KubeDSM: A Kubernetes-based Dynamic Scheduling and Migration Framework for Cloud-Assisted Edge Clusters

Amirhossein Pashaeehir, Sina Shariati, Shayan Shafaghi, Manni Moghimi, Mahmoud Momtazpour

[a]*Computer Eng. Dep. of Amirkabir University of Technology, Hafez Ave., Tehran, , Tehran, Iran*

Abstract

Edge computing has become critical for enabling latency-sensitive applications, especially when paired with cloud resources to form cloud-assisted edge clusters. However, efficient resource management remains challenging due to edge nodes' limited capacity and unreliable connectivity. This paper introduces KubeDSM, a Kubernetes-based dynamic scheduling and migration framework tailored for cloud-assisted edge environments. KubeDSM addresses the challenges of resource fragmentation, dynamic scheduling, and live migration while ensuring Quality of Service (QoS) for latency-sensitive applications. Unlike Kubernetes' default scheduler, KubeDSM adopts batch scheduling to minimize resource fragmentation and incorporates a live migration mechanism to optimize edge resource utilization. Specifically, KubeDSM facilitates three key operations: intra-edge migration to reduce fragmentation, edge-to-cloud migration during resource shortages, and cloud-to-edge migration when resources become available, thereby lowering latency. Experimental evaluations on a simulated edge-cloud Kubernetes cluster show that KubeDSM significantly reduces latency compared to baseline schedulers while achieving near-optimal resource utilization. The proposed framework advances the orchestration of containerized applications in cloud-assisted edge clusters, providing a robust solution to real-time resource scheduling challenges.

Keywords: dynamic scheduling, live migration, container orchestration.

1. Introduction

In recent years, we have witnessed an increasing demand for edge computing environments. Public edge platforms such as Google Distributed Cloud Edge [1], AWS Local Zones [2] and Azure Public MEC [3] has been introduced to enable latency-sensitive workloads to run on edge server clusters and optimize users' quality of experience. Furthermore, the lower latency and higher bandwidth of the 5G technology enables edge-based deployment of many latency-sensitive applications, such as online gaming, remote surgery, real-time video analytics, and edge intelligence. However, there is still a gap for ultra-low-latency applications such as virtual and augmented reality (VR/AR) and autonomous driving (AD)[4]. For example, as characterized by Mohan et al. [4], AR/VR applications require sub-20ms latency, out of which 13ms should be reserved for display technology, and only around 7ms remains to perform all communications, processings, modellings, and output formation tasks. Next-generation telecommunication technologies like 6G might partially mitigate this challenge. However, efficient resource management techniques still play a significant role in guaranteeing the quality of service (QoS) under such a tight latency constraint.

Edge computing systems often comprise heterogeneous nodes with limited computing and storage resources connected through unreliable links. These limitations make resource management, scheduling, and migration particularly challenging in edge-based server clusters. Moreover, due to the stringent latency requirements, resource management techniques must be fast, adding to the complexity of their design. Given the limited computing capacity of edge nodes, they may only have enough resources to handle some computational tasks locally. Consequently, to enhance the scalability of edge platforms, several studies have implemented cloud-assisted edge computing, which dynamically allocates resources from the cloud to address the resource shortage at edge sites ([5], [6], [7]).

Furthermore, due to the cost associated with offloading computation from the edge to the cloud, combined with the limited resources of edge servers, optimizing resource utilization on edge nodes is crucial. One way to achieve this is through reducing resource fragmentation by implementing effective migration strategies. By enhancing the utilization of edge resources and ensuring that resources are utilized to their fullest potential, we can significantly reduce the need for computational offloading to the cloud during periods of resource shortage. This not only decreases operational costs but also lowers the average latency of applications running on cloud-assisted edge platforms, thereby improving overall system performance and user experience.

This paper introduces a fragmentation-aware, QoS-aware dynamic scheduling and migration framework for cloud-assisted edge server clusters. The framework has been designed and implemented as a live component of Kubernetes and can be directly used in production container orchestration systems such as Kubernetes, K3s and KubeEdge. To the best of our knowledge, this is the first attempt to incorporate a fragmentation-aware scheduling and migration framework for cloud-assisted edge computing into Kubernetes. The main contributions of this work are as follows:

1. In contrast to Kubernetes' default scheduler (Kube-scheduler), where pods are scheduled one at a time, the proposed framework handles batch binding of pods to nodes of a Kubernetes cluster to efficiently reduce resource fragmentation
2. The proposed framework adds a live migration component to Kubernetes to reduce resource fragmentation and efficiently utilize edge resources. The added live component actively monitors the remaining resources on servers of a local edge cluster and tries to:
   (a) Migrates pods between nodes on the edge cluster to reduce resource fragmentation
   (b) Migrates pods from edge to cloud nodes in the event of resource shortage
   (c) When enough resources become available, migrate pods from the cloud back to the edge nodes to reduce the applications' average latency. This behaviour, in turn, helps tenant services to maximize their use of available edge resources.

Experimental results showed that...

The rest of the paper is organized as follows. In Section 2, we review the related work. Section 3 defines the system model, detailing the key components and interactions within our proposed architecture. Section 4 outlines the problem formulation, where we formulate the problem as a multi-objective Mixed-Integer Linear Programming (MILP) model. In Section 5, we present the proposed approach, followed by the evaluation and analysis of the results of our experiments in Section 6. Finally, Section 7 summarizes the key findings, discussing their implications and suggesting potential directions for future work.

2. Related Work

The field of edge computing has evolved significantly in recent years, driven by the need for efficient resource management and low-latency services. In this section, we focus on two key research areas: workload migration and adapting Kubernetes for edge clusters. The first subsection discusses strategies and techniques for resource management in edge clusters, while the second subsection examines efforts to tailor Kubernetes, the leading container orchestration platform, to the unique requirements of edge computing environments.

2.1. Resource Management in Edge Clusters

To optimize resource usage on edge clusters and hence the QoS, several scheduling and migration techniques have been proposed in the literature. The authors in [5] addressed scalability in edge platforms with the Cloud Assisted Mobile Edge (CAME) framework by outsourcing mobile requests to cloud instances, which accommodates dynamic requests and various quality of service requirements. They proposed Optimal Resource Provisioning (ORP) algorithms to optimize edge computation capacity and dynamically adjust cloud tenancy. Evaluations showed these algorithms outperform local-first and cloud-first benchmarks in flexibility and cost-efficiency. Li et al. [7] studied a cloud-assisted edge computing system (CAECS) to address the challenges of edge computing, such as handling randomly varying workloads. They proposed a replica placement strategy to meet diverse user demands and reduce response time, and a data migration strategy to ensure data reliability. Additionally, a heterogeneity-aware elastic provisioning strategy was introduced to manage cloud instance rentals. The authors in [8] introduced three migration algorithms and developed an algorithm-selector mechanism that chooses the most appropriate algorithm based on the characteristics of the container. This approach enabled them to achieve live migration durations of less than one second in their tests. Moreover, [9, 10] approached migration by modeling it as partially observable Markov decision processes and employing multi-objective, multi-constraint optimization techniques. This enabled them to achieve improvements in average delay and reductions in power consumption. Similarly, in [11], Wang et al. proposed an actual cost predictive model. By assuming pre-determined upper bounds, they were able to make sub-optimal placement decisions based on their predicted costs in their simulations.

Additionally, several studies have leveraged service migration to enhance service quality in mobile edge clusters. For example, Chi et al. [12] proposed a method for live migration of services in mobile edge clusters using a multi-criteria decision-making algorithm based on TOPSIS. This method optimizes migration processes and alleviates traffic congestion. Furthermore, Rong et al. [13] introduced three approaches for migrating video analysis applications within edge clusters, ensuring minimal noticeable impact on service quality for users during the transition.

2.2. Adapting Kubernetes to Edge clusters

Numerous articles address the challenges related to Kubernetes adaptation within edge infrastructure. For instance, Ghafouri et al. [14] proposed a reinforcement learning-based scheduler aimed at reducing energy consumption. This study demonstrated that learning-based solutions could effectively replace traditional algorithmic approaches through innovative methods. Similarly, Lai et al. [15] introduced a solution that combines a scheduler and a scorer. The scheduler uses the scorer in a filtering stage to select nodes with minimum network latency and the most available resources to serve the pods, thereby achieving a balance between processing delays and network latency among pods. Moreover, Zhang et al. [16] analyzed the user's probability function of sojourn time to characterize user mobility intensity and service deployment overhead models. The resulting scheduler was able to reduce user-perceived latency, constrain service migration, and optimize user experience quality in software-defined (SDN) networks.

