Container Orchestration in Heterogeneous Edge Computing Environments

Daghash K. Alqahtani and Adel N. Toosi

Abstract The emerging latency-sensitive applications and Internet of Things technology have resulted in the development of Edge computing. Therefore, improving Quality-of-Service (QoS) requirements such as response time is a fundamental goal in Edge environments. However, as edge devices are heterogeneous and resource-constrained, placing replicas of software containers in these environments is not a trivial task. Yet, the scheduler of well-known container orchestration tools such as Kubernetes places pods in nodes by only considering available resources (CPU, memory, etc.). Our study aims to minimize application latency by optimizing resource allocation through efficient scheduling. We customize the Kubernetes scheduler to assign pods to nodes with the least response time and integrate it with a customized autoscaler to scale up/down replicas. We evaluate our algorithm on a small-scale cluster with two types of deployments (web and object detection services), and evaluation results show better response time and throughput than the default scheduler.

Keywords Edge computing · Latency · Kubernetes · Scheduling · Replicas

1 Introduction

The term "containerization" refers to a method of developing software in which a single "container image" contains an application or service, all its required dependencies, and its configuration (represented abstractedly in deployment manifest files). The containerized software may then be individually tested and deployed to the host OS as an instance of the container image. Containers are a standard unit of software deployment that may include varied code and dependencies, like shipping containers that enable items to be delivered by ship, rail, or truck independent of the contents

D. K. Alqahtani (✉) · A. N. Toosi
Monash University, Melbourne, Australia
e-mail: daghash.Alqahtani@monash.edu

A. N. Toosi
e-mail: adel.n.toosi@monash.edu

A. Mukherjee et al. (eds.), *Resource Management in Distributed Systems*, Studies in Big Data 151, https://doi.org/10.1007/978-981-97-2644-8_8

within. Programmers and system administrators may efficiently distribute applications to other platforms with few changes by packaging applications in containers. In the same way, containers partition applications running on the same OS. Containerized applications are deployed to and executed by a container host, which is deployed to and managed by the operating system (Linux or Windows). This means that container images need far less space than VM images [3].

Kubernetes becomes a de facto tool to orchestrate containers in scale clusters. Famous companies and organizations adopt Kubernetes to manage and maintain their services, such as Amazon, Alibaba, and IBM. This is because it is open source and compatible with large clusters which consist of hundreds or thousands of devices. Kubernetes scheduler is responsible for assigning coming workloads to the ideal node among other nodes in a group. The default scheduler considers consistently placing a pod to a node with the most available resources (CPU and memory). Kubernetes was invented for cloud computing, and many works have been created to optimize it for edge and fog computing. The main goal of these works is customizing the scheduler to be aligned with their purposes.

At the same time, the tremendous rise and increased computational capability of IoT devices have resulted in previously unheard-of data quantities. Also, volumes of data will keep rising as the number of mobile devices linked to 5G networks increases. At the same time, the promise of cloud and AI was to automate and accelerate innovation by generating actionable insights from data. However, the extraordinary volume and complexity of data generated by connected devices have exceeded network and infrastructure capacities. By 2025, [17] predicts that 75% of all data will be handled outside the conventional data centre or on the cloud. Bandwidth and latency problems emerge when all this data is sent from disparate devices to a single location in a data centre or the cloud. Data can be processed and analysed much more quickly since it is done much closer to the creation site using edge computing. There is far less delay when processing data locally than sending it over the network to a cloud or data centre. Edge computing makes faster and more thorough data analysis possible, especially mobile edge computing on 5G networks, which may lead to more significant insights, quicker reaction times, and enhanced consumer experiences. Edge computing is defined by [17] as technologies that enable data processing at or near the source of data production. In the context of the Internet of Things (IoT), for example, the origins of data creation are often things with sensors or embedded devices. Edge computing is a decentralized extension of campus networks, cell phone networks, data centre networks, and cloud computing.

As one of the reasons the demand for releasing edge computing is latency, so Kubernetes scheduler has been adjusted to achieve that. Many works attempt to improve response time by placing pod to node, considering the delay between nodes. However, the variety of devices in edge computing can result in different performances. In other words, resource scheduling in heterogeneous edge computing has not received adequate studies, so it must be investigated further. Our goals are customizing the Kubernetes scheduler to improve response time and optimize resource utilization in the heterogenous edge computing environment. We propose a scheduling algorithm that considers the delay between nodes and the execution time of

nodes in a heterogenous edge cluster for placing replicas and integrating Kubernetes Event-Driven Autoscaler (KEDA) tool to scale up/down replicas when the number of requests increases/decreases.

