

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

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

^aComputer Eng. Dep. of Amirkabir University of Technology, Hafez Ave., Tehran, , Tehran, Iran

Abstract

Example abstract for the astronomy and computing journal. Here you provide a brief summary of the research and the results.

Keywords: keyword 1, keyword 2, keyword 3, keyword 4

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 could enable 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 via unreliable links. These limitations make resource management, scheduling, and migration challenging in edge-based server clusters. Moreover, these resource management techniques should be super fast under such a tight latency constraint, which adds to the complexity of designing such techniques.

Due to the limited computing capacity of edge nodes, sufficient resources may only exist to cover some computational tasks on the edge. Therefore, to increase the scalability of the edge platforms, several works in the literature have used cloud-assisted edge computing to dynamically allocate resources from the cloud to cover the shortage of resources in edge sites ([5], [6], [7]).

Due to the cost of computation offloading from the edge to the cloud and the limited resources of edge servers, perfect resource utilization on edge nodes is essential. Efficiently uti-

lizing edge resources reduces the need to offload computation from the edge servers to the cloud in resource shortage, thus lowering the average latency of applications running on the cloud-assisted edge platform.

Edge infrastructure providers now use virtualization technologies such as containerization to efficiently share these limited resources among different edge application services in a multi-tenant environment. Kubernetes, now the de-facto standard for container orchestration, has been introduced to deploy, scale, and manage containerized applications in the cloud. Several challenges must be addressed to make Kubernetes compatible with edge computing environments' sporadic and unreliable nature:

1. The Kubernetes distribution should be **lightweight** so it can be efficiently deployed on the limited resources of edge servers.
2. The distribution should support the network, application deployment, and metadata **synchronization between cloud and edge**.
3. The resource management policies incorporated into Kubernetes (such as the default scheduler) should be **adapted to the edge environment**.

Fortunately, several attempts were made to address the first two challenges, including SuSE's K3s [12] and KubeEdge [13]. Also, Several studies in the literature have tried to adapt the default behaviour of Kubernetes to edge environments, but it is still in its infancy. For example, authors in [14] have proposed a deep reinforced learning-based scheduler aiming for reducing energy consumption, being able to prove and suggest that common heuristic-based algorithms can be replaced with learning-based solutions or, in [15], three approaches for video analytic applications have been proposed, achieving near-transparent migrations from the user's point of view. Likewise, studies such as [16] have worked on ultra-large-scale mobile edge clusters using a new multi-criteria decision-making algorithm based on TOPSIS, whose results showed global optimal dynamic service migration and were able to release the centralized traffic burden of cloud servers. As reported in [17], the user's probability

function of sojourn time was analyzed to characterize user mobility intensity and service deployment overhead model, and the resulting scheduler was able to reduce user-perceived latency, constrain service migration, and optimize user experience quality in SDN(software-defined) networks.

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 (Kubescheduler), 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.

2. Related Work

Prior proposals target two main areas: (I) workload migration and (II) Adapting Kubernetes to Edge Clusters. However, to our knowledge, more work needs to be done on using Kubernetes in edge clusters equipped with migration-able schedulers.

2.1. Workload Migration

To optimize resource usage on edge clusters and hence the QoS, several scheduling and migration techniques have been proposed in the literature, such as [8], which proposed three migration algorithms and an algorithm-selector mechanism that selects the suitable algorithm based on container characteristics, allowing them to obtain sub-1-second live migration duration in their tests. Furthermore, [9, 10] modelled migration as partially observable Markov decision processes and multi-objective multi-constraint optimization to achieve better average delay and lower power consumption. Likewise, in [11], wang et al. have suggested an actual cost predictive model; assuming they have pre-determined upper bounds, they achieved

sub-optimal placement decisions based on their predicted costs in their simulations.

Additionally, some other studies used service migration to optimize 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, which results in optimizing migration and alleviating traffic congestion. Furthermore, Rong et al. [13] have proposed three approaches for migrating video analysis applications within edge clusters, ensuring that users hardly notice any change in service quality 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 to reduce energy consumption. This article has demonstrated that learning-based solutions can replace algorithmic approaches based on innovative methods. Also, Lai et al. [15] proposed a solution composed of a scheduler and a scorer, where the scheduler follows a filtering stage using the scorer and selects nodes with minimum network latency and the most available resources to serve the pods. This method can achieve a good balance between processing delays and network latency among pods. Moreover, As reported in [16], the user's probability function of sojourn time was analyzed to characterize user mobility intensity and service deployment overhead model, and the resulting scheduler was able to reduce user-perceived latency, constrain service migration, and optimize user experience quality in SDN(software-defined) networks.

