Real-Time Resource Scaling and Service Allocation for Mobile Devices in Edge-Cloud Continuum

Nasim Samimi[1][0000−0002−6296−8792], Daniel Casini[2][0000−0003−4719−3631], Luca Abeni[2][0000−0002−7080−9601], Twan Basten[1][0000−0002−2274−7274], Marc Geilen[1][0000−0002−2274−7274], Mitra Nasri[1][0000−0002−1052−8437], and Alessandro Biondi[2][0000−0002−6625−9336]

[1] Eindhoven University of Technology, Netherlands
[2] Scuola Superiore Sant'Anna, Italy

Abstract. Edge computing established itself as a key paradigm to complement Cloud infrastructures by decentralising the processing closer to where data originates, reducing energy consumption and the amount of data sent to the Cloud, with further benefits in improving privacy and latency. This paper proposes dynamic task allocation algorithms for mobile IoT users to offload workloads in mixed Edge/Cloud resources while optimising quality of service (QoS), cost, and guaranteeing timing constraints. Our approach uses CPU reservations, such as those provided by the SCHED_DEADLINE policy, to guarantee real-time constraints and isolation between mutually untrusted applications potentially belonging to different tenants. The evaluation reports the impact of various system configurations, including node selection, reallocation strategies, and partitioning heuristics, demonstrating the effectiveness of our approach in ensuring real-time performance while reducing costs.

1 Introduction

With the growing demand for low-latency and intensive computing for mobile IoT devices, Edge computing has emerged as a crucial paradigm to complement traditional Cloud infrastructures. Although Cloud computing provides scalable resources, it often suffers from high network delays, making it unsuitable for real-time applications. Edge computing addresses this challenge by leveraging geographically distributed data centres that are closer to data origin than centralised Cloud servers, with several typical benefits such as reduced latency and energy consumption and improved privacy by sending less data to the Cloud [36].

Mixed Edge-Cloud computing can be used by mobile IoT devices [16, 61], which are often called to offload compute-intensive workloads through the network due to their resource constraints. In this context, designing allocation and offloading strategies to guarantee timing performance, maximize quality of service, and minimize cost is of key importance [46].

Edge and Cloud servers naturally co-locate mutually untrusted applications from different tenants, potentially exposing each other to potentially harmful

timing interferences that can considerably delay their completion if not properly handled: in extreme cases, e.g., in case of a cyber-attack or a severe software bug, malicious applications can monopolize the processing bandwidth, causing a denial of service to co-located applications. The SCHED_DEADLINE scheduling class of Linux [32] provides an effective solution to this issue: it implements the Constant Bandwidth Server [2], a reservation algorithm that both allows to have a fine-grained partitioning of the overall CPU bandwidth, while also enforcing to not exceed the resource assignment.

This setup can be useful for Function-as-a-Service (FaaS) [28] and serverless functions, where computational workloads are encapsulated in lightweight and dynamically scalable execution environments, and it is especially useful for applications requiring real-time processing, such as video analytics, autonomous vehicle decision-making, and IoT-based events.

Allocation of offloaded applications in Edge-Cloud systems presents several challenges. On one hand, resource availability at the Edge is inherently limited compared to the Cloud. The cost of the resources in the Cloud can be higher than in the Edge, and it introduces a longer network delay, often making the Edge a better fit. On the other hand, Edge nodes, which are part of smaller-scale data centres, have limited processing capacity. This creates non-trivial trade-offs in allocation and offloading process, highlighting the need to optimise resource allocation.

This paper. Here, we propose new service allocation and offloading algorithms for real-time applications in Edge-Cloud systems, addressing the noted challenges. We consider SCHED_DEADLINE reservations and partitioned scheduling to provide real-time guarantees and resource isolation in multi-tenant multi-core edge and cloud nodes. Additionally, partitioned scheduling avoids potential issues with dynamic voltage and frequency scaling (DVFS) that arise with global scheduling, as each core voltage and frequency can be scaled independently. Our key contributions are as follows.

- A comprehensive model of a modern distributed system composed of mobile IoT nodes that offload computation to both the Edge and Cloud, based on a task-level model.
- QoS- and cost-aware algorithms for the allocation of dynamic real-time workloads offloaded by mobile IoT devices, providing hard schedulability guarantees and dynamic resource scaling to optimise QoS and cost.
- Reallocation techniques for refining existing allocations using fast heuristics to improve task placement with a low overhead.
- An interval-based allocation method, where requests are buffered and processed in batches, reducing decision-making overhead and fragmentation.
- A thorough evaluation of the algorithms and techniques through extensive experiments

2 Related work

Regardless of the offloading technology (e.g., hypervisors or OS-level virtualisation), a key challenge is allocating node resources to virtual environments

Table 1: Comparison of related works with this paper

Related Works	Distributed	Real-Time Guarantees	Workload Model	Dynamic resource Scaling
[9, 19, 24, 25, 31, 40, 43, 44, 51]	No	Hard	Task	No
[12, 13]	No	Hard	Task	No
[7, 50]	Yes	Hard	Task	No
[48, 49, 65]	Edge	Hard	Task	No
[14]	Edge	Hard	Task	Yes
[39]	Edge	Soft	Task	Yes
[8, 54]	Edge/Fog	Soft	Task	No
[52]	Cloud	Soft	Job	Yes
[3, 16, 27, 29, 30, 35, 38, 41, 53, 61, 66]	Edge(or Fog)-Cloud	Soft	Job	No
[4, 6, 11, 18, 21, 26, 33, 36, 42, 46, 55, 56, 60, 64, 67]	Edge-Cloud	None	Job	-
[5, 20, 23, 58]	Edge-Cloud	Soft	Job	Yes
[57]	Edge-Cloud	Soft	Task	No
[17, 34, 37, 59]	Edge-Cloud	Soft	Task (VNF)	Yes
[45]	Edge-Cloud	Soft	Task (container)	No
[15]	Edge-Cloud	Hard	Job	No
This work	**Edge-Cloud**	**Hard**	**Task**	**Yes**

hosting the offloaded applications. This requires good trade-offs to avoid both resource over- and under-provisioning. The problem of optimising resource allocation to maximise the utility of real-time applications has been studied in prior work. Q-RAM (QoS-Aware Resource Allocation Model) [43] models multiple real-time applications on a single node. It was later extended to more complex cases [44], including discrete CPU allocations [31] and multi-core systems [24,25]. However, it does not address distributed systems or Edge/Cloud environments with offloaded applications from mobile IoT nodes. Timing guarantees for real-time applications on single nodes are often based on Compositional Scheduling Frameworks (CSF) [9, 19, 40, 51], which may lead to resource over-allocation. Other works consider local load balancing with real-time guarantees on multi-core embedded systems [12, 13], but do not address distributed or Edge-Cloud environments.

In Edge-Cloud related studies, some works address real-time constraints, aiming to meet deadlines [3,5,15–17,20,23,27,29,30,35,38,41,53,58,61,66]. However, these approaches typically model tasks as jobs that are executed and completed upon offloading. In contrast, our work considers a task-level model, where tasks are deployed to Edge or Cloud nodes and continuously process multiple incoming jobs over time. Some adopt similar task models [45,57], but with soft guarantees.

Other works provide soft or hard real-time guarantees but do not fully integrate with the Edge-Cloud environment. Some are limited to Fog or Edge-only systems [8, 14, 39, 48, 49, 54, 65], others target distributed real-time embedded systems [7, 50], or focus on Cloud [52] only. While relevant in localised settings, these approaches do not address broader resource allocation challenges across the Edge-Cloud continuum. Specialising in resource allocation for specific application classes can improve performance at the cost of generality. For example, Network Function Virtualisation (NFV) has been explored [17, 34, 37, 59]. These approaches focus on optimising network-related operations such as routing and load balancing. Accordingly, they model tasks as NFVs and do not address CPU-level allocation under real-time constraints.

Buffering and processing tasks in intervals naturally aligns with strategies like Deferred Acceptance (DA), a classic stable matching algorithm. In DA, tasks and Edge servers iteratively propose and respond based on preferences until a stable assignment is reached [42, 55]. However, DA aims for stability, not optimal allocation or offloading based on QoS or schedulability. It proceeds in multiple rounds, making it less suitable in real-time systems with frequent task arrivals. Several works in the literature have tackled task scheduling and offloading in Edge or Edge-Cloud systems with the goal of minimising latency, but without enforcing real-time guarantees [4, 6, 11, 18, 21, 26, 33, 36, 46, 56, 60, 64, 67].

To our knowledge, this is the first work that supports hard real-time guarantees for task-level models, mobile, multi-tenant workloads in Edge-Cloud systems, with QoS- and cost-aware allocation, and dynamic scaling (see Table 1).

3 System model and problem definition

We consider a smart mobility system where connected mobile IoT nodes (e.g. vehicles) dynamically request services, such as data aggregation or analytics, to nearby Edge domains. These IoT nodes move through different geographic locations, such as urban road segments or highway sections, where Edge domain nodes are deployed to provide low-latency computation.

