Journal of Grid Computing (2021) 19: 31
https://doi.org/10.1007/s10723-021-09573-z

# Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes

László Toka

Received: 3 November 2020 / Accepted: 21 June 2021 / Published online: 17 July 2021

**Abstract** Novel applications will require extending traditional cloud computing infrastructure with compute resources deployed close to the end user. Edge and fog computing tightly integrated with carrier networks can fulfill this demand. The emphasis is on integration: the rigorous delay constraints, ensuring reliability on the distributed, remote compute nodes, and the sheer scale of the system altogether call for a powerful resource provisioning platform that offers the applications the best of the underlying infrastructure. We therefore propose Kubernetes-edge-scheduler that provides high reliability for applications in the edge, while provisioning less than 10% of resources for this purpose, and at the same time, it guarantees compliance with the latency requirements that end users expect. We present a novel topology clustering method that considers application latency requirements, and enables scheduling applications even on a worldwide scale of edge clusters. We demonstrate that in a potential use case, a distributed stream analytics application, our orchestration system can reduce the job completion time to 40% of the baseline provided by the default Kubernetes scheduler.

L. Toka (✉)
MTA-BME Network Softwarization Research Group, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics,
Budapest, Hungary
e-mail: toka.laszlo@vik.bme.hu

**Keywords** Orchestration · Kubernetes · Edge computing · Resource provisioning · High-reliability · Low-latency

## 1 Introduction

Future applications, e.g., extended reality applications or 5G and beyond telco services, will require ultra-reliability and low-latency communication from the hosting compute and network infrastructure. Hence we are in the process of extending the traditional cloud with an emerging architecture. Edge computing, built on the fast access network of 5G, is capable of fulfilling such strict delay criteria. Remote edge nodes are prone to failures and their downtime might be longer than that of a central infrastructure, i.e., data centers. Therefore, while the edge deployment of compute elements of the service minimizes service delay, ensuring the high reliability of services is a challenge.

The evolution and the growing interest of virtualization technologies led to the appearance of centralized data centers that host cloud-native applications and have advanced infrastructure managers to ensure the seamless operation of those applications. Kubernetes has become the most popular cluster manager during the past 5 years: it is used primarily for orchestrating data center deployments running web applications. Its powerful features, e.g., self-healing and scaling, have attracted a huge community, which in turn, is inducing a meteoric rise of this open source project. We venture to reshape Kubernetes' "heart-and-soul", the scheduler, to be suited for an edge infrastructure and for delay-sensitive applications to be deployed on the edge of a global network. As the edge infrastructure is highly prone to failures, and is considered to be expensive to build and maintain, self-healing features must receive more emphasis than in the baseline Kubernetes, therefore a topology-aware system is needed that extends its widely-used feature set with regard to network latency.

Today's services strive for worldwide availability and geographic reach might be even more crucial in the future. In order for a system to meet all the requirements, e.g., low latency and high availability practically everywhere, it should have tens of thousands of computing nodes geographically distributed and fully connected to serve all the clients. Generally, we can state that managing such a huge infrastructure is far from trivial, and exacerbated by the geographical spread. Answering simple questions, like, how do we measure network characteristics effectively, how can we react to topology changes, do become more and more difficult. Besides availability, reaching high reliability is also challenging: service providers must ensure that they can respond to different failures, so their users will not be affected by a service outage for a long time.

In this article we give potential answers to the above challenges and provide a conceptual solution for the fulfillment of strict time criteria of future applications. We propose our advanced edge-scheduler that takes into account the underlying network latency, and the applications' latency requirement in the scheduling decisions about the application components. Our backup resource multiplexing technique provides high reliability for the applications with aware of latency requirements by carefully provisioning resources for this aim. The system works on large scale with huge number of worker nodes and service requests thanks to our dynamic clustering method that can also organize a federated system dynamically. As a proof of concept, we implement the edge-scheduler in Kubernetes, and evaluate its performance in selected scenarios.

The paper is organized as follows. In Section 2 we introduce our model for ensuring high reliability and ultra-low latency with economical edge resource provisioning. In Section 3 we show our proposed scheduler solution that is based on an advanced heuristic scheduling algorithm, which dynamically handles incoming events of a geographically widespread virtual infrastructure, supporting several latency critical 5G applications. In Section 4 we present the operation of our re-scheduler that further decreases the provisioned resources in our system periodically, in an offline orchestration operation. We address scalability and present our edge node clustering solution based on network delay in Section 5. In Section 6 we describe the implementation choices we made and we present our experiment results in Section 7. We discuss related prior art in Section 8, and we conclude the paper in Section 9.

## 2 Scalable and Economical Edge Scheduling for Latency-Critical and Operation-Critical Applications

Our proposed concept turns a virtual infrastructure scheduler into a manager of geographically widespread infrastructure. It is built on two advanced scheduling algorithms that support latency critical applications. In this section we introduce our proposition for reserving backup resources, and we build a model for describing the problem of minimizing the amount of those while conveniently providing reliability for delay-critical applications on top of them.

In the architecture of the system we call the physical entities with computational resources, processor, memory, network bandwidth, as nodes. In this sense, a node can represent a single server at the edge of the network, or an abstraction of an entire cloud data center. Since our system considers network latency in every aspect of application deployment, we use a delay matrix: the values in the delay matrix represent the smallest delay value between each node pair. The deployable units of application components are called as Pods. The users can define delay criterion for each latency critical Pod which gives the maximum network latency that is tolerated by the application from an arbitrary point defined by the application provider, which we call *origin*.

**Definition 1** The application provider submits its requests to deploy each and every Pod in the system along with the Pod's origin and with the respective

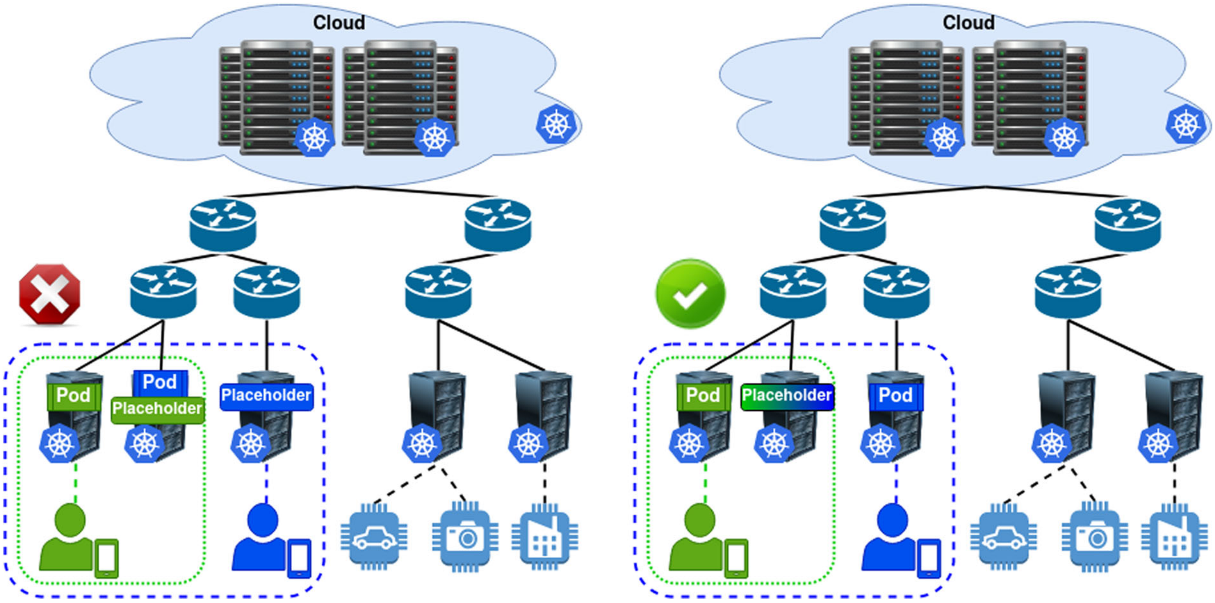

**Fig. 1** Proposed system architecture design for edge computing

radius. The origin of a Pod can be defined either as one of the nodes (that is close to the location of the users that the Pod will serve), or as another Pod deployed previously in the system (with which affinity is required for the Pod currently being deployed).

To provide high reliability for the applications, we provision backup compute resources on edge nodes, which we call *placeholders*. We prepare for only one node failure at a given moment, so we dimension the placeholders for the maximum number of Pods on any node to fail at once. Therefore, each placeholder has a resource demand depending on two factors: i) how many Pods it supports simultaneously; ii) how the backed up Pods are distributed among the nodes. A placeholder's size is not necessarily equal to the sum of the supported Pods' sizes: it can be less if the supported Pods are placed on different nodes. A Pod's placeholder must be assigned to a node that differs from the host of the Pod, and the placeholder must also fulfill the Pod's latency requirement.

Since edge nodes are prone to failures and we strive to ensure high reliability for all the applications, we want to make sure if a single node failure occurs, placeholders in the system have enough reserved resource to restart all Pods of the failed node. We consider that the resources on edge nodes are expensive, since edge nodes have limited resource capacity compared to the large data centers.

The Pods' have computational characteristics that our system needs to ensure, i.e., processor, memory, network bandwidth. Regarding of these two properties, one of our main goals is to minimize the resources reserved for placeholders in the system.

An example view of our system architecture with two simple scheduling result is presented in Fig. 1 with a central cloud, and several edge nodes. On the left side of Fig. 1, a non-optimal scheduling example is presented. In this case, the amount of backup resource (placeholder) reservation is greater than what the optimal solution would need. The result of how an advanced scheduler would deploy both the Pods and the placeholders can be seen on the right side. Since both of the Pods have common servers in their latency radius, their placeholders can be multiplexed in order to decrease the provisioned extra resources but still high reliability is ensured for the Pods. Our edge-scheduler, an advanced scheduler extension, integrates and improves the rudimentary solutions described in [8, 27]. The architecture of our edge-scheduler with some mandatory operations of each component is visualised in Fig. 2. The main components in our edge-scheduler are the Monitoring, Event handler, Clustering, Scheduler, and Re-scheduler. We introduce these components throughout the next section.

## 3 Scheduler: Online Pod Scheduling for Fulfilling Delay Requirements

Our online scheduler component is in charge of deploying the incoming Pod requests on the fly, with the awareness of their delay and computational requirements, and also of deploying respective placeholders for ensuring high reliability. It works in polynomial time and its approximation ratio is 3 in terms of total amount of placeholder resources sacrificed for guaranteeing high reliability against single node failures.

### 3.1 The operation of our online Pod scheduler

The scheduler works in an online manner, it processes the users' application requests one-by-one at the time of their submission. The major steps of scheduling are showcased in the schedule box in Fig. 2 and Algorithm 1.

