# An Optimization Framework for Edge-to-Cloud Offloading of Kubernetes Pods in V2X Scenarios

Estela Carmona [1] and Muhammad Shuaib Siddiqui [2]

[1]i2CAT Foundation
[2]Affiliation not available

October 30, 2023



**Abstract**

Authors' pre-print version of the following work: E. Carmona Cejudo and M. S. Siddiqui, "An Optimization Framework for Edge-to-Cloud Offloading of Kubernetes Pods in V2X Scenarios," 2021 IEEE Globecom Workshops (GC Wkshps).

# An Optimization Framework for Edge-to-Cloud Offloading of Kubernetes Pods in V2X Scenarios

Estela Carmona Cejudo, Muhammad Shuaib Siddiqui
i2CAT Foundation, Barcelona, Spain
{estela.carmona, shuaib.siddiqui}@i2cat.net

***Abstract*—Vehicle-to-everything V2X applications usually have strict latency requirements that can be difficult to meet in traditional cloud-centric networks. By pushing resources to edge servers located closer to the users, end-to-end latency can be greatly reduced. Task offloading in edge-cloud environments refers to the optimization of which tasks should be offloaded between the edge and cloud. Moreover, the use of containers to virtualize applications can further reduce resource and time consumption and, in turn, the latency of V2X applications. Even though Kubernetes has become the de facto container orchestrator, the offloading of Kubernetes pods has not been previously studied in the literature, to the best of the authors' knowledge. In this paper, a theoretical optimization framework is proposed for edge-to-cloud offloading of V2X tasks implemented as Kubernetes pods. First, an utility function is derived in terms of the cumulative pod response time, weighted by the priority levels and resource usage requirements of pods. Based on the optimal theoretical solution to this problem under memory and central processing unit (CPU) constraints, an edge-to-cloud offloading decision algorithm (ECODA) is proposed. Numerical simulations demonstrate that ECODA outperforms first-in, first-served (FIFS) in terms of utility, average pod response time, and occupancy of edge resources. Further, ECODA achieves a good trade-off between performance and computational complexity, and therefore it can help achieve strict latency requirements of V2X applications.**

## I. Introduction

Novel fifth-generation (5G) vehicle-to-everything (V2X) applications are capable of providing road users with network connectivity and road-safety functionalities. However, V2X applications often have strict latency, reliability and bandwidth requirements that pose a challenge to existing networks. The use of containers to virtualize applications has recently gained popularity, particularly in cloud environments. Compared to virtual machines, containers can offer a more lightweight, and less resource and time consuming solution, thus reducing the overall computational cost of V2X applications. Docker [1] and Kubernetes [2] have become the most popular solutions for container runtime and container orchestration, respectively, due to their high levels of maturity, resiliency and flexibility [3]. The smallest computing units that can be deployed in Kubernetes are referred to as *pods*. These are groups of application containers, with shared storage and network resources and, often, initialization containers [2].

However, V2X applications usually have strict latency requirements that can be difficult to meet in traditional cloud-centric networks, even when applications are virtualized. In this sense, edge computing can facilitate the achievement of ever-growing computational demands in vehicular networks, by extending the cloud computing capabilities to the network edge [4]. Edge computing pushes resources to edge servers, located closer to the users, thus reducing end-to-end latency. Computational capabilities at the edge tier are normally smaller than at the cloud tier. However, this can be largely addressed by combining edge and cloud computing, and running each task either at the cloud or the edge tier, depending on overall network conditions and application constraints. In settings with a large number of users, edge computing is unlikely to provide enough resources to meet all the requirements of users' tasks, and it is necessary to offload some tasks to the cloud.

Task offloading in edge-cloud environments refers to the optimization of which tasks should be offloaded between the edge and the cloud tiers, depending on a set of optimization parameters and system constraints. The work in [4] carried out a thorough survey and taxonomy on existing task offloading research, and found several challenges that have not yet been properly addressed in the literature. For example, previous works that have considered the task response time as an optimization parameter mainly considered the average response time of all tasks [5] or the long-term average response of each task [6], which could lead to service level agreement (SLA) violations in some cases. Moreover, only a few works have considered multi-objective optimization. For example, [7] optimized the minimum weighted sum of delay and consumed energy for each user under fairness and maximum tolerable delay constraints. However, this and other works [8] only considered the benefit of users, and did not study the resource usage cost optimization. [9] studied the tradeoff between end-to-end delay of tasks and resource usage cost, but considered separate allocation of tasks with small and large workloads.

