DC nodes for the feature fusion, it is more expensive than CIP+RS which uses Edge nodes.

In the last scenario, where edge nodes are twice as expensive as the data center nodes, there is less room for improvement. The baseline chooses data center nodes because they are the less allocated nodes, and it gets results similar to the LIP and LIP+RS approaches. However, the CIP approach manages to find 9% lower total costs. But, these cost improvements imply an impact on the end-to-end latency. CIP+RS performs better than the baseline with a limited impact on the QoS.

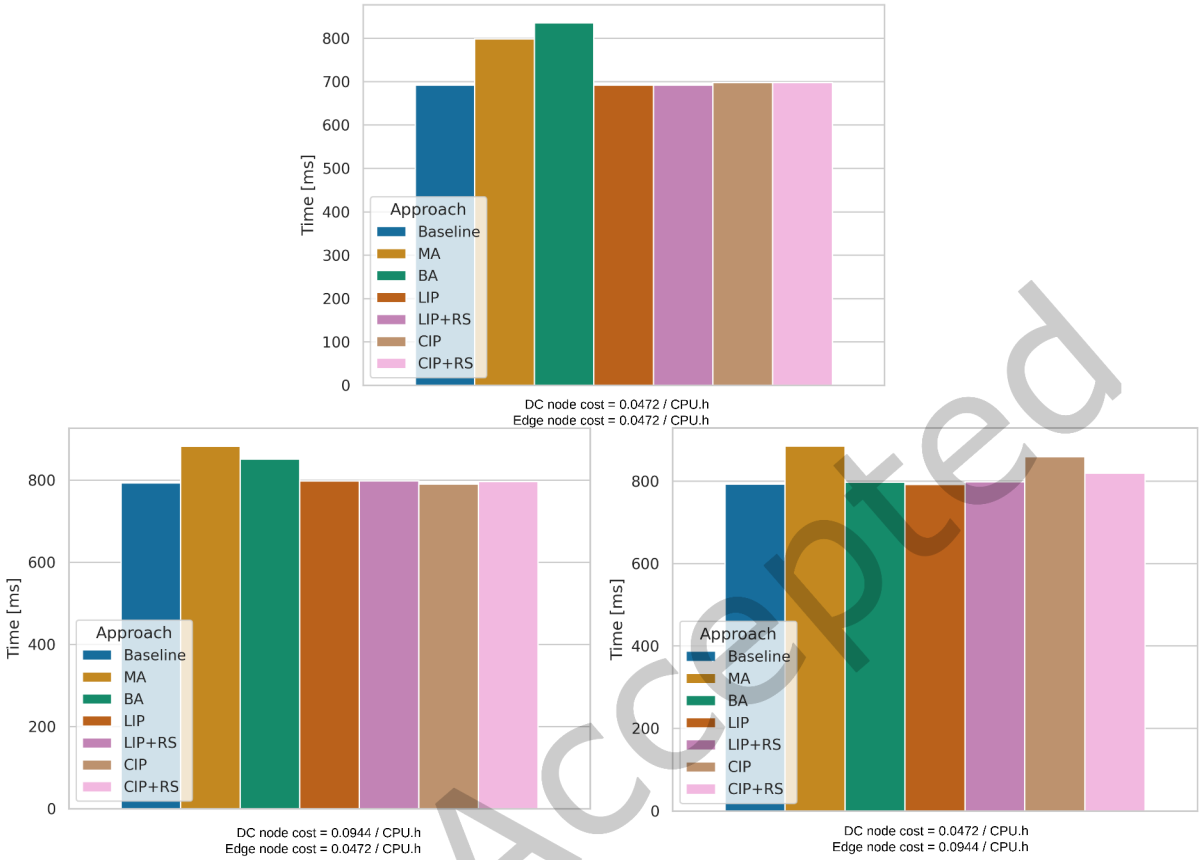

Fig. 12. 95^{th} percentile of the end-to-end latency for each approach (ms)

The Balanced Allocation approach is not appropriate for the edge-to-cloud computing continuum. It provides higher total costs and lower QoS than the baseline in every scenario. Using Balanced Allocation on heterogeneous resources is inefficient.