Since our scheduler must give a solution that meets the delay requirements, the scheduling starts with the identification of the options that the Pod's requirements allow. More precisely, it pinpoints the nodes that are in the radius of the given origin (Line 8 of Algorithm 1). In case of the origin is a Pod, the algorithm defines its current host and gets the nodes around that. It is possible that none of the nodes in the radius have enough computational resource for the new application, i.e., processor, memory, network bandwidth. If none of the listed nodes have enough computational resource, our algorithm tries to migrate Pods from their current hosts to somewhere else, in order to free up some resource for the actual request (Line 11 of Algorithm 1). We present the dynamic operational challenges that our algorithm must solve, i.e., migration and fail-over, in Section 3.4.

**Algorithm 1:** Online schedule algorithm.

```
   Data: pod, nodes
   Result: pod.node and pod.placeholder are set
1  if pod.reschedule then
2      bind(pod, pod.target);
3      pod.reschedule ← False;
4  end
5  if pod.placeholder then
6      patch(pod, pod.placeholder);
7  else
8      nodes_in_radius ← nodes.filter(pod.origin,
       pod.delay) ;
9      suitable_nodes ←
       nodes_in_radius.filter(pod.capacity);
10     while suitable_nodes.size = 0 do
11         nodes_in_radius ←
           find_and_migrate_pods(nodes_in_radius) ;
12         suitable_nodes ←
           nodes_in_radius.filter(pod.capacity);
13     end
14     if suitable_nodes.size > 1 then
15         hosting_nodes, placeholder ←
           find_placeholder(suitable_nodes, pod) ;
16     else
17         No placeholder will be assigned to this
           Pod! ;
18     end
19     if hosting_nodes.size > 0 then
20         chosen_node ←
           min_utilized(hosting_nodes) ;
21     else
22         chosen_node ←
           min_utilized(suitable_nodes) ;
23     end
24     bind(pod, chosen_node);
25     if placeholder then
26         patch(pod, placeholder) ;
27     else
28         create_placeholder(suitable_nodes, pod) ;
29     end
30 end
```

During the Pod scheduling, we have to keep in mind the scarcity of the edge resources. Therefore, our algorithm first tries to place each Pod without increasing the total placeholder size in the system, keeping in mind the delay requirement (Lines 15 and 26 of Algorithm 1). When we can not find any solution that keeps the total backup resource size intact, we have to deploy the Pod first and then create a new placeholder or increase an existing placeholder's size for the Pod (Line 28 of Algorithm 1). Our node selection strategy for Pods favors dispersing them among nodes, leading to a balanced utilization in the system, which in turn decreases the necessary placeholder resources to support single node failures (Lines 20 and 22 of Algorithm 1). In contrast, the placement of placeholders favors those nodes that have a high number of nodes in their vicinity in terms of delay: these "central" nodes

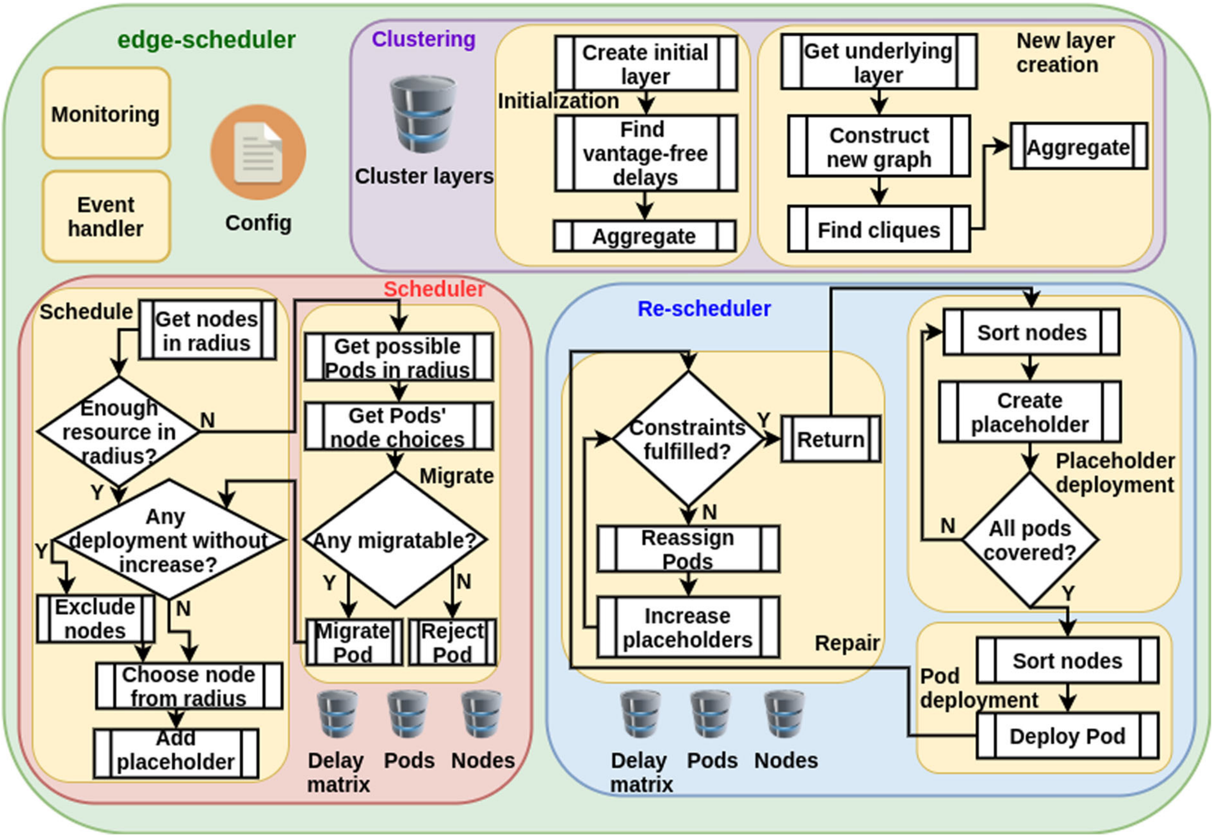

**Fig. 2** The components of our edge-scheduler

are good choices for placeholders, since they can support Pods on many nodes around them. The scheduler does not cover a Pod with a placeholder if the available computational resources do not allow the placeholder creation or size increase, or the delay requirement is so strict that only the starting node appears in the radius (Line 17 of Algorithm 1).

### 3.2 Complexity Analysis of our Proposed Scheduler

The scheduler algorithm processes the incoming Pod requests at the time of their arrival. Therefore, in a globally available system, the scheduler component need to act fast when a request comes in. We state that our scheduler runs in polynomial time, which we state in Theorem 1.

**Theorem 1** *Our proposed online scheduling algorithm has polynomial complexity.*

*Proof* Let us denote the set of nodes with $N$ and the set of Pods with $P$. In the beginning, getting the nodes in the radius around the origin can be done in $O(|N|)$. Then, the online scheduling algorithm tries to deploy the incoming Pod without increasing the total placeholder size in the system. Regarding that, the algorithm collects the placeholders in the radius and checks the network and computational constraints. This collection and constraint check have the following complexity: $O(|N|^2 + |P|^2)$. In the next step, the algorithm sorts the nodes based on their number of deployed Pods and their number of network connections that fulfill the delay requirements. In the worst case scenario, the algorithm has to do this sorting two times, which means, its complexity can be approximated with $O(2|N| \log |N|) = O(|N| \log |N|)$. After the sorting, the selection of the best fitting node and the deployment takes constant time. To summarize, the complexity of our online Pod scheduling algorithm (without migration) can be approximated by $O(|N|+|N|^2+|P|^2+|N|\log|N|) = O(|N|^2+|P|^2)$, which equals with $O(|N|^2)$ when $|N| > |P|$, and $O(|P|^2)$ when $|N| < |P|$. □

### 3.3 Approximation Bound on Placeholder Provisioning

In this section we prove that our scheduler is a 3-approximation algorithm in terms of the amount of placeholder allocation for Pods. As the first step, let us create a graph $G = (V, E)$, where the vertices represent the nodes and the edges of the graph present the connection between the nodes. We denote the set of Pods as $P$. In the proofs of the approximation we use the graph's diameter $d(G)$, which is the length of $max_{u,v \in V} d(u, v)$ the "longest shortest path" between any two graph vertices $(u, v)$, where $d(u, v)$ is the distance between the vertices. We define the group of vertices that we call *buds* in Definition 2.

**Definition 2** A vertex is a bud, if it connects to at least one leaf.

Furthermore, we make an assumption about the graph model and the latency requirements of the Pods in order to render the approximation analysis of our scheduler algorithm analytically tractable. The first part of the assumption is about the size of the topology and the resource capability of each node. The second part simplifies the number and the requirement of Pods to be deployed.

**Assumption 1** $G$ is a simple, connected graph, with $|V| = n > 3$, each vertex in $G$ represents a node in the Kubernetes cluster and has infinite capacity. Edges in $G$ represent unit latency distance between the vertices. $|P| = |V|$, moreover each Pod $p \in P$ has unit resource requirement, i.e., homogeneous Pod sizes, and there is a one-to-one mapping between the Pods' origins and the vertices in the graph: $p_i \rightarrow v_i$; $p_i \in P, v_i \in V$, i.e., every vertex is origin for a Pod. The delay requirement of each Pod makes the neighboring nodes of the Pod's origin eligible, no other nodes, i.e., nodes farther than 1 hop yield too much delay for the service deployed in the Pod.

Note that in the following we consider Assumption 1 to hold. It is partly a relaxing assumption, e.g., in terms of Pod-, and placeholder placement as infinite node capacities are supposed, but partly specific, e.g., in the aspect of origin selection. In terms of latency requirements, the assumption considers an extremely restrictive scenario.

The goal of an economical scheduler is to find the minimum amount of placeholders that can support all Pods in the system in case of one node's failure. Let us denote by $OPT$ the optimal solution and by $HEUR$ the solution that our online scheduler algorithm yields. Let us denote the number of buds as $b$, and the diameter of the graph $d$. The lower bound of the optimal solution can be deduced from the number of buds and the diameter of the graph. Therefore, we define the lowest amount of placeholders that can be theoretically achieved in Lemmas 1 and 2 using the diameter and number of buds respectively.

**Lemma 1** $OPT \geq \frac{d+1}{3}$

*Proof* $G$ with diameter $d$ has at least one shortest path with length $d$ and must have $d + 1$ nodes. Thus, there is a subgraph $G'$ in $G$ that can be represented as a path graph, which has $d + 1$ vertices. The Pods' delay requirement allows only the origin node and its neighbors (see Assumption 1) as their hosting node. Since every node is an origin for a Pod, the number of vertices in each Pod's radius (whose origin is in $G'$) is 2 or 3 in $G'$.