Moreover, not many existing researches considered the offloading of containerized applications. While the work in [10]

**This paper is the authors' pre-print version of the following work: E. Carmona Cejudo and M. S. Siddiqui, "An Optimization Framework for Edge-to-Cloud Offloading of Kubernetes Pods in V2X Scenarios," *2021 IEEE Globecom Workshops (GC Wkshps)*.**

assumed the use of containerized tasks, it ignored the heterogeneous provisioning of resources between edge and cloud. In [11], a single-objective optimization framework was presented in which the use of Docker containers was assumed, and the offloading energy cost was optimized under application completion time, computing resources and memory constraints. To the best of the authors' knowledge, no works have so far considered the offloading of Kubernetes pods.

The contributions of this work are as follows. Unlike other researches, the use of Kubernetes as a container orchestrator is considered, and the granularity of applications to be offloaded is defined in terms of pods. First, an utility function is derived in terms of the cumulative pod response time, weighted by the priority levels and resource usage requirements of pods. Then, a theoretical optimization framework is proposed, where the sum of the weighted pod response time is minimized under central processing unit (CPU) and memory constraints. The modeling of pod response time includes pod instantiation and node-to-node latency. The optimal theoretical solution to the offloading optimization problem is found by applying the Lagrange dual function method [12]. An edge-to-cloud offloading decision algorithm (ECODA) is proposed, based on the optimal solution. Numerical simulations demonstrate that ECODA outperforms first-in, first-served (FIFS) in terms of utility, average pod response time, and occupancy of edge resources. Further, ECODA achieves a good trade-off between performance and computational complexity, and therefore it can help achieve strict latency requirements of V2X applications.

The remainder of this paper is organized as follows. Section II provides the system model for this study. Section III formally elaborates the problem at hand. Section IV introduces the proposed ECODA. Numerical results are provided in Section V. Section VI concludes the paper.

## II. System Model

The system model considered in this work consists of a Kubernetes-based cluster formed by an edge node and a cloud connected through a fiber optic link. OpenStack [13] is used as the virtual infrastructure manager. The OpenStack environment is configured with one controller node and three compute nodes in the cloud tier. Kubernetes is installed inside the three compute nodes in the cloud, and in a bare-metal edge server. One node in the cloud serves as the Kubernetes master node, and the remaining cloud and edge nodes serve as Kubernetes worker nodes. It is assumed that the offloading of tasks is between the edge node and any of the cloud nodes, indistinctively, and that the Kubernetes controller node manages the load balancing among cloud nodes. It is also assumed that the provisioning of cloud resources can be extended as required and, therefore, the cloud tier is not resource constrained.

It is assumed that a central service level agreement (SLA) module manages the memory and CPU resources of all nodes and feeds resource usage information to a central decision support system (DSS) module. The DSS decides which tasks must be offloaded to the cloud and then informs the Kubernetes master node, which performs pod offloading as required.

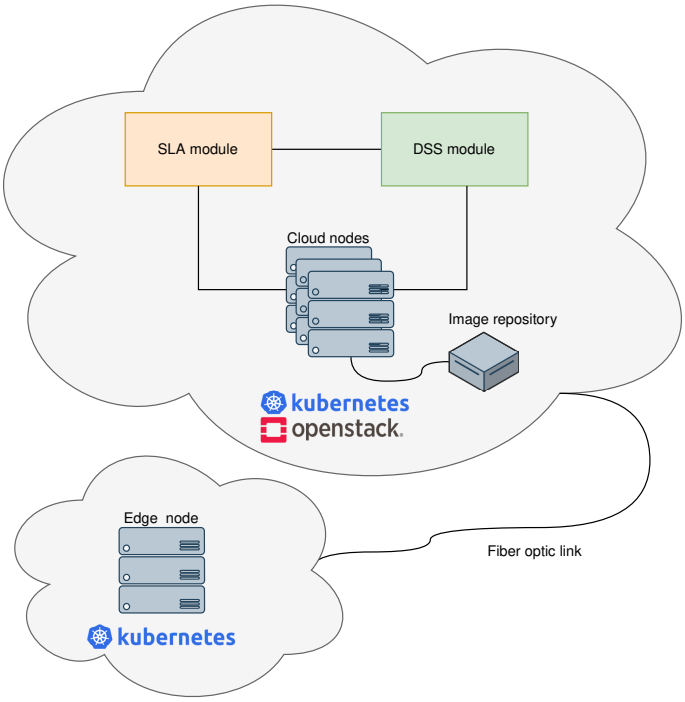


Fig. 1. Architecture setup.