We consider a reactive scenario where mobile IoT nodes enter a new geographic area and request some services for computation-intensive tasks. For example, for vehicles, the example applications include traffic density estimation or road condition monitoring. The service orchestrator must decide where and how to allocate the services. Once selected, the requested service is deployed as a containerised microservice or a Function-as-a-Service (FaaS) function, allowing for rapid instantiation. The mobile node then establishes a connection to the offloaded service, ensuring continuous processing as it moves in the network.

System definition. The system is composed of a dynamic set $\mathcal{C}$ of Cloud nodes and a dynamic set $\mathcal{E}$ of Edge nodes. A Cloud node is, in general, a virtual machine, while an Edge node can either be a whole computing (physical) platform or a virtual machine running on a shared cluster of Edge nodes (e.g., accessed under a rental/leasing scheme). Edge nodes are partitioned into multiple *domains*, where $\mathcal{E}_k$ denotes the Edge nodes within the k-th domain, denoted by D_k. The

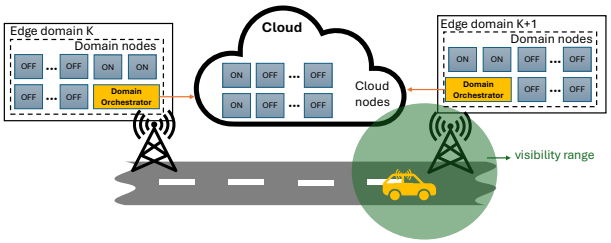


Fig. 1: Offloading example from mobile nodes to Edge/Cloud.

Edge nodes within the same domain share the same capabilities in terms of radio connectivity (e.g., they are connected to the same radio station), meaning that they can all communicate with the same set of IoT nodes. As IoT nodes move within the environment covered by the system, they can change the domain to which they connect. Furthermore, the fact that the set of Cloud nodes and the set of Edge nodes are dynamic means that Cloud nodes and Edge nodes can be powered on or off depending on the current service requests by IoT nodes. For each domain, an orchestrator node is considered to determine a suitable allocation for each service requested by IoT nodes. An overview of the system architecture is shown in Figure 1.

Resource model. Each node N_q in $\mathcal{E}_k$ or $\mathcal{C}$ includes M_q processor cores (either virtual or physical) and implements *partitioned* Earliest Deadline First (EDF) scheduling [10]. The x-th core of N_q is denoted with $c_{q,x} \in C_q$, where C_q denotes the set of cores in N_q. Both Cloud and Edge nodes run a dynamic set of applications composed of multiple tasks, each encapsulated within a reservation server (e.g., implemented by the SCHED_DEADLINE [32] scheduler in Linux). While mainline SCHED_DEADLINE maps one reservation per task, hierarchical SCHED_DEADLINE [1] allows encapsulating multiple tasks. Each node $N_q \in \mathcal{E} \cup \mathcal{C}$ is characterised by a corresponding set of allocated reservation servers $\mathcal{R}_q$. Similarly, we denote the set of reservations allocated to core $c_{q,x}$ of node N_q in domain D_k by $\mathcal{R}_{q,x}$.

Workload model. Each application τ_i is modelled as a triplet (Q_i, P_i, m_i) and is served by m_i reservation servers r_i, each allocated on a different core of the node, with the same budget Q_i and period P_i. This follows the parallel task execution model (see [19] for more details on how to configure these parameters in order to guarantee timing requirements). The set of applications running on a node N_q is denoted by $\Gamma(N_q)$. The set of applications on each node is dynamic because applications can be activated or deactivated depending on the current services requested by IoT nodes.

We consider a predefined set of applications available for offloading. When a user requests one of these applications, a dedicated instance is deployed to serve that request. Once instantiated, this instance is referred to as a *service*. The service remains persistently deployed, processing requests from the user until a deallocation is triggered.

Dynamic scaling. IoT nodes dynamically issue service activation and deactivation requests to the domain to which they are connected. We assume that each service S_i is provided by the system in two quality-of-service (QoS) modes:

- *Standard*: The service S_i is entirely provided by an application τ_i running on an Edge node. This mode allows the service to be offered with the highest QoS, and not using the Cloud nodes prevents any additional latency.
- *Reduced*: The service S_i is cooperatively provided by both an Edge and a Cloud node, through two corresponding applications τ_i^E and τ_i^C. The Edge-side component τ_i^E handles communication with the requesting IoT node, while the compute-intensive logic is offloaded to the Cloud-side component τ_i^C. This mode introduces additional latency due to Cloud communication and therefore results in reduced QoS. While the service remains functional in reduced mode, not only is part of the application offloaded to the Cloud and it may suffer from lower responsiveness, but the application itself may also adapt its internal behaviour to operate under constrained conditions. For example, a video analytics service may reduce its frame rate or processing resolution, or a sensing application may lower its sampling frequency.

Satisfying a service activation request requires finding first an Edge node to which a new application τ_i can be allocated, passing an *admission test*. If the allocation is successful, the service is provided with standard QoS. Otherwise, three options are available:

1. Offer the service with Standard QoS by powering on a new Edge node in the requested domain (if available), allowing allocating τ_i to execute the service;
2. Offer the service with Reduced QoS by finding an Edge node and a Cloud node to allocate τ_i^E and τ_i^C, passing two corresponding admission tests without powering on a new node;
3. Offer the service with Reduced QoS by both finding an Edge node to allocate τ_i^E and powering on a new Cloud node to allocate τ_i^C.

Conversely, satisfying service deactivation requests implies deallocating the corresponding application(s). These events may trigger a reallocation of applications to reduce the number of active nodes or improve the QoS of some service(s).

Admission test. When a service request S_i is received, the domain-specific orchestrator node is responsible for finding a suitable allocation for the corresponding application that satisfies the resource requirements, assessed by submitting it to an *admission test* to evaluate the schedulability conditions. Partitioned SCHED_DEADLINE reservations, based on the EDF schedulability theory [10, 32], are capable of providing a guaranteed processor bandwidth and a worst-case service latency to the workloads they serve by guaranteeing a budget of Q_i time units of service every period P_i, provided that the sum of the requested bandwidths of all the reservations served on each core is less than or equal to one. Please refer to [19] for more details on schedulability analysis (MPR analysis), and determining the budget and period that guarantee the application-specified deadlines.

Standard Mode. Since each application is composed of a triplet (Q_i, P_i, m_i), mapped to m_i reservations with bandwidth $\alpha_i = Q_i/P_i$, the application τ_i corresponding to the incoming service request S_i can be feasibly allocated in a given domain D_k if and only if

$$\exists_{N_q \in D_k} \exists_{C'_q \subseteq C_q} |C'_q| = m_i \wedge \forall_{c_{q,x} \in C'_q} \sum_{r_j \in \mathcal{R}_{q,x}} \alpha_j + \alpha_i \leq 1, \tag{1}$$

with C'_q being an arbitrary subset of C_q of size m_i. Note that a feasible allocation cannot be obtained if the node was overloaded before issuing the request.

Reduced Mode. When an Edge-Cloud split solution is applied, S_i is applied at a reduced QoS mode, which is characterised by two applications τ_i^E (Edge) and τ_i^C (Cloud) with two sets of `SCHED_DEADLINE` parameters: (Q_i^E, P_i^E, m_i^E) and (Q_i^C, P_i^C, m_i^C). The first set of parameters refers to the part of the application running on the Edge τ_i^E with the purpose of coordinating the work that occurs in the Cloud. Clearly, the parameters (Q_i^E, P_i^E, m_i^E) are characterised by a much lower budget and possibly a lower number of required cores than for the case of an Edge allocation, as the actual computation occurs on more powerful platforms in the Cloud. This allows favouring the allocation at the Edge: indeed, if Eq. (1) is not satisfied for a given domain D_k, it could be satisfied with the reduced QoS mode. Thus, the admission of the service in the reduced QoS mode, composed of two applications τ_i^E and τ_i^C, must pass two admission tests: **(i)** the admission test for τ_i^E at the Edge, reported in Eq. (1), applied using $\alpha_i^E = Q_i^E/P_i^E$ instead of α_i; **(ii)** a similar admission test for τ_i^C in the Cloud, reported in Eq. (2):

$$\exists_{N_q \in \mathcal{C}} \exists_{C'_q \subseteq C_q} |C'_q| = m_i^C \wedge \forall_{c_{q,x} \in C'_q} \sum_{r_i \in \mathcal{R}_{q,x}} \alpha_j^C + \alpha_i^C \leq 1, \tag{2}$$

where $\alpha_i^C = Q_i^C/P_i^C$. The admission test in Eq. (2) may fail if the computing resources are not enough to accept the new application. To handle admission control, it is convenient to keep the cores ordered according to standard bin-packing heuristics: **best-fit (BF)**, considering cores ordered by decreasing utilisation; or **worst-fit (WF)**, considering cores ordered by increasing utilisation.