In the path graph representation the minimum number of sets that cover all nodes at least once and each set contains only neighboring nodes, equals to dividing the nodes into groups of three. One can see that the number of sets gives the minimum number of placeholders in $G$, that should be deployed. □

**Lemma 2** $OPT \geq b$

*Proof* We know that a bud is connected with at least one leaf, and each Pod's latency constraint allows only the neighbors of the origin node. Therefore, only two nodes (a bud and the leaf) are in the radius of the Pods, whose origin node is a leaf. Regarding that, one of the nodes in each bud-leaf pairs must hold a placeholder. From this statement, one can see that the number of placeholders must be greater or equal to the number of buds. □

We state, with Lemma 3, that our heuristic solution will have at least one Pod, which shares its placeholder with at least one other Pod.

**Lemma 3** $HEUR \leq n - 1$.

*Proof* $HEUR \leq n$, as $|P| = n$. By the heuristics applied in our online scheduling algorithm, equality occurs only in the case when placeholders cannot be multiplexed. This would occur only in a $G$ with 1-degree vertices, which is impossible with $n > 3$, hence the statement. □

In order to prove the approximation bound of our scheduler, we have to identify the proportion between: i) the diameter and the optimal amount of placeholders; ii) the number of buds and the amount of placeholders provided by our heuristic solution. Therefore in Lemma 5 and Lemma 6 (in the Appendix) we prove that the number of placeholders is directly proportional with the number of buds and the diameter value, as well.

Simple, connected graphs can have diverse combinations of diameter value and number of buds that affect the number of placeholders provisioned in the system. In Lemma 7 we present the possible graph architectures that simple, connected graphs can have with diverse diameter and bud value combinations. From Lemma 5 and Lemma 6 we can draw the following relationship:

$$\max\left(\frac{HEUR}{OPT}\right) \propto \max\left(\frac{d}{b}\right).$$

Based on this observation, we define the approximation bound of our heuristic solution in the combinations of diameter and the number of buds where the latter is minimal and the former is maximal. To summarize the previously presented results, we state and prove the approximation bound of our scheduler algorithm in Theorem 2.

**Theorem 2** *Our online scheduling solution is a 3-approximation algorithm for providing joint placement of placeholders of Pods ($HEUR \leq 3OPT$).*

*Proof* In Lemma 8 (in the Appendix) we prove that on all possible inputs, the approximation ratio between our heuristic solution and the optimal solution is always less than or equal to 3. Therefore, our scheduler is a 3-approximation algorithm in terms of the amount of placeholder allocation under Assumption 1. □

### 3.4 Pod Migration and Fail-Over

There are certain dynamic operational challenges that scheduling algorithms must face; for remedy we propose a migration policy. Network-aware migration of deployed Pods is triggered when a new Pod request comes in, but the available resources are not sufficient. In these situations we migrate the affected Pods to new nodes to avoid disruptions. The major steps of migration, and the flow of the process between them is presented in the “Migrate” box in Fig. 2. Although we strive to make room for the incoming Pod in the system, we migrate Pods only if their relocation frees enough resources and their assigned placeholders’ size remains the same. While the online scheduling will inevitably lead to suboptimal resource allocation for the placeholders, i.e., more resources will be dedicated to backup than the absolute minimal amount at the highest attainable multiplexing scheme for single node failures, we are not sure how often migration events will need to take place. As the authors of [9] argue, edge computing is the strongest candidate for providing low-latency responses, but it is not yet clear what edge infrastructures will be like. In addition to that, the edge applications’ dynamics and their latency requirements will greatly affect the frequency of migrations.

Our solution can also handle topology changes dynamically. The fail-over process is triggered, when our scheduler perceives that a worker node is unreachable, or a delay deterioration in the infrastructure spoils Pods’ delay constraints. In these cases we use the already provisioned placeholders to restart the respective Pods within their placeholders’ resources. After the restart, we remove the Pods from their original placeholders, and try to find or create new placeholders for them.

Both Pod migration and fail-over appear in our online algorithm. Since we consider the delay requirements as hard constraints, both of these methods take the delay requirements into account. Our re-scheduler operates on all Pods at once, which renders migration or fail-over meaningless during its execution.

### 3.5 Complexity analysis of Pod migration calculation

In every system, the migration of virtual entities, e.g. Pods, is an expensive process in terms of execution time and operational steps. Although it is a costly operation, we show that our Pod migrating algorithm runs in polynomial time, and we prove its polynomial complexity in Theorem 3.

**Theorem 3** *The migration calculation in our scheduler has polynomial complexity.*

*Proof* Let us denote the set of nodes with $N$ and the set of Pods with $P$. In the migration process the algorithm knows the new Pod (that cannot be deployed in the system due to lack of resources) characteristics and the nodes that are in the latency radius of the Pod's origin node. Our solution iterates over all those nodes' Pods and try to migrate them till one of the nodes has enough free resource to host the new Pod. This means in the worst case we have to try the migration in $O(|P|)$ times. When we examine a Pod if its "migratable", we check the following constraints: i) the actual node will have enough free resource for hosting the new Pod, in case we migrate the examined Pod to another node (can be done in $O(1)$); ii) at least one of the nodes in the examined Pod's radius has enough free resource for that Pod ($O(|N|)$); iii) when a placeholder is assigned to the examined Pod, we do not have to increase its size if we deploy the Pod to a new host (if $|P| > |N|$ then $O(|P|^2)$, otherwise $|N|^2 log|N|$). If all constraints are met, we migrate the examined Pod, so we can deploy the new Pod to its original host. The complexity of Pod migration is $O(|P||N|^2 log|N|)$ in cases, when $|P| < |N|$, else ($|P| > |N|$) it is $O(|P|^3)$. □

As for the technical migration overhead, we argue that stateless [26] application components can be migrated with minimal extra resources. The stateless design, of course, must be supported by a distributed cloud database [24, 25], which transforms the punctual migration overhead into a continuous synchronization of application states onto multiple database instances running on nodes potentially hosting the stateless application, which leads to an extra consumption in terms of compute, memory and network resources.

## 4 Re-scheduler: An Offline Orchestrator to Minimize Provisioned Backup Resources

Operating besides the scheduler, our re-scheduler is responsible for the offline minimization of the total provisioned backup resources in the system. The main difference between the two solutions is in the submission pattern of the Pods. While the scheduler works in an online manner, the re-scheduler better approximates the minimum amount of necessary placeholders as it works in an offline manner and it is fed with the batch of all deployed Pods.

### 4.1 The Operation of our Re-Scheduler

Our re-scheduler has three major phases: i) placeholder deployment; ii) Pod deployment; iii) repair phase. The flowchart of the phases are presented inside the "Re-scheduler" box in Fig. 2. As for the first phase, according to our intuition, the nodes that could host the most Pods are the best choices for placeholders: placeholders on them can cover all those Pods if they are placed elsewhere, which maximizes the multiplexing effect, hence the least possible resources reserved for placeholders. Therefore in the first phase, as shown in Algorithm 2, we reserve the minimum amount of placeholders on the nodes (Lines 2 and 3 of Algorithm 2) that could possibly host all Pods to be deployed.

**Algorithm 2:** Offline placeholder deployment.

```
   Data: pods, nodes
   Result: pod.placeholder are set for all pods
1  while pods do
2      node ←
       nodes.sort(by_number_of_deployable_pods,
       decreasing_order, pods).first ;
3      placeholder ← create_placeholder(node,
       maximum_capacity(node.deployable_pods)) ;
4      for pod in node.deployable_pods do
5          patch(pod, placeholder);
6          pod.placeholder ← node;
7      end
8      pods.remove(node.deployable_pods);
9      node.deployable_pods ← ∅;
10 end
```

The deployment of Pods that can be hosted only on a subset of nodes, e.g., in a strict latency radius, is challenging. The order of Pod deployment follows the number of possible nodes that could host a Pod (Line 6 of Algorithm 3), which mainly corresponds to the tightness of their delay requirements. We deploy the Pods with the fewest options first, then move forward to Pods with looser latency requirements. At the end of this phase, each Pod is deployed and all of them are covered with a placeholder as Algorithm 3 indicates.

**Algorithm 3:** Offline pod deployment.

```
  Data: pods, nodes
  Result: pods are deployed
1 foreach pod in pods do
2     nodes_in_radius ← nodes.filter(pod.origin,
      pod.delay);
3     suitable_nodes ←
      nodes_in_radius.filter(pod.capacity);
4     pod.choosable_nodes ←
      suitable_nodes.filter(pod.placeholder);
5 end
6 foreach pod in
  pods.sort(by_number_of_choosable_nodes,
  decreasing_order) do
7     chosen_node ←
      min_utilized(pod.choosable_nodes);
8     bind(pod, chosen_node);
9 end
```

Pod migration is an expensive operation since during the migration the behavior of the application can be non-deterministic and the service provider has to guarantee the seamless relocation of the components. Therefore, the cost of migration is not negligible in the minimization process in our re-scheduler. When the re-scheduler is triggered, a deployment that defines the host node, determined by the online scheduler, is in effect for each submitted Pod. Relying on that predefined deployment, in the second phase the scheduler strives to deploy the Pods on those nodes that already host the Pod. Due to this behavior, our solution can minimize the number of migrations while still minimizing the amount of total provisioned resources.

It is possible that some of the reserved placeholders' size might not be enough to back up all the Pods which have been assigned to them. In order to ensure full reliability, we have to repair those failed placeholders, one input to Algorithm 4. Since we minimize the total footprint of provisioned placeholders, we reassign Pods from each failed placeholder to other placeholders, or re-deploy them to other nodes if migration is favored (Line 2 of Algorithm 4). If we can not find any solution that would keep the amount of total placeholder size on the same level, we have to increase the broken placeholders' size (Line 6 of Algorithm4), or instantiate new placeholders (Line 10 of Algorithm 4).

**Algorithm 4:** Offline repair.

```
   Data: bad_placeholders, nodes
   Result: no bad placeholders left
 1 foreach placeholder in bad_placeholders do
 2     success ← reassign(placeholder, nodes) ;
 3     if success then
 4         bad_placeholders.remove(placeholder);
 5 foreach placeholder in bad_placeholders do
 6     success ← increase(placeholder) ;
 7     if success then
 8         bad_placeholders.remove(placeholder);
 9 foreach placeholder in bad_placeholders do
10     success ← create_placeholder(placeholder) ;
11     if success then
12         bad_placeholders.remove(placeholder);
```

### 4.2 Complexity Analysis of our Re-Scheduler

Although our re-scheduler works in an offline manner, i.e., it processes the batch of all the deployed Pods' requirements, we must not let its execution time increase unpredictably. The state of the system is continuously changing: Pods come and go, nodes might fail. If such events occur while the re-scheduler is running, the placement result yield by the algorithm may not be valid anymore. Therefore it is of paramount importance to design the re-scheduling algorithm to be fast. In Theorem 4 we prove that our proposed re-scheduler algorithm runs in polynomial time.