The efficient utilization of edge resources reduces the necessity of offloading computations from edge servers to the cloud in case of resource scarcity. Consequently, it reduces the average latency of running applications on the edge platform with the assistance of the cloud cluster. Although several studies, like the ones discussed above, have addressed the migration and resource allocation in edge clusters, none focused on decreasing 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 considering migration and scheduling simultaneously on a cloud-assisted cluster aiming to achieve near-optimal resource allocation in Kubernetes-managed environments while accounting for the ever-changing dynamics of the cluster through migration of pods in a batched manner.

3. System model

Edge clusters are typically made up of resource-constrained, heterogeneous, failure-prone nodes placed at the network's edge, enabling the processing of user requests with reduced latency. To address these shortcomings, they are typically paired with cloud clusters consisting of nodes with more resources and fault-tolerant architectures, forming a more extensive cluster of cloud and edge nodes, often called cloud-assisted edge clusters. illustrates the cloud-assisted edge cluster with its components.

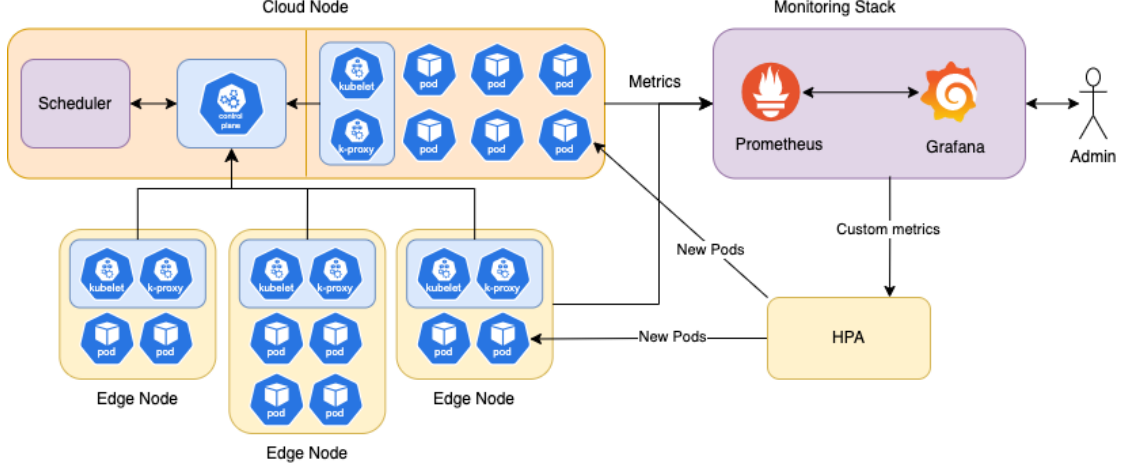


Figure 1: Kubernetes-managed cloud-assisted edge cluster model

Kubernetes, the de facto container orchestration standard, can manage application deployments and their life cycles on the cluster. As shown in Figure 1, several critical parts are necessary for Kubernetes to manage containers. These components generally include a scheduler, KubeProxy, a monitoring system, and horizontal pod autoscaling (HPA).

3.1. Scheduler

The scheduler is a critical Kubernetes control plane component responsible for assigning newly created pods to nodes. Through filtering and a scoring phase, the scheduler determines the node with the highest score for binding. The default scheduler installed alongside control plane components in Kubernetes is called a kube-scheduler [17]. By default, the Kube scheduler tries to distribute pods evenly across nodes to increase availability [18].

3.2. Kube Proxy

A component of Kubernetes that runs on each node and maintains network rules that allow connectivity between services and pods. This service can facilitate access to pods from inside or outside the cluster.

3.3. Monitoring Stack

Due to the failure-prone nature of edge nodes, a monitoring stack is even more essential to monitor each node's state and consider its state in different system behaviours. Prometheus [19] for metric recording, typically deployed alongside Grafana [20] for metric observability, is a common monitoring stack in the industry.

3.4. Horizontal Pod Autoscaler (HPA)

HPA is an API resource that scales the number of pods of a service based on observed CPU utilization, memory usage, or other custom metrics fetched from the monitoring stack. This capability allows the Kubernetes cluster to adapt to varying task

loads and optimize resource allocation by dynamically changing the number of replicas. HPA ensures efficient resource utilization and maintains application performance within the Kubernetes environment.