Further, the central cloud hosts a centralized repository of V2X application images. Kubernetes provides container orchestration and enables the automatic deployment of pods, which are preferably instantiated on the edge node in order to reduce network delays.

In our deployment, a total of $N$ pods compete for edge resources. Different pods may correspond to the same or different applications, i.e. there might be multiple replicas of a pod running simultaneously at any given time. Further, it is assumed that applications are stateless, with no logic dependencies among tasks. The total CPU power of the edge server is represented by $\mathcal{R}$, and its available memory is given by $\mathcal{M}$. Pod $n$ requests a guaranteed amount of CPU, $R_n$, and memory, $M_n$. If $\sum_{n=1}^{N} R_n \leq \mathcal{R}$ and $\sum_{n=1}^{N} M_n \leq \mathcal{M}$, the CPU and memory requirements of all pods are guaranteed to be met by the edge server. Otherwise, the edge server is unable to handle all pods' requirements and it is necessary to offload some pods to the cloud. It is assumed that pod images are always pre-loaded in the edge node. At the cloud, pod images must be downloaded first from a local image repository, unless another equal pod has previously been instantiated.

Note that Kubernetes pods can have two types of resource quotas: a request quota is a minimum guaranteed compute or memory resource per pod, whereas a limit quota refers to the maximum compute or memory resources that can be assigned to a pod. In this work, it is assumed that pods only have a request-type quota. When more resources are required, for example because there is a large number of end users, additional pods are deployed, without modifying the resources allocated to already running pods.

## III. Problem Formulation

The problem formulation presented in this work seeks to minimize the cumulative pod response time, weighted by the priority levels and resource usage requirements of pods. CPU and memory constraints are assumed.

$\delta_n$ models the total task response time associated to pod $n$ (node-to-node latency, instantiation time and pre-loading time), and it is given by

$$\delta_n = x_n(l_n + p_n) + i_n, \tag{1}$$

where $l_n$ is the node-to-node latency, $i_n$ is the instantiation time and $p_n$ is the pre-loading time. It is assumed that all pod images are already pre-loaded in the edge node. Only images of pods that have previously been offloaded to the cloud server are pre-loaded there. Further, $x_n$ is a binary variable defined as

$$x_n = \begin{cases} 1, & \text{if pod } n \text{ is offloaded to the cloud,} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Note that, for simplicity, the execution time of pods is ignored in the definition of $\delta_n$.

V2X applications normally have strict latency and application response time requirements, and can be classified according to these requirements and their execution priority. Therefore, the utility function to be optimized is the weighted sum of pod reponse times, i.e.

$$U = \sum_n w_n \delta_n. \tag{3}$$

The weight factor $w_n$ is given by

$$w_n = \frac{T_n}{\omega_n}, \tag{4}$$

where $T_n$ is the normalized resource usage of pod $n$, defined as the sum of normalized memory and CPU requirements. $\omega_n$ is a discrete priority value, where a larger value of $\omega_n$ models a higher execution priority.

The problem formulation to be considered in this work is given by

$$\min_{x_n} \quad \sum_n w_n \delta_n \tag{5a}$$

$$\text{s.t.} \quad x_n \in \{0,1\}, \forall n, \tag{5b}$$

$$\sum_n (1 - x_n) R_n \leq \mathcal{R}, \tag{5c}$$

$$\sum_n (1 - x_n) M_n \leq \mathcal{M}. \tag{5d}$$

Constraint (5b) indicates that pod $n$ can only be deployed either in the edge or in the cloud. Constraint (5c) ensures that the sum of required CPU resources of pods running in the edge is lower than or equal to available CPU resources at the edge server. Constraint (5d) guarantees that the sum of required memory resources of pods running in the edge is lower than or equal to available memory resources at the edge server.