Service migration. Whenever an IoT node leaves the area covered by a domain D_a and enters the one covered by a domain D_b, it disconnects from D_a to connect to D_b. As a result, all services activated by the IoT node on D_a need to be *migrated* to D_b. Migration consists in (i) issuing a service deactivation request to D_a and (ii) issuing a service activation request to D_b. Note that the migration process can lead to improving or reducing the QoS of a service.

System cost and overall QoS. Powering on a new Cloud or Edge node has a monetary cost, e.g., charged by the Cloud/Edge providers. We consider the instantaneous cost of the nodes that are powered on at any time by means of two parameters: $\gamma^C(m)$, denoting the cost for each Cloud node with m cores, and $\gamma^E(m)$, denoting the cost for each Edge node with m cores. The overall system cost is hence given by

$$\gamma^{\text{tot}} = \sum_{N_q \in \mathcal{E}} \gamma^E(M_q) + \sum_{N_q \in \mathcal{C}} \gamma^C(M_q). \tag{3}$$

Cost is related to the overall QoS of the services activated by IoT nodes. Each service S_i is weighted by an *importance factor* $\beta_i \in [0, 1]$ and assigned a QoS value W^{Std}, if running with Standard QoS, or W^{Red}, if running with Reduced QoS. It represents the reduction in QoS when service S_i switches to reduced mode. Given a set $\mathcal{S}$ of services that are active at a certain time, the overall QoS at that time is defined as

$$W^{\text{tot}} = \sum_{S_i \in \mathcal{S}} \beta_i \times W_i, \tag{4}$$

where W_i is either equal to W^{Std} or W^{Red} depending on QoS of the service S_i.

Problem definition. The objective of this work is to design algorithms to handle dynamic scaling and service migration that aim to maximise the QoS per cost ratio $W^{\text{tot}}/\gamma^{\text{tot}}$.

4 Service orchestration algorithms

In this section, we introduce the service orchestrator, which is responsible for the allocation of incoming services to maximise QoS while minimising cost. It also manages service deallocation and continuously monitors and controls the QoS and cost efficiency of the system.

4.1 Arrival of a new service request

Our proposed algorithm processes incoming service requests immediately upon arrival, aiming to allocate resources efficiently while prioritising Edge computing (due to its low latency). When a request is received in an Edge domain D_k, the algorithm first attempts to allocate the service to D_k in standard mode. If the allocation fails, the algorithm considers alternative strategies to accommodate the request, which are: **(a)** using the reduced QoS mode for the incoming service request, allocating two interacting applications τ_i^E and τ_i^C, one at the Edge and one in the Cloud, respectively, or **(b)** reallocating another, previously allocated, service S_j, possibly involving reducing its QoS mode, to make room for τ_i. If neither strategy succeeds, the algorithm proceeds to **(c)** scale up resources by powering on additional Edge or Cloud nodes, thereby increasing the system cost. When no more Edge nodes are available, a Cloud node will be powered on. The following subsections provide details of each strategy. In the final subsection, we detail the algorithm and how these strategies are employed.

Allocation on the Edge The allocation produces the first attempts to allocate the application in standard mode considering the triplet $\tau_i = (Q_i, P_i, m_i)$. To this end, the algorithm uses the *residual per-core reservation bandwidth* as a metric, which, for a core $c_{q,x}$, is defined as $U_{q,x} = 1 - \sum_{r_j \in \mathcal{R}_{q,x}} \alpha_j$. First, the algorithm filters out all nodes $N_q \in D_k$ that do not satisfy Eq. (1), that is, those that are characterised by $U_{q,x} < \alpha_i$ for more than $M_q - m_i$ cores. The remaining

nodes are said to be *eligible* and denoted by $D'_k \subseteq D_k$. If $D'_k \neq \emptyset$, the application can be allocated in the standard mode in at least one of the nodes in the current edge domain. However, there may be multiple choices for allocating τ_i, both in terms of nodes and cores within the nodes. Different choices lead to a different system configuration, which may be more or less prone to host the allocation of future workloads.

Node selection strategies for Edge allocation. We first consider the order of nodes in D'_k by which to perform the selection of nodes. These strategies select nodes from D'_k and require C'_q, the subset of cores in Eq. (1) that satisfies the admission control. The subset C'_q is identified by means of a partitioning heuristic, either worst-fit or best-fit: as such, variants of the strategies reported next can be obtained by combining them with different partitioning schemes.

- **Max-Max Residual Bandwidth (MMRB).** $\max_{N_q \in D'_k} \max_{c_{q,x} \in C'_q} U_{q,x}$: Considers the maximum residual bandwidth among all cores C'_q in all eligible nodes. The node is selected with a rationale of privileging nodes with lightly loaded cores.
- **Min-Min Residual Bandwidth (mmRB).** $\min_{N_q \in D'_k} \min_{c_{q,x} \in C'_q} U_{q,x}$: Considers the minimum residual bandwidth among all cores C'_q in all eligible nodes. The node is selected with a rationale of filling the cores in a node.

Edge-cloud split allocation with available nodes It is possible that application τ_i in the standard mode cannot be allocated for a given service request. Strategies for allocating applications in reduced mode are hence also required. In this case, the split application τ_i^E and τ_i^C (Edge and Cloud parts, respectively) must pass the corresponding admission tests. The QoS per cost ratio of the system after allocating service S_i using split allocation is given by

$$\frac{W^{\text{tot}} + \beta_i \times W^{\text{Red}}}{\gamma^{\text{tot}}}, \tag{5}$$

where the total QoS of the system is increased by the QoS of S_i admitted in reduced mode, i.e., $\beta_i \times W^{Red}$, while the total cost remains the same, as no new nodes are powered on.

Service reallocation This approach allows to achieve higher values of the QoS-per-cost ratio $W^{\text{tot}}/\gamma^{\text{tot}}$ by reallocating an existing service to a better location, thus facilitating the allocation of a new service, or moving an existing service with a lower importance factor to a reduced QoS mode.

We propose two reallocation methods: **(i)** *intra-node reallocation*, and **(ii)** *edge-cloud split reallocation*. Intra-node reallocation does not involve a reduction in QoS of S_j but is characterised by an increasing reallocation delay.

(i) *Intra-node reallocation* is defined as follows. When a new service S_i needs to be admitted, another service S_j is selected to be reallocated. The algorithm first checks if there exists another set of cores within the same node that allows

both S_j and S_i to pass the admission test. This is done by first trying to allocate S_i and then S_j using the algorithms discussed previously.

(ii) *Edge-cloud split reallocation* is a second variant. Instead of reducing the QoS of S_j, it checks if it is possible to allocate both τ_j^E and τ_j^C in such a way that both pass the admission test. Reallocating a service S_j in reduced mode, to allow the new service S_i to be served at the Edge, leads to the following tentative value of the objective function:

$$\frac{W^{\text{tot}} + \beta_i \times W^{\text{Std}} - \beta_j \times \left(W^{\text{Std}} - W^{\text{Red}}\right)}{\gamma^{\text{tot}}}, \tag{6}$$

where the contribution from the new service S_i in standard mode is added ($\beta_i \times W^{\text{Std}}$), while the QoS of the pre-admitted service S_j is adjusted by subtracting the difference between its standard and reduced modes. The cost remains the same, as no nodes are powered on or off.

Next, we present several strategies to select the service S_j to be reallocated, among those in a given node $N_q \in D_k$.

- **Lowest Bandwidth service (LB).** This strategy reallocates the service characterised by the lowest per-core α_j, meaning the reallocated service has low bandwidth. It is helpful to perform the reallocation as smaller tasks can lead to fragmentation if not placed efficiently.
- **Lowest Importance service (LI).** This strategy considers the importance factor, i.e., β_j. This strategy is used for split reallocation only and reduces the W^{tot} reduction.

Powering on edge or cloud nodes Powering on a new Edge or Cloud node comes with a monetary cost and, therefore, a negative effect on the objective function $W^{\text{tot}}/\gamma^{\text{tot}}$. However, allocating a new service increases the value of the objective function. Therefore, whether it is more convenient to perform a reallocation, power on a new Edge node, or power on a new Cloud node is based on the variation in the objective function. If a new Edge node with M_q cores is powered on (i.e., rented) for serving S_i, then the objective function is:

$$\frac{W^{\text{tot}} + \beta_i \times W^{Std}}{\gamma^{\text{tot}} + \gamma^E(M_q)}, \tag{7}$$

where the system cost increases by the cost of the newly rented domain node $\gamma^E(M_q)$, while the total QoS increases by $\beta_i \times W^{Std}$ from allocating the new service S_i in standard mode.

If a new Cloud node with M_q cores is powered on to admit S_i with Edge-Cloud splitting, the objective function becomes:

$$\frac{W^{\text{tot}} + \beta_i \times W^{\text{Red}}}{\gamma^{\text{tot}} + \gamma^C(M_q)}, \tag{8}$$

where the system cost increases by the cost of the newly rented Cloud node $\gamma^C(M_q)$, while the total QoS increases by $\beta_i \times W^{\text{Red}}$ from allocating the new service S_i in reduced mode.