**Theorem 4** *Our proposed re-scheduler algorithm has polynomial complexity.*

*Proof* Let us denote the set of nodes with $N$ and the set of Pods with $P$. In the re-scheduler algorithm's first phase we deploy the placeholders. As the first step, it calculates the nodes in each Pod's radius. This calculation can be estimated with $O(|P||N|)$. After the calculation, the re-scheduler sorts the nodes by the number of the Pods that could be hosted on them. The complexity of this sorting is $O(|N|^2)$. In the next step, it deploys each placeholder and reorders the list after every deployment. This step and the whole first phase can be estimated with $O(|P||P||N|(|N|-1)) = O(|P|^2|N|^2 - |N|) = O(|P|^2|N|^2)$.

In the second phase the re-scheduler places the Pods on the nodes. First, it sorts the Pods by their number of fitting nodes. The complexity of the sorting is $O(|P|^2)$. Then, the algorithm iterates through the sorted Pods, and deploys them on the least utilized node. The worst case complexity of this iteration can be estimated by $O(|P||N|)$. The worst case complexity of the whole second phase, if $|P| > |N|$, is $O(|P|^2)$, anyway the worst case complexity is $O(|P||N|)$.

In the third phase, the algorithm checks if any broken placeholders exist, and repairs the failed ones. When it checks the placeholder constraints, it iterates through all the placeholders and for each of them it also iterates through all the nodes and the Pods. The worst case complexity of this validation is $O(|P||N|^2)$, as in the worst case each node contains one placeholder. Then, the re-scheduler goes through all the broken placeholders, and tries to fix them with Pod reassignment to different, already instantiated placeholders, or to other nodes. During the reassignment for each broken placeholder, the algorithm gets those nodes whose constraints are not fulfilled and tries to reassign the specified Pods. In case of $|P| > |N|$, the reassignment's complexity is $O(|P||N|^2)$, otherwise the complexity is $O(|N|^3)$. In the second repair method the algorithm increases each broken placeholder's size, if possible, which has the worst case complexity of $O(|P||N|)$. The last repair attempt is the new placeholder, which has the same worst case complexity as the Pod reassignment $O(|P||N|^2)$or $O(|N|^3)$. If $|P| > |N|$ the worst case complexity of the third phase equals to $O(|P||N|^2 + |P||N|^2 + |P||N| + |P||N|^2) = O(|P||N|^2)$, otherwise it is $O(|P||N|^2 + |N|^3 + |P||N| + |N|^3) = O(|N|^3)$.

Summarizing, our proposed heuristic algorithm's complexity in the worst case scenario equals $O(|P|^2|N|^2 + |P|^2 + |P||N|^2) = O(|P|^2|N|^2)$ in case of $|P| > |N|$, otherwise $O(|P|^2|N|^2 + |P||N| + |N|^3) = O(|P|^2|N|^2 + |N|^3)$. □

## 5 Providing Scalability with Node Clustering

As the size of the system grows, not only finding the best placement for the service components, but even measuring the network characteristics becomes challenging. The continuous measurement is induced by the fact that the underlying network and topology may change, and such events can cause application failures and delay requirement violations. Our clustering solution not only reduces the overhead of determining network characteristics, but also helps the scheduling of the applications by compressing the topology. By topology compression, we mean that since a clustering algorithm forms groups (clusters) from a set of nodes, the scheduling algorithms do not need to iterate through all the nodes, it is sufficient to calculate with the clusters. Furthermore, our solution provides valuable input for service providers who want to implement self-organizing network features to dynamically organize their federated system hierarchically, based on their network topology.

### 5.1 Dynamic, Delay-Based Clustering Problem

We propose a clustering algorithm that groups the physical nodes into clusters in order to ensure that dynamic application placement and network delay measurements can scale effectively in large topologies. Our solution is an agglomerative clustering algorithm that creates cluster layers hierarchically, where each layer contains clusters that are constructed with a new delay requirement belonging to a Pod request. The input of our clustering algorithm is a topology (that maybe already clustered before), and a set of delay values that will be used for clustering the nodes (or the clusters) inside topology. In this agglomerative clustering approach, each node starts in its own cluster, then the clusters are merged as we build up the hierarchy, where each layer (and their clusters) are built based on increasing delay requirement values.

In cases when the topology is not clustered with a given delay value yet, a new layer is being created (visualized in Fig. 2, in "New layer creation" box inside the “Clustering” box) relying on the underlying layer that is clustered with the delay that is the greatest and closest, but still less than the new one. Regarding the application placement, our clustering mechanism guarantees that all nodes inside a cluster fulfill a Pod’s delay requirement in case the cluster is an output of the clustering with that delay value. Hence service delay requirement violations can not happen inside a cluster for the given delay value, no matter which member node hosts the given Pod.

The clustering may lead to different outcomes, in which a node may belong to different clusters. An illustrative example can be viewed in Fig. 3, where on the top, a simple topology is depicted with delays on network connections between the nodes. On the bottom, we present how the topology can be clustered based on different delay requirements. The red circled clusters (bottom middle) are non-deterministic, since they overlap each other and the clustering result depends on the processing order of the nodes.

There are some delay values (denoted as $d$ in Fig. 3), for which the clustering is deterministic, i.e., each node belongs to only one determined cluster no matter the order of nodes during the clustering process. Formally we define this problem in Definition 3.

**Definition 3** Given a $G = (V, E)$ graph and a $d$ delay value. $G$ is an undirected complete graph with weighted edges that fulfill the triangle inequality and $d$ is a positive number that represents the delay requirement of a service.

DETERMINISTIC CLUSTERING PROBLEM: Can $G$ be clustered based on $d$ in a deterministic manner?

A key feature of our solution is to find those delay values (for the given topology) that provide such deterministic clustering results. We call these delays as *vantage-free delays* and we seek these delays at the initialization of the clustering component (“Initialization” box inside “Clustering” box in Fig. 2). These *vantage-free* clustering layers can be defined and

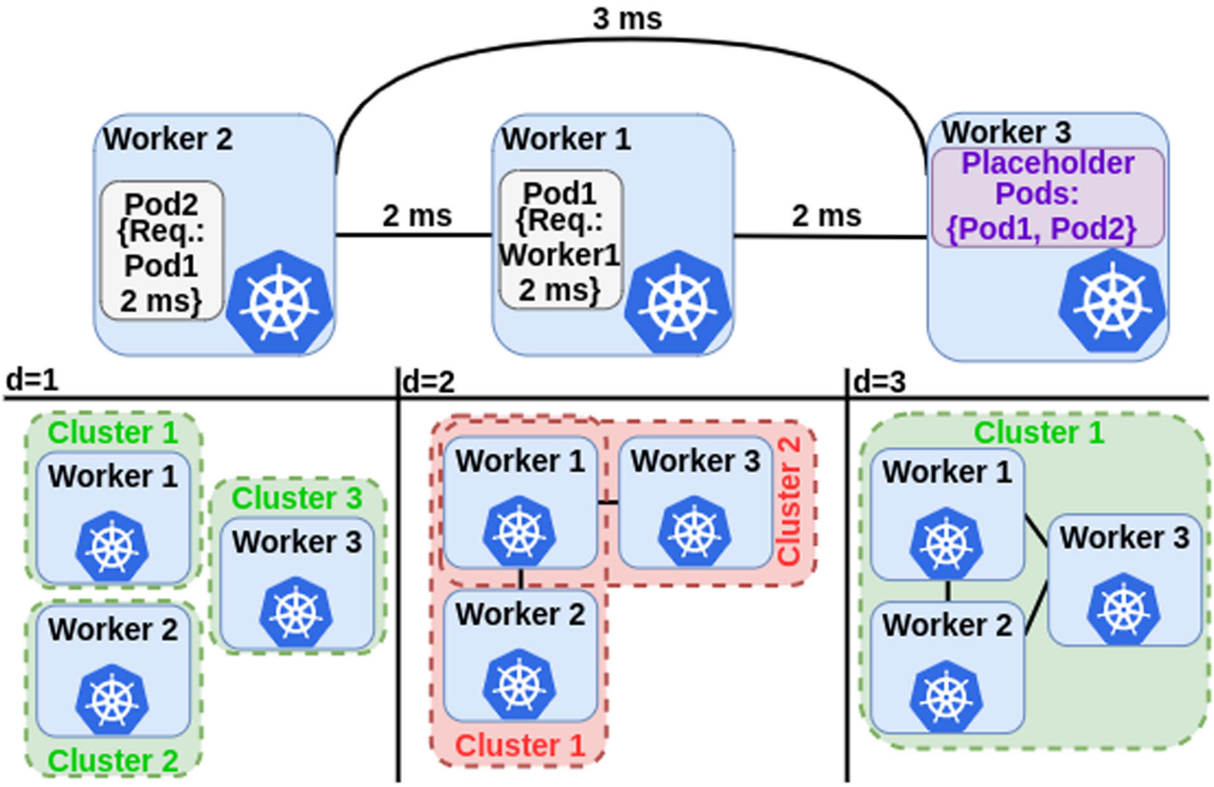

**Fig. 3** Deterministic and non deterministic clustering examples

created in polynomial time by using only the delay values that appear between the nodes in the topology (see Lemma 4).

**Lemma 4** *We can find an answer for the* DETERMINISTIC CLUSTERING PROBLEM *in polynomial time.*

*Proof* Let us represent our topology with a complete graph $G$, where the nodes are the vertices and the edges are the smallest available latency values between each node pair. The proof consists two steps. First, we delete the edges from $G$ if their weight is greater than $d$, in $O(|V|^2)$. Then, we examine each disconnected component, if they are complete subgraphs. This examination can be done in $O(|V|^2)$. If all of the disconnected components are complete subgraphs (cliques), the output is positive (yes), $G$ can be clustered based on $d$ deterministically, otherwise it cannot. □

The purpose of defining these delay values and their clustering layers is that they serve well as underlying layers for clustering with other, non *vantage-free delay* values, since these *vantage-free delays* do not change unless the topology changes. In order to support the large scale scheduling, the purpose of our clustering solution is to create the least clusters that cover all the nodes and the nodes cannot violate the given delay requirement within their cluster. We call this problem as DELAY BASED CLUSTERING PROBLEM. A formal definition of the DELAY BASED CLUSTERING PROBLEM is given in Definition 4. We state and prove that DELAY BASED CLUSTERING PROBLEM is a hard problem in general in Theorem 5.

**Definition 4** Given a graph $G = (V, E)$ that represents the physical topology, and $d$, a positive number that equals to the delay requirement of the service. $G$ is an undirected complete graph, whose edges fulfill the triangle inequality.

DELAY BASED CLUSTERING PROBLEM: Clustering the vertices with minimum number of clusters with the awareness of the following requirements: i) a cluster has to be a clique; ii) the weight of every edge inside the clusters is less than or equal to $d$; iii) clusters can not overlap with each other.