In order to simplify the analysis, CPU and memory resource usage constraints (5c) and (5d) can be combined into a single overall resource usage constraint. For this purpose, since $R_n$ and $M_n$ represent units of different dimensions, it is necessary to convert them into equivalent dimensionless units, i.e.

$$\hat{R}_n = \frac{R_n}{\mathcal{R}}, \tag{6}$$

and

$$\hat{M}_n = \frac{M_n}{\mathcal{M}}, \tag{7}$$

where $0 \leq \hat{R}_n \leq 1$ and $0 \leq \hat{M}_n \leq 1$. Therefore, constraints (5c) and (5d) become

$$\sum_n (1 - x_n) \hat{R}_n \leq 1, \tag{8}$$

$$\sum_n (1 - x_n) \hat{M}_n \leq 1, \tag{9}$$

respectively. Further, the linear combination of (8) and (9) yields

$$\sum_n (1 - x_n) T_n \leq 2, \tag{10}$$

where $T_n = \hat{R}_n + \hat{M}_n$ represents the overall normalized resource usage by pod $n$.

Problem (5) has multiple solutions depending on the values taken by the set of variables $\{\hat{R}_n, \hat{M}_n\}$. Therefore, problem (5) is tackled by dividing it up into three equivalent single-constrained sub-problems, where either (8), (9) or (10) are considered as a constraint. The single-constrained sub-problem of interest is selected dynamically depending on changing resource usage in the system. This strategy is explained in more detail in Section IV, where ECODA is introduced.

### A. CPU-constrained system

In the scenario where the CPU constraint (8) is considered, but not the memory constraint (9), problem (5) can be simplified as

$$\min_{x_n} \quad \sum_n w_n \delta_n \tag{11a}$$

$$\text{s.t.} \quad x_n \in \{0,1\}, \forall n, \tag{11b}$$

$$\sum_n (1 - x_n) \hat{R}_n \leq 1. \tag{11c}$$

The binary nature of variable $x_n$ in constraint (5b) adds complexity to problem (5). In order to simplify the analysis, the binary allocation variable $x_n$ is relaxed and transformed into an equivalent real version, $\hat{x}_n \in \mathbb{R}$. Assuming that $\hat{\delta}_n$ is obtained by substituting $\hat{x}_n$ into (1), the equivalent to problem (11) is given by

$$\min_{\hat{x}_n} \quad \sum_n w_n \hat{\delta}_n \tag{12a}$$

$$\text{s.t.} \quad \sum_n (1 - \hat{x}_n) \hat{R}_n \leq 1. \tag{12b}$$

Problem (12) is a linear program in standard form, and it can be solved by applying the Lagrange dual function method [12]. The Lagrangian of (12) is given by

$$\mathcal{L} = \sum_n w_n \hat{\delta}_n + \lambda \left( \sum_n (1 - \hat{x}_n) \hat{R}_n - 1 \right), \tag{13}$$

and the dual function is then given by

$$g = \inf_{\hat{x}_n} \left( w_n (l_n + p_n) - \lambda \hat{R}_n \right) \hat{x}_n + w_n i_n + \lambda \left( \hat{R}_n - 1 \right), \tag{14}$$

where $\inf$ denotes infimum. Therefore, $g = -\infty$, except when

$$\left( w_n (l_n + p_n) - \lambda \hat{R}_n \right) = 0, \tag{15}$$

or, equivalently, when

$$\lambda = \frac{w_n (l_n + p_n)}{\hat{R}_n}. \tag{16}$$

When (16) holds, the nontrivial lower bound to problem (12) is given by

$$\hat{x}_n = w_n i_n + \lambda \left( \hat{R}_n - 1 \right), \tag{17}$$

By substituting (16) in (17), the optimal $\hat{x}_n$ is given by

$$\hat{x}_n = w_n \left( i_n + (l_n + p_n) \left( 1 - \frac{1}{\hat{R}_n} \right) \right). \tag{18}$$

Assuming

$$\mathcal{X} = [\hat{x}_1, \cdots, \hat{x}_n, \cdots, \hat{x}_N], \tag{19}$$

the value in $\mathcal{X}$ that minimizes the utility function $U$ is $\min \mathcal{X}$. Thus, it is optimal to schedule on the edge the pod that yields $\min \mathcal{X}$. Similarly, the value that maximizes $U$ is $\max \mathcal{X}$, and it is optimal to offload the pod that yields $\max \mathcal{X}$ to the cloud. Thus, the optimal solution to the original problem in (11) is

$$x_n = \begin{cases} 1, & \text{for } n \text{ such that } \hat{x}_n = \max \mathcal{X}, \\ 0, & \text{otherwise.} \end{cases} \tag{20}$$