Algorithm 1 Pseudo-code of the service allocation algorithm.

```
1: function ALLOCATE(S_i, D_k, realloc)
2:     if (Allocate_Edge(S_i, D_k)) then
3:         return ALLOCATED
4:     if (realloc) then
5:         for each N_q ∈ D_k do
6:             if (Intra-node_reallocation(S_i, N_q)) then
7:                 return ALLOCATED
8:         if (Eq. (6)≥ Eq. (7) and Eq. (6)≥Eq. (5)) then
9:             for each N_q ∈ D_k do
10:                if (Edge-Cloud_split_realloc(S_i, D_k)) then
11:                    return ALLOCATED
12:    if (Eq. (5) ≥ Eq. (7) ) then
13:        if (Allocate_edge_cloud_split(S_i, D_k)) then
14:            return ALLOCATED
15:    else if (Eq. (7) ≥ Eq. (8) ) then
16:        if (Scale_edge(S_i, D_k)) then
17:            return ALLOCATED
18:    else if (Admission_test(S_i,D_k)) then
19:        if (Scale_cloud(S_i, D_k)) then
20:            return ALLOCATED
21:    return REJECTED
```

Service allocation algorithm The overall allocation procedure can be defined by combining all the strategies introduced above and is summarised in Algorithm 1. The algorithm starts with the standard mode allocation at Line 2 by finding a node that satisfies the admission test of Eq. (1). If this fails and `realloc` is `true`, i.e., reallocation is enabled, it proceeds with reallocation from Line 5. Otherwise, the algorithm proceeds with Line 12. For each node, an existing service S_j from the node is selected for intra-node reallocation according to the reallocation strategies in Section 4.1. If intra-node reallocation fails for one node, reallocation will be re-attempted with the next node. If intra-node reallocation is unsuccessful for all the nodes, the algorithm must decide between Edge-Cloud split reallocation, Edge-Cloud split allocation, or powering on a new Edge or Cloud node. To prioritise these options, the cost function is computed for each method (Equations (5)-(8)) and used to make a decision.

The algorithm attempts an Edge-Cloud split reallocation (Line 10), if the split reallocation objective function meets the condition (Line 8), i.e., split reallocation provides a higher QoS per cost ratio than split allocation and powering on new nodes. If it fails, the algorithm attempts an Edge-Cloud split allocation (Line 13) if its cost function meets the condition (Line 12). If Edge-Cloud split allocation fails, whether the admission test on the Edge or the Cloud has failed, the algorithm attempts to scale up the Edge domain by powering on a node (Line 16), provided that it guarantees a higher QoS per cost ratio than split allocation (Line 15). If it is successful, the service will be allocated in standard mode. If scaling up the Edge nodes fails, i.e., there are no domain nodes available

(or the condition at Line 15 is not satisfied), if the admission test of the task (i.e., τ_i^E) is already passed on the Edge, and the cost function has a value higher than Edge-Cloud split reallocation (Line 18), a Cloud node will be powered on. However, if the admission test of τ_i^E fails on the Edge, the service is rejected (Line 21), remains on the device, and continues local execution. The device may request allocation again later.

4.2 Exit of a service

When a service S_i exits an Edge domain D_k, the corresponding reservations are deallocated. Furthermore, an exit event allows for improving QoS by reallocating a service that was allocated in reduced mode to standard mode. We call this action *service upgrade*. To this end, we consider the **Highest QoS improvement (HQ)** strategy, which selects the service S_j with the highest QoS improvement (subject to the importance), i.e., with the highest $\beta_j \times (W^{\text{Std}} - W^{\text{Red}})$.

Services are considered only if allocated in reduced QoS mode. The selected service S_j is then tried to be allocated to the Edge with the algorithms of Section 4.1. Furthermore, once an Edge or Cloud node becomes empty due to a service exit, the corresponding physical machine can be turned off, reducing the overall cost γ^{tot}.

4.3 Interval-based service orchestration

An advanced approach is proposed to enhance allocation capabilities. Deallocation events are processed immediately, while allocation events are buffered and processed at the end of a fixed time interval of length Δ. At the end of each interval, the algorithm sorts the buffered allocation requests based on **Highest Importance Lowest Bandwidth-Core (HILBC)**, prioritising services with higher importance per required utilisation ratio. The sorted requests are then processed sequentially, each as described in Algorithm 1.

This approach improves efficiency by ensuring that resources are immediately freed when a service exits, making them available for new requests. Buffering allocation events allows the system to make more strategic placement decisions, reducing suboptimal allocations caused by frequent and unpredictable arrivals. This is particularly important in online settings as considered in this work, where decisions must be made without knowledge of future workloads, where optimal scheduling has been shown to be impossible in general [22].

Since allocation requests are delayed by up to some amount of time Δ, careful selection of the interval is critical to maintain responsiveness. A small Δ ensures near-immediate processing but limits batching benefits, leading to more frequent allocation decisions. In contrast, a large Δ improves efficiency by allowing better decisions, but introduces noticeable delays in service allocation. Thus, this interval must be adjusted based on the latency constraints of the services.

To further prevent excessive queueing delays, buffering can be paused if the number of pending allocation events reaches a pre-defined threshold. For this

Algorithm 2 Psuedo-code for Interval-Based Task Allocation

```
1: function INTERVAL_BASED_ALLOCATE(Δ, ths)
2:     T ← current_time + Δ
3:     while true do
4:         while current_time < T or length(buffered_events) < ths do
5:             if new event then
6:                 buffered_events.append(event)
7:         for event in buffered_events do
8:             Sort(buffered_events)
9:             ALLOCATE(event.Domain, event.Service, realloc)
10:        buffered_events.clear()
11:        if length(buffered_events) == ths then
12:            T ← current_time +Δ
13:        else
14:            T ← T + Δ
```

reason, the interval of length Δ is either periodically repeated or ends prematurely when the threshold is reached. The threshold is determined based on the latency tolerance of the application, ensuring request backlogs do not grow too large, leading to degraded performance.

Algorithm 2 illustrates the interval-based service allocation. The algorithm buffers incoming requests until either a time T or threshold `ths` is reached before scheduling them. It begins by initialising the buffering window by setting $T =$ `current_time` $+ \Delta$ (Line 2) and then enters an infinite loop to continuously handle incoming tasks (Line 3). The algorithm waits until one of the two stop conditions is reached `ths` (Line 4). While waiting, if a new event arrives, it is added to the buffer (Lines 5-6). Once exiting the waiting loop, buffered events are sorted by HILBC. All buffered events are processed sequentially (Line 7) by function ALLOCATE from Algorithm 1 (Line 9). Then, the buffer is cleared (Line 10). Also, if the number of buffered events reached threshold `ths`, the buffering window is reset to the current time plus Δ (Lines 11-12). Otherwise, the buffering window repeats periodically to end at time $T + \Delta$ in the next iteration (Line 14).

5 Evaluation

We conducted experiments to demonstrate and compare the effectiveness of the proposed algorithms in managing resource scaling and allocation. To achieve this, we base our experiments on a realistic use case following the system model of Section 3. According to our use case, under normal conditions, vehicles enter the system in a predictable manner, following planned routes and expected activation patterns. The system efficiently handles offloading and migration, and all algorithms may perform similarly. However, in dynamic conditions, such as sudden congestion during rush hours or unpredictable user mobility, the workload can fluctuate significantly. To evaluate algorithm performance in real-world

settings, we introduce abrupt changes in user behaviour, including random unpredicted vehicles requesting services.

5.1 Experimental setup

To perform these experiments, we must generate services, users, system configurations, and events based on user behaviour. We begin by generating a set of services that run on both Edge domain and Cloud nodes and are required by the users. Next, we establish an initial estimation of user behaviour to generate the system's initial configuration, ensuring sufficient resources based on the estimated user behaviours. Finally, we generate a series of events that are constrained by the user's behaviour, and within each series, we gradually introduce unexpected user behaviour.

Service generation. We generate 100 services where the period P_i of the services is randomly selected in a range from 100 ms to 1000 ms, with a step of 100 ms, with uniform distribution. To adjust the budget, the bandwidth (α_i) of each service is randomly and uniformly selected from 10 % to 80 %, and then each budget is computed accordingly ($Q_i = \alpha_i \times T_i$). The number of cores ($m_i$) for each service is randomly selected from 1 to 8, with uniform distribution.

The importance factor for each service is a unique number between 1 and 100, which are randomly assigned to the services. Furthermore, we need to generate the number of cores and bandwidth to allocate the service with Edge-Cloud splitting in reduced mode. In this case, as the application running on the Edge is mostly responsible for connectivity and offloading the request, for the Edge part we consider $m_i^E = 1$ and bandwidth $\alpha_i^E = \alpha_i/2$. For the Cloud part, we consider $\alpha_i^C = \alpha_i/2$ and $m_i^C = \max(m_i - 1, 1)$. For QoS parameters, we consider $W^{Std} = 1$ and $W^{Red} = 0.5$.