**Theorem 5** *The* DELAY BASED CLUSTERING PROBLEM *is NP-hard.*

*Proof* In the proof of this theorem, we use Karp-reduction for a known NP-hard problem, the CLIQUE COVER PROBLEM, which is the algorithmic problem of finding a minimum clique cover. A clique cover of a given undirected graph is a partition of the vertices into cliques.

As a preparation step, we construct $G'$ with deleting all of the edges with greater weight than $d$ from $G$, since those edges surely will not appear inside any cluster. After this step, we can ignore the edge weights in $G'$. In this case we strive to find the minimum number of not overlapping clusters that are cliques, and cover all the vertices. Note that, since service delay requirement violations can not happen inside a cluster for the given delay value, the clusters can only contain cliques in our solution. Let $G'$ be the input of the clique cover problem. In this case, one can see that finding the minimum clique cover (the clique cover that uses as few cliques as possible), equals with finding the minimum clusters (that are also cliques) that solves the DELAY BASED CLUSTERING PROBLEM. Also a solution for DELAY BASED CLUSTERING PROBLEM gives a good clique cover for the MINIMUM CLIQUE COVER PROBLEM. Since $G'$ can be constructed in polynomial time ($O(|V|^2)$) from $G$, and covering with minimum clusters on $G$ is fully complaint with CLIQUE COVER PROBLEM on $G'$, DELAY BASED CLUSTERING PROBLEM is NP-hard. □

We proved that the DELAY BASED CLUSTERING PROBLEM is a hard problem in general, although in some cases it is solvable in polynomial time. In Theorem 6 we state that the DELAY BASED CLUSTERING PROBLEM is solvable in polynomial time in cases when DETERMINISTIC CLUSTERING PROBLEM gives positive answer for the same input.

**Theorem 6** *On a given complete, weighted graph G (whose edges fulfill the triangle inequality) and d positive number for which answering the* DETERMINISTIC CLUSTERING PROBLEM *gives positive outcome (yes), a solution can be found for the* DELAY BASED CLUSTERING PROBLEM *in polynomial time.*

*Proof* In Lemma 4 we proved that we can find an answer for the question of Definition 3 in polynomial time. In cases, when the question of Definition 3 has a positive answer if we delete all edges with greater weight than $d$ from $G$, then all the disconnected components of $G$ (after the deletion) are complete subgraphs. Since none of the remaining edges has greater weight than $d$, the optimal solution for DELAY BASED CLUSTERING PROBLEM equals with the disconnected complete subgraphs. □

### 5.2 Operational Steps of our Clustering Algorithm

When our clustering component is initialized, the scheduler receives the node clusters from our clustering component when they are looking for nodes in a certain delay radius. The scheduler calls the clustering component with the origin node and the delay radius required by the Pod. If there is already a constructed cluster layer with that delay value, the algorithm finds the appropriate cluster (that holds the starting node), and returns that to the scheduler. If the topology is not clustered with the given delay value yet, the algorithm creates a new layer that is based on the underlying layer that is clustered with the delay that is the greatest and closest to the new one, but still less than the given one. Relying on that underlying layer and its delay matrix, the algorithm creates the new layer in six steps: 1) delete the edges that are greater than the delay requirement; 2) find a maximal clique in the graph; 3) create cluster from the found maximal clique; 4) remove the vertices in the clique from the graph: 5) return to step (2), until all of the vertices are deleted from the graph; 6) create the new delay matrix.

After we defined the clusters with the new delay requirement, a new delay matrix must be created that holds the delay values between the clusters. We apply the conservative complete-linkage clustering method (also known as farthest neighbour clustering) for delay matrix creation to calculate the delay values between clusters. The complete-linkage clustering method means that the delay value between two clusters equals to the delay between those two nodes (one in each cluster) that are farthest away (have the highest delay) from each other.

## 6 Implementation Choices

Today's most widely used resource and service manager is Kubernetes [17]: it orchestrates containers, provides automatic scheduling, scaling and self-healing features. We have built a prototype of our edge-scheduler that we integrate with Kubernetes. In this section we present Kubernetes extensions, including ours, that propose edge computing support.

### 6.1 Kubernetes on the Edge

Kubernetes distinguishes two types of nodes, which might be either virtual or physical machines: i) the *master*, who is responsible for coordinating the cluster; and ii) the *worker* nodes. *Pods* are the smallest deployable units of computing in Kubernetes; several Pods can be instantiated on a node, and the Kubernetes master schedules and manages Pods across nodes in the cluster. A Pod contains one or more containers, e.g., Docker containers, with shared storage/network, and a specification for how to run them.

As edge computing becomes rather the norm, than the exception, the community has started to work on extending Kubernetes with several capabilities that support the operation in edge computing systems. KubeEdge [15] is built upon Kubernetes to extend application scheduling capabilities to nodes in an edge computing environment. It provides infrastructure support for networking and application deployment between cloud and edge. Beside KubeEdge, another multi-cluster oriented framework has attained the community's attention. Kubefed [16] (Kubernetes Cluster Federation) is the official Kubernetes cluster federation implementation. It allows the management and configuration of multiple Kubernetes clusters with a single set of APIs from a "host cluster". Neither KubeEdge and Kubefed feature network characteristic measurements between the worker nodes nor make guarantees about fulfilling delay requirements that novel applications would pose. Both of these frameworks apply the default Kubernetes scheduler, and they work on manually and preliminary constructed clusters, i.e., they do not support dynamic cluster creation based on e.g., network properties. K3s [12] and Microk8s [19] are frameworks that make Kubernetes lightweight by eliminating some of its components and by reducing the operational overhead. Although, a distributed system can enjoy the small footprint operation capability that K3s and Microk8s provide, in edge computing, beside the limited physical resources there are other issues that the system has to face with, e.g. unreliability, large scale, service level agreements, etc.

None of the listed frameworks are fully capable of handling an edge cloud system with latency-critical application requests. They proposed some steps forward in order to support an edge cloud system, but we argue that they are missing features that must be considered when a fully operational geo-distributed manager is proposed. These deficiencies call for an advanced system design that we propose throughout this article.

### 6.2 Extending Kubernetes with our Prototype

Our prototype extends Kubernetes [17] with the ability to work in a large scale edge computing topology and make latency critical Pod scheduling possible, while ensuring high reliability. Our solution works next to Kubernetes' default scheduler and extends its functionality with awareness to network characteristics.

Beside the previously seen scheduler, re-scheduler, and clustering modules, there are two more major components that make Kubernetes-edge-scheduler a fully operable edge-scheduler. The Monitoring component in Fig. 2 is responsible for collecting the underlying network delays. The measurement of the delay values can be performed in three ways with our implementation: i) with "ping pods" that are distributed between all nodes and send the measured delay values (between all node pairs) to Kubernetes-edge-scheduler; ii) with Goldpinger [6]; iii) with static files. The Event handler (also presented in Fig. 2) connects Kubernetes system events, e.g. Pod submission, Pod removal, etc. with our solution. It subscribes to important events, and after some pre-processing, it forwards the event to our system.

The source code of our implemented solutions has been released [18].

## 7 Comprehensive Evaluation of our Kubernetes Scheduler

In this section we present the efficiency of our Kubernetes-edge-scheduler with comprehensive evaluation scenarios. First, we run large scale simulations to evaluate the performance and scalability of our scheduler and re-scheduler algorithms, then we deploy a streaming analytics application in a Kubernetes cluster featuring our proposed scheduler to demonstrate the benefits of the delay-awareness in the edge.

In a realistic operation mode, customers send their requests in an online manner, one after another distributed in time. The offline placeholder minimization process, however, takes a single batch request that contains all, already submitted Pods. To make the best of the two worlds, our online scheduler and our offline re-scheduler work together: while the scheduler does the dynamic operations, like Pod scheduling, the re-scheduler can periodically minimize the size of placeholders provisioned in the system. Although the provider has to find the balance, how frequently the re-scheduler should run, and how they manage the changes that the re-scheduler proposes.

The offline minimization process is expensive and the state of the system may even change during its execution. The cost of the placeholder minimization in fact relies on two factors: i) the calculation demands extra computational resources; ii) resulting Pod migrations cause instability to applications. Multiple triggering criteria can be defined to control the start of executing the offline minimization procedure. A feasible triggering criteria can be a threshold of the ratio between the size of the deployed Pods and the size of the provisioned placeholders. This value can give a good insight about the efficiency of the placeholder provisioning and the utilization of the system.

### 7.1 Large Scale Simulation Setting

In our experiment setup we simulate high numbers of edge nodes in the range of 500 to 10000, and a number of Pods in the same order of magnitude to be scheduled (and then re-scheduled for optimization) on those edge nodes. The resource capacity of edge nodes are identical, and are represented by 12GB of memory. On the other hand, the inter-node delay values are heterogeneous, due to their hierarchical network setting: we assume 3 edge nodes in one server rack with 1ms delay between any pair of them, and 2-3 racks in close vicinity with 5ms delay between the edge nodes within. The network topology of our simulations is similar to the one depicted in Fig. 1 with delays of 10ms, 20ms, and 40ms on the links from bottom up. Therefore the delay between any 2 edge nodes is between 1ms and 152ms. Central cloud is supposed to be reachable at the top of the network hierarchy, so the network latency between any edge node and the cloud is 76ms.

The computational resource demand of Pods are represented homogeneously by 1GB of memory for simplicity. Their delay requirements, however, are randomly drawn from the following values: 1, 20, 50, 60, 70, 80, 90, 150, 160, 170 ms. The corresponding origins for which these delay limits are defined are randomly scattered over the locations of edge nodes. The order of deploying the Pods' with the online scheduler is totally randomized over the measurements. For all parameter settings we run 100 measurements to account for the randomized infrastructure, Pod delay requirements and Pod deployment order.

### 7.2 Evaluation of Placeholder Provisioning

We compare the performance of our scheduler and re-scheduler in terms of how effectively they provision the resources to provide high reliability. In the left side of Fig. 4 we compare the created placeholders' total size to the total amount of resources of deployed Pods with both of our scheduler and re-scheduler. The right plot in Fig. 4 shows the size of the Pods, that need to be migrated after the re-scheduler finished, compared to the total amount of resources of the deployed Pods. In each scenario we scheduled Pods triple the number of edge nodes. All scenarios, distinguished with different colors per plot in Fig. 4, ran on topologies of various sizes. The number of edge nodes is depicted on the *x-axis*. The *y-axis* shows the achieved size ratio in percentage (lower value is better).

We evaluate our re-scheduler in two scenarios. The scenarios differ in the strategy of Pod relocation in the repair phase. In the first scenario, we favor migration: the algorithm migrates a Pod when the migration fixes a failed placeholder in the repair phase. In the other scenario, we rule out the migration possibility during the repair phase.

The re-scheduler always achieves better performance, i.e., more compacted placeholders, compared to the online scheduler's results (see the left chart). This is expected, since the re-scheduler operates on all Pod requests at once, in contrast to the online algorithm, which strives to minimize the placeholders' size, but it processes only one request at a time.

