Cost-Effective Scheduling for Kubernetes in the Edge-to-Cloud Continuum

Samuel Rac
SnT, University of Luxembourg
Luxembourg
samuel.rac@uni.lu

Mats Brorsson
SnT, University of Luxembourg
Luxembourg
mats.brorsson@uni.lu

***Abstract*—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 container orchestration. It is very efficient in a data center environment, but it fails to give the same performance when adding 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. A cost-aware scheduler enables this optimization. We are also monitoring the Key Performance Indicators of the applications to ensure that cost optimizations do not impact negatively 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 application. To demonstrate the performance of our scheduling approach, we evaluate it on a vehicle cooperative perception use case, a representative 5G application.

***Index Terms*—Cloud computing, Edge computing, Scheduling, Container Orchestration, Resource allocation, 5G, Kubernetes**

I. 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 data center to the edge computing continuum is the aggregation of resources located in both traditional data centers and at the edge of the network. An application can run on any node from the computing continuum, however, the geographical position of a node changes the delays with end-users. Thus, an application scheduler for the edge-to-cloud continuum should consider more parameters than one 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 delays and the available bandwidth are different from node to node. Also, network conditions can evolve over time; the network is more likely to be congested due to more limited bandwidth, or the delays can change if a user is moving. Edge nodes should not be chosen only for their processing capabilities but also for their geographical location; the location of a node modifies the delays with end users.

Deploying applications in the cloud-to-edge computing continuum should be easy for developers [1]. Application developers should define requirements (e.g. CPU, memory, delays, and other Service Level Objectives) instead of static locations. Otherwise, it will not be possible to scale up the application if it is placed manually by developers. 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)* [2]. Therefore, the customers pay only for the resource they use. E.g. customer pays 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 saves the outgoing network costs; data stays local and traffic between the edge and the data center is not charged.

We propose a scheduler to lower the costs of deploying applications in the cloud-to-edge continuum. To make application deployment easy and as close as possible to industrial standards, we implement our methodology as a Kubernetes scheduler plugin [3] based on network requirements and costs. Any containerized application can be deployed using our scheduler. 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 migrate the service pod[1] automatically. Also, when a cheaper

[1]A *pod* is the smallest schedulable unit in Kubernetes, containing at least one container

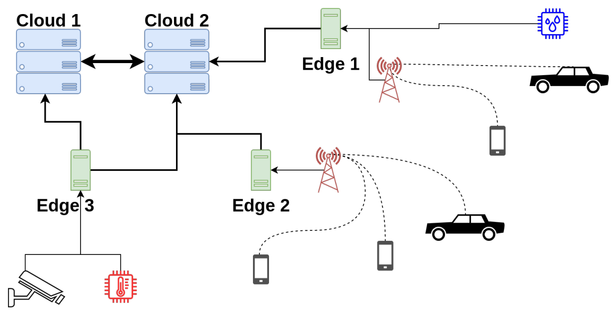

Fig. 1. The Edge-to-Cloud Computing Continuum - Computing resources are in blue and green.

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 a cost-aware Kubernetes scheduling plugin.
2) Design and implementation of a network-aware rescheduler to keep application placement decisions optimal when end users are moving.
3) Demonstration of our scheduling methodology performance on a realistic 5G use-case using cooperative perception for autonomous vehicles.

From our experiments, we have verified that our custom scheduler for initial placement can achieve in most cases about 15% lower costs than the default Kubernetes scheduler and that with the rescheduler more than 20% cost reduction can be achieved.

The rest of the article is organized as follows. Section II presents the relevant background and motivation. Then we present the relevant related work in section III. Our scheduling methodology is exposed in section IV and in section V 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 VI.

II. Background and Motivation

A. 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 is changing 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.

B. Motivation

There are many use cases of AI software running at the edge that can motivate the need for a scheduler for the whole cloud to edge computing continuum.

For example, analyzing a crowd during a demonstration using body-worn cameras. These cameras can simply record the scene, or they can enable advanced features like estimating the number of participants or detecting dangerous individuals. Using AI this information can be extracted from the video streams in real-time and be quickly reported to police officers.