Efficient utilization of edge resources reduces the necessity of offloading computations from edge servers to the cloud during resource shortages, thus lowering the average latency of applications running on the edge platform with cloud cluster assistance. Although several studies, like those mentioned earlier, have addressed migration and resource allocation in edge clusters, they have not focused on reducing latency by optimizing the utilization of edge nodes through simultaneous service migration and dynamic scheduling in Kubernetes-managed clusters. To the best of our knowledge, this is the first work that considers both migration and scheduling simultaneously on cloud-assisted edge clusters. Our approach aims to achieve efficient resource allocation in Kubernetes-managed environments while accommodating the cluster's dynamic nature through pod migration. Table 5 summarizes the related works.

3. System model

Edge clusters are typically composed of resource-constrained, heterogeneous, failure-prone nodes placed at the network's edge. These nodes enable the processing of user requests with reduced latency. To mitigate these limitations, edge clusters are typically paired with cloud clusters that feature nodes with greater resources and fault-tolerant architectures. This forms an extensive system of cloud and edge nodes, commonly referred to as cloud-assisted edge clusters. This configuration enhances the overall capability and reliability of the network, ensuring efficient processing and improved service delivery. Figure 1 illustrates the cloud-assisted edge cluster with its components.

Kubernetes, the de facto standard for container orchestration, efficiently manages application deployments and their life cycles within a cluster. As depicted in Figure 1, several essential components are crucial for Kubernetes to perform this task. These components typically include a scheduler for resource allocation, KubeProxy for network management, a monitoring system for performance tracking, and horizontal pod autoscaling (HPA) for dynamic scaling of application instances.

3.1. Scheduler

The scheduler is a vital component of the Kubernetes control plane, responsible for assigning newly created pods to nodes. It operates through a filtering and scoring phase to determine the optimal node for pod binding. The default scheduler, known as kube-scheduler, is installed alongside other control plane components in Kubernetes [17]. By default, the kube-scheduler aims to evenly distribute pods across nodes to enhance availability and reliability [18].

3.2. KubeProxy

KubeProxy is a key Kubernetes component that runs on each node and manages network rules to ensure connectivity between services and pods. This service allows for seamless communication and access to pods both from within and outside the cluster, facilitating efficient interaction across the entire network.

3.3. Kubelet

Kubelet is another crucial component of Kubernetes that runs on each node within the cluster. It ensures that containers are running in a pod by interacting with the container runtime and the API server. Kubelet receives PodSpecs from the API server and ensures that the containers described are running and healthy. By constantly monitoring the state of pods and containers, Kubelet plays a key role in maintaining the desired state of the cluster and ensuring that applications are running as intended.

3.4. Monitoring Stack

Given the failure-prone nature of edge nodes, a robust monitoring stack is crucial for continuously tracking each node's state and integrating this data into various system behaviors. Prometheus [19] is widely used for metric recording, often deployed in conjunction with Grafana [20] for metric visualization and observability. This combination provides a comprehensive monitoring solution that is commonly adopted in the industry to ensure the health and performance of edge infrastructure.

3.5. Horizontal Pod Autoscaler (HPA)

Horizontal Pod Autoscaler (HPA) is a crucial API resource in Kubernetes that dynamically adjusts the number of pods for a service based on observed CPU utilization, memory usage, or custom metrics obtained from the monitoring stack. This functionality enables the Kubernetes cluster to respond to fluctuating workloads and optimize resource allocation by altering the number of replicas as needed. Consequently, HPA ensures efficient resource utilization and maintains application performance within the Kubernetes environment.

4. Problem Definition

This section defines the scheduling and migration problems that the scheduler aims to solve. Before defining the problems, it is necessary to state the assumptions the scheduler considers from its environment, input, and output.

4.1. Assumptions

To reduce the complexity of the scheduling space, we have made the following assumptions.

4.1.1. Cluster Assumptions

We assume that there is a single Kubernetes cluster, consisting of both edge nodes and cloud nodes. The Kubernetes scheduler is aware of the node types. The edge and cloud nodes are represented by the sets E and C respectively. Without loss of generality, we assume a single cloud node with infinite resource capacity in this work.

The edge nodes are heterogeneous, and the resources of the ith node are represented as a two-dimensional vector R_{E_i}, comprising the number of processor cores, and the volume of memory of this node in gigabytes.

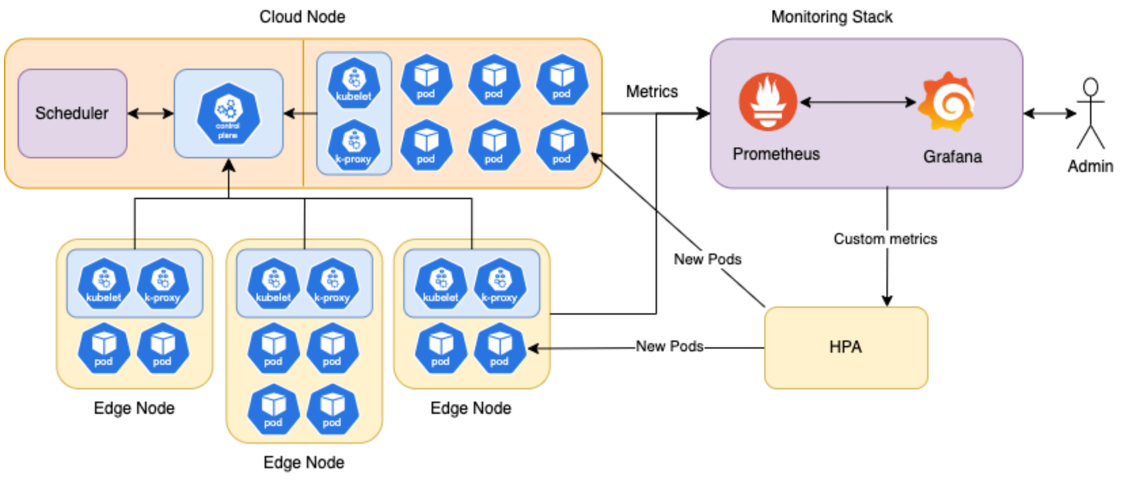

Figure 1: Kubernetes-managed cloud-assisted edge cluster model

4.1.2. Service Assumptions

The scheduler assumes that users may request one of the services from the service set S, each comprising several pods that aim to respond to user requests. The set P_i denotes the pods of the ith service deployment S_i. The scheduler is not responsible for determining the number of pods for each service, but must make placement decisions based on the assumption that this number can vary by HPA. It is also assumed that all pods of service S_i require the same resources presented as a two-dimensional vector R_{S_i}, where the dimensions correspond to the number of processor cores, and the volume of memory required by that service.
Furthermore, the service provider provides a QoS level for each service, stating that at least Q_i fraction of P_i should be deployed on edge.

4.1.3. Scheduler Assumptions

At any given moment, the scheduler has a collection, potentially with duplicate members, denoted by N, representing the deployments of newly created pods. The scheduler outputs a binary allocation matrix, u, where $u_{i,j,k} = 1$ if the pod $P_{i,j}$ has been placed on the node E_k in the edge cluster.
All the notations are summarized in Table 1.

4.2. Problem Statement

The main objective of the scheduler is to keep users satisfied based on the given guarantees on QoS, i.e. maximizing $QoS(u)$. As the secondary objective, the scheduler tries to perform the minimum number of migrations. We define the overall quality of service ($QoS(u)$) as the accumulated QoS of all services as follows:

$$QoS(u) = \sum_{i \le |S|} QoS(i, u) \tag{1}$$

where $QoS(i, u)$ is the QoS level of service S_i. The scheduler aims to establish the maximum number of QoS guarantees possible; if it is not possible, it strives to get as close as possible. To do this, we define $\Delta(i, u)$ as the difference between the current level of QoS and the expected level of QoS (Q_i) for service S_i.

$$\Delta(i, u) = \frac{\sum_{j \le |P_i|, k \le |E|} u_{i,j,k}}{|P_i|} - Q_i \tag{2}$$

Then, we define F and $QoS(i, u)$ as follows:

$$F(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ \beta x + \gamma & \text{if } x \ge 0 \end{cases} \tag{3}$$

$$QoS(i, u) = F(\Delta(i)) \tag{4}$$

The F function is designed to transform the difference $\Delta(i, u)$ into a QoS value that reflects the satisfaction level of each service. When $\Delta(i, u)$ is negative, the current QoS is below the expected level, hence we apply a linear penalty with severity of α. This motivates the scheduler to increase the QoS level, if it does not meet the expected level. When it is non-negative, meaning the current QoS meets or exceeds the expected level, we increase the reward by a large value γ to incentivize the scheduler to maximize QoS. We also apply a linear reward by the rate of β to motivate the scheduler to keep increasing the QoS level, even if all QoS requirements are met. Overall, these constants follow the inequality below:

$$0 \le \beta < \alpha << \gamma \tag{5}$$

After each scheduling event, the scheduler changes the allocation mapping from u' to u. The number of migrations performed for the pod $P_{i,j}$ that is not in N can be defined as follows:

$$\theta(i, j) = \frac{\sum_{k \le |E|} \left| u_{i,j,k} - u'_{i,j,k} \right| + \left| \sum_{k \le |E|} u_{i,j,k} - \sum_{k \le |E|} u'_{i,j,k} \right|}{2} \tag{6}$$