User generation. We consider a total of 300 types of users that can connect to the domains in our system. Each type of user passes a random number of domains, in a random order.

Multiple user instances, referred to in the remainder as just users, can be generated for each user type. Users may join and leave the system over time and remain connected to the system for a variable amount of time called *active time*. The active time of the user type is chosen from a range of 20 to 60 minutes. We assume that each user distributes their active time equally across the domains they connect to. For example, if a user connects to five domains and their total active time is 50 minutes, they will spend 10 minutes in each domain.

Each user type is characterised by a minimum and maximum inter-arrival time between consecutive connections to the system. The minimum inter-arrival time of each user is chosen randomly with a uniform distribution between 1.25 and 4 times its active time. The maximum inter-arrival time is randomly chosen between 2 to 4 times its minimum inter-arrival time. Each user can request between 15 and 20 randomly assigned services.

System parameter generation. We consider 10 Edge domains. We initialise each domain with 1 node. We assume that all nodes within a domain are homo-

Algorithm 3 Pseudo-code for Baseline Service Allocation

```
1: function BASELINE_ALLOCATE(S_i, D_k)
2:     if (Allocate_edge(S_i, D_k)) then
3:         return ALLOCATED
4:     if (Allocate_edge_cloud_split(S_i, D_k)) then
5:         return ALLOCATED
6:     return REJECTED
```

geneous, with 8 or 16 cores. We set the cost of a core in Edge and Cloud nodes to 1 and 3, respectively, reflecting the relative cost difference between the two types of nodes in a simplified manner.

Event generation according to early estimation of user characteristics. Each event includes the user's arrival time (constrained by the minimum and maximum inter-arrival time), their total active time, the order in which they traverse domains, time spent per domain, and the services they require. During event generation, each user is assigned an arrival time and a randomly determined order for the domains they will traverse. Other parameters, such as total active time and service requirements, are predefined user characteristics. A series of events includes a total of 1000 events per user.

Event generation with unpredictable user behaviour. We generate additional random events, where we add some random unpredictable users requesting for random services to observe algorithm performance when there are unexpected workloads and events. To maintain control over the level of extra workload, we use the first generated series of events from the previous step and gradually increase the total user active time and utilisation in the system. We compute the active time of each user times its requested utilisation and accumulate across all users and we call it *workload*.

In the first series of events, we increase the workload by 20% by generating more requests for random services in different domains. We call this additional workload *extra workload*. Then, the workload of the second series of events is increased again by 20% to generate the third set of events with 40% extra workload. This procedure continues to create some series of events with extra workloads ranging from 20% to 200% with a step of 20%.

Real-world mobility dataset To incorporate real-world workload characteristics, we used the T-Drive Trajectory dataset [62, 63], which contains GPS traces from 10,357 taxis in Beijing, with approximately 15 million location points. Since no dataset provides vehicle mobility along with domain assignments, service requests, and resource usage, we synthesised the missing components. We randomly selected 10 fixed locations as domain centres and generated a timeline of domain connections for each taxi. Service requests were also generated synthetically for each vehicle. To study the impact of workload density, we derived multiple versions of the dataset by randomly selecting subsets of users (e.g., 10%, 25%, 50%, 75%, and 100%) and repeated the experiments for each configuration.

5.2 Evaluation criteria

We measure QoS, cost, and QoS per cost over time after each allocation or deallocation event. The values of these metrics are computed by multiplying the metric values by the time interval during which the system maintained that value. The accumulated values across all intervals are then averaged over time, providing a representative long-term measure of the system performance. For each experimental configuration, we compute the average values of these metrics for different levels of extra workload in the system. The results are then visualised by plotting the average values per extra workload according to the configurations of each experiment.

Baseline algorithm for comparison To evaluate the effectiveness of our algorithm, we compare it with a baseline algorithm that follows a simpler allocation strategy. The baseline is a reference point to measure QoS, cost, QoS per cost, and runtime. The baseline service allocation in Algorithm 3 tries to allocate a service S_i to a domain D_k by following a structured approach that prioritises Edge nodes before the split allocation. The process starts by trying to allocate the service on the Edge node using `Allocate_edge` (Line 2). If successful, it returns `ALLOCATED`. If edge resources are insufficient, the algorithm considers partial task allocation across Edge and Cloud resources using `Allocate_edge_cloud_split` (Line 4). If none of these succeeds, the task is rejected (Line 6).

Additionally, we assume a different node selection strategy for the baseline, which selects the first node, where the admission test is satisfied. Our experiments demonstrate that best-fit and worst-fit perform similarly. Therefore, we use best-fit for the baseline in our subsequent comparisons.

We assume that two cloud nodes are reserved for the baseline in all experiments. In the following setups, we define a maximum scaling size, which represents the upper bound on the total number of cores available within a domain. For the baseline, we assume that all these cores are pre-reserved and remain fixed throughout execution.

5.3 Evaluation results

The mmRB and MMRB node selection methods, as well as the BF and WF partitioning heuristics, showed similar performance. Thus, we report results only for the BF and mmRB methods in the rest of the experiments.

Reallocation evaluation In this part, we evaluate the strategies used in the algorithm to evaluate their impact on the allocation efficiency and runtime of the algorithm compared to the baseline. Fig. 2 shows the performance for a node size of 8, where we assume a maximum threshold of 32 cores in each domain. We evaluated the impact of reallocation and dynamic scaling by comparing three configurations: (1) the *baseline*, which uses fixed reserved resources (32 cores per domain); (2) the *enhanced baseline (e-baseline)*, which enables dynamic scaling but disables reallocation; and (3) the full version of our algorithm, including dynamic scaling and reallocation.

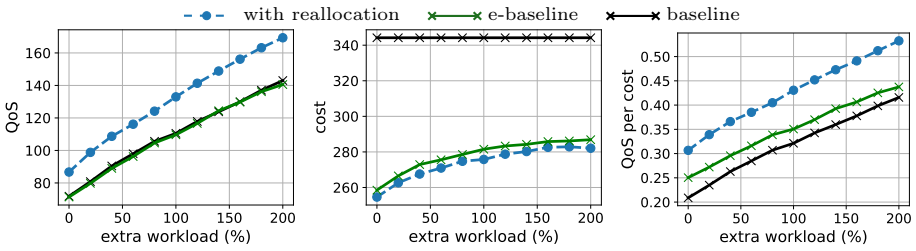


Fig. 2: Comparison among baseline, e-baseline, and our algorithm with reallocation

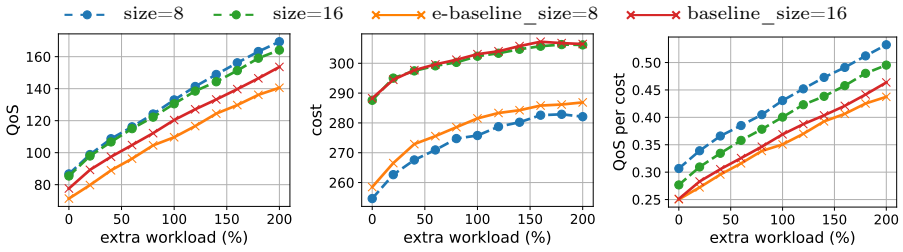
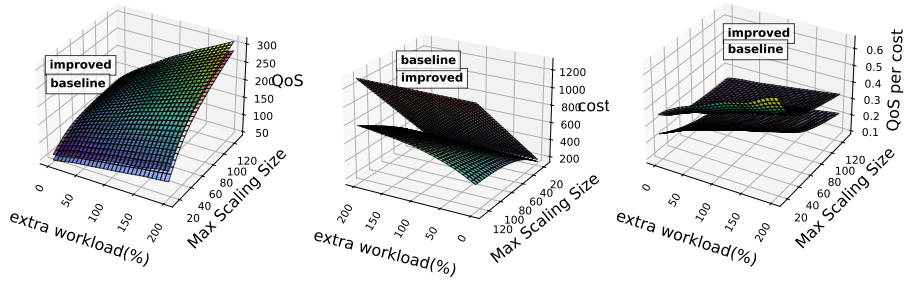


Fig. 3: Comparison of our algorithm against e-baseline for node sizes 8 and 16.

E-baseline. Fig. 2b indicates a more than 20% cost improvement over the baseline. That is, dynamic resource allocation prevents under-utilisation of the resources and reduces the cost by releasing resources when not used. On the other hand, Fig. 2a illustrates that the QoS of the system is similar to the baseline. Fig. 2c shows that QoS per cost can be improved by nearly 5% on average, and by 19% when there is no unpredicted workload in the system.

With reallocation. Fig. 2a illustrates that the QoS of the system improves slightly with reallocation strategies. This is because, by reallocation, more services can be allocated without powering on a new node. Fig. 2b indicates that reallocation strategies have the same cost. Fig. 2c shows that QoS per cost can be improved compared to the baseline by almost 41% at 0 extra workload, and 28% at 200% extra workload, i.e., 20% and 22% higher improvement compared to e-baseline, when there are 0 and 200% extra workload, respectively. This is due to improving QoS by admitting more services with high QoS.