However, the computing capabilities embedded in the cameras are limited, they cannot process the video stream analysis; the application should run outside of the cameras. The application can be executed in a traditional data center, but communication costs will apply and could be expensive for many high-resolution video streams. The AI application can also run at the edge, on a server, or in a micro data center located close to the 5G base stations. The video streams would be processed locally and communication costs will be lower. Only extracted information would go to the traditional data centers.

The challenge is to identify which is the best server to host the video stream analysis application. Also, body-worn cameras are moving. The optimal server selection evolves over time, this issue should be addressed as well.

C. Proposed solution

We think that Kubernetes is one of the best candidates for orchestrating applications over the whole cloud-to-edge continuum. Kubernetes is the de facto industrial standard for container orchestration. It is not only limited to Docker containers [4] but it can manage any container 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 for orchestrating 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. The geographical location of the servers is important to have low delays. It is therefore not possible to efficiently place applications in the whole computing continuum if not considering the networking resources.

We can use the Kubernetes scheduler framework [3] to implement an ad hoc scheduling methodology for the edge-to-cloud computing continuum. The scheduling framework is easy to extend, every scheduling stage is defined in 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 between 0 and 100. Normalization is necessary to get an averaged 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.

Our solution is twofold: a *custom initial placement* (CIP) method and a *rescheduler* (RS).

The custom 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 delays with 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 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 is changing 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.

III. 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.

a) Cost-aware scheduling: Lai et al present a cost-aware scheduler in [6]. They are using a heuristic approach (most capacity first) to maximize the number of allocated edge users while minimizing the number of necessary servers at the edge. In this work, there is 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: [7], [8] 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. [7] 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. [8] 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.

b) Network-aware scheduling: Kaur et al. [9] 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 [10]–[12]. In [10], they present a network-aware scheduler plugin and a descheduler that is checking 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 [11], 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. [12] 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 [13], 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.

c) QoS-aware scheduling: Mattia and Beraldi [14] 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.

Polaris scheduler is presented in [15]. 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 [17] is a run-time system that maps micro-services to nodes based on communication overhead, resource utilization, and IO pressure.

None of the approaches above use a rescheduling mechanism.

IV. 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 plugin, and ii) the rescheduler. Finally, we expose how we implement this scheduling methodology on a Kubernetes cluster.

A. 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. delays with other machines, available bandwidth) matter for the scheduling decision.

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, if edge data is processed locally, no additional network cost is charged. However, if data traffic is sent to a data center, then network traffic is charged.

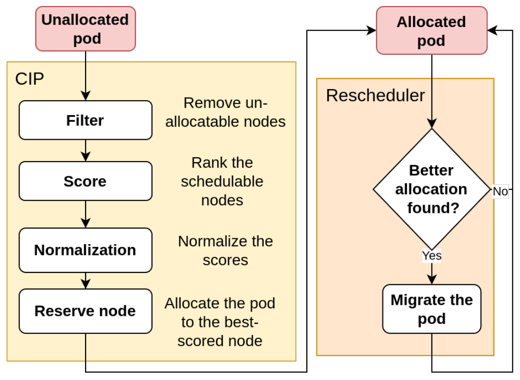

Fig. 2. Scheduling workflow: Custom Initial Placement (yellow), Rescheduler (orange)

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. E.g. it is cheaper to deploy a network-intensive workload at the edge where data is produced rather than in a data center.

Fig. 2 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.

B. Service initial placement

The objective 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.

Algorithm 1 describes our node scoring method. The objective is to find an initial location for deploying a pod. We run the 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). To improve the end-to-end latency and lower the networking costs (saving bandwidth) we chose a node close to the connected services.

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. If the service is not deployed yet, we do not add the server to the list. Once the server list is built, we evaluate the delays between these servers and the evaluated node. If the delay is lower than α, we add β to the final score (α is a delay distance in ms and β is a score modifier.). The more connected services in a radius of α the higher the score.

Algorithm 1 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 $n_s$
```

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

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

C. 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-user are moving), ii) we can use data collected when the service is running to improve the scheduling decision (e.g. we can estimate data traffic which is not possible at the initial placement stage since this data is unknown at that stage). 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, at the end 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 IV-D, we can consult how much data was sent to which destination. Using this information, we can estimate future data usage and networking costs.

D. Implementation on Kubernetes

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

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
```

For the custom initial placement we use the scheduler plugin framework, [3], which we have extended with our scoring plugin for initial placement.