The proposed scheduler algorithm enables the clients to state the response time constraint. Then, the nodes are filtered based on the latency threshold by comparing the response time of each node with these constraints. First, our algorithm calculates the delay between nodes and the execution time of the service in worker nodes. After that, it excludes the nodes with a response time that violates the threshold latency. Next, we sort nodes based on the latency value of nodes, then place the first replica to a node with a minimum value. In addition, when replicas increase, the scheduler places them in different nodes unless all the nodes become busy, then places a pod in the same node with the first replica. In contrast, the pods that have lately been created are terminated when replicas decrease.

We evaluate our scheduler in a small cluster and compare it with the default scheduler. We utilize Kind to emulate a small real cluster which consists of one master node and four worker nodes. The configuration of nodes has been customized to mimic the heterogeneity of edge devices. Furthermore, we create Locust files to stress the workloads and simulate multiple clients sending requests.

The key contributions of this paper are summarized as

- We provide a review of recent works on Kubernetes schedulers in edge computing and discuss the gap in the knowledge.
- We design and implement our proposed scheduler to improve response time alongside integrating it with an autoscaler to maximize resource utilization.
- We perform experiments to evaluate our proposed algorithm in an emulated setup and compare it with Kubernetes default scheduler.

The rest of this paper is organized as follows: Sect. 2 introduces Kubernetes and container autoscaling along with other related studies. Next, the system design and proposed algorithm are presented in Sect. 3. After describing the evaluation environment in Sect. 4, the results are analysed in Sect. 5. Section 6 discusses the findings and the work's limitations. Finally, we conclude the work and suggest future works in Sect. 7.

2 Background and Related Works

2.1 Kubernetes

Kubernetes is a sophisticated open-source technology created by Google that is used to orchestrate containerized applications in a clustered environment. It strives to improve the management of connected dispersed components and services across various infrastructures. At its core, Kubernetes groups separate physical or virtual computers into a cluster that communicates through a shared network. This cluster

serves as the physical substrate for all Kubernetes components, capabilities, and workloads.

Each node in the cluster is assigned a role in the Kubernetes ecosystem. The master (control plane) node is one node (or a small group in highly available installations). This node serves as the cluster's gateway and brain, presenting an API to users and clients, assessing the health of other nodes, choosing how to break up and allocate work (known as "scheduling"), and organizing communication between different components. The master node is the principal point of contact with the cluster and is in charge of the majority of the centralized logic provided by Kubernetes. The other workstations in the cluster are known as workers: nodes that accept and perform tasks utilizing local and external resources. Kubernetes runs applications and services in containers to help with isolation, administration, and flexibility. Hence each node must be equipped with a container runtime (like Docker). The node gets work instructions from the master node and builds or destroys containers as needed, modifying networking rules as needed to route and forward traffic [15].

Kubernetes's control plane includes the *Kube-scheduler*, which is the platform's default scheduler. The *Kube-scheduler* framework was created so that you may replace it with your custom scheduling mechanism if necessary. *Kube-scheduler* picks a node for the pod in a two-step operation: Filtering and Scoring. As a result of the filtering process, a list of nodes where it is possible to schedule the pod is identified. For instance, the Pod Fits Resources filter determines whether a potential node has enough resources to fulfil a pod's requirements. As a result of this process, the node list will include all relevant Nodes; in most cases, this will be more than one. If there are no pods on the list, they cannot be scheduled at this time. In the scoring stage, the scheduler rates the remaining nodes to find the optimal pod placement. The nodes that made it through the scheduler's filtering process are given scores based on the currently applied scoring criteria. At last, the *Kube-scheduler* places the pod on the best rated node. If there are many equally scored nodes, the *Kube-scheduler* will choose one at random [14].

A pod is a single instance of a Kubernetes-running application. Containerized applications are the workloads that you run on Kubernetes. Containers cannot be run directly on Kubernetes as they can in a Docker environment. Instead, the container is packaged into a Kubernetes object known as a pod. A pod is the smallest item that Kubernetes can produce. A single pod may house a collection of one or more containers. On the other hand, a pod does not often include numerous instances of the same application. A pod contains information on shared network and storage setup, as well as instructions for running its bundled containers. Pod templates are used to describe information about the pods that run in a cluster. To handle pod deployments, pod templates are YAML-coded files that can be reused and embedded in other objects.

A replication controller employs pod templates and specifies the number of pods that must execute. The controller makes it possible to run numerous instances of the same pod and guarantees that pods are constantly operating on one or more cluster nodes. If running pods fail, are removed, or are terminated, the controller replaces them with fresh pods in this manner [15].

2.2 Autoscaling

To automatically scale a workload in Kubernetes to meet demand, a Horizontal Pod Autoscaler can change a workload resource (such as Deployment or StatefulSet). With horizontal scalability, additional pods are added to handle the extra work as demand rises. In contrast, when scaling vertically with Kubernetes, developers would add other resources (such as memory or CPU) to the pods currently operating for the task. Horizontal Pod Autoscaler directs the workload resource (the Deployment, StatefulSet, or similar resource) to de-scale if the demand reduces and the number of pods exceeds the defined minimum [13].