### *B. Memory-constrained system*

Assume that only the memory constraint in problem (5) (9) is considered, but not the CPU constraint (8). Then, problem (5) can be simplified as

$$\min_{x_n} \quad \sum_n w_n \delta_n \tag{21a}$$

$$\text{s.t.} \quad x_n \in \{0, 1\}, \forall n, \tag{21b}$$

$$\sum_n (1 - x_n) \hat{M}_n \leq 1. \tag{21c}$$

Further, assume that $x_n$ is relaxed and transformed into an equivalent real version, $\hat{x}_n \in \mathbb{R}$, and that $\hat{\delta}_n$ is obtained by substituting $\hat{x}_n$ into (1). Then, (21) can be simplified as

$$\min_{\hat{x}_n} \quad \sum_n w_n \hat{\delta}_n \tag{22a}$$

$$\text{s.t.} \quad \sum_n (1 - \hat{x}_n) \hat{M}_n \leq 1. \tag{22b}$$

By applying the same methodology of section III-A, the optimal solution to problem (22) is found:

$$\hat{x}_n = w_n \left( i_n + (l_n + p_n) \left( 1 - \frac{1}{\hat{M}_n} \right) \right). \tag{23}$$

The solution to the original problem (21) is given by replacing (23) into (19), and the resulting set $\mathcal{X}$ into (20).

### *C. Mixed-resource-constrained system*

The original problem formulation (5) can be re-written by combining constraints (5c) and (5d) into the normalized linear combination of both, as given by (10). Therefore, problem (5) can be re-written as

$$\min_{x_n} \quad \sum_n w_n \delta_n \tag{24a}$$

$$\text{s.t.} \quad \sum_n (1 - x_n) T_n \leq 2. \tag{24b}$$

By transforming $x_n$ into an equivalent real version, $\hat{x}_n \in \mathbb{R}$, and by writing $\delta_n$ in (1) as $\hat{\delta}_n$ in terms of $\hat{x}_n$, problem (24) can be simplified as

$$\min_{\hat{x}_n} \quad \sum_n w_n \hat{\delta}_n \tag{25a}$$

$$\text{s.t.} \quad \sum_n (1 - \hat{x}_n) T_n \leq 2. \tag{25b}$$

After applying the same methodology of section III-A, the optimal solution to problem (25) is found as:

$$\hat{x}_n = w_n \left( i_n + (l_n + p_n) \left( 1 - \frac{2}{T_n} \right) \right). \tag{26}$$

The solution to the original problem (21) is given by replacing (26) into (19), and the resulting set $\mathcal{X}$ into (20).

## IV. Edge-to-Cloud Offloading Decision

The offloading optimization problem in (5) has multiple solutions, since there are many combinations of the values $\hat{R}_n$ and $\hat{M}_n$ that represent the same overall resource usage severity.

In order to deal with the distinct degrees of resource usage severity, three different scenarios are considered:

*1) $\mathcal{R} > \mathcal{M}$ and $\mathcal{R} \geq \alpha$, where $\alpha$ represents a given CPU resource usage threshold:* In this case, the CPU usage is more critical than the memory usage. Therefore, in order to carry out offloading, a CPU-constrained system as given in Section III-A is considered, and the optimal offloading decision is found by replacing (18) into (19), and the resulting set $\mathcal{X}$ into (20).

*2) $\mathcal{M} > \mathcal{R}$ and $\mathcal{M} \geq \beta$, where $\beta$ is a given memory usage threshold:* In this scenario, the memory usage is considered more critical than the CPU usage, and a memory-constrained system as given in III-B is considered. Therefore, the optimal offloading decision is given by replacing (23) into (19), and the resulting set $\mathcal{X}$ into (20).

*3) $\mathcal{R} < \alpha$ and $\mathcal{M} < \beta$, or $\mathcal{R} = \mathcal{M}$:* In this situation, the memory and CPU usage constraints are deemed equally critical, and a mixed-resource-constrained system is considered, as given in Section III-C. Therefore, the optimal offloading decision is found by replacing (26) into (19), and the resulting set $\mathcal{X}$ into (20).