The Most Allocated node approach can have lower total costs than the baseline in two scenarios. The QoS of the MA approach is lower than the baseline in every scenario. The major issue of this approach is that it does not adapt to different cluster configurations. It always chooses the edge nodes (which are the most allocated nodes), even if they are not close to the end users. A first edge node is picked randomly and then filled at its maximum capacity, then the next one is selected. If the first edge node picked is close to the end users, the overall results will be better than the baseline. This approach will be even less performant if we add more edge nodes in the cluster. This method has a lot of variability due to randomly picked edge nodes. It makes that approach less reliable than our methodology.

The End-to-End latency is similar for most of the approaches, except for the above-mentioned case. Overall cost improvement does not negatively impact the Quality of Service.

Table 3. Normalized total costs for different approaches

Approach	Optimal	LIP	CIP (without initial data)	CIP (with initial data)
Normalized cost	1	1.32	1.22	1.10

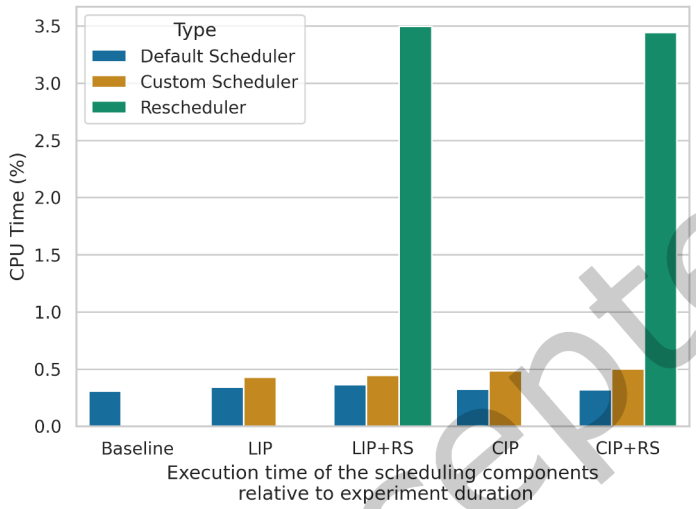

Fig. 13. CPU time overhead of the scheduling components

The CIP approach is the best of our approaches to minimizing the total costs. However, there is one scenario (when edge nodes are twice as expensive as data-center nodes) where the QoS is impacted by cost optimization. The LIP approach performs less aggressive cost optimizations, and results are lower than CIP but never negatively impacts the QoS. The rescheduler helps reduce the costs or improve the QoS. But, to propose the best performances possible, the rescheduling routine needs to be shorter than the systems evolve.

5.3.2 CIP performances and initial data. The performances of the CIP scheduler vary according to the available data. To quantify this variation, we run an experiment where we use the CIP with and without initial data. Table 3 presents the normalized total costs for different approaches. When the CIP scheduler can access initial data, the total cost is 12% lower than it is without initial data. However, the CIP approach outperforms the LIP approach in all cases, with and without initial data.

The CIP approach has better performance when it already has data about the application it is deploying.

5.3.3 Custom scheduler and rescheduler overhead. We measure the overhead of using our custom scheduling methodologies. It is important to ensure that the cost of running our approaches is lower than the savings they can achieve.

Fig. 13 presents the execution time of the scheduling components relative to the experiment duration for each approach studied. Using our scheduling approaches has a limited impact on CPU consumption. The CPU time of the additional scheduling components is below 4% of the experiment duration. The scheduling components are idle most of the time. Our custom schedulers consume around 1.5 times more CPU than the default Kubernetes scheduler.

Overall, saving 10 to 25% of the costs of running the experiment is worth the expense of 4% of a CPU core for the duration of the experiment.

It is worth noting that the overhead of scheduling does not impact any applications in any negative way as it executes on a separate node.

5.4 Limitations

5.4.1 Workload. In this set of experiments, we consider that Edge and Cloud have the same processing speed (but with different numbers of CPUs). The workload consists of handling a video stream in real time; more computing capabilities cannot make it faster, but it can improve the accuracy. Further studies may investigate different framerate and resolutions, or test other workloads.