The first term of the equation 6 counts the number of changes in the edge allocation of pods (the value will be increased by two for each migration between edge nodes and by one for edge-to-cloud/cloud-to-edge migration). The second term

Table 1: Summary of all notations

Symbol	Definition
E	Set of all edge nodes in the edge cluster
E_i	ith node in the edge cluster
R_{E_i}	Resource vector of edge node E_i
t_C	Request latency when executed on cloud
t_E	Request latency when executed on edge nodes
S	Set of all deployments
P_C	The set of all cloud pods
P_E	The set of all edge pods
$P_{i,j}$	The jth pod of the ith service
R_{S_i}	Resource vector for pods of service S_i
Q_i	Quality of service guaranteed for service S_i
N	Set of all newly created pods, which is input to the scheduler
u	Allocation mapping for all pods
u'	Allocation mapping for all pods before scheduler decision
$u_{i,j,k}$	Whether the pod $P_{i,j}$ scheduled on the kth node
M_{C2E}	The maximum number of pods that the scheduler will migrate from the cloud to the edge in each suggestion
M_{ER}	The maximum number of pods that the scheduler will reorder in the edge (migrate from one edge node to another edge node) in each suggestion or scheduling request
M_{CPU}	The maximum amount of CPU cores provided by the edge nodes
M_{MEM}	The maximum amount of memory in GB provided by the edge nodes

calculates how many pods have migrated from cloud to edge and vice versa. As each migration is counted twice, the overall value is divided by two.

The following are the problem constraints:

1. Pod-to-node allocation constraints: Each pod can be assigned to at most one of the cluster nodes. Hence, we have:

$$\begin{aligned} & u_{i,j,k} \in \{0,1\}, \\ & \sum_{k \le |E|} u_{i,j,k} \le 1 \qquad \forall i \le |S|, j \le |P_i| \end{aligned} \tag{7}$$

When a pod is not allocated to any edge node (i.e. $\sum_{k\le|E|} u_{i,j,k} = 0$), the pod is deployed on the cloud.

2. Resource constraints: The sum of resources required by the pods allocated to each node must be less than that node's resources:

$$\sum_{i\le|S|, j\le|P_i|} u_{i,j,k} \times R_{s_i} \le R_{E_k} \qquad \forall k \le |E| \tag{8}$$

The scheduler aims to maximize user QoS (primary goal) and minimize migrations count (secondary goal) subject to the above constraints:

$$\begin{aligned} & \max_{u\in U} \sum_{i\le|S|} QoS(i,u) \\ & \min_{u\in U} \sum_{i\le|S|, j\le|P_i|, P_{i,j}\notin N} \theta(i,j) \end{aligned} \tag{9}$$

The presented problem is a MILP[1] optimization problem over integers, and its optimal solution is NP-hard. We approach solving this problem by proposing KubeDSM, a Kubernetes scheduler that incorporates various combinatorial methods.

5. Proposed Approach

In this section, we present the system architecture of the proposed KubeDSM scheduler in detail, including its key components, their functionalities, and the proposed scheduling algorithms.

5.1. Scheduler Overview

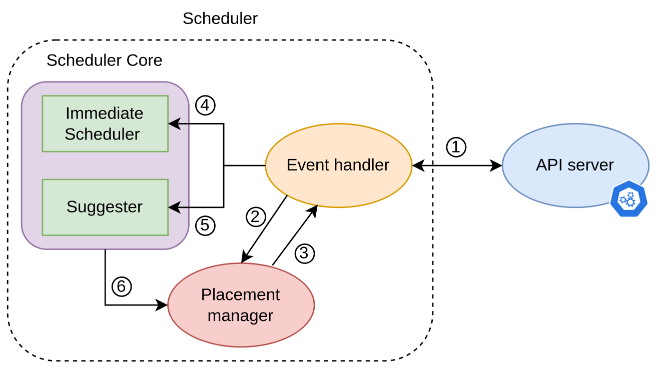

Figure 2: KubeDSM component diagram

As illustrated in Figure 2, the scheduler is composed of three main components.

Event Handler: The event handler serves as the communication bridge between the scheduler and Kubernetes. It subscribes to the events channel in the Kubernetes API server to capture all events occurring within the scheduler's namespace ①. Additionally, it periodically requests information about all nodes and pods to account for any missed or unexpected events. Each event received is forwarded to the Placement Manager component ②, which then sends a response detailing the action to be taken ③. Possible actions include waiting for another event, ignoring the current event, or executing tasks such as deleting a pod or deploying a pod on a specific node. Using the information gathered from events or periodic requests, the event handler constructs and maintains a cluster state (CS), representing the scheduler's understanding of the current cluster state. The event handler continuously updates this cluster state over time.

[1]Mixed-integer linear programming

Table 2: Possible steps, actions and their verification

Step	Action	Verification
Create a pod for a deployment	Nothing (will be created by HPA)	A pod creation event for the same deployment
Bind a pod to a node	Submit a pod-target binding request to the API server	A pod changed event for the desired pod with the pod's node being the target node
Delete a pod	Submit a pod deletion request to the API server	A pod deletion event for the desired pod

Placement Manager: This component is responsible for implementing current pod placement plans. Each plan comprises a sequence of steps, and if any step fails or is canceled, the subsequent steps will also be canceled, resulting in only partial execution of the plan. A step includes two parts:
Action, which is the specific action required to execute the step, and
Verification, that is the method used to verify that the action was successfully and completely executed. Verification is achieved by receiving a specific type of event from the event handler.

Table 2 outlines the possible steps. When a new event is received from the event handler, one of the following scenarios occurs:

- The event matches the current state of one of the plans: the step is verified, and the next step in that plan is executed (or the plan is completed).
- The event is incompatible with some plans (e.g., it pertains to the same pod, but the type or expected information does not match): the related plans are canceled. In this case, any remaining pending pods will be deployed to the cloud.
- The event concerns the creation of a new pod: it is forwarded to the scheduler core (first via step ③, then step ④).
- The event is deemed irrelevant, so it is safely ignored.

Scheduler Core: The scheduler core is composed of two main components:
Immediate scheduler, this component is responsible for scheduling newly created pods. It receives a list of new pods from the event handler ④ and generates plans based on the scheduling algorithm to bind the pods to appropriate nodes. This process does not involve migration, resulting in single-step plans that only focus on binding new pods to selected nodes.
Suggester, this component is responsible for suggesting plans for pod reordering. It is periodically called by the event handler ⑤ to provide migration suggestions. These suggestions are formed by creating multiple plans based on suggestion algorithms for pod migration, aiming to place more pods on edge nodes and optimize the utilization of edge resources.
The plans generated by both components are then forwarded to the Placement Manager ⑥.
The scheduler defines the following constants (as its configuration) and utilizes them in the algorithm:

- M_{C2E}: The maximum number of pods that the scheduler will migrate from the cloud to the edge in each suggestion.
- M_{ER}: The maximum number of pods that the scheduler will reorder in the edge (migrate from one edge node to another edge node) in each suggestion or scheduling request.
- M_{CPU} and M_{MEM}: The maximum amount of resources provided by the edge nodes (CPU cores and memory in gigabytes respectively).