### A. Edge-to-Cloud Offloading Decision Algorithm (ECODA)

After the optimal results to problems (11), (21) and (24) are found, an ECODA is proposed, as summarized in Algorithm 1. In the proposed ECODA, the system is first classified as CPU-constrained, memory-constrained or mixed-resource-constrained. The system classification result depends on the CPU and memory usage at the edge, i.e. $\mathcal{R}$ and $\mathcal{M}$, respectively; and on the CPU and memory usage thresholds, $\alpha$ and $\beta$, respectively. Based on this classification, the optimal offloading decision is given by replacing either (18), (23) or (26) into (19), and the resulting $\mathcal{X}$ into (20). ECODA iteratively calculates the optimal edge-to-cloud offloading decision, based on the system classification, and then performs offloading of a pod. This procedure is repeated in a loop, and ECODA finishes when the overall resource usage at the edge is reduced to $\mathcal{R} < \alpha$ and $\mathcal{M} < \beta$, simultaneously.

**Algorithm 1** Edge-to-cloud offloading decision algorithm

```
Known: α, β
Compute R, M
while R ≥ α or M ≥ β do
    if R > M and R ≥ α then
        Replace (18) into (19), replace X into (20)
        Compute R, M
    else if M > R and M ≥ β then
        Replace (23) into (19), replace X into (20)
        Compute R, M
    else
        Replace (26) into (19), replace X into (20)
        Compute R, M
    end if
end while
```

During each iteration, ECODA takes the optimal offloading decision based on the problem classification. The optimal solutions to problems (12), (22) or (25), are also optimal solutions to the original problem (5).

### B. Computational Complexity of Algorithm 1 (ECODA)

Algorithm 1 performs one single computation per offloaded pod. In the worst case scenario, all pods are offloaded from edge to cloud. Therefore, it is trivial that the computational complexity of Algorithm 1 is linear with the number of pods $N$, i.e. it is of order $\mathcal{O}(N)$.

## V. Numerical Results

The performance of ECODA is validated through Monte Carlo simulations, performed in Matlab with $10^4$ repetitions per experiment in order to guarantee statistical significance of results. Simulation parameters have been extracted from measurements in our experimental testbed in Barcelona, and are as follows.

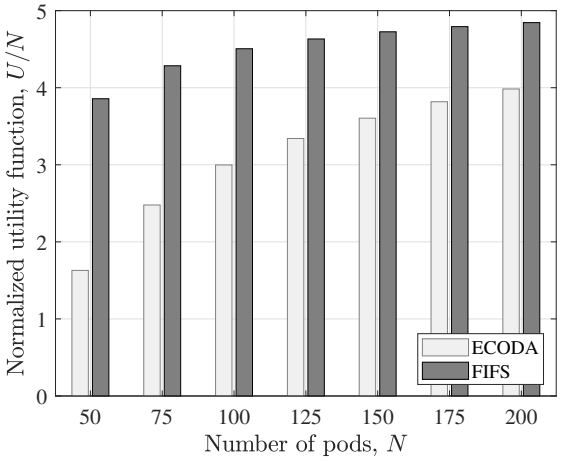


Fig. 2. Normalized utility function vs. number of pods.

The node-to-node latency is assumed to have a fixed value of $l_n = 30$ millisecongs, $\forall n$. The pre-loading time of pod images, $p_n$, is modeled as a uniformly distributed random variable with values in the range between 10 seconds and 100 seconds. After pre-loading, the additional required pod instantiation time $i_n$ is modeled as a uniformly distributed random variable with values in the range between 200 milliseconds and 5 seconds.

It is assumed that pods are classified according to their priority level, which is modeled as an integer variable $\rho_n$ that takes values from the range $I_\rho = [1, 4]$, with one denoting the lowest priority value, and four denoting the highest priority value. $\omega_n$ is calculated by normalizing the priority value, i.e. $\omega_n = \rho_n / (\max I_\rho - \min I_\rho)$. Further, the normalized resource requirement of pod $n$, $T_n$, is modeled as a uniformly distributed random variable with values in the range 0.02-0.2. It is assumed that individual normalized memory and CPU requirements per pod are uniformly and randomly distributed between 0.01 and 0.1. In addition, it is assumed that servers in the cloud tier have enough CPU and memory resources for all the offloaded pods.