5.4.2 Rescheduling. The rescheduling routine sometimes is too long for optimal performance. It could be counter-intuitive that a static approach (CIP) outperforms the dynamic one (CIP+RS). The explanation is that the system evolves faster than the rescheduler can react.

For example, when a pod is placed on node E3 to be close to a vehicle located on E-U4, this solution is optimal at this given time. But, if the vehicle moves to E-U6, the solution is no longer optimal. If the rescheduler takes a minute to move the pod to E5, the pod stays in an inadequate location for one minute, and the costs grow higher very fast. If the rescheduler cannot react fast enough, it can be better to have a static approach without a rescheduler or to improve the reactivity of the rescheduler. To avoid these situations, the period of rescheduling should be significantly lower compared to the changes that occur in the clusters (vehicle movement or new node availability).

The minimum duration of the rescheduling routine is fixed by execution and migration time. Section 4.4.3 provides more details about the rescheduling duration. Future works may investigate ways to improve the rescheduling speed.

5.4.3 Quality of Service. Even if our proposed methodology has similar or better QoS than the baseline, there is no mechanism to ensure that Service Level Objectives (SLOs) are met.

The LIP approach minimizes inter-service latency to lower communication costs. It reduces the end-to-end latency and improves the QoS. However, there are no hard requirements to avoid choosing nodes with high latency if it is the only choice possible.

The CIP approach minimizes the total costs. If the nodes of the cluster have similar prices, this approach will minimize the data traffic. Otherwise, it will choose a tradeoff between the instance and the communication costs. Reducing data traffic by processing data locally reduces the end-to-end latency, and improves the QoS.

The CIP approach can reach lower total costs than the LIP approach, even if the QoS is lower. The LIP approach performs less aggressive optimizations, it can get higher QoS.

Future work can investigate adding a node filter in the scheduler and the rescheduler to remove the nodes that do not meet SLOs from the list of schedulable nodes.

6 CONCLUSION

We propose a cost-effective scheduling methodology that simplifies the deployment of distributed applications in the cloud-to-edge computing continuum as it does not need manual placement of edge services. Doing that robustly lowers the costs of running the applications while keeping the same Quality of Service. This scheduling methodology works for clusters that aggregate resources from traditional data centers and the servers located at the edge of the network. We implement our scheduling methodology on a Kubernetes cluster, and we demonstrate its benefits using a realistic workload: a vehicular cooperative perception. Experiments on this workload show

that using our approach reduces costs by 10% to 25% compared to the default Kubernetes scheduler for the same quality of service. Also, it is possible to use our methodology with any containerized workload.

Although we use monetary cost as our optimization target, any measurable metric could be used, e.g., energy consumption can be minimized instead.

ACKNOWLEDGMENTS

This research has been partly funded by the Luxembourg National Research Fund (FNR) under contract number 16327771 and has been supported by Proximus Luxembourg SA. For the purpose of open access, and in fulfillment of the obligations arising from the grant agreement, the author has applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any Author Accepted Manuscript version arising from this submission.