5.2. Algorithms overview

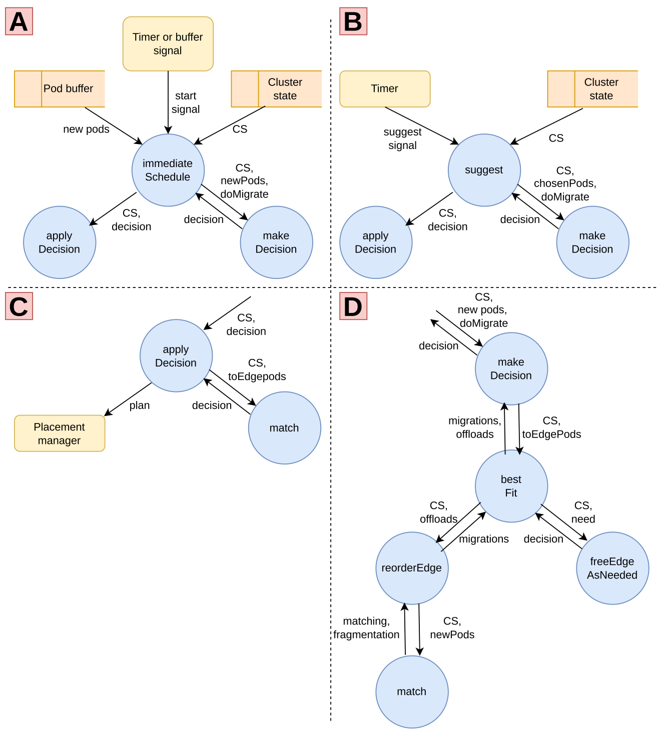

Figure 3: Scheduler Complete Call Graph

Figure 3 demonstrates the data flow diagram of the scheduler's algorithm. It features two entry points: `suggest` and `ImmediateSchedule`. The remaining parts of the algorithm, presented in reverse dependency order, include `match`, `reorderEdge`, `freeEdgeAsNeed`, `bestFit`, `makeDecision`, and `applyDecision`. We describe each component in this order.

immediateSchedule: As shown in Algorithm 1, this algorithm is a simple call to `makeDecision` with the list of new pods, with migration disabled (as new pods need to be scheduled as quickly as possible to avoid potential connection loss for certain applications). Following this, it invokes `applyDecision` with the generated decision.

suggest: The goal of the suggest algorithm (Algorithm 2) is to suggest some migrations to offload (migrate) as many pods as possible from the cloud to the edge (to increase QoS) and

Algorithm 1: Immediate Schedule Algorithm

```
Input: Cluster State (CS), newPods
Function ImmediateSchedule(CS, newPods):
    decision ← makeDecision(CS, newPods, false);
    applyDecision(CS, decision);
```

reorder (migrate) some pods inside the edge cluster, to decrease the fragmentation if needed. It begins by creating a list of pods to be placed on edge nodes. It then makes a decision using the selected pods as new pods, with migration enabled. Finally, it applies the decision. As shown in Algorithm 2, for creating the list, in each iteration, as long as the number of selected pods is less than M_{C2E} and there is at least one candidate pod in the cloud, it sorts the candidate pods by their score and checks the pod with the highest score. If the pod, along with the already selected pods, fits on the edge, it is added to the list. The pod is then removed from the candidate list, and the iterations continue. The scoring function is defined as:

$$score(pod) = \frac{QoS(CS' - \{pod\ on\ cloud\} \cup \{pod\ on\ edge\}) - QoS(CS')}{size(pod)} \tag{10}$$

The score function aims to capture the increase in QoS per resource unit moved to the edge. The QoS increase is calculated as the difference between when the pod is on the edge and the cloud. To find the increase per resource unit moved, score is divided by the pod's size, which is calculated as:

$$size(pod) = \sqrt{\frac{pod_{CPU}}{M_{CPU}} \times \frac{pod_{MEM}}{M_{MEM}}} \tag{11}$$

makeDecision: As demonstrated in algorithm 3, the makeDecision function begins by choosing to deploy all new pods to the cloud as the default decision. It then iteratively evaluates each subset of new pods, denoted as *toEdge*, to determine the possibility of deploying *toEdge* on edge nodes. If the combined resources of *toEdge* exceed the total available edge resources, that subset is discarded. If migration is allowed, the function calculates the optimal and minimal set of migrations required to accommodate *toEdge* on edge using the bestFit function. If migration is not allowed, feasibility is checked approximately by comparing the total resources of *toEdge* against the available resources in the edge nodes. Finally, the Quality of Service (QoS) score for both the current decision and the best decision is evaluated. If the current decision yields a higher QoS score than the previously recorded best decision, it is selected as the new best decision. The function ultimately returns the best decision found through this process.

bestFit: Algorithm 4 aims to determine the optimal set of migrations needed to free up a specified amount of resources on the edge (referred to as *need*). The term "optimal" here means: First, to meet the required amount of freed resources on the edge, the algorithm prioritizes minimizing the associated QoS loss. Second, among the possible sets of offloads that

Algorithm 2: Suggest Algorithm

```
Input: Cluster State (CS)
Function suggest(CS):
    candidPods ← all cloud pods;
    chosenPods ← [];
    CS' ← CS ;      // a copy of cluster state
    currentFreeResources ← freeResources in edge;
    while len(chosenPods) < $M_{C2E}$ and len(candidPods) > 0 do
        sort candidPods by score (Eq 10) in decreasing order;
        firstCandidate ← candidPods.pop();
        if firstCandidate.resources ≤ currentFreeResources then
            chosenPods += firstCandidate;
            currentFreeResources -= firstCandidate.resources;
            CS' ← CS' - {firstCandidate on cloud} ∪ {firstCandidate on edge};
        end
    end
    decision ← makeDecision(CS, chosenPods, true);
    applyDecision(CS, decision);
```

Algorithm 3: Make Decision Algorithm

```
Input: Cluster State, newPods, doMigrate
Output: bestDecision
Function makeDecision(CS, newPods, doMigrate):
    bestDecision ← place all new pods on cloud;
    foreach subset toEdge ⊂ newPods do
        if sum of resources of toEdge > edge's resources then
            continue;
        end
        currentDecision ← deploying toEdge on edge and other new pods on cloud;
        if doMigrate then
            migrationsNeeded ← bestFit(CS, toEdge);
            currentDecision += performing migrationsNeeded;
        end
        else
            if sum of resources of toEdge > free resources in edge then
                continue;
            end
        end
        if QoS(CS after bestDecision) > QoS(CS after currentDecision) then
            bestDecision ← currentDecision;
        end
    end
    return bestDecision;
```

achieve the same QoS, the algorithm selects the set that involves the fewest number of migrations. After identifying the best set of migrations to free up edge resources, the algorithm then attempts, if possible, to perform up to M_{ER} migrations. The goal is to reorder the edge in a manner that reduces fragmentation, making it easier for future pods to be accommodated on the edge nodes.

Algorithm 4: Best Fit Algorithm

```
Input: Cluster State (CS), toEdgePods
Output: migrations
1 Function bestFit(CS, toEdgePods):
2     offloads ← freeEdgeAsNeeded(CS, sum resources in
        toEdgePods) ;  // Offloads are migrations
        from the edge to the cloud to increase
        QoS .
3     CS' ← CS after offloads;
4     reorderings ← reorderEdge(CS') ;
        // Reorderings are migrations inside
        the edge for fragmentation reduction.
5     return offloads ∪ reorderings;
```

freeEdgeAsNeeded: The `freeEdgeAsNeeded` algorithm (Algorithm 5) operates similarly to the `suggest` algorithm, but in reverse. Its goal is to minimize score loss while freeing up the required amount of resources on the edge. It sorts the edge pods using a similar scoring function as the `suggest` algorithm and proceeds to free them one by one until the needed resources are released. Since the scoring function changes each time a pod is freed, it is crucial to re-sort the candidate list after each pod is removed. The score function is defined as follows:

$$score(pod) = \frac{QoS(CS' - \{pod\ on\ edge\} \cup \{pod\ on\ cloud\}) - QoS(CS')}{size(pod)} \tag{12}$$

reorderEdge: This function (Algorithm 6) aims to reduce fragmentation within edge nodes by identifying potential pod migrations within the edge. The resource fragmentation for each node is calculated based on following equation:

The algorithm evaluates each subset of edge pods, up to a size of M_{ER}, and considers migrating that subset. Using the `match` algorithm, it determines the optimal target nodes for each pod. The resulting fragmentation is then calculated for each subset, and the subset with the least fragmentation, along with its best target nodes, is selected as the optimal set of reorder migrations.

appleDecision: A decision may contain multiple components, and the applyDecision function (Algorithm 7) manages each of these accordingly. First, it may include a set of migrations, which could involve migrating pods from cloud to edge, from edge to cloud, or reordering within the edge itself. The applyDecision function generates a multi-step plan for each migration and forwards it to the placement manager. Additionally, there could be a list of new pods designated for deployment on the edge, each with a specific target node. For these pods, the function creates a single-step plan for binding them to their re-

Algorithm 5: Free Edge As Needed Algorithm

```
Input: Cluster State, need
Output: freedPods
1  Function freeEdgeAsNeeded(CS, need):
2      candidPods ← all edge pods;
3      freedPods ← [];
4      CS' ← CS ;       // a copy of cluster state
5      currentFreeResources ← freeResources in edge;
6      while len(candidPods) > 0 and
        currentFreeResources < need do
7          sort candidPods by score in decreasing order;
8          firstCandidate ← candidPods.pop();
9          freedPods += firstCandidate;
10         currentFreeResources +=
             firstCandidate.resources;
11         CS' ← CS' - {firstCandidate on edge} ∪
             {firstCandidate on cloud};
12     end
13     return freedPods;
```