We achieve the best placeholders/Pods ratios with our re-scheduler when the migration is favored. In these cases, the size ratios are around 8%, while in other scenarios, where we consider the migration too expensive, we achieve results around 8.5% and the online algorithms scores between 9 to 10% (see the left plot in Fig. 4). Although beside the improvement in the total size of backup resources, the number of migrations can be relatively high, around 18% of the total Pods, when we favor the migration. In contrast, when the migration is avoided during the repair phase, the number of migrated Pods is around 6% (see the right plot in Fig. 4). The achieved size ratio results are

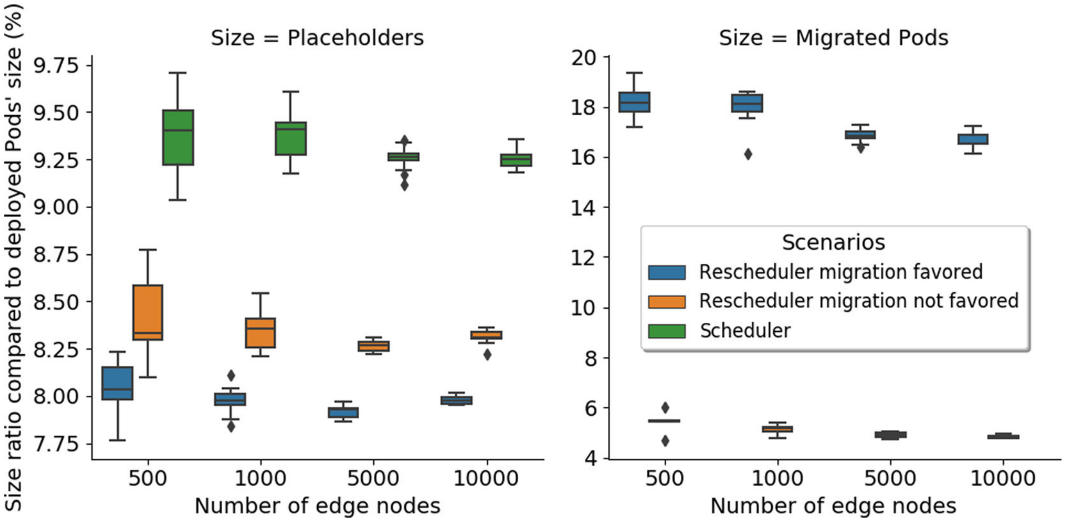

**Fig. 4** Size ratio differences compared to deployed Pods' size

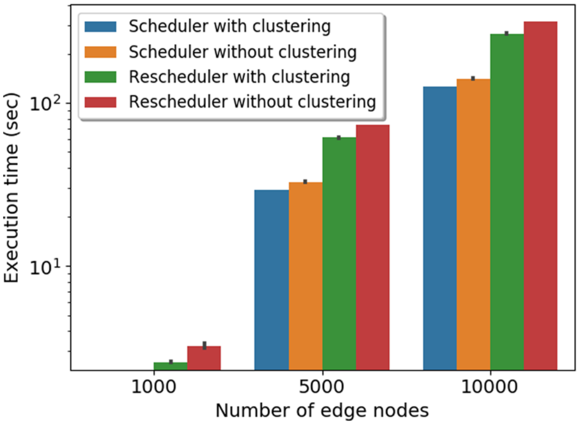

**Fig. 5** Execution time of placement of 30000 Pods with and without clustering

greatly affected by the delay values in the topology and the Pods' delay requirements, hence the relatively large boxplots showing high variance.

To conclude the results presented in Fig. 4 we state that both our scheduler and re-scheduler can effectively provision resource for placeholders and Pods, since in all cases the size ratios are less than 10%, which means our schedulers provision less than 10% of the requested resource of applications to provide high reliability for all applications with the aware of their delay criteria.

### 7.3 Evaluation of Execution Times

We evaluated our scheduler and our re-scheduler with and without the clustering feature on topologies of different sizes. The achieved execution times are presented in Fig. 5, where the *x-axis* shows the number of edge nodes and the *y-axis* presents the achieved runtime. In all scenarios, we submitted Pods, triple of the number of edge nodes. An evaluation started when the first request was submitted and ended when the algorithms gave their final deployment.

Although the re-scheduler has polynomial complexity (Section 4.2), it took the most time to return its result deployment. In all scenarios, we either deployed all Pods with the online algorithm one-by-one, or the re-scheduler received the same request set in one batch to work with. Both of the scheduler and re-scheduler benefit from clustering in terms of execution time, since in every case the algorithms finished earlier with clustering, e.g., on 10000 edge nodes the re-scheduler with clustering deployed all 30000 Pods on average in 263 seconds, but it took 317 seconds without clustering. A summarizing table, Table 1, shows the complexities for both the offline minimization, the online scheduling and the migration methods. We denote the set of nodes with $N$ and the set of Pods with $P$.

The presented results exclude the initialization phase, since it is not an integral part of the scheduling process, and we have to find the vantage-free layers once. Since we use hierarchical clustering, as the number of layers increases, the clustering becomes faster. This effect can be seen in Fig. 5. The clustering has negligible overhead, as the execution times are improved with the clustering feature, since the base layers were constructed with the *vantage-free delays*. To conclude the results presented in Fig. 5 we state that Kubernetes-edge-scheduler scales well for both the growing topology and for the increasing number of requests. Even the slower re-scheduler algorithm took less than 300 seconds for mapping 30000 Pods, which means it took an average of 10ms to deploy a single Pod.

While concerned about the quadratic complexity in terms of number of nodes, we strive to decrease the search space of the algorithms with the help of clustering the nodes based on latency measurements among them, we note that the runtime is majorly affected by the latency requirements of the applications and the capacity of edge nodes: if both the former and the latter are stringent, then the search space is greatly reduced, possibly resulting in orders of magnitude lower runtimes. We argue that for typical delay-critical applications and edge node infrastructures this will be the case.

**Table 1** Summarizing the complexities of the algorithms

| | Offline | Online | Migration |
|---|---|---|---|
| $\|P\| < \|N\|$ | $O(\|P\|^2\|N\|^2 + \|N\|^3)$ | $O(\|N\|^2)$ | $O(\|P\|\|N\|^2 log\|N\|)$ |
| $\|P\| > \|N\|$ | $O(\|P\|^2\|N\|^2)$ | $O(\|P\|^2)$ | $O(\|P\|^3)$ |

### 7.4 Stream Analytics in the Edge

We demonstrate the benefits of using Kubernetes-edge-scheduler in a real-world Spark [30] streaming application that is deployed over an emulated edge computing topology. In our use-case, we have a streaming application that receives a text file through a network socket and executes a word-count application on it on the fly. In our topology we had 10 edge nodes that are close together, and 1 worker node that represented the cloud. We compared the completion time of the word-count application deployed with both the default Kubernetes scheduler and Kubernetes-edge-scheduler. The results are presented in Fig. 6, where the *x-axis* shows the number of allocated executors (Pods) for the application, the *y-axis* shows the execution time in seconds.

The results show that the Spark streaming application always finished earlier in the cases when Kubernetes-edge-scheduler deployed its Pods. This outcome confirms that network delay affects the streaming applications performance, and as Kubernetes-edge-scheduler takes into account the network delay between the application components, it achieves better performance. More precisely, our solution reduced the average execution time by more than 60%. Using the default scheduler leads to higher variance in execution times, which is explained by the default scheduler policy that chooses nodes randomly, disregarding the characteristics of the underlying network connections. In contrast, Kubernetes-edge-scheduler considers network latency during scheduling, so the variance of execution times is lower.

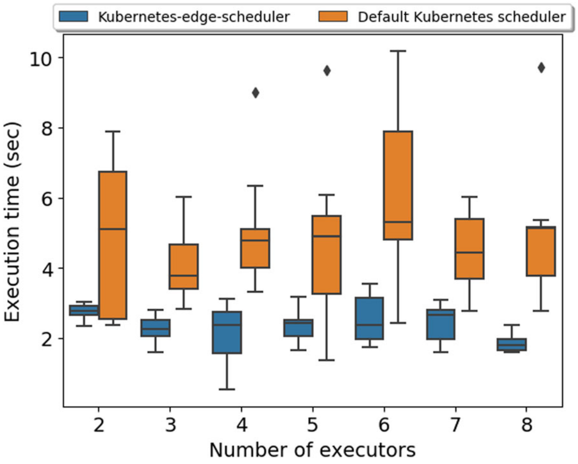

**Fig. 6** Execution times for a Spark streaming application

## 8 State-of-the-art on Reliability and Delay Guarantees of Scalable Edge Cloud Platforms

In this section we present the major achievements in the literature related to our Kubernetes-based edge cloud orchestrator. We divide the discussion of the state-of-the-art into parts on i) the application requirements and features of the edge cloud, ii) high reliability and availability concepts in virtual resource environments, and finally, iii) the implementation efforts towards scalable edge cloud platforms.

### 8.1 Latency-Critical Cloud-Native Applications and the Edge Cloud

Latency-sensitive and data-intensive applications, such as IoT or mobile services, are leveraged by edge computing, which extends the cloud ecosystem with distributed computational resources in proximity to data providers and consumers. This brings significant benefits in terms of lower latency and higher bandwidth. However, by definition, edge computing has limited resources with respect to cloud counterparts; thus, there exists a trade-off between proximity to users and resource utilization. Moreover, service availability is a significant concern at the edge of the network, where extensive support systems as in cloud data centers are not usually present. To overcome these limitations, [1] proposes a score-based edge service scheduling algorithm that evaluates network, compute, and reliability capabilities of edge nodes. The algorithm outputs the maximum scoring mapping between resources and services with regard to critical aspects of service quality. [9] introduces a new platform for enabling an edge infrastructure according to a disaggregated distributed cloud architecture and an opportunistic model based on bare-metal providers. Results from a multi-server online gaming application deployed in a real geo-distributed edge infrastructure show the feasibility, performance and cost efficiency of the solution.

In order to meet the rapidly changing requirements of the cloud-native dynamic execution environment, without human support and without the need to continually improve one's skills, autonomic features need to be added. Embracing automation at every layer of performance management enables us to reduce costs while improving outcomes. Kosińska and Zielinski [14] lists the definition of autonomic management requirements of cloud-native applications, and the authors propose that the automation is achieved via high-level policies, while autonomy features are accomplished via the rule engine support. One such feature of online scheduling in a cloud-native context is migration. A large body of research has tackled the issues around migration of virtual machines, containers, etc. in the cloud. E.g., [10] proposes an energy-aware virtual machine migration technique for cloud computing, which is based on the Firefly algorithm. The proposed technique migrates the maximally loaded virtual machine to the least loaded active node while maintaining the performance and energy efficiency of the data centers.