Algorithm evaluation with immediate event handling The next experiment evaluates the performance of the proposed algorithm where reallocation is enabled. Each node has 8 cores, and events are processed once they arrive. The scaling threshold varies. Here, we only compare with the baseline to quantify the differences between different methods and evaluate their impact.



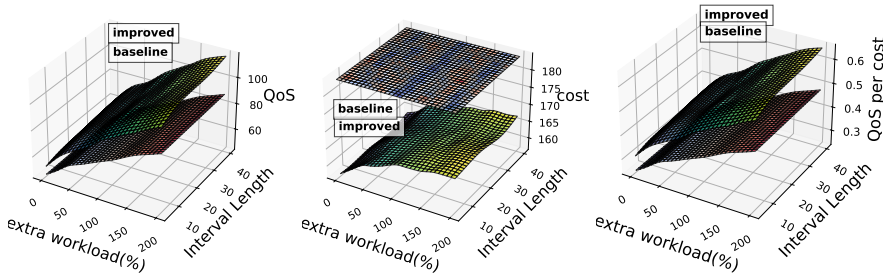
(a) (b) (c)

Fig. 4: Comparison of our algorithm with reallocation against baseline

Impact of node size on performance. Fig. 3 depicts the performance of the proposed algorithm compared to the e-baseline across different sizes of nodes. The maximum scaling threshold is 32 cores. This impact is also evaluated compared to the e-baseline. As shown in Fig. 3a, the QoS is reduced in e-baseline when the node size increases from 8 to 16, while this reduction is negligible when deploying reallocation in the algorithm. Fig. 3b depicts that with reallocation, the achieved cost can be lower in node size of 8, while the e-baseline and reallocation based algorithm perform similarly in node size of 16. Finally, Fig. 3c shows that with lower node sizes, we can achieve higher QoS per cost due to both lower cost and higher QoS.

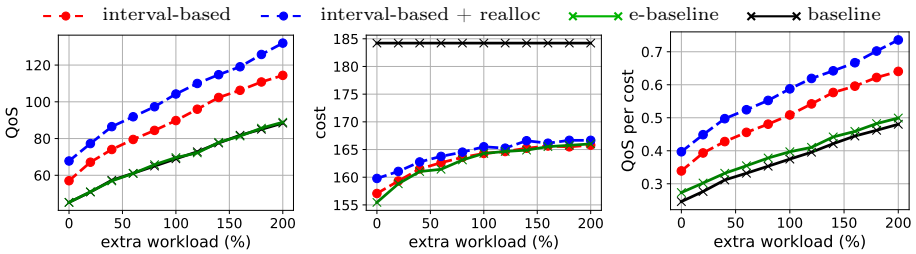
Impact of maximum scaling size on performance. Fig. 4 illustrates the performance of the algorithm with different maximum scaling thresholds compared to the baseline, where the number of available cores can be 16, 32, 64, and 128. As shown, QoS, cost, and QoS per cost increase by increasing the maximum scaling size in domains or the workload but, our algorithm performs better in all aspects.

Fig. 4a illustrates that the proposed algorithm achieves a higher QoS than the baseline. When the maximum scaling size increases from 16 to 128, the QoS improvement decreases from 10% to 0%,at 0 extra workload, and decreases from 33% to 8%, at 200% extra workload. This is because with higher core counts in the system, the baseline has abundant resources to allocate the service, hence the QoS increases. Fig. 4b shows that the algorithm reduces the cost. When the maximum scaling size increases from 16 to 128, the cost reduction is increased from 13% to 57%, at 0 extra workload, and increases from 9% to 35%, at 200% extra workload. The higher costs in the baseline at larger scaling sizes are due to the increase in reserved nodes from 2 to 16 per domain. In the proposed algorithm, the smaller cost reduction under higher workload randomness is due to the need for powering on more nodes, which raises overall cost. Finally, Fig. 4c shows that, when the maximum scaling size increases from 16 to 128, the QoS per cost improvement increases from 15% to 136% at 0, and increases from 10% to 54%, at 200% extra workload.



(a) (b) (c)

Fig. 5: Comparison of interval-based method with baseline for max. scaling size of 16.



(a) (b) (c)

Fig. 6: Comparison of interval-based method against baseline and e-baseline with interval length of 30 ms.

Runtime of the algorithm. The experiments were conducted on a machine with an Intel Core i7-9750H processor (6 cores, 12 threads, 2.60 GHz) and 16 GB of DDR4 RAM. The algorithms are configured in different maximum scaling sizes, where the number of available cores is 16, 32, 64, and 128. The worst observed runtime of the e-baseline reaches 70 μs, while the typical runtime remains below 20 μs. The worst observed runtime of the algorithm, including reallocation reaches 120 μs, while the typical runtime remains below 40 μs, which is acceptable for applications in our use case. Plots are omitted due to space limits.

The reallocation step has a time complexity of $\mathcal{O}(m \cdot n)$, where n is the number of nodes and m is the number of pre-allocated services per node. Each strategy evaluates all pre-allocated services on a node to select a reallocation candidate, resulting in $\mathcal{O}(m)$ complexity per node.

Impact of Cloud monetary cost and $W^{\mathbf{Std}}/W^{\mathbf{Red}}$. As the Cloud-to-domain cost ratio increases, the cost rises for the baseline and our algorithm. As the $W^{\mathrm{Std}}/W^{\mathrm{Red}}$ ratio increases, the QoS improvement of our algorithm increases, while the cost remains the same. Plots are omitted due to space limits.

Algorithm evaluation with interval-based event handling The goal of this experiment is to evaluate the performance of our algorithm when reallocation is disabled and event buffering is enabled. Each node has 8 cores. In these experiments, we assume no threshold on the number of buffered events, as each is typically handled in under 20μs, making the buffering delay negligible.

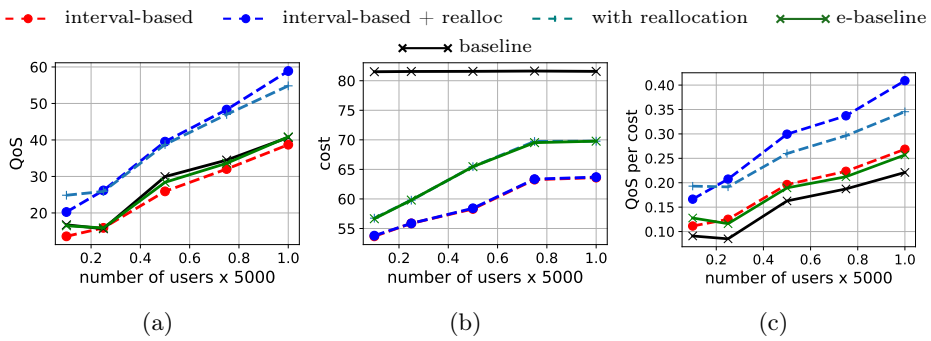
Impact of interval length on performance. Fig. 5 illustrates the performance of the algorithm in different interval lengths compared to the baseline, where the maximum scaling size of the nodes is 16 and the interval length is 5, 10, 20, 30, and 40 ms. Fig. 5a illustrates that when the interval length is increased from 5 ms to 40 ms, the achieved QoS improvement of the proposed algorithm increases from 13% to 36% at 0 extra workload and from 22% to 33% at 200% extra workload. The increase in the QoS improvement with the increasing interval length is because a longer interval allows more events to be deallocated and more arrivals to be buffered, enabling better allocation decisions. Similarly, Fig. 5b shows that the algorithm decreases the cost by 9% for 200% extra workload at any interval length. Finally, Fig. 5c shows that with interval length increasing from 5 and 40 ms, the algorithm achieved QoS per cost increases from 24% to 39%, at 0 extra workload and increases from 29% to 40% at 200% extra workload.

Impact of interval-based allocation. Fig. 6 illustrates the performance of the algorithm with and without reallocation with an interval length of 30 ms and a maximum scaling size of 16, compared to the baseline, e-baseline.

Fig. 6a illustrates the system QoS, where the interval-based algorithm achieves 27% and 30% higher QoS than the e-baseline at 0 and 200% extra workload, respectively. This shows that as the extra workload increases, the interval-based algorithm outperforms both the baseline and e-baseline. This improvement is likely because more events arrive within each interval, allowing for more effective QoS-driven allocation decisions. Fig. 6b shows that the algorithm decreases the cost slightly lower than the e-baseline, while maintaining QoS. Finally, Fig. 6c shows that the algorithm improves QoS per cost by 25% and 30% compared to e-baseline at 0 and 200%, respectively.