Algorithm 6: Reorder Edge Algorithm

```
Input: Cluster State
Output: bestMigrations
1  Function reorderEdge(CS):
2      bestMigrations ← {};
3      leastFrag ← frag(CS);
4      foreach $podsToReorder \in P_E$ where
        $|podsToReorder| < M_{ER}$ do
5          CS' ← CS - {podsToReorder on edge};
6          mapping, _ ← match(CS', podsToReorder);
7          CS' ← CS' ∪ {deploying podsToReorder by
             mapping};
8          currentFrag ← frag(CS');
9          if currentFrag < leastFrag then
10             leastFrag ← currentFrag;
11             bestMigrations ← {(pod → mapping[pod])
                 for pod in podsToReorder};
12         end
13     end
14     return bestMigrations;
```

spective edge nodes. Lastly, for pods that need to be placed on the cloud, a single-step plan is created for each to be bound to the cloud node. After selecting the best decision, the next step is to apply it. The algorithm begins by assigning each pod, intended for deployment on the edge, to an appropriate edge node, using the match algorithm based on the cluster state after the migrations. This results in a mapping of each pod to its target edge node. However, there may be cases where not all the pods intended for edge deployment can be successfully matched to a node, as the best decision was initially derived based on approximate feasibility. In such cases, as many pods as possible are deployed on the edge, while the remaining pods are redirected to the cloud.
The placement manager then executes each plan, step by step. For any given plan, the scheduler ensures that no action begins until all preceding actions have been successfully completed. With this in mind, as detailed in algorithm 7, plans for deploying pods on the cloud are independent and can proceed without inter-dependencies. However, the edge deployment plan is conditional upon the successful completion of all migrations (with an additional dependency created to ensure proper pod deployment on the edge). If a plan fails partway through, any remaining pods within that plan are deployed on the cloud, where they await further reordering by the suggester to move them to the edge.

Algorithm 7: Apply Decision Algorithm

```
Input: Cluster State, decision
1  Function applyDecision(CS, decision):
2      CS' ← CS after migrating decision.migrations;
3      matching, remainingPods ← match(CS',
         decision.toEdgePods);
4      toCloudPlans ← [];
5      foreach pod in decision.toCloud ∪ remainingPods
         do
6          toCloudPlans ← append(toCloudPlans, [deploy
             pod on cloud]);
7      end
8      toEdgePlan ← [];
9      foreach migration in decision.migrations do
10         toEdgePlan ← (toEdgePlan..., then migration);
11     end
12     foreach pod in matching do
13         toEdgePlan ← (toEdgePlan..., then deploy pod
             on edge);
14     end
15     foreach plan in toCloudPlans do
16         execute plan;
17     end
18     execute toEdgePlan;
```

match: The goal of the Algorithm 8 is to match new pods to edge nodes while satisfying the following conditions: maximize the number of pods matched, and prioritize matchings that result in the least amount of fragmentation among different options. To achieve this, the `match` algorithm employs a dynamic programming approach. Let $dp[(i, pods)]$ represent the minimum fragmentation possible when deploying a set of pods to edge nodes from 1 to i. Based on this definition, for each i from 1 to the number of edge nodes $|E|$, and for each subset of new `pods` $pods$, a state is defined for calculating dp. The base state, representing the initial state of the cluster, is:

$$dp[(0, \emptyset)] = frag(CS) \tag{13}$$

All other states are initially marked as impossible (i.e. setting dp value to ∞). The recurrence relation is formulated as follows: for each state $(i, pods)$, the goal is to deploy a (possibly empty) subset of $pods$ to the ith node. We refer to this subset as cur_{pods}. First, it must be verified if cur_{pods} can be deployed on the ith node, which is determined by comparing the remaining resources of the node with the total resources required by cur_{pods}. If the deployment is feasible, a possible solution for $dp[(i, pods)]$ can be derived as:

$$\begin{aligned} dp[(i, pods)] = \\ & dp[(i-1, pods - cur_{pods})] \\ & + frag(E_i \cup cur_{pods}) - frag(E_i) \end{aligned} \tag{14}$$

Based on this, the recurrence relation for dp can be expressed as:

$$\begin{aligned} dp[(i, pods)] = \\ & min_{\forall cur_{pods} \subseteq pods}\Big(dp[(i-1, pods - cur_{pods})] + \\ & frag(E_i - cur_{pods}) - frag(E_i)\Big) \end{aligned} \tag{15}$$

The final solution is obtained by finding the largest subset of input pods (denoted as $final_{pods}$ such that $dp[(|E|, final_{pods})]$ is not ∞ and has the minimum fragmentation. To determine the specific matching, the algorithm keeps track of dynamic programming updates in an additional array called par. For each state S, $par[S]$ stores S', which is the state from which $dp[S]$ was updated. By tracing back from $par[final_{state}]$ to the base state $(0, \emptyset)$, the deployment of each pod to its respective node can be determined.

5.3. Complexity analysis

Let's analyze each algorithm in the reverse dependency order.

match: The matching algorithm has two main steps. First, it calculates the dp value for all $|E| \times 2^{|newPods|}$ states, which is $O(|E| \times \sum_{N' \subset newPods} 2^{|N'|})$ that is equal to $O(|E| \times 3^{|newPods|})$. Second, it uses these values to find the best match. This second step takes at most $|E|$ iterations to track the best solution and extract targets for each new pod, so it is $O(|E| + |newPods|)$. Overall the complexity is $O(|E| \times 3^{|newPods|})$.

applyDecision: This algorithm first uses the `match` algorithm to fit $decision.toEdgePods$ to the edge, this parts will be done in $O(|E| \times 3^{|decision.toEdgePods|})$ complexity, and then will simply create a plan for each pod in the decision which is done in $|decision|$, in the end, the complexity is $O(|E| \times 3^{|decision.toEdgePods|} + |decision|)$.

Algorithm 8: Match Algorithm

```
Input: Cluster State, newPods
Output: targets, the minimum fragmentation achieved
Function match(CS, newPods):
    dp ← mapping from all states ($i$, pod_set) to $\infty$;
    par ← mapping from all states to None;
    $dp(0, \emptyset) \leftarrow$ frag(CS);
    $par(0, \emptyset) \leftarrow$ Nil;
    for $i \leftarrow 1 \textbf{to} |E|$ do
        foreach $pods \subseteq newPods$ do
            foreach $cur_{pods} \subseteq pod_{set}$ do
                $fragChange \leftarrow frag(CS[E_i] \cup cur_{pods})\text{-}frag(CS[E_i])$
                $achievedFragmentation \leftarrow dp(i-1, pods - cur_{pods}) + fragChange;$
                if $dp(i, pods) > achievedFrag$ then
                    $dp(i, pods) \leftarrow achievedFrag;$
                    $par(i, pods) \leftarrow (i-1, pods - cur_{pods});$
                end
            end
        end
    end
    max_sub_set ← $\emptyset$;
    foreach $pods \subseteq newPods$ do
        if $dp(|E|, pods) = \infty$ then
            continue;
        end
        if $|pods| > |max\_sub\_set|$ or ($|pods| = |max\_sub\_set|$ and $dp(|E|, pods) < dp(|E|, max\_sub\_set)$) then
            $max\_sub\_set \leftarrow pods;$
        end
    end
    $targets \leftarrow \{\};$
    $stateIterator \leftarrow (|E|, max\_sub\_set);$
    while $stateIterator \neq None$ do
        $i, pods \leftarrow stateIterator;$
        $cur_{pods} \leftarrow pods - par(i, pods);$
        foreach $pod \in cur_{pods}$ do
            $targets[pod] \leftarrow E_i$ end
            $stateIterator \leftarrow par(i, pods);$
        end
        foreach $pod \in newPods\text{-}max\_sub\_set$ do
            $targets[pod] \leftarrow cloud\ node;$
        end
        return $targets,\ dp(|E|, max\_sub\_set);$
```

reorderEdge: The algorithm invokes the `match` function for each subset of P_E that has at most M_{ER} pods in it. So the complexity is always less than $O(|P_E|^{M_{ER}} \times |E| \times 3^{M_{ER}})$.

freeEdgeAsNeeded: In this algorithm, all edge pods are sorted in each iteration by a score function, and the best one is chosen. The scores can be computed at first for each deployment and modified whenever a pod from that service is chosen. With this implementation, the time complexity will be $O(|P_E|^2 \times log(|P_E|))$.

bestFit: This algorithm simply calls two `freeEdgeAsNeeded` and `reorderEdge` functions in order. So the complexity is $O(|P_E|^{M_{ER}} \times |E| \times 3^{M_{ER}} + |P_E|^2 \times log(|P_E|))$.

makeDecision: For each subset of new pods, if the `doMigrate` is enabled, it will invoke the `bestFit` function, otherwise, it will compare the resources. In both cases, it will calculate the QoS after and before the decision that can be implemented efficiently as described in `freeEdgeAsNeeded`.This means, if `doMigrater` is enabled then the complexity is $O(2^{|newPods|} \times Complexity_{bestFit})$, and otherwise $O(2^{|newPods|} \times |newPods|)$.