REFERENCES

[1] Bloomberg. 2023. https://github.com/bloomberg/goldpinger
[2] Docker. 2023. https://www.docker.com
[3] Yucong Duan, Guohua Fu, Nianjun Zhou, Xiaobing Sun, Nanjangud C. Narendra, and Bo Hu. 2015. Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. In *2015 IEEE 8th International Conference on Cloud Computing*. 621–628. https://doi.org/10.1109/CLOUD.2015.88 ISSN: 2159-6190.
[4] ETCD. 2023. https://etcd.io
[5] Linux Fondation. 2023. https://opencontainers.org
[6] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, and Minyi Guo. 2022. Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (Aug. 2022), 1825–1840. https://doi.org/10.1109/TPDS.2021.3128037 Conference Name: IEEE Transactions on Parallel and Distributed Systems.
[7] Kiranpreet Kaur, Fabrice Guillemin, Veronica Quintuna Rodriguez, and Francoise Sailhan. 2022. Latency and network aware placement for cloud-native 5G/6G services. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*. 114–119. https://doi.org/10.1109/CCNC49033.2022.9700582 ISSN: 2331-9860.
[8] Colin Ian King. 2023. https://github.com/ColinIanKing/stress-ng
[9] Kubernetes. 2023. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/
[10] Kubernetes. 2023. https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins
[11] Phu Lai, Qiang He, John Grundy, Feifei Chen, Mohamed Abdelrazek, John G Hosking, and Yun Yang. 2020. Cost-Effective App User Allocation in an Edge Computing Environment. *IEEE Transactions on Cloud Computing* (2020), 1–1. https://doi.org/10.1109/TCC.2020.3001570 Conference Name: IEEE Transactions on Cloud Computing.
[12] Hongjian Li, Jie Shen, Lei Zheng, Yuzheng Cui, and Zhi Mao. 2023. Cost-efficient scheduling algorithms based on beetle antennae search for containerized applications in Kubernetes clouds. *The Journal of Supercomputing* (Feb. 2023). https://doi.org/10.1007/s11227-023-05077-7
[13] Angelo Marchese and Orazio Tomarchio. 2022. Extending the Kubernetes Platform with Network-Aware Scheduling Capabilities. In *Service-Oriented Computing (Lecture Notes in Computer Science)*, Javier Troya, Brahim Medjahed, Mario Piattini, Lina Yao, Pablo Fernández, and Antonio Ruiz-Cortés (Eds.). Springer Nature Switzerland, Cham, 465–480. https://doi.org/10.1007/978-3-031-20984-0_33
[14] Angelo Marchese and Orazio Tomarchio. 2022. Network-Aware Container Placement in Cloud-Edge Kubernetes Clusters. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 859–865. https://doi.org/10.1109/CCGrid54584.2022.00102
[15] Gabriele Proietti Mattia and Roberto Beraldi. 2021. Leveraging Reinforcement Learning for online scheduling of real-time tasks in the Edge/Fog-to-Cloud computing continuum. In *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*. 1–9. https://doi.org/10.1109/NCA53618.2021.9685413 ISSN: 2643-7929.
[16] Adrián Orive, Aitor Agirre, Hong-Linh Truong, Isabel Sarachaga, and Marga Marcos. 2022. Quality of Service Aware Orchestration for Cloud–Edge Continuum Applications. *Sensors* 22, 5 (Jan. 2022), 1755. https://doi.org/10.3390/s22051755 Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.
[17] Prometheus. 2023. https://prometheus.io/
[18] Prometheus. 2023. https://github.com/prometheus/pushgateway/
[19] Thomas Pusztai, Stefan Nastic, Andrea Morichetta, Víctor Casamayor Pujol, Philipp Raith, Schahram Dustdar, Deepak Vij, Ying Xiong, and Zhaobo Zhang. 2022. Polaris Scheduler: SLO- and Topology-aware Microservices Scheduling at the Edge. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. 61–70. https://doi.org/10.1109/UCC56403.2022.00017
[20] Samuel Rac and Mats Brorsson. 2021. At the Edge of a Seamless Cloud Experience. *arXiv preprint arXiv:2111.06157* (2021).

[21] Samuel Rac and Mats Brorsson. 2023. Cost-Effective Scheduling for Kubernetes in the Edge-to-Cloud Continuum. In *2023 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 153–160.

[22] Samuel Rac, Rajarshi Sanyal, and Mats Brorsson. 2023. A Cloud-Edge Continuum Experimental Methodology Applied to a 5G Core Study. *arXiv preprint arXiv:2301.11128* (2023).

[23] László Toka. 2021. Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes. *Journal of Grid Computing* 19, 3 (July 2021), 31. https://doi.org/10.1007/s10723-021-09573-z

[24] Lukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. 2021. NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM42981.2021.9488670 ISSN: 2641-9874.

[25] Runsheng Xu, Zhengzhong Tu, Hao Xiang, Wei Shao, Bolei Zhou, and Jiaqi Ma. 2022. *CoBEVT: Cooperative Bird's Eye View Semantic Segmentation with Sparse Transformers*. https://doi.org/10.48550/arXiv.2207.02202

[26] Zhiheng Zhong and Rajkumar Buyya. 2020. A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources. *ACM Transactions on Internet Technology* 20, 2 (April 2020), 15:1–15:24. https://doi.org/10.1145/3378447