Fig. 6a shows that interval-based algorithm with reallocation achieves 20% and 15% higher QoS than the algorithm with reallocation at 0 and 200% extra workload, respectively, i.e., 34% and 53% higher QoS than the baseline at 0 and 200% extra workload, respectively. As shown, while both the reallocation and interval-based algorithms individually improve performance, their combined effect is not cumulative. This is likely because the interval-based algorithm may introduce sub-optimal allocation decisions, which limit the effectiveness of subsequent reallocation. Fig. 6b shows that the cost is similar to the algorithm with reallocation. Finally, Fig. 6c shows that the algorithm improves QoS per cost by 17% and 14% compared to the interval-based algorithm at 0 and 200%, respectively, i.e., 60% and 50% compared to baseline at 0 and 200%, respectively.

Evaluation on the T-Drive Trajectory dataset Fig. 7 shows the performance of the algorithm with and without reallocation with an interval length



(a) (b) (c)

Fig. 7: Comparison of algorithms using traces of 5000 taxi from Trajectory dataset.

of 30 ms and a maximum scaling size of 16, compared to the baseline and e-baseline, using the T-Drive Trajectory dataset. As shown, the interval-based algorithm with reallocation achieves significantly better performance: It delivers up to 50% higher QoS when all users are present, requires approximately 10% lower cost, and consequently provides a 80% higher QoS per cost under the same conditions, compared to the e-baseline. This improvement results from the QoS gains achieved by the reallocation-only algorithm and the cost reductions provided by the interval-based algorithm.

6 Conclusion and future work

This paper presents real-time resource allocation and offloading algorithms for mobile IoT applications in Edge-Cloud environments. By integrating service allocation and dynamic scaling strategies that leverage SCHED_DEADLINE reservations, our system achieves predictable task execution while adapting to dynamic workloads. Our experimental evaluation shows that the proposed algorithms outperform a baseline approach. In particular, reallocation and interval-based allocation strategies show clear advantages in adapting to unpredictable service demands. Additionally, the runtime analysis confirms the feasibility of our methods to be used online, showing typical runtimes of a few tens of microseconds. In future work, we aim to implement our orchestrator on Kubernetes to support container-based deployment (building upon [47]) and use mobility patterns for proactive container warm-up, mitigating cold-start delays.

Acknowledgments

This work has been supported by the EU ECSEL project TRANSACT grant agreement No. 101007260, the European Union's Horizon Europe Framework Programme project NANCY under the grant agreement No. 101096456, and the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

References

1. Abeni, L., Balsini, A., Cucinotta, T.: Container-based real-time scheduling in the Linux kernel. ACM SIGBED Review **16**(3), 33–38 (2019)
2. Abeni, L., Buttazzo, G.: Integrating multimedia applications in hard real-time systems. In: Real-Time Systems Symposium. pp. 4–13. IEEE (1998)
3. Aburukba, R.O., Landolsi, T., Omer, D.: A heuristic scheduling approach for fog-cloud computing environment with stationary IoT devices. Journal of Network and Computer Applications **180**, 102994 (2021)
4. Adhikary, T., Das, A.K., Razzaque, M.A., Alrubaian, M., Hassan, M.M., Alamri, A.: Quality of service aware cloud resource provisioning for social multimedia services and applications. Multimedia Tools and Applications **76**, 14485–14509 (6 2017). https://doi.org/10.1007/S11042-016-3852-X/FIGURES/7, `https://link.springer.com/article/10.1007/s11042-016-3852-x`
5. Agbaje, P., Nwafor, E., Olufowobi, H.: Deep reinforcement learning for energy-efficient task offloading in cooperative vehicular edge networks. In: 2023 IEEE 21st International Conference on Industrial Informatics (INDIN). pp. 1–8 (2023). https://doi.org/10.1109/INDIN51400.2023.10218113
6. Ammavasai, S.: Dynamic task scheduling in edge cloud systems using deep recurrent neural networks and environment learning approaches. Journal of Intelligent & Fuzzy Systems (Preprint), 1–16 (2024)
7. Ashjaei, M., Mubeen, S., Behnam, M., Almeida, L., Nolte, T.: End-to-end resource reservations in distributed embedded systems. In: International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). pp. 1–11. IEEE (2016)
8. Azizi, S., Shojafar, M., Abawajy, J., Buyya, R.: Deadline-aware and energy-efficient IoT task scheduling in fog computing systems: A semi-greedy approach. Journal of network and computer applications **201**, 103333 (2022)
9. Bini, E., Bertogna, M., Baruah, S.: Virtual multiprocessor platforms: Specification and use. In: Real-Time Systems Symposium. pp. 437–446 (2009)
10. Buttazzo, G.C.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition. Springer (2011)
11. Cai, J., Liu, W., Huang, Z., Yu, F.R.: Task decomposition and hierarchical scheduling for collaborative cloud-edge-end computing. IEEE Transactions on Services Computing (2024)
12. Casini, D., Biondi, A., Buttazzo, G.: Semi-partitioned scheduling of dynamic real-time workload: A practical approach based on analysis-driven load balancing. In: Euromicro Conference on Real-Time Systems (ECRTS). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (Jun 2017)
13. Casini, D., Biondi, A., Buttazzo, G.: Task splitting and load balancing of dynamic real-time workloads for semi-partitioned EDF. IEEE Transactions on Computers **70**(12), 2168–2181 (2020)
14. Casini, D., Pazzaglia, P., Becker, M.: Managing real-time constraints through monitoring and analysis-driven edge orchestration. Journal of Systems Architecture (2025)
15. Chen, B., Huang, W., Wang, L., Zhang, Y.: Joint optimization of service caching and task scheduling in energy-efficient mobile edge computing. In: 2024 4th International Conference on Intelligent Technology and Embedded Systems (ICITES). pp. 207–213 (2024). https://doi.org/10.1109/ICITES62688.2024.10777462

16. Cho, C., Shin, S., Jeon, H., Yoon, S.: QoS-aware workload distribution in hierarchical edge clouds: A reinforcement learning approach. IEEE Access **8**, 193297–193313 (2020)
17. Cohen, I., Chiasserini, C.F., Giaccone, P., Scalosub, G.: Dynamic service provisioning in the edge-cloud continuum with bounded resources. IEEE/ACM Transactions on Networking **31**(6), 3096–3111 (2023). https://doi.org/10.1109/TNET.2023.3271674
18. Deng, X., Li, J., Liu, E., Zhang, H.: Task allocation algorithm and optimization model on edge collaboration. Journal of Systems Architecture **110**, 101778 (2020)
19. Easwaran, A., Shin, I., Lee, I.: Optimal virtual cluster-based multiprocessor scheduling. Real-Time Systems **43**(1), 25–59 (2009)
20. Fan, Y., Ge, J., Zhang, S., Wu, J., Luo, B.: Decentralized scheduling for concurrent tasks in mobile edge computing via deep reinforcement learning. IEEE Transactions on Mobile Computing **23**(4), 2765–2779 (2024). https://doi.org/10.1109/TMC.2023.3266226
21. Fang, J., Ma, A.: IoT application modules placement and dynamic task processing in edge-cloud computing. IEEE Internet of Things Journal **8**(16), 12771–12781 (2020)
22. Fisher, N., Goossens, J., Baruah, S.: Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. Real-Time Systems **45**, 26–71 (2010)
23. Ghahari-Bidgoli, M., Ghobaei-Arani, M., Sharif, A.: An efficient task offloading and auto-scaling approach for IoT applications in edge computing environment. Computing **107**(5), 1–44 (2025)
24. Ghosh, S., Rajkumar, R., Hansen, J., Lehoczky, J.: Scalable resource allocation for multi-processor QoS optimization. In: International Conference on Distributed Computing Systems, 2003. Proceedings. pp. 174–183 (2003)
25. Ghosh, S., Hansen, J., Rajkumar, R., Lehoczky, J.: Integrated resource management and scheduling with multi-resource constraints. In: Real-Time Systems Symposium. pp. 12–22. IEEE (2004)
26. Guo, S., Liu, J., Yang, Y., Xiao, B., Li, Z.: Energy-efficient dynamic computation offloading and cooperative task scheduling in mobile cloud computing. IEEE Transactions on Mobile Computing **18**(2), 319–333 (2018)
27. Hoseiny, F., Azizi, S., Shojafar, M., Ahmadiazar, F., Tafazolli, R.: PGA: A priority-aware genetic algorithm for task scheduling in heterogeneous fog-cloud computing. In: IEEE INFOCOM 2021-IEEE conference on computer communications workshops (INFOCOM WKSHPS). pp. 1–6. IEEE (2021)
28. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al.: Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383 (2019)
29. Khaledian, N., Razzaghzadeh, S., Haghbayan, Z., Völp, M.: Hybrid markov chain-based dynamic scheduling to improve load balancing performance in fog-cloud environment. Sustainable Computing: Informatics and Systems **45**, 101077 (2025)
30. Lakhan, A., Mohammed, M.A., Kadry, S., AlQahtani, S.A., Maashi, M.S., Abdulkareem, K.H.: Federated learning-aware multi-objective modeling and blockchain-enable system for IIoT applications. Computers and Electrical Engineering **100**, 107839 (2022)
31. Lee, C., Lehoezky, J., Rajkumar, R., Siewiorek, D.: On quality of service optimization with discrete QoS options. In: Real-Time Technology and Applications Symposium. pp. 276–286 (1999)