suggest: This is one of the main entry points of the scheduler. It consists of three parts, first choosing the cloud pods to suggest, which are the same as `freeEdgeAsNeeded` pod choosing phase but it is for cloud pods, so the complexity is $O(M_{C2E} \times |P_C| \times log(|P_C|))$. The second part is calling `makeDecision` on the chosen pods, with migration being enabled, which is done in $O(2^{M_{C2E}} \times (|P_E|^{M_{ER}} \times |E| \times 3^{M_{ER}} + |P_E|^2 \times log(|P_E|)))$. The third part is applying the decision made for the suggested pods with $O(|E| \times 3^{M_{C2E}} + M_{C2E} + |P_E|)$ complexity. The $M_{C2E} + |P_E|$ term accounts for the maximum size of a decision (may migrate all $|P_E|$ edge pods to the cloud and migrate M_{C2E} pods from the cloud to the edge). Overall, the complexity is:

$$\begin{aligned} O(&M_{C2E} \times |P_C| \times log(|P_C|)+ \\ &2^{M_{C2E}} \times |P_E|^{M_{ER}} \times |E| \times 3^{M_{ER}}+ \\ &2^{M_{C2E}} \times |P_E|^2 \times log(|P_E|) \\ &|E| \times 3^{M_{C2E}}) \end{aligned} \tag{16}$$

immediateSchedule: This is the other main entry point of the scheduler. It includes two steps, first invoking `makeDecision` on the *newPods*, with migration being disabled, which is done in $O(2^{|N|} \times |N|)$, and then apply the decision made using `applyDecision` that is done in $O(|E| \times 3^{M_{C2E}} + M_{C2E} + |P_E|)$. Given this, the overall complexity is:

$$\begin{aligned} O(&2^{|N|} \times |N|+ \\ &|E| \times 3^{M_{C2E}} + M_{C2E} + |P_E|) \end{aligned} \tag{17}$$

For further analysis, let's evaluate how large M_{C2E} and M_{ER} can get. The migration is not a built-in feature in Kubernetes and to simulate a migration, we have to delete the pod where it is while creating the new one. Because Kubernetes ensures having the HPA set number of pods always running, it won't terminate the pod, until the new pod is running. This may cause some migrations to take a two-step approach (i.e. migrate the pod first to the cloud, and then migrate it from the cloud to the target edge node) when the resources allocated to the pod are in need (for other migrations and scheduling). This entire process can take a couple of seconds to fulfill. Have this in mind, M_{ER} is set from 1 to 3, meaning the scheduler can do at most three migrations per suggestion event (that happens periodically), where the source of the migration is in the edge. Because migrations from the cloud is easier (the cloud resources are infinite) M_{C2E} can be set to larger numbers up to ten. The `suggest` algorithm is linear to the number of pods on cloud $|P_C|$ and as the number of pods on the edge $|P_E|$ gets lower, it can search for more migrations on edge. The `immediateSchedule` algorithm performance is mostly influenced by the number of new pods $|N|$. With this complexity we can schedule a batch of 20 pods simultaneously in less than a second, which is a great improvement over Kubernetes default or other greedy schedulers' ability to schedule one pod at a time.

6. Evaluation

We built our scheduler using Golang, which runs as a pod inside the cluster. Our scheduler's pod does not replace the Kube-scheduler pod. Instead, we designed it so that workloads can request allocation using the "schedulerName" property in Kubernetes, allowing them to choose their scheduler [21]. To connect with the API gateway, we used the Kubernetes Golang SDK and set up the necessary cluster role bindings (RBACs) so the scheduler could allocate pods to nodes. Moreover, we have implemented another scheduler Sencillo[2], providing baseline algorithms to compare our scheduler's performance. Project Scenilo provides the following baselines: Random, Biggest Edge Node First, Smallest Edge Node First, and Cloud First. Additionally, we have used Prometheus [19] as our time-series database to collect metrics of the cluster and Grafana [20] to monitor changes in the cluster.
The source code for all of these projects can be found in our GitHub repositories [22, 23], which supports the reproducibility of the research we propose in this paper.

6.1. Cluster Setup

Table 3: Summary of Nodes configuration used in our evaluations

Node Name	Role	Memory (GB)	Cores
N1	Master	6	4
N2	Edge Node	5	5
N3		4	4
N4		5	7
N5	Cloud Node	17	22

As detailed in Table 3, we created a K3S cluster [24], consisting of one master node, three edge nodes, and one resource-rich cloud node. The cloud node is assigned more resources, and edge nodes are assigned heterogeneous resources to emulate the typical configurations found in cloud-assisted edge clusters.

Table 4: Summary of service deployments' resource requests

Service Name	Cores	Memory (Mi)
A	1	950
B	1	1900
C	1	950
D	2	1900

Our custom load generation tool, DrStress [25], performs stress tests by simulating multi-threaded HTTP requests to our pods. We utilized Kubernetes's Horizontal Pod Autoscaler (HPA) to adjust the pod count dynamically based on incoming request rates.

Furthermore, We assume four deployments on our cluster: A, B, C, and D, with their resource configuration detailed in Table 4. Each scenario begins with one pod from each service deployment. As the request rate rises, Kubernetes HPA automatically increases the number of pods, enabling us to assess the scheduler's performance in various situations.

6.2. Scenario Design

We created several load-testing scenarios, demonstrated as frequency sequences for each deployment, which DrStress uses to drive tests. The scenarios generate request rates based on the normal distribution to evaluate how well the system allocates resources. Each scenario has different request rates, requiring varying amounts of cluster resources. For clarification, we named the scenarios by the amount of edge resources they need. For example, a scenario labeled scenario_1.0_0.5 indicates a case where the generated workload fully utilizes the edge resources on average (100%), with a standard deviation of 0.5 in the normal distribution.
Each scenario consists of a series of cycles, for each cycle the amount of resources used is calculated by sampling the usage fraction from the distribution. Then these resources are equally divided between deployments. In the end, for each cycle, the frequency of requests is calculated in such a way that triggers the HPA to create the desired number of pods for each deployment. These frequencies are then passed to DrStress to run the scenario. For our experiments, each scenario consists of 12 cycles, each lasting for 90 seconds.
Because the cloud and edge nodes are physically on the same cluster, the latencies do not differ. To address this issue, we assumed that the cloud response time is 350ms and the edge response time is 50ms.
Since the Kube scheduler does not differentiate between cloud and edge nodes by default, comparing it with KubeDSM and other edge-oriented schedulers would be unfair. To address this issue, we utilized Kubernetes's "Taint & Toleration" feature, a mechanism that allows a node to repel a set of pods, to prioritize edge nodes for pod allocation. We have set node affinity for our pods, attracting them to our edge nodes.

6.3. Experiment Set 1: Compare with Baselines

In the first set of experiments, we analyzed the scheduler under various scenarios. We initially varied only the standard deviation, which simulates sudden changes in user requests while