4. Problem Definition

This section defines the scheduling and migration problems of the proposed framework. 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 i th 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 megabytes.

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 i th service deployment. The scheduler is not responsible for determining the number of pods for each service, but must make decisions based on the assumption that this number can vary by HPA. It is also assumed that all pods of service i require the same resources presented in a two-dimensional vector R_{S_i} , where the dimensions correspond to the number of processor

Symbol	Definition
E	Set of all edge nodes
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 services
P_i	Set of all pods related to service i
R_{S_i}	Resource vector for pods of service i
Q_i	Quality of service guaranteed for service i
N	Set of all newly created pods, which is input to the scheduler
N_i	Set of all pods related to i th service
u	Allocation matrix for all pods
u'	Allocation matrix for all pods before scheduler decision
u_N	Allocation matrix for newly created pods
$u_{i,j,k}$	Boolean value stating that j th pod of i th service scheduled on the k th node
$u_{N_i,j}$	Boolean value stating that newly created pod of service N_i is scheduled on the j th node

Table 1: Summary of all notations

cores, and the volume of memory required by that service. Meanwhile, The response time of user requests serviced on the cloud and edge nodes are t_C and t_E respectively ($t_E \leq t_C$). The reason for considering a uniform response time for edge nodes is based on the assumption of a uniform probability of user presence throughout the entire geographic area covered by this edge cluster.

Furthermore, the service provider provides a QoS¹ guarantee level for each service, stating that the average response time for the i th service must be less than a fixed value Q_i .

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. N_i is the i th service in this collection in the input set. Additionally, the scheduler's output comprises two allocation matrices, u and u_N , with possible values of 0 and 1, where $u_{i,j,k}$ signifies, for all pods, whether the j th pod of the i th deployment has been placed on the k th node in the edge cluster. Furthermore, $u_{N_i,j}$ indicates whether or not the newly arrived i th pod has been allocated to the k th node 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 while performing the minimum number of migrations. The scheduler tries to establish the maximum number of QoS guarantees. If it cannot maintain a guarantee for a service, it strives to get as close as possible to establishing it. We know that the difference between the average expected response time of service i and the guaranteed value (Q_i) is equal to:

$$\Delta(i) = \frac{\sum_{j \in P_i, k \in E} (u_{i,j,k}) \cdot t_E + (|P_i| - \sum_{j \in P_i, k \in E} (u_{i,j,k})) \cdot t_C}{|P_i|} - Q_i$$

Now, we define the user satisfaction level for the i th service as follows:

$$F(i) = -ReLU(\Delta(i))$$

in which the ReLU function ensures that if the average time exceeds the guaranteed value, the satisfaction decreases linearly. Also, the number of migrations performed for the j th pod from the i th deployment can be defined as follows:

$$\theta(i, j) = \frac{\sum_{k \in E} |u_{i,j,k} - u'_{i,j,k}| + |\sum_{k \in E} (u_{i,j,k}) - \sum_{k \in E} (u'_{i,j,k})|}{2}$$

We also define the problem constraints as follows:

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

$$u_{i,j,k} \in \{0, 1\}$$

$$\forall i \in S, j \in P_i \left(\sum_{k \in E} (u_{i,j,k}) \right) \leq 1$$

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

$$\forall k \in E \left(\sum_{i \in S, j \in P_i} (u_{i,j,k} \times R_{S_i}) \right) \leq R_{E_k}$$

The scheduler aims to maximize user QoS and minimize migrations count subject to the above constraints:

$$Max_{u \in U} \left(\sum_{i \in S} F(i) \right)$$

¹Quality of service

$$\text{Min}_{u \in U} \left(\sum_{i \in S, j \in P_i} \theta(i, j) \right)$$

The presented problem is a MILP² 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 modules, functionalities, and design scheduling algorithms.

5.1. Scheduler Overview

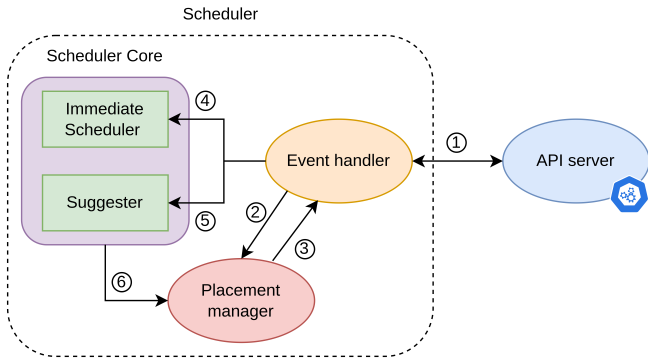


Figure 2: KubeDSM component diagram

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, representing the scheduler’s understanding of the current cluster state. The event handler continuously updates this cluster state over time.