In the era of cloud services, there is a strong desire to improve the elasticity and reliability of applications in the cloud. The standard way of achieving these goals is to decouple the life-cycle of important application states from the life-cycle of individual application instances: states, and data in general, are written to and read from cloud databases, deployed close to the application code. Rooted in cloud-native computing, the stateless design outsources the state embedded in computing entities, e.g., virtual machines, containers, Pods, virtual network functions, to a dedicated state storage layer, facilitating elastic scaling and resiliency [26]. In [26] the authors propose a system design that can be adapted to any cloud application without the need for complex coordination among the network control, the stateless application elements, and the state storage backend. They present the first product-phase realization of the stateless paradigm, an operational virtualized IP Multimedia Subsystem that can restore the live call records of thousands of mobile subscribers under a couple of seconds with half the resources required by a traditional "stateful" design.

The high performance requirements on the application impose strict latency limits on these cloud storage solutions for state access. Cloud database instances are therefore distributed on multiple hosts in order to strive to ensure data locality for all applications. However, the shared nature of certain states, and the inevitable dynamics of the application workload necessarily lead to inter-host data access within the data center (or even across data centers, if the application requires a multi-data center setup). In order to minimize the inter-host communication due to state externalization, the authors of [25, 26] propose an advanced cloud scheduling algorithm that places applications' states across the hosts of a data center. In such a cloud-native setting, stateless cloud applications and an adaptively self-synchronizing distributed cloud database alleviate the long-standing issues of live migration within the cloud.

### 8.2 Ensuring High Reliability and Availability on Virtualized Resources

Several research papers have been published that all propose some kind of scheme for improving the availability and reliability of applications in the inherently untrustworthy context of edge cloud infrastructure.

Javed et al. [11] tackled the problem of separate software stacks between the edge and the cloud with no unified fault-tolerant management, which hinders dynamic relocation of data processing. In such systems, the data must also be preserved from being corrupted or duplicated in the case of intermittent long-distance network connectivity issues, malicious harming of edge devices, or other hostile environments.

A self-adapting scheme named SAB is proposed in [23]: SAB uses static and dynamic backups for VNFs (Virtualized Network Function) over both the edge and the cloud in order to provide high availability. Fan. et al. [4] propose a framework to provision availability of SFC (Service Function Chain) requests in a data center. None of these research works consider multiplexing backup resources for multiple virtual instances like our Kubernetes-edge-scheduler does. Yala et al. [28] propose a solution for their optimization problem that strive to optimize the trade-off between availability and latency. However, their work deploys VNFs without deploying backup resources.

Although, the solution of Kanizo et al. [13] and Raba [31] both multiplex backup resources, however they ensure high availability for VNFs with dedicated backup nodes. In contrast, Kubernetes-edge-scheduler does the resource provisioning on general nodes that contains Pods as well. In [2, 5] the authors consider the replica and virtual function placement to achieve lower migration time, however they did not consider minimizing the provisioned resources assigned to replicas. Authors of [29] investigate the fog resource provisioning problem for deadline-driven IoT services to minimize the cost considering the probability of resource failures. They assume that virtual machine failures are temporary and recoverable. In contrast, we argue that each node's failure should be prepared for.

### 8.3 Latency-Aware Cloud Platforms

An online resource orchestration algorithm which takes into account network aspects is proposed in [7]. The algorithm enables the orchestrator of OpenStack to manage a distributed cloud-fog infrastructure. An embedding algorithm is proposed in [21], which instantly deploys end-to-end delay-constrained services while applying a cost-aware VNF migration strategy. The authors' hybrid orchestration approach unites the advantages of online heuristics and offline optimization in their service orchestration method, with the goal of providing fast service placement and minimizing the cost due to VNF migrations.

The research community have already started extending the Kubernetes scheduler to support edge computing architectures with network awareness [3, 20, 22]. In contrast to our work, the scheduling method in [3] does not take into account the delay directly between the edge nodes. Authors of [20] proposes a content delivery method that improves Kubernetes scheduler with awareness to network distance using AS path of BGP. In contrast, we use the measured delay values as the network distance property. An extension, called Network-Aware Scheduler, is implemented in [22] enabling Kubernetes to make resource provisioning decisions based on the network infrastructure properties like latency and bandwidth. Although, the application requests in [20, 22] do not define latency requirement that has to be met during their scheduling. Furthermore, none of the previous works consider providing high reliability for the applications, and dynamic topology clustering to relax the difficulties that a large scale architecture poses. We argue that the main goal of edge computing, i.e., hosting delay critical applications, must be aware of not only network latency, but it must also take into account the unreliability and the large number of edge nodes.

A key difference between previous works and our solution is that none of [3, 7, 20–22] deal with reliability, while our proposed solution achieves high reliability with the consideration of minimizing the resources provisioned for this cause.

## 9 Conclusion

In this article we proposed Kubernetes-edge-scheduler, a novel scheduler that extends a Kubernetes system to operate on an edge computing architecture and to manage latency critical, novel applications. Our contribution is fourfold: i) we defined placeholders that help to guarantee high-reliability for edge applications with the provision of backup resources; ii) we proposed an online Pod scheduler algorithm that deploys latency critical Pods on the fly, reacts to network and Pod related system events, and provides high reliability for applications; iii) an offline re-scheduler is presented that reduces the provisioned backup resources that ensure the high reliability in the system; iv) a latency based clustering method is proposed for addressing the difficult task a possibly worldwide-scale topology would pose in network latency measurements, Pod scheduling, etc. Using both emulated and simulated experiments we showcased the effectiveness of our solution in terms of end-to-end application delay, the amount of provisioned resources and the scaling quality of our Kubernetes-edge-scheduler.

**Funding** Open access funding provided by Budapest University of Technology and Economics.

## Appendix

**Lemma 5** *In case of a fix number of vertices ($|V| = n$) and a fix diameter ($d$), with the increase of the number of buds ($b$), the optimal solution ($OPT$) monotonically increases.*

*Proof* Referring on the proof of Lemma 2, one can see that the number of placeholders is directly proportional to the number of buds. □

**Lemma 6** *In case of a fix number of vertices ($|V| = n$) and a fix number of buds ($b$), with the increasing diameter of the graph $d$, the number of placeholders, given by our heuristic algorithm ($HEUR$) monotonically increases.*

*Proof* Let us induce a path graph $G'$ from a path in $G$ whose length equals to $d$. We can give a sequence of Pod requests, for which the amount of placeholders provisioned by our heuristic solution would be equal to $HEUR = d$. Since $G'$ has $k = d + 1$ vertices, the $HEUR = d = k - 1$ solution is the worst solution that our algorithm can give on $G'$ (Lemma 3). Regarding this, one can see that with the increase of a graph's diameter, the number of placeholders, given by our heuristic algorithm, monotonically increases. □

**Lemma 7** *We deduce the possible number of buds in simple, connected graphs with a given diameter.*

1. *If* $d = 1$*, then* $b = 0$*;*
2. *If* $d = 2$*, then* $0 \leq b \leq 1$*;*
3. *If* $3 \leq d \leq \frac{n}{2}$*, then* $0 \leq b \leq \frac{n}{2}$*;*
4. *If* $d = \frac{n}{2} + k, k > 0 (\frac{n}{2} + 1 \leq d \leq n - 3)$*, then* $0 \leq b \leq (\frac{n}{2}) - k + 1$*;*
5. *If* $d = n - 2$*, then* $1 \leq b \leq 3$*;*
6. *If* $d = n - 1$*, then* $b = 2$.

*Proof* The indices of the following proofs refer to the indices of cases listed in the lemma.

1. Only complete graphs (from the class of connected, simple graphs) have diameter 1. Complete graphs do not have any leaf vertices, therefore they do not have buds either.
2. The diameter of a star graph, with one central node is 2. This graph has exactly 1 bud node (the central node), since all other nodes are leaves. We can construct several different graphs that have diameter $d = 2$ and 0 buds. One trivial example is a graph that is constructed from a complete graph by deleting a single edge. There cannot be more buds in graphs with $d = 2$, since the smallest combination of vertices and edges where $b = 2$ is a leaf-bud-bud-leaf subgraph that already has $d = 3$.
3. Let us create a cycle with $n$ vertices. This cycle has $d = \frac{n}{2}$ and $b = 0$. We can add extra diagonal edges to this graph so that it will have any diameter value between 3 and $\frac{n}{2}$ and the number of buds remains 0.

   Now let us present the other cases, when the graphs have $0 < b \leq \frac{n}{2}$. For every diameter value $3 \leq d \leq \frac{n}{4}$ we can create a cycle $C$ with $\frac{n}{2}$ vertices and adding extra diagonal edges so that $G$ has $3 \leq d \leq \frac{n}{4}$. At this point, we can connect the remaining $(G - C)$ $\frac{n}{2}$ vertices (from "outside of the circle") to the cycle so the number of buds is $0 < b \leq \frac{n}{2}$.

   We can also construct graphs for which $\frac{n}{4} < d \leq \frac{n}{2}$ and $0 < b \leq \frac{n}{2}$. Let us create a path graph with $d + 1$ vertices. The number of internal nodes (that are neither buds, nor leaves in the initial path graph) is $d + 1 - 4 = d - 3$. We can add $d - 3$ extra leaves connected to the internal nodes in order to increase $b$. To increase further the value of $b$, we can connect extra bud-leaf node pairs to any of the internal nodes till we reach the desired number of buds, or we consumed all the vertices in the graph. In case we reach the desired number of buds, but there are more unconnected vertices, we can construct a clique from them and connect it to any internal node.

   We showcased possible diameter and bud combinations between the upper and lower bounds of both the diameter and the number of buds. We also showed how we can construct graphs that have any $d$ and $b$ values between the defined bounds. Since the maximum number of buds in a graph with $n$ vertices is $\frac{n}{2}$, there is no other combination that can be constructed regarding the given diameter range.
4. Since the diameter of the graph is greater than $\frac{n}{2}$, the graph can not be a cycle. Let us construct a path graph with $\frac{n}{2} + k + 1$ vertices (so the diameter equals to $\frac{n}{2} + k$). Following on Lemma 7, we can close the two ends of the path graph with two vertices, and connect the remaining $(\frac{n}{2} - k - 3)$ vertices so that the graph will not have any buds. After the induction of the path graph with $\frac{n}{2}+k+1$ vertices, it has $\frac{n}{2} + k + 1 - 4 = \frac{n}{2} + k - 3$ vertices and we have $\frac{n}{2} - k - 1$ unconnected nodes. The maximum number of buds that is achievable in these scenarios is $\frac{n}{2}-k-1+2 = \frac{n}{2}-k+1$, since the path graph already has 2 buds, one at each end.
5. Let us construct a path graph with $n - 1$ vertices with $d = n - 2$ and $b = 2$. The remaining one node can be connected to the graph in three possible ways: i) the graph will have $b = 1$ bud, if we form a triangle at one end of the path graph (so we connect the last node to a leaf and to the connected bud); ii) the graph will have $b = 2$ buds, if we connect the last node to two adjacent internal nodes; iii) the graph will have $b = 3$ buds, if we connect the last node to one of the internal nodes.
6. A path graph with $n$ nodes is the only connected, simple graph that has $d = n - 1$. This graph must

have $b = 2$ buds, i.e., the second vertex on both ends. □

**Lemma 8** *The approximation ratio between our heuristic and the optimal solution for given maximal diameter and minimal number of buds:*

1. *if* $d = 1$ *and* $b = 0$*, then* $HEUR = OPT = 2$*;*
2. *if* $d = n - 3$ *and* $b = 0$*, then* $HEUR \leq 3OPT$*;*
3. *if* $d = n - 2$ *and* $b = 1$*, then* $HEUR \leq 3OPT$*;*
4. *if* $d = n - 1$ *and* $b = 2$*, then* $HEUR \leq 3OPT$*.*

*Proof* The indices of the following proofs refer to the indices of cases listed in the lemma.