[2]Sencillo means simple in Spanish

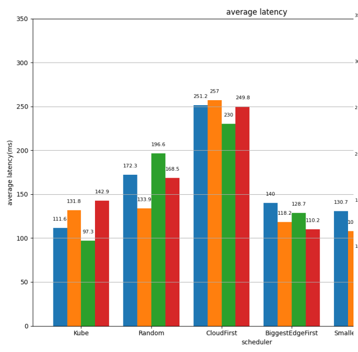

Figure 4: scenario 1.1 0.4

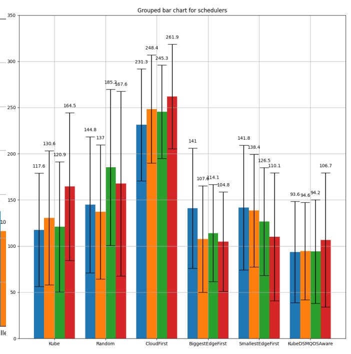

Figure 5: scenario 1.2 0.4

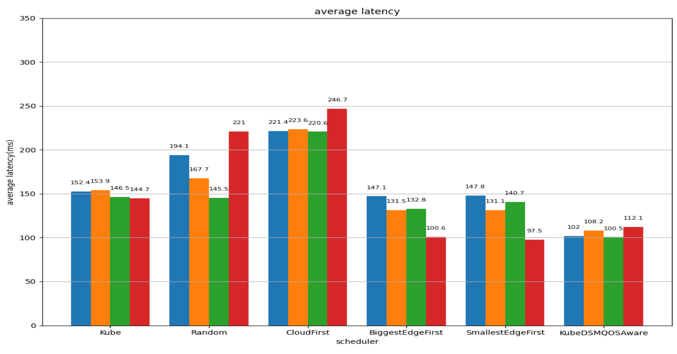

Figure 12: scenario 1.5 0.5

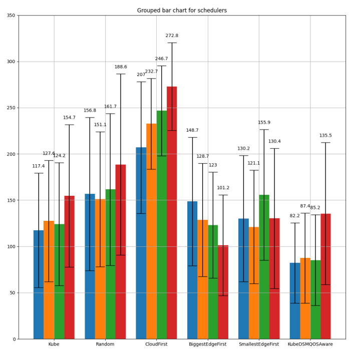

Figure 6: scenario 1.3 0.4

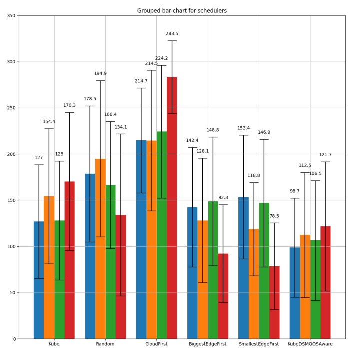

Figure 7: scenario 1.4 0.4

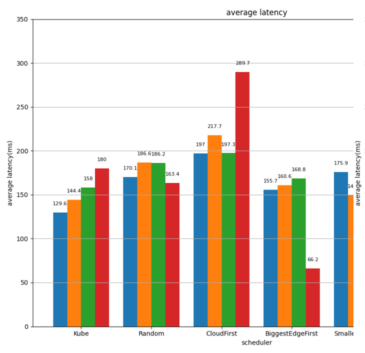

Figure 8: scenario 1.5 0.1

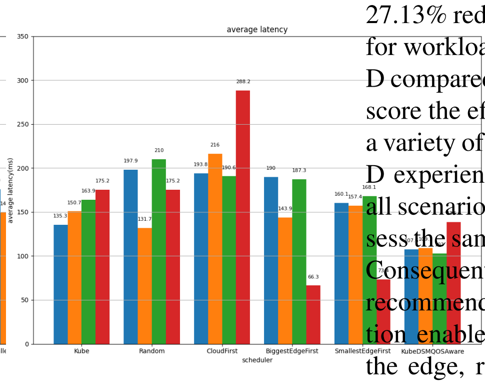

Figure 9: scenario 1.5 0.2

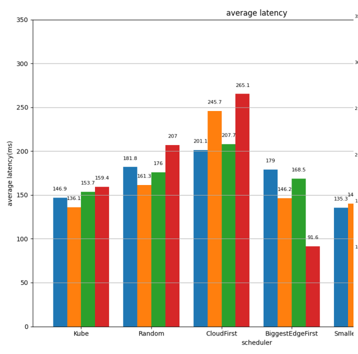

Figure 10: scenario 1.5 0.3

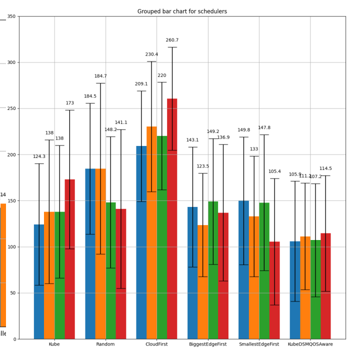

Figure 11: scenario 1.5 0.4

keeping the edge resource usage (mean parameter for the normal distribution) constant. After that, we fixed the standard deviation and adjusted the edge resource usage rate simulating the traffic increase. Our experiments revealed significant performance improvements with KubeDSM. As shown in Figure 4, KubeDSM reduced latency by 25%, 40%, 12.5%, and 36.8% for workloads A, B, C, and D, respectively, compared to the kube-scheduler. These results demonstrate KubeDSM's superior performance, achieving the lowest average latency compared to all other baselines. Figure 5 further supports these findings, showing that KubeDSM reduced latency by 20.4%, 27.5%, 22%, and 35.15%. Similarly, Figures 13 and 14 show a 29.9% and 22.28% drop in latency for workload A, a 31.5% and 27.13% reduction for workload B, a 31.4% and 16.7% lowering for workload C, and a 12.4% and 39.9% decrease for workload D compared to kube-scheduler. These impressive results underscore the effectiveness of KubeDSM in reducing latency across a variety of workloads. Our observations indicate that workload D experiences higher latency than the other workloads across all scenarios, primarily due to its larger size. All workloads possess the same Quality of Service (QoS) factor in these scenarios. Consequently, the suggester component, detailed in Section 5, recommends migrating most D pods to the cloud. This migration enables the allocation of additional A, B, and C pods at the edge, resulting in lower latency for workloads A, B, and C. Moreover, in these experiments, we showed that migration can assist in scenarios where the sum of resource requests exceeds the total available resources at the edge. Next, we kept the edge resource usage rate constant and changed the standard deviation, simulating changes in the request rate. In Figure 8, KubeDSM reduced latency by 17.5%, 27.8%, 29.5%, and 42.79% for A, B, C, and D, respectively. However, the BiggestEdgeFirst and SmallestEdgeFirst schedulers surpassed it in performance. We believe this observation is due to the greedy nature of BiggestEdgeFirst and SmallestEdgeFirst, which try to allocate as many pods to the edge as possible. However, this policy will create unfairness against A, B, and C, as indicated by their higher average latency. Considering an aggregation of all workloads, KubeDSM has shown better performance as its average latency considering all workloads is much lower than other competitors. Figures 9, 10, 11, and 12 exhibit a similar pattern. Specifically, there is a 20.47%, 23.2%, 14.8%, and

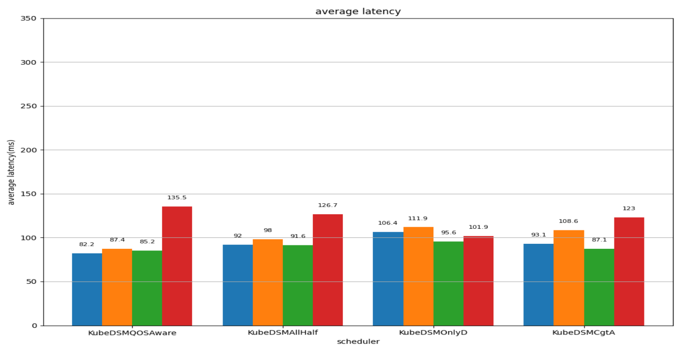

Figure 13: scenario 1.3 0.4

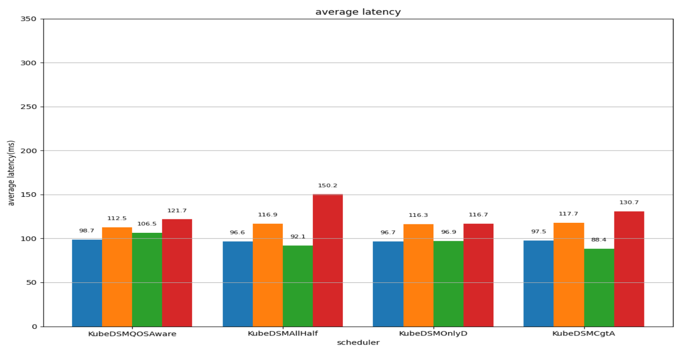

Figure 14: scenario 1.4 0.4

33.07% reduction for A; a 27.6%, 21.45%, 22.3%, and 31.3% reduction for B; and a 37.15%, 29.9%, 22.3%, and 31.3% reduction for C in each respective figure. For D, the performance gap decreases as the request rate increases. Initially, the performance gap was 39.7% and 12.55% compared to the BiggestEdgeFirst and SmallestEdgeFirst schedulers, respectively, as shown in Figure 8. By Figure 12w, this gap narrows to 10.25% and 13%.

6.4. *Experiment Set 2: QoS Factor Impact Analysis*

Our second experiment set sought to understand how QoS factors influence the scheduler's decisions and how different scenarios can impact average latency in the system. To achieve this, we will compare **KubeDSMQoSAware**, detailed in experiment set 1, with three of its variations. The first variation, **KubeDSMAllHalf**, sets the QoS factor for each workload to 0.5. In the second, **KubeDSMOnlyD**, a non-zero QoS factor is assigned only to workload D, while the QoS factors for all other workloads are set to zero. Finally, in **KubeDSMCgtA**, a QoS factor greater than the QoS factor assigned to workload C is assigned to workload A.
In Figure 13, we observe that by assigning a higher Quality of Service (QoS) factor to C than to A, the scheduler has adjusted its decisions to position more C pods at the edge, reflected in the lower average latency achieved by C. Moreover, by assigning a non-zero QoS factor solely to workload D, the scheduler has adjusted its strategy to place as many D pods at the edge as possible. This change has reduced latency for D but increased latency for all other workloads. Conversely, by setting the QoS factor to 0.5 for all workloads, we observe nearly identical performance between the mutant and base versions, reflecting the scheduler's effort to maximize satisfied QoS factors.
On the same note, figure 14 shows almost the same behavior from the scheduler. By setting a more significant QoS factor for C compared to A, we have reduced C's average latency by 17% and almost the same performance for A at the cost of worse performance for B and D. Moreover, in Figure 6, by setting a non-zero QoS factor only for D, we have been able to reduce its latency by 5%. However, for KubeDSMAllHalf, we have yet to have the same performance, which we do not know why yet!