The performance of ECODA is compared to that of a FIFS algorithm [11]. In FIFS, pods are allocated to the edge server in a first-come, first-saved basis; when there are no available resources at the edge server, pods are offloaded to the cloud. Fig. 2 shows the value of the normalized utility function, defined as $U/N$, for a varying number of pods, $N$. It is proved that ECODA outperforms FIFS, yielding a mean average reduction of a $31.9\%$ of the value of $U$ achieved with FIFS. The benefit is larger for a smaller $N$, since a larger percentage of pods can be run on the edge, thus reducing the average pod response time. This, in turn, reduces the value of $U$.

Fig. 3 represents the value of $U/N$ for fixed $N = 100$ and varying CPU and memory usage thresholds, where it is assumed that $\alpha = \beta$. The mean average reduction of the value of $U$ is of approximately $15.9\%$. Further, the value of $U$ decreases faster

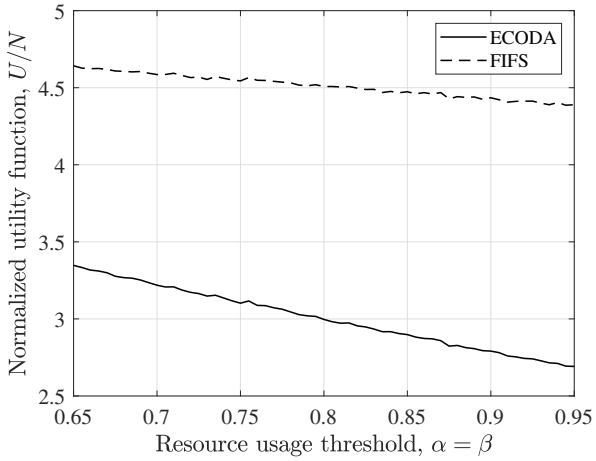


Fig. 3. Utility function vs. resource usage threshold.

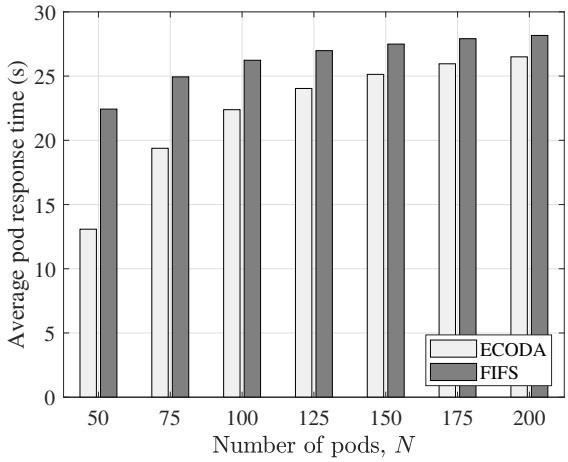


Fig. 4. Average pod response time.

for ECODA than for FIFS when thresholds $\alpha$ and $\beta$ increase. The reason for this is that, for increasing values of $\alpha$ and $\beta$, more pods are executed in the edge server, and therefore the cumulative end-to-end delay is reduced. This, in turn, decreases the value of $U$. Since ECODA allocates pods in an optimal manner, the impact on $U$ is larger than that yielded by the application of FIFS.

For fixed threshold levels $\alpha$ and $\beta$, ECODA also improves the pod response time with respect to that achieved with FIFS by nearly fours seconds on average, as demonstrated in Fig. 4. According to (3), the utility function $U$ is directly proportional to the response time per pod, $\delta_n$. A larger benefit is obtained for smaller $N$, since $U$ decreases when a larger share of pods are run at the edge. For $N = 50$, the average pod response time is over nine seconds lower for ECODA than for FIFS.

Last, Fig. 5 compares the performance of ECODA and FIFS in terms of average resource usage at the edge, for $\alpha = 0.8$, $\beta = 0.8$ and varying $N$. ECODA optimizes CPU and memory usage at the edge tier, since the total resource usage approximately equals the threshold values set by $\alpha$ and $\beta$. In contrast, the

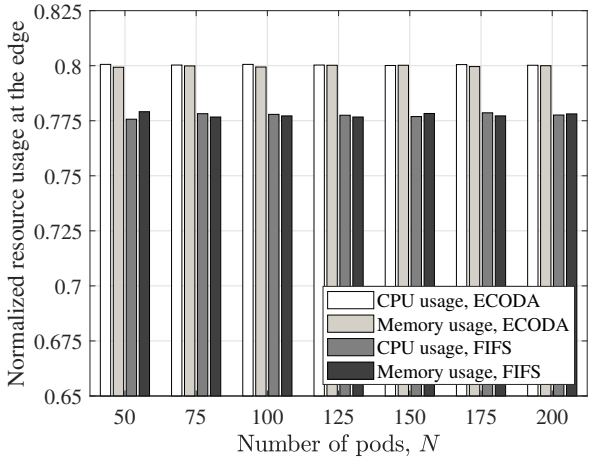


Fig. 5. Normalized resource usage at the edge.

edge CPU and memory usage achieved with FIFS with respect to ECODA are reduced by $2.86\%$ and $2.78\%$, respectively. This result implies that the cumulative response time of pods is lower when ECODA is applied, thus lowering the value of the utility function $U$.