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. *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 plans that are single-step and 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.

5.2. Algorithms overview

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

²Mixed-integer linear programming

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

Table 2: Possible steps, actions and their verification

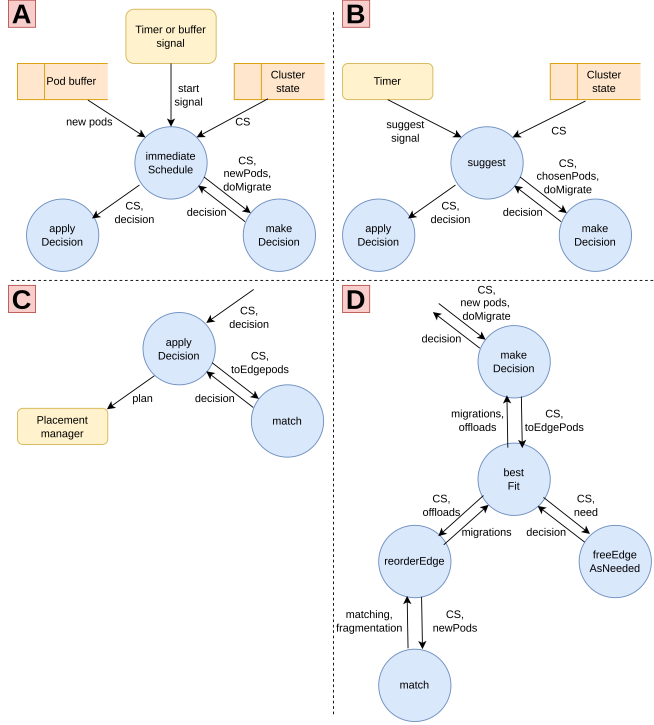


Figure 3: Scheduler Complete Call Graph

Algorithm 1: Immediate Schedule Algorithm

Input: Cluster State, newPods

```

1 Function ImmediateSchedule(CS, newPods):
2   decision ← makeDecision(CS, newPods, false);
3   applyDecision(CS, decision);

```

some pods inside edge (to decrease 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 `MAX_CLOUD_OFFLOAD` (TODO change it to a parameter) 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 func-

tion is defined as:

$$score(pod) = \frac{QoS(CS' - \{pod \text{ on cloud}\} \cup \{pod \text{ on edge}\}) - QoS(CS')}{size(pod)} \quad (1)$$

The score function can be viewed as QoS increased per resource unit moved to the edge. The former is calculated as difference of QoS between when the pod is on the edge and the cloud. The latter is achieved by dividing the score by the pod's size. The size of a pod is calculated as:

$$size(pod) = \sqrt{\frac{pod_{CPU}}{MAX_{CPU}} \times \frac{pod_{MEM}}{MAX_{MEM}}} \quad (2)$$

Algorithm 2: Suggest Algorithm

Input: Cluster State

```

1 Function suggest(CS):
2   candidPods ← all cloud pods;
3   chosenPods ← [];
4   CS' ← CS; // a copy of cluster state
5   currentFreeResources ← freeResources in edge;
6   while len(chosenPods) < MAX_CLOUD_OFFLOAD
   and len(candidPods) > 0 do
7     sort candidPods by score (Eq 1) in decreasing
       order;
8     firstCandidate ← candidPods.pop();
9     if firstCandidate.resources ≤
       currentFreeResources then
10      chosenPods += firstCandidate;
11      currentFreeResources -=
        firstCandidate.resources;
12      CS' ← CS' - {firstCandidate on cloud} ∪
        {firstCandidate on edge};
13    end
14  end
15  decision ← makeDecision(CS, chosenPods, true);
16  applyDecision(CS, decision);

```

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.