6.5. *Experiment Set 3: Migration Strategy Impact Analysis*

Our final set of experiments focuses on the impact of various migration strategies on average latency. Our primary scheduler, KubeDSMQoSAware, implements various migration strategies, including migrations from edge to cloud, cloud to edge, and between edge nodes. To assess the effectiveness of our migration strategies, we compare it with four alternative versions of the scheduler. The first, KubeDSMNoMigration, does not perform migrations at all. The second, KubeDSMNoCloudOffload, disables cloud-to-edge migrations, while the third, KubeDSMNoEdgeMigration, disables migrations between edge nodes. Lastly, KubeDSMMidMigration reduces the frequency of migrations to half that of our primary scheduler.

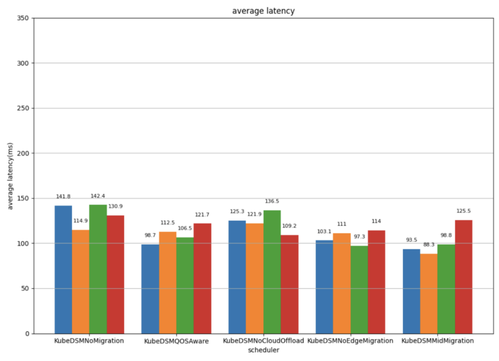

Figure 15: Average Latency for Workloads A, B, C, and D Across Five Schedulers in milliseconds. Workload colors: A (Purple), B (Orange), C (Green), D (Red).

Based on the diagram in Figure 15, we can conclude that migrations play a significant role in the performance of our scheduler. Disabling migrations leads to a drastic increase in average latency across all workloads compared to our primary scheduler. This effect is especially evident for workloads A and C, where latency increases by nearly 40%. In contrast, the latency for workload D shows only a slight increase, while workload B exhibits almost no difference in latency between the two schedulers.

Similarly, disabling cloud-to-edge migrations results in an increase in average latency for models A, B, and C. Although this increase is not as significant for workloads A and C compared to the previous alternative scheduler, workload B shows a more noticeable increase, with latency increasing by nearly 8% relative to the main scheduler. Interestingly, workload D shows a contrasting trend, with latency decreasing by approximately 10% compared to the main scheduler. This value represents the lowest recorded latency for workload D in all five evaluated schedulers. This behavior can likely be due to the high resource demands of workload D. When resource demands are substantial, running the workload in the cloud appears to be more effective than deploying it on edge servers, as more resources are readily available in the cloud compared to edge nodes.

As with disabling migrations between edge servers, there is a slight decrease in latency for workloads B, C, and D, while a minor increase is observed for workload A. Although the overall impact of disabling this type of migration appears positive, the changes are relatively small, with all fluctuations remaining within a 10% range.

Lastly, reducing the migration frequency to half that of the main scheduler results in a decrease in the average latency for workloads A, B, and C. Among these, workload B shows the most significant improvement, with a reduction of approximately 12% compared to the main scheduler. However, an opposite trend is observed for workload D, which experiences an increase in latency relative to the main scheduler.

7. conclusion

there is something here:

References

[1] Google cloud — distributed cloud. [Online]. Available: https://cloud.google.com/distributed-cloud

[2] Amazon web services — local zones. [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/localzones/

[3] Azure private multi-access edge computing. [Online]. Available: https://docs.microsoft.com/en-us/azure/private-multi-access-edge-compute-mec/overview

[4] N. Mohan, L. Corneo, A. Zavodovski, S. Bayhan, W. Wong, and J. Kangasharju, "Pruning edge research with latency shears," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. ACM, pp. 182–189. [Online]. Available: https://dl.acm.org/doi/10.1145/3422604.3425943

[5] X. Ma, S. Wang, S. Zhang, P. Yang, C. Lin, and X. Shen, "Cost-efficient resource provisioning for dynamic requests in cloud assisted mobile edge computing," vol. 9, no. 3, pp. 968–980. [Online]. Available: https://ieeexplore.ieee.org/document/8660570/

[6] X. Ma, S. Zhang, W. Li, P. Zhang, C. Lin, and X. Shen, "Cost-efficient workload scheduling in cloud assisted mobile edge computing," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/7969148/

[7] C. Li, J. Bai, Y. Ge, and Y. Luo, "Heterogeneity-aware elastic provisioning in cloud-assisted edge computing systems," vol. 112, pp. 1106–1121. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X20300339

[8] T. Kim, M. Al-Tarazi, J.-W. Lin, and W. Choi, "Optimal container migration for mobile edge computing: algorithm, system design and implementation," vol. 9, pp. 158 074–158 090. [Online]. Available: https://ieeexplore.ieee.org/document/9628116/

[9] W. Chen, Y. Chen, and J. Liu, "Service migration for mobile edge computing based on partially observable markov decision processes," vol. 106, p. 108552. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0045790622007674

[10] X. Li, Z. Zhou, Q. He, Z. Shi, W. Gaaloul, and S. Yangui, "Re-scheduling IoT services in edge networks," vol. 20, no. 3, pp. 3233–3246. [Online]. Available: https://ieeexplore.ieee.org/document/10039683/

[11] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," vol. 28, no. 4, pp. 1002–1016. [Online]. Available: http://ieeexplore.ieee.org/document/7557016/

[12] H. R. Chi, R. Silva, D. Santos, J. Quevedo, D. Corujo, O. Abboud, A. Radwan, A. Hecker, and R. L. Aguiar, "Multi-criteria dynamic service migration for ultra-large-scale edge computing networks," vol. 19, no. 11, pp. 11 115–11 127. [Online]. Available: https://ieeexplore.ieee.org/document/10043024/

[13] C. Rong, J. H. Wang, J. Wang, Y. Zhou, and J. Zhang, "Live migration of video analytics applications in edge computing," pp. 1–15. [Online]. Available: https://ieeexplore.ieee.org/document/10049158/

[14] S. Ghafouri, A. Karami, D. B. Bakhtiarvan, A. S. Bigdeli, S. S. Gill, and J. Doyle, "Mobile-kube: Mobility-aware and energy-efficient service orchestration on kubernetes edge servers," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, pp. 82–91. [Online]. Available: https://ieeexplore.ieee.org/document/10061810/

[15] W.-K. Lai, Y.-C. Wang, and S.-C. Wei, "Delay-aware container scheduling in kubernetes," vol. 10, no. 13, pp. 11 813–11 824. [Online]. Available: https://ieeexplore.ieee.org/document/10044213/

[16] Q. Zhang, C. Li, Y. Huang, and Y. Luo, "Effective multi-controller management and adaptive service deployment strategy in multi-access edge computing environment," vol. 138, p. 103020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1570870522001925

[17] Kube-scheudler. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/

[18] Kube-scheduler eviction. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[19] Prometheus website. [Online]. Available: https://prometheus.io/

[20] Grafana website. [Online]. Available: https://grafana.com/

[21] Kubernetes documentations. [Online]. Available: https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers/

[22] KubeDSM github. [Online]. Available: https://github.com/AUT-Cloud-Lab/ecmus

[23] sencillo github. [Online]. Available: https://github.com/AUT-Cloud-Lab/sencillo

[24] k3s — lightweight kubernetes. [Online]. Available: https://k3s.io

[25] DrStress github. [Online]. Available: https://github.com/AUT-Cloud-Lab/drstress

Table 5: Summary of Related Works on Migration Strategies in Cloud-Assisted Edge Environments

	Environment		Factors		Migration Strategy			
Paper	Kubernetes	Cloud-Assisted Edge Cluster	Fragmentation	Quality of Service	Container Placement	Cloud to Edge	Edge to Cloud	Edge to Edge
[7]	✗	✓	✗	✓	✗	✗	✓	✗
[5]	✗	✓	✗	✓	✗	✗	✓	✗
[11]	✗	✓	✗	✓	✗	✓	✗	✓
[8]	✗	✗	✗	✓	✗	✗	✗	✓
[9]	✗	✗	✗	✓	✓	✗	✗	✓
[10]	✗	✗	✗	✓	✓	✗	✗	✓
[12]	✓	✓	✗	✗	✗	✓	✗	✓
[13]	✓	✗	✗	✗	✗	✗	✗	✓
[14]	✓	✗	✗	✗	✓	✗	✗	✓
[15]	✓	✗	✗	✗	✓	✗	✗	✓
Proposed Approach	✓	✓	✓	✓	✓	✓	✓	✓