32. Lelli, J., Scordino, C., Abeni, L., Faggioli, D.: Deadline scheduling in the Linux kernel. Software: Practice and Experience **46**(6), 821–839 (2016)
33. Li, G., Lin, Q., Wu, J., Zhang, Y., Yan, J.: Dynamic computation offloading based on graph partitioning in mobile edge computing. IEEE access **7**, 185131–185139 (2019)
34. Li, Y., Xuan Phan, L.T., Loo, B.T.: Network functions virtualization with soft real-time guarantees. In: INFOCOM- International Conference on Computer Communications. pp. 1–9 (2016)
35. Li, Y., Dai, W., Gan, X., Jin, H., Fu, L., Ma, H., Wang, X.: Cooperative service placement and scheduling in edge clouds: A deadline-driven approach. IEEE Transactions on Mobile Computing **21**(10), 3519–3535 (2022). https://doi.org/10.1109/TMC.2021.3061602
36. Ma, X., Zhou, A., Zhang, S., Li, Q., Liu, A.X., Wang, S.: Dynamic task scheduling in cloud-assisted mobile edge computing. IEEE Transactions on Mobile Computing **22**(4), 2116–2130 (2021)
37. Ma, Y., Liang, W., Li, J., Jia, X., Guo, S.: Mobility-aware and delay-sensitive service provisioning in mobile edge-cloud networks. IEEE Transactions on Mobile Computing **21**(1), 196–210 (2020)
38. Materwala, H., Ismail, L., Hassanein, H.S.: QoS-SLA-aware adaptive genetic algorithm for multi-request offloading in integrated edge-cloud computing in Internet of vehicles. Vehicular Communications **43** (2023)
39. Meng, J., Tan, H., Xu, C., Cao, W., Liu, L., Li, B.: Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In: IEEE INFOCOM 2019-IEEE Conference on Computer Communications. pp. 2287–2295. IEEE (2019)
40. Mok, A., Feng, X., Chen, D.: Resource partition for real-time systems. In: Real-Time Technology and Applications Symposium. pp. 75–84 (2001)
41. Nikoui, T.S., Balador, A., Rahmani, A.M., Bakhshi, Z.: Cost-aware task scheduling in fog-cloud environment. In: 2020 CSI/CPSSI International Symposium on Real-Time and Embedded Systems and Technologies (RTEST). pp. 1–8. IEEE (2020)
42. Pandeeswari, S.T., Padmavathi, S., Kabilan, D., Krishna, B.A.: Resource-aware fog service placement using deferred acceptance in edge computing. Journal of Engineering Research (2024)
43. Rajkumar, R., Lee, C., Lehoczky, J., Siewiorek, D.: A resource allocation model for QoS management. In: Real-Time Systems Symposium. pp. 298–307 (1997)
44. Rajkumar, R., Lee, C., Lehoczky, J., Siewiorek, D.: Practical solutions for QoS-based resource allocation problems. In: Real-Time Systems Symposium. pp. 296–306 (1998)
45. Rosmaninho, R., Raposo, D., Rito, P., Sargento, S.: Edge-cloud continuum orchestration of critical services: A smart-city approach. IEEE Transactions on Services Computing pp. 1–15 (2025). https://doi.org/10.1109/TSC.2025.3568251
46. Russo, G.R., Ferrarelli, D., Pasquali, D., Cardellini, V., Presti, F.L.: QoS-aware offloading policies for serverless functions in the Cloud-to-Edge continuum. Future Generation Computer Systems **156**, 1–15 (2024)
47. Samimi, N., Abeni, L., Casini, D., Marinoni, M., Basten, T., Nasri, M., Geilen, M., Biondi, A.: Enabling containerisation of distributed applications with real-time constraints. In: Mancuso, R. (ed.) 37th Euromicro Conference on Real-Time Systems (ECRTS 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 335, pp. 3:1–3:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025). https://doi.org/10.4230/LIPIcs.ECRTS.2025.3, `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2025.3`

48. Samimi, N., Nasri, M., Basten, T., Geilen, M.: Guaranteeing weakly-hard timing constraints of real-time server-based systems. In: 2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA). pp. 1–8 (2024). https://doi.org/10.1109/ETFA61755.2024.10711035
49. Samimi, N., Nasri, M., Basten, T., Geilen, M.: Online admission test for real-time tasks with arrival curves for server platforms. In: Proceedings of the 32nd International Conference on Real-Time Networks and Systems. pp. 266–277 (2024). https://doi.org/10.1145/3696355.3696359
50. Schantz, R.E., Loyall, J.P., Rodrigues, C., Schmidt, D.C., Krishnamurthy, Y., Pyarali, I.: Flexible and adaptive QoS control for distributed real-time and embedded middleware. In: Middleware 2003: ACM/IFIP/USENIX International Middleware Conference. pp. 374–393. Springer (2003)
51. Shin, I., Lee, I.: Periodic resource model for compositional real-time guarantees. In: Real-Time Systems Symposium (2003)
52. Sohani, M., Jain, S.: A predictive priority-based dynamic resource provisioning scheme with load balancing in heterogeneous cloud computing. IEEE access **9**, 62653–62664 (2021)
53. Stavrinides, G.L., Karatza, H.D.: A hybrid approach to scheduling real-time IoT workflows in fog and cloud environments. Multimedia Tools and Applications **78**, 24639–24655 (2019)
54. Struhár, V., Craciunas, S.S., Ashjaei, M., Behnam, M., Papadopoulos, A.V.: Hierarchical resource orchestration framework for real-time containers. ACM Transactions on Embedded Computing Systems (2023)
55. Swain, C., Sahoo, M.N., Satpathy, A., Muhammad, K., Bakshi, S., Rodrigues, J.J.: A-DAFTO: Artificial cap deferred acceptance-based fair task offloading in complex IoT-fog networks. IEEE Transactions on Consumer Electronics **69**(4), 914–926 (2023)
56. Tran, T.X., Chan, K., Pompili, D.: Costa: Cost-aware service caching and task offloading assignment in mobile-edge computing. In: sensing, communication, and networking (SECON). pp. 1–9. IEEE (2019)
57. Wang, S., Li, Y., Pang, S., Lu, Q., Wang, S., Zhao, J.: A task scheduling strategy in edge-cloud collaborative scenario based on deadline. Scientific Programming **2020**(1), 3967847 (2020)
58. Xu, Y., Chen, L., Lu, Z., Du, X., Wu, J., Hung, P.C.K.: An adaptive mechanism for dynamically collaborative computing power and task scheduling in edge environment. IEEE Internet of Things Journal **10**(4), 3118–3129 (2023). https://doi.org/10.1109/JIOT.2021.3119181
59. Xu, Z., Liang, W., Jia, M., Huang, M., Mao, G.: Task offloading with network function requirements in a mobile edge-cloud network. IEEE Transactions on Mobile Computing **18**(11), 2672–2685 (2018)
60. Yang, L., Liu, B., Cao, J., Sahni, Y., Wang, Z.: Joint computation partitioning and resource allocation for latency sensitive applications in mobile edge clouds. IEEE Transactions on Services Computing **14**(5), 1439–1452 (2019)
61. Yao, J., Ansari, N.: QoS-aware fog resource provisioning and mobile device power control in IoT networks. IEEE Transactions on Network and Service Management **16**(1), 167–175 (2018)
62. Yuan, J., Zheng, Y., Xie, X., Sun, G.: Driving with knowledge from the physical world. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '11). ACM, New York, NY, USA (2011)

63. Yuan, J., Zheng, Y., Zhang, C., Xie, W., Xie, X., Sun, G., Huang, Y.: T-drive: Driving directions based on taxi trajectories. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '10). pp. 99–108. ACM, New York, NY, USA (2010)
64. Zhang, Q., Gui, L., Hou, F., Chen, J., Zhu, S., Tian, F.: Dynamic task offloading and resource allocation for mobile-edge computing in dense cloud ran. IEEE Internet of Things Journal **7**(4), 3282–3299 (2020)
65. Zhang, Y., Chen, X., Chen, Y., Li, Z., Huang, J.: Cost efficient scheduling for delay-sensitive tasks in edge computing system. In: 2018 IEEE International Conference on Services Computing (SCC). pp. 73–80 (2018). https://doi.org/10.1109/SCC.2018.00017
66. Zhang, Y., Tang, B., Luo, J., Zhang, J.: Deadline-aware dynamic task scheduling in edge–cloud collaborative computing. Electronics **11**(15), 2464 (2022)
67. Zhu, J., Zhao, L., Zhou, J., Ye, W., Xiao, F.: Service degradation-tolerated online user allocation in edge computing. IEEE Transactions on Services Computing **17**(4), 1806–1819 (2024). https://doi.org/10.1109/TSC.2024.3353290