Algorithm 3: Make Decision Algorithm

Input: Cluster State, newPods, doMigrate

Output: bestDecision

```

1 Function makeDecision(CS, newPods, doMigrate):
2   bestDecision ← place all new pods on cloud;
3   foreach subset toEdge of newPods do
4     if sum of resources of toEdge > edge's resources
5       then
6         continue;
7     end
8     currentDecision ← deploying toEdge on edge
9       and other new pods on cloud;
10    if doMigrate then
11      migrationsNeeded ← bestFit(CS, toEdge);
12      currentDecision += performing
13        migrationsNeeded;
14    end
15    else
16      if sum of resources of toEdge > free
17        resources in edge then
18        continue;
19      end
20    end
21    if QoS(CS after bestDecision) > QoS(CS after
22      currentDecision) then
23      bestDecision ← currentDecision;
24    end
25  end
26  return bestDecision;

```

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 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 MAX_REORDER_MIGRATION (TODO change it to a parameter) 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.

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

Algorithm 4: Best Fit Algorithm

Input: Cluster State, toEdgePods

Output: migrations

```

1 Function BestFit(CS, toEdgePods):
2   offloads ← freeEdgeAsNeeded(CS, sum resources in
3     toEdgePods); // Offloads are migrations
4     from the edge to the cloud to increase
5     QoS.
6   CS' ← CS after offloads;
7   reorderings ← reorderEdge(CS');
8   // Reorderings are migrations inside
9     the edge for fragmentation reduction.
10  return offloads ∪ reorderings;

```

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:

$$score(pod) = \frac{QoS(CS' - \{pod \text{ on edge}\} \cup \{pod \text{ on cloud}\}) - QoS(CS')}{size(pod)} \quad (3)$$

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
7     currentFreeResources < need do
8     sort candidPods by score in decreasing order;
9     firstCandidate ← candidPods.pop();
10    freedPods += firstCandidate;
11    currentFreeResources +=
12      firstCandidate.resources;
13    CS' ← CS' - {firstCandidate on edge} ∪
14      {firstCandidate on cloud};
15  end
16  return freedPods;

```

reorderEdge: This function (Algorithm 6) aims to reduce fragmentation within edge nodes by identifying potential pod migrations within the edge. It evaluates each combination of edge pods, up to a size of MAXIMUM_MIGRATIONS (TODO change it to a parameter), and considers migrating that combination. Using the match algorithm, it determines the optimal target nodes for each pod. The resulting fragmentation is then calculated for each combination, and the combination with the

least fragmentation, along with its best target nodes, is selected as the optimal set of reorder migrations.

Algorithm 6: Reorder Edge Algorithm

Input: Cluster State
Output: bestMigrations

```

1 Function CS:
2   bestMigrations  $\leftarrow \{\}$ ;
3   leastFrag  $\leftarrow \text{frag}(\text{CS})$ ;
4   foreach combination comb of edge pods where
      | comb < MAXIMUM_MIGRATIONS do
5     CS'  $\leftarrow \text{CS} - \{\text{comb on edge}\}$ ;
6     mapping, _  $\leftarrow \text{match}(\text{CS}', \text{comb})$ ;
7     CS'  $\leftarrow \text{CS}' \cup \{\text{deploying } \text{comb} \text{ by } \text{mapping}\}$ ;
8     currentFrag  $\leftarrow \text{frag}(\text{CS}')$ ;
9     if currentFrag < leastFrag then
10      leastFrag  $\leftarrow \text{currentFrag}$ ;
11      bestMigrations  $\leftarrow \{(\text{pod} \rightarrow \text{mapping}[\text{pod}])$ 
        | for pod in comb};
12    end
13  end
14  return bestMigrations;

```

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 moving pods from cloud to edge, edge to cloud, or 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 respective 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'  $\leftarrow \text{CS}$  after migrating decision.migrations;
3   matching, remainingPods  $\leftarrow \text{match}(\text{CS}',$ 
      | decision.toEdgePods);
4   toCloudPlans  $\leftarrow \{\}$ ;
5   foreach pod in decision.toCloud  $\cup$  remainingPods
      do
6     toCloudPlans  $\leftarrow \text{append}(\text{toCloudPlans}, [\text{deploy}$ 
      | pod on cloud]);
7   end
8   toEdgePlan  $\leftarrow \{\}$ ;
9   foreach migration in decision.migrations do
10    toEdgePlan  $\leftarrow (\text{toEdgePlan}..., \text{then migration})$ ;
11  end
12  foreach pod in matching do
13    toEdgePlan  $\leftarrow (\text{toEdgePlan}..., \text{then deploy pod}$ 
      | 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 $dp[(i, \text{pods})]$ 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, 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:

$$dp[(0, 0)] = \text{frag}(\text{CS}) \quad (4)$$

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 i th node. We refer to this subset as cur_{pods} . First, it must be verified if cur_{pods} can be deployed on the i th 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 $dp[(i, \text{pods})]$ can be derived as:

$$\begin{aligned}
dp[(i, \text{pods})] = & \\
& dp[(i - 1, \text{pods} - cur_{\text{pods}})] \\
& + \text{frag}(\text{node}_i \cup cur_{\text{pods}}) - \text{frag}(\text{node}_i)
\end{aligned} \quad (5)$$

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

as:

$$dp[(i, pods)] = \min_{cur_{pods} \subseteq pods} (dp[(i-1, pods - cur_{pods})] + frag(node_i - cur_{pods}) - frag(node_i)) \quad (6)$$

The final solution is obtained by finding the largest subset of input pods (denoted as $final_{pods}$ such that $dp[(n, final_{pods})]$ 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.

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, meaning simple in Spanish), 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

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

Table 3: Summary of Nodes configuration used in our evaluations

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. Our custom load generation tool, DrStress [25], performs stress tests by simulating multi-threaded HTTP requests to our pods.

Algorithm 8: Match Algorithm

Input: Cluster State, newPods

Output: targets, $dp(N, max_sub_set)$