The Rescheduler is implemented as a Go application running in a dedicated pod. We are using 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 when it is up the old pod is deleted.

We are using open-source tools, [18], [19], to monitor the state of the cluster and collect Key Performance Indicators (e.g. end-to-end latency).

V. 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 co-operative perception application. This workload is a distributed application consisting of multiple services which 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 custom initial placement (CIP) and rescheduling (RS) algorithms in a realistic environment.

A. 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

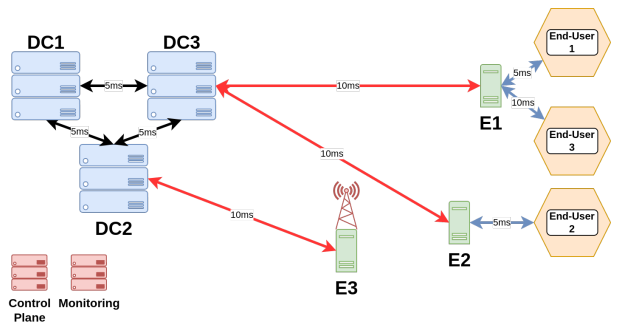

Fig. 3. Experimental cluster infrastructure graph

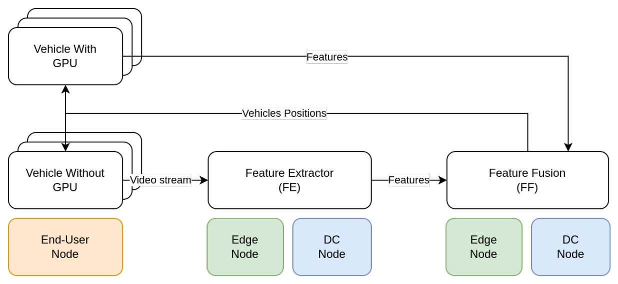

Fig. 4. Workload architecture: Workload pods and the nodes where they can run

nodes. The details on how this is done can be found in our previous work [20]. We deploy a Kubernetes cluster using all these nodes. In this infrastructure, only the delays are emulated, we deploy the Kubernetes cluster on real nodes.

Fig. 3 shows the infrastructure graph of the experimental cluster with all nodes and delays 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 real life environment, end-user nodes can be replaced by dedicated equipment such as a smartphone or a car.

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.

B. Workload: Vehicular cooperative perception

The vehicular cooperative perception workload leverages computer vision to help detect nearby vehicles. As described in [21], 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 that cannot see others in their blind spot. 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 at 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. 4 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.

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 I 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 [22]. Services send randomly generated data over the network using the sizes defined in Table I.

C. 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.

TABLE I
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 define a scenario where 2 vehicles are embedding a GPU (to extract the features locally) and 3 vehicles use a feature-extractor (FE) deployed in the computing continuum. There is one feature-fusion (FF) instance for all the vehicles.

We evaluate this scenario using three scheduling approaches. *Baseline*: the default Kubernetes scheduler. *Custom Initial Placement* (CIP): the initial placement algorithm described in section IV-B. *CIP + Rescheduler* (CIP+RS): the initial placement algorithm in addition to the rescheduling methodology described in section IV-C.

For each approach, we are testing different cluster configurations. Table II summarizes the experimental parameters and defines the different *experiments*. Experiments 1 to 3 test different α parameters and experiments 4 and 5 explore different node costs. Since the experiments are run on real machines there is significant variability between each execution and therefore we repeat each experiment ten times and use the average value in the figures here.

TABLE II
COOPERATIVE PERCEPTION WORKLOAD CLUSTER PARAMETERS

Experiment	DC cost (per CPU)	Edge Cost (per CPU)	Network cost (per GB)	α (ms)	β
1	0.0472	0.0472	0.01	20	1000
2	0.0472	0.0472	0.01	15	1000
3	0.0472	0.0472	0.01	40	1000
4	0.0472	0.0944	0.01	20	1000
5	0.0944	0.0472	0.01	20	1000

For every *experiment*, we measure the total costs of running the workload. The total costs are the sum of the computing and networking costs.