1. The optimal solution in every complete graph is 2. The anti-affinity requirement hinders to have only one placeholder. Our heuristic algorithm also achieves the value of 2, since after the deployment of the first request there will be one Pod and one placeholder in the system. Let us denote the hosting node of this placeholder with $v$. Every other Pod whose origin differs from $v$ will be placed in the system without the need of increasing the number of the total placeholders. For the Pod, whose origin host is $v$, our solution creates the second placeholder, and deploys the Pod on another node. Therefore, with any order of Pod submissions, the heuristic algorithm will deploy only two placeholders.
2. Let us construct a path graph with $n - 2$ nodes with $d = n - 3$ and $b = 2$. Now let us close both ends of this path graph with two triangles made by the last two nodes on both ends and the two unconnected nodes. Therefore, the graph will have $b = 0$. In this case the optimal solution equals to $OPT = \frac{d+1}{3} = \frac{n-2}{3}$ (following on Lemma 1). Our heuristic solution will deploy maximum two placeholders in each triangle, so at least one of the Pods (per side) whose origin is in the triangle will benefit from a previously deployed placeholder, i.e., two Pods will share a placeholder on both ends of the graph. From this, we can state that at least two nodes will not have any placeholder on them, so $HEUR \leq n - 2$. Therefore, $HEUR \leq 3OPT = n - 2 \leq 3\frac{n-2}{3}$.
3. Regarding to Lemma 1, the optimal solution can not be less than $OPT = \frac{d+1}{3} = \frac{n-1}{3}$. Therefore, even when the heuristic solution achieves the worst solution ($HEUR \leq n - 1$), we can state that $HEUR \leq 3OPT = n - 1 \leq 3\frac{n-1}{3}$. One can see that our statement $HEUR \leq 3OPT$ is proven in graphs with $d = n - 2$ and $b = 1$.
4. The only connected, simple graph that has $d = n - 1$ and $b = 2$ is the path graph with $n$ nodes (see Lemma 7). The Pods' delay requirements allow only the origin node and its neighbors (see Assumption 1) as their hosting node. Therefore, the number of vertices in each Pod's radius is 2 or 3. Therefore, the minimum number of sets that cover all nodes, each set containing only nodes that are connected with each other, give the optimal solution for the number of placeholders. Therefore the optimal solution is $OPT = \lceil \frac{n}{3} \rceil$ in a path graph with $n$ vertices. In Lemma 3 we showed that the worst case result of our heuristic algorithm is $HEUR = n - 1$ (which is the case in a path graph), hence $HEUR \leq 3OPT = n - 1 \leq 3\frac{n}{3}$. □

**Acknowledgements** Projects no. 135074 and 2019-2.1.13-TÉT-IN-2020-00021 have been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the FK20 and 2019-2.1.13-TÉT-IN funding schemes.

## References

1. Aral, A., Brandic, I., Uriarte, R.B., De Nicola, R., Scoca, V.: Addressing application latency requirements through edge scheduling. J. Grid Comput., 17. https://doi.org/10.1007/s10723-019-09493-z (2019)
2. Bose, S.K., Brock, S., Skeoch, R., Rao, S.: Cloudspider: Combining replication with scheduling for optimizing live migration of virtual machines across wide area networks. In: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (2011)

3. Chima Ogbuachi, M., Gore, C., Reale, A., Suskovics, P., Kovács, B.: Context-Aware K8S Scheduler for Real Time Distributed 5G Edge Computing Applications. In: 2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM), pp. 1–6 (2019). https://doi.org/10.23919/SOFTCOM.2019.8903766
4. Fan, J., Jiang, M., Rottenstreich, O., Zhao, Y., Guan, T., Ramesh, R., Das, S., Qiao, C.: A framework for provisioning availability of nfv in data center networks. IEEE J. Sel. Areas Commun. **36**(10), 2246–2259 (2018). https://doi.org/10.1109/JSAC.2018.2869960
5. Farris, I., Taleb, T., Flinck, H., Iera, A.: Providing ultra-short latency to user-centric 5g applications at the mobile network edge. Trans. Emerging Telecommun. Technol. **29**(4). https://doi.org/10.1002/ett.3169 (2018)
6. Goldpinger: Debugging tool for Kubernetes. https://github.com/bloomberg/goldpinger. Accessed on: 21 March 2020
7. Haja, D., Szabo, M., Szalay, M., Nagy, A., Kern, A., Toka, L., Sonkoly, B.: How to Orchestrate a Distributed OpenStack. In: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 293–298 (2018). https://doi.org/10.1109/INFCOMW.2018.8407014
8. Haja, D., Szalay, M., Sonkoly, B., Pongracz, G., Toka, L.: Sharpening kubernetes for the edge. In: Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, SIGCOMM Posters and Demos '19, pp. 136–137. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3342280.3342335
9. Huedo, E., Montero, R.S., Moreno-Vozmediano, R., Vázquez, C., Holer, V., Llorente, I.M.: Opportunistic deployment of distributed edge clouds for latency-critical applications. J. Grid Comput., 19. https://doi.org/10.1007/s10723-021-09545-3 (2021)
10. Jain, N., Chana, I.: Energy-aware Virtual Machine Migration for Cloud Computing - A Firefly Optimization Approach. J. Grid Comput., 14. https://doi.org/10.1007/s10723-016-9364-0 (2016)
11. Javed, A., Robert, J., Heljanko, K., Främling, K.: IoTEF: A Federated Edge-Cloud Architecture for Fault-Tolerant IoT Applications. J. Grid Comput., 18. https://doi.org/10.1007/s10723-019-09498-8 (2020)
12. K3s: Lightweight Kubernetes. https://k3s.io/. Accessed on: 21 March 2020
13. Kanizo, Y., Rottenstreich, O., Segall, I., Yallouz, J.: Optimizing virtual backup allocation for middleboxes. IEEE/ACM Trans. Netw. **25**(5), 2759–2772 (2017). https://doi.org/10.1109/TNET.2017.2703080
14. Kosińska, J., Zielinski, K.: Autonomic management framework for cloud-native applications. J. Grid Comput., 18. https://doi.org/10.1007/s10723-020-09532-0 (2020)
15. KubeEdge: A Kubernetes Native Edge Computing Framework. https://kubeedge.io. Accessed on: 21 March 2020
16. Kubernetes Cluster Federation. https://github.com/kubernetes-sigs/kubefed. Accessed on: 21 March 2020
17. Kubernetes: Production-grade Container Orchestration. https://kubernetes.io. Accessed on: 21. March 2020
18. kubernetes-edge-scheduler. https://github.com/davidhaja/kubernetes-edge-scheduler. Accessed on: 21 March 2020
19. MicroK8s: Lightweight upstream K8s. https://microk8s.io Accessed on: 21 March 2020
20. Nakanishi, K., Suzuki, F., Ohzahata, S., Yamamoto, R., Kato, T.: A Container-Based Content Delivery Method for Edge Cloud over Wide Area Network. In: 2020 International Conference on Information Networking (ICOIN), pp. 568–573 (2020). https://doi.org/10.1109/ICOIN48656.2020.9016481
21. Németh, B., Szalay, M., Dóka, J., Rost, M., Schmid, S., Toka, L., Sonkoly, B.: Fast and Efficient Network Service Embedding Method with Adaptive Offloading to the Edge. In: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 178–183 (2018). https://doi.org/10.1109/INFCOMW.2018.8406882
22. Santos, J., Wauters, T., Volckaert, B., De Turck, F.: Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications. In: 2019 IEEE Conference on Network Softwarization (Netsoft), pp. 351–359 (2019). https://doi.org/10.1109/NETSOFT.2019.8806671
23. Shang, X., Huang, Y., Liu, Z., Yang, Y.: Reducing the Service Function Chain Backup Cost over the Edge and Cloud by a Self-Adapting Scheme. In: IEEE INFOCOM -IEEE Conference on Computer Communications (2020)
24. Szalay, M., Mátray, P., Toka, L.: Minimizing State Access Delay for Cloud-Native Network Functions. In: 2019 IEEE 8th International Conference on Cloud Networking (Cloudnet), pp. 1–6 (2019). https://doi.org/10.1109/CloudNet47604.2019.9064048
25. Szalay, M., Mátray, P., Toka, L.: State management for cloud-native applications. Electronics **10**(4). https://doi.org/10.3390/electronics10040423. https://www.mdpi.com/2079-9292/10/4/423 (2021)
26. Szalay, M., Nagy, M., Géhberger, D., Kiss, Z., Mátray, P., Németh, F., Pongrácz, G., Rétvári, G., Toka, L.: Industrial-Scale Stateless Network Functions. In: 2019 IEEE 12Th International Conference on Cloud Computing (CLOUD), pp. 383–390 (2019). https://doi.org/10.1109/CLOUD.2019.00068
27. Toka, L., Haja, D., Kőrösi, A., Sonkoly, B.: Resource Provisioning for Highly Reliable and Ultra-Responsive Edge Applications. In: 2019 IEEE 8Th International Conference on Cloud Networking (Cloudnet), pp. 1–6 (2019). https://doi.org/10.1109/CloudNet47604.2019.9064131
28. Yala, L., Frangoudis, P.A., Ksentini, A.: Latency and Availability Driven VNF Placement in a MEC-NFV Environment. In: IEEE Global Communications Conference (GLOBECOM) (2018)
29. Yao, J., Ansari, N.: Reliability-Aware Fog Resource Provisioning for Deadline-Driven IoT Services. In: IEEE Global Communications Conference (GLOBECOM) (2018)

30. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, p. 10. USENIX Association, USA (2010)
31. Zhang, J., Wang, Z., Peng, C., Zhang, L., Huang, T., Liu, Y.: Raba: Resource-Aware Backup Allocation for a Chain of Virtual Network Functions. In: IEEE INFOCOM 2019-IEEE Conference on Computer Communications (2019)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.