```

1 Function match(CS, newPods):
2   dp ← mapping from all states (i, pod_set) to  $\infty$ ;
3   par ← mapping from all states to None;
4   dp(0,  $\emptyset$ ) ← frag(CS);
5   par(0,  $\emptyset$ ) ← Nil;
6   for i ← 1 to N do
7     foreach pods  $\subseteq$  newPods do
8       foreach cur_pods  $\subseteq$  pod_set do
9         fragChange ← frag(CS[nodei]  $\cup$ 
10          cur_pods) - frag(CS[nodei])
11         achievedFragmentation ←
12           dp(i - 1, pods - cur_pods) + fragChange;
13         if dp(i, pods) > achievedFrag then
14           dp(i, pods) ← achievedFrag;
15           par(i, pods) ← (i - 1, pods - cur_pods);
16         end
17       end
18     end
19   end
20   max_sub_set ←  $\emptyset$ ;
21   foreach pods  $\subseteq$  newPods do
22     if dp(N, pods) =  $\infty$  then
23       continue;
24     end
25     if |pods| > |max_sub_set| or
26       (|pods| = |max_sub_set| and dp(N, pods) <
27        dp(N, max_sub_set)) then
28       max_sub_set ← pods;
29     end
30   end
31   targets ← {};
32   stateIterator ← (N, max_sub_set);
33   while stateIterator  $\neq$  None do
34     i, pods ← stateIterator;
35     cur_pods ← pods - par(i, pods);
36     foreach pod  $\in$  cur_pods do
37       targets[pod] ← nodei end
38       stateIterator ← par(i, pods);
39     end
40   end
41   foreach pod  $\in$  newPods - max_sub_set do
42     targets[pod] ← cloud node;
43   end
44   return targets, dp(N, max_sub_set);

```

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

Table 4: Summary of service deployments' resource requests

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 JSON files, which are used by DrStress 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.

6.3. Results

7. conclusion

there is something here: 5

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. Aboud, 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>

Paper	Enviromenent		Factors			Container placement	Migration Starategy						
	Kubernetes	cloud-assisted edge environment	Fragmentation	Quality of service	Fairness		Live-checkpoint	Event-based	Periodic	Real-time	Cloud to edge	edge to cloud	Edge to Edge
[?]	X	✓	X	✓	X	X	X	X	✓	X	X	✓	X
[?]	X	✓	X	✓	X	X	X	X	X	✓	X	✓	X
[?]	X	✓	X	✓	X	X	X	X	X	✓	✓	✓	X
[?]	X	✓	X	✓	X	X	X	X	X	✓	✓	✓	✓
[?]	X	X	X	✓	X	X	✓	✓	X	✓	X	X	✓
[?]	X	X	X	✓	✓	✓	X	X	X	✓	X	X	✓
[?]	X	X	X	✓	X	✓	X	X	X	✓	X	X	✓
[?]	✓	✓	X	X	✓	X	X	✓	X	?	✓	X	✓
[?]	✓	X	X	X	X	X	X	?	?	?	X	X	✓
[?]	✓	X	X	X	X	✓	X	X	✓	✓	X	X	✓
[?]	✓	X	X	X	X	✓	X	✓	X	X	X	X	✓
proposed approach	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 5: Related work