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 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 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. 5 presents the average of the total costs. The total costs are normalized to the baseline approach. Error bars represent the 95% confidence interval. The lower the costs the better.

Fig. 6 shows the 95^{th} percentile of the end-to-end latency for the different experiments. We can use this figure to ensure

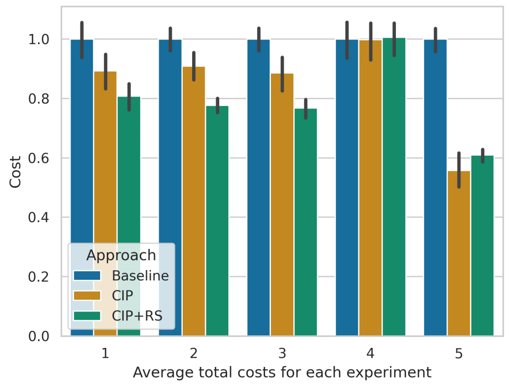

Fig. 5. Average of total costs for each approach: Baseline, Custom Initial Placement, and CIP + Rescheduler. Costs are normalized to the baseline approach

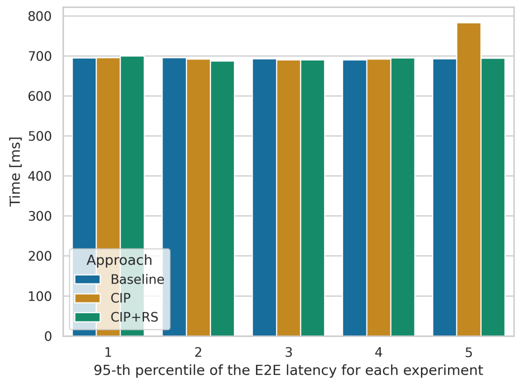

Fig. 6. 95^{th} percentile of the end-to-end latency for each approach: Baseline, Custom Initial Placement, and CIP + Rescheduler.

that the Quality of Service remains at the same level.

By analyzing the node allocation of the services over time, 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 CIP and the CIP+RS approaches use both edge and data center nodes (it depends on the cluster configuration).

Experiments 1 to 3 use different values of α. The CIP and the CIP+RS approaches show lower costs than the baseline approach. The CIP+RS provides an improvement of around 20%. According to the confidence intervals, the CIP+RS costs are significantly lower than the baseline approach. In these three experiments, the computing cost is the same for all kinds of nodes. Therefore computing costs are the same for every approach. The total costs can be lower only if the networking costs are lower because the computing costs are constant. The CIP and the CIP+RS get lower total costs because they managed to get smaller networking costs.

In experiment 4, the edge nodes are twice as expensive as the data center nodes. By analyzing the allocation of the services, we observe that most of the services are deployed in data centers. Data center nodes are the best solution in this configuration. The three approaches are all choosing data center nodes, therefore the results are very similar.

In experiment 5, the data center nodes are twice as expensive as the edge nodes. When using our methodology, the services are mostly placed at the edge. The CIP and the CIP+RS achieve an improvement of around 40% compared to the baseline approach. They are significantly lower.

For each experiment, the E2E latency is similar for every approach. Even if our methodology is lowering the costs, the E2E latency is not impacted, and this KPI remains the same. However, due to huge optimization in experiment 5, the CIP approach gets higher costs. This approach chooses only edge nodes that are cheaper and have low latency, but the vehicles are moving and the services stay at the same location. This location at the edge is too specific and provides higher E2E latency. Using a central location like a data center may help to avoid this issue. To benefit from the cost optimization of using edge nodes, we need to use the rescheduler to keep the same quality of service.

The results show that we can have lower costs when running the same services with the same end-to-end latency. Our scheduling methodology can lower the costs of deployment of the edge-to-cloud computing continuum. However, to ensure lower costs it is better to use the rescheduler in addition to our initial placement approach.

VI. Conclusion

We propose a cost-effective scheduling methodology that can lower the costs of deploying 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 20% to 40% compare to the default Kubernetes scheduler for the same quality of service. Also, it is possible to use our methodology with any containerized workload.