Overall, ECODA achieves a good performance with respect to FIFS, with a very low implementation complexity. Moreover, in the particular case of V2X verticals, applications normally have strict latency constraints, and it is usually preferable to execute applications at the edge tier as much as possible, in order to minimize end-to-end latency. In this sense, ECODA can aid fulfilling the requirements of V2X applications, by minimizing the response time and maximizing the edge occupancy, as demonstrated in Fig. 4 and Fig. 5, respectively.

## VI. Conclusions

In this paper, an optimization framework was presented for edge-to-cloud offloading of V2X tasks implemented as Kubernetes pods. First, an utility function was derived in terms of cumulative weighted pod response time, and a utility minimization problem was proposed, under edge CPU and memory occupancy constraints. Based on the optimal solution to this problem, a computationally efficient ECODA was presented for edge-to-cloud offloading of Kubernetes pods. Through numerical simulations, it was demonstrated that ECODA outperforms FIFS in terms of utility, average pod response time, and occupancy of edge resources. Further, it was demonstrated that ECODA provides a good trade-off between performance and computational complexity. Therefore, ECODA can help achieve strict latency requirements of V2X applications.

## Acknowledgment

This work is currently supported by the European Union's Horizon 2020 Research and Innovation Programme, grant agreement No: 871536 (PLEDGER).

## REFERENCES


[1] "Docker," https://www.docker.com, accessed: 2021-06-22.

[2] "Production-grade container orchestration," https://kubernetes.io, accessed: 2021-06-22.

[3] A. Pereira Ferreira and R. Sinnott, "A performance evaluation of containers running on managed Kubernetes services," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2019, pp. 199–208.

[4] B. Wang, C. Wang, W. Huang, Y. Song, and X. Qin, "A survey and taxonomy on task offloading for edge-cloud computing," *IEEE Access*, vol. 8, pp. 186 080–186 101, 2020.

[5] F. Sun, F. Hou, N. Cheng, M. Wang, H. Zhou, L. Gui, and X. Shen, "Cooperative task scheduling for computation offloading in vehicular cloud," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 11, pp. 11 049–11 061, 2018.

[6] N. Cheng, F. Lyu, W. Quan, C. Zhou, H. He, W. Shi, and X. Shen, "Space/aerial-assisted computing offloading for IoT applications: A learning-based approach," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 5, pp. 1117–1129, 2019.

[7] J. Du, L. Zhao, J. Feng, and X. Chu, "Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee," *IEEE Transactions on Communications*, vol. 66, no. 4, pp. 1594–1608, 2018.

[8] R. Duan, J. Wang, C. Jiang, Y. Ren, and L. Hanzo, "The transmit-energy vs computation-delay trade-off in gateway-selection for heterogenous cloud aided multi-UAV systems," *IEEE Transactions on Communications*, vol. 67, no. 4, pp. 3026–3039, 2019.

[9] Z. Wang, S. Zheng, Q. Ge, and K. Li, "Online offloading scheduling and resource allocation algorithms for vehicular edge computing system," *IEEE Access*, vol. 8, pp. 52 428–52 442, 2020.

[10] O. Chabbouh, S. B. Rejeb, Z. Choukair, and N. Agoulmine, "A strategy for joint service offloading and scheduling in heterogeneous cloud radio access networks," *EURASIP Journal on Wireless Communication Networks*, vol. 2017, no. 196, Nov. 2017.

[11] J. Tang, R. Yu, S. Liu, and J.-L. Gaudiot, "A container based edge offloading framework for autonomous driving," *IEEE Access*, vol. 8, pp. 33 713–33 726, 2020.

[12] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge University Press, 2004.

[13] "The most widely deployed open source cloud software in the world," https://www.openstack.org, accessed: 2021-06-22.