In the future, we want to investigate ways to extend this methodology to improve initial service placement. We think that the knowledge from previously deployed instances of a service can be used to get better results. Furthermore, we would like to see how our methodology can be used to optimize other metrics such as energy consumption.

Acknowledgment

This work has been partly funded by the Luxembourg National Research Fund (FNR) under contract number 16327771 and has been supported by Proximus Luxembourg SA.

References

[1] S. Rac and M. Brorsson, "At the edge of a seamless cloud experience," *arXiv preprint arXiv:2111.06157*, 2021.

[2] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu, "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends," in *2015 IEEE 8th International Conference on Cloud Computing*, Jun. 2015, pp. 621–628, iSSN: 2159-6190.

[3] Kubernetes, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[4] Docker, 2023. [Online]. Available: https://www.docker.com

[5] L. Fondation, 2023. [Online]. Available: https://opencontainers.org

[6] P. Lai, Q. He, J. Grundy, F. Chen, M. Abdelrazek, J. G. Hosking, and Y. Yang, "Cost-Effective App User Allocation in an Edge Computing Environment," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020, conference Name: IEEE Transactions on Cloud Computing.

[7] H. Li, J. Shen, L. Zheng, Y. Cui, and Z. Mao, "Cost-efficient scheduling algorithms based on beetle antennae search for containerized applications in Kubernetes clouds," *The Journal of Supercomputing*, Feb. 2023. [Online]. Available: https://doi.org/10.1007/s11227-023-05077-7

[8] Z. Zhong and R. Buyya, "A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources," *ACM Transactions on Internet Technology*, vol. 20, no. 2, pp. 15:1–15:24, Apr. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3378447

[9] K. Kaur, F. Guillemin, V. Q. Rodriguez, and F. Sailhan, "Latency and network aware placement for cloud-native 5G/6G services," in *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*, Jan. 2022, pp. 114–119, iSSN: 2331-9860.

[10] A. Marchese and O. Tomarchio, "Network-Aware Container Placement in Cloud-Edge Kubernetes Clusters," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2022, pp. 859–865.

[11] ——, "Extending the Kubernetes Platform with Network-Aware Scheduling Capabilities," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, Eds. Cham: Springer Nature Switzerland, 2022, pp. 465–480.

[12] L. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, May 2021, pp. 1–9, iSSN: 2641-9874.

[13] L. Toka, "Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes," *Journal of Grid Computing*, vol. 19, no. 3, p. 31, Jul. 2021. [Online]. Available: https://doi.org/10.1007/s10723-021-09573-z

[14] G. P. Mattia and R. Beraldi, "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)*, Nov. 2021, pp. 1–9, iSSN: 2643-7929.

[15] T. Pusztai, S. Nastic, A. Morichetta, V. C. Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, "Polaris Scheduler: SLO- and Topology-aware Microservices Scheduling at the Edge," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, Dec. 2022, pp. 61–70.

[16] A. Orive, A. Agirre, H.-L. Truong, I. Sarachaga, and M. Marcos, "Quality of Service Aware Orchestration for Cloud–Edge Continuum Applications," *Sensors*, vol. 22, no. 5, p. 1755, Jan. 2022, number: 5 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1424-8220/22/5/1755

[17] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive Resource Efficient Microservice Deployment in Cloud-Edge Continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, Aug. 2022, conference Name: IEEE Transactions on Parallel and Distributed Systems.

[18] Prometheus, 2023. [Online]. Available: https://prometheus.io/

[19] Bloomberg, 2023. [Online]. Available: https://github.com/bloomberg/goldpinger

[20] S. Rac, R. Sanyal, and M. Brorsson, "A cloud-edge continuum experimental methodology applied to a 5g core study," *arXiv preprint arXiv:2301.11128*, 2023.

[21] R. Xu, Z. Tu, H. Xiang, W. Shao, B. Zhou, and J. Ma, *CoBEVT: Cooperative Bird's Eye View Semantic Segmentation with Sparse Transformers*, Jul. 2022.

[22] C. I. King, 2023. [Online]. Available: https://github.com/ColinIanKing/stress-ng