Kubernetes Event-Driven Autoscaling (KEDA) is a piece of software that helps automate the scaling of applications. The KEDA's event count may control any Kubernetes container's scalability. KEDA is a minimal, single-function component that can be integrated into any existing Kubernetes environment. It is compatible with the default Kubernetes components, such as the Horizontal Pod Autoscaler, and may add new features without replacing or duplicating existing ones. In addition, KEDA allows developers to selectively define which applications benefit from event-driven scaling while leaving others unaffected. This ensures that KEDA can safely coexist with any Kubernetes framework or application [10].

2.3 Related Works

Numerous Kubernetes-based studies have been performed to build optimized container schedulers for various use case scenarios [1, 2, 4–6, 8, 9, 16, 18, 20–24, 26, 27]. Many of these require either the introduction of a custom scheduler or modifications to the existing scheduler. In this section, we review the literature on customizing the Kubernetes scheduler and classify them based on some categories, as depicted in Table 1. Various studies have focussed on reducing power consumption, optimizing resource utilization, or improving quality of service, using different strategies such as leveraging application states, resource metrics or network states. As a result, there is no single optimal scheduler that can fulfill all use cases and circumstances. For example, some papers place emphasis on reducing power consumption [16, 18, 22], while many others aim to optimize resource utilization [1, 8, 9, 20, 21, 27]. Besides, Kubernetes' scheduler has been the subject of several QoS optimization research. Many of these methods aim to satisfy the user's need for fast response while maintaining a reasonable latency. This is because fog and edge computing came into existence due to recent years' substantial breakthroughs and the widespread distribution of apps and devices. Therefore, in this section, we will describe the related work that emphasizes delay.

Network-aware scheduling (NAS) addition for Kubernetes is being proposed by [23]. This extension offers up-to-date information on the latest characteristic of the network infrastructure (bandwidth and latency). The paper proposes a network-aware

Table 1 Summary of related literature on scheduling strategies

Paper	Objective	Evaluation	Heterogeneity	Autoscaling	Target environment
[22]	E	Testbed	Size & H	No	Cloud
[16]	E	Emulation	Size	No	Cloud
[18]	E	Simulation	No	No	Edge
[27]	R	Testbed	Size	No	Edge
[20]	R	Testbed	Size	No	Edge
[1]	R	Testbed	No	No	Fog
[9]	R	Testbed	No	No	Fog
[21]	R	Testbed	Size & CPU-GPU	No	Edge
[8]	R	Testbed	Size & H	No	Fog
[23]	QoS	Testbed	Size	No	Fog
[4]	QoS	Testbed	H	No	Fog
[2]	QoS	Testbed	No	No	Edge
[6]	QoS	Testbed	No	No	Edge
[26]	QoS	Testbed	No	No	Edge
[24]	QoS	Testbed	No	No	Cloud & Edge
[5]	QoS	Simulation & Testbed	No	Yes	Fog

E: Energy Consumption, R: Resource utilization, QoS: Quality of Service, H: Hardware architecture

scheduling method for Smart City container-based apps, which makes resource allocation choices based on the actual state of the network. First, the suggested NAS analyses the pod configuration file to determine the optimal deployment site for a service, taking into account the Round-Trip Time (RTT) labels set at key nodes. After the filtering process is complete, the node selection is made based on the minimization of the RTT depending on the service's intended destination. In addition, NAS uses the service's bandwidth requirement label to determine whether the best candidate node has sufficient bandwidth to handle the service. Yet, this work is specific for placing one pod, and it fails to consider the variety of edge nodes.

Eidenbenz et al. [4] present a fog layer architecture to handle the computation and deployment of latency-aware industrial applications on top of Kubernetes. Consequently, the fog layer automatically optimizes the allocation of resources and delivers containerized applications across the networks of automation systems. Moreover, it does so without actively altering Kubernetes, making it ideal for environments where such changes would be undesirable. It is also superior to a vendor-specific solution since it is not limited by infrastructure and proprietary protocols. This article describes a case study in which a generic latency-aware resource allocation method was modified to be Kubernetes-compatible. To that purpose, they created K8S-GBA, based on the Greedy Border Allocation (GBA) heuristic. However, this paper makes no attempts to address autoscaling and the heterogeneity of fog nodes.

To dynamically orchestrate Industrial IoT (IIoT) applications, [2] suggest considering environmental, functional, and network aspects in addition to software states. They develop a method that gives the custom scheduler more responsiveness to external factors and programmatic alterations. With 5G radio and connection installed close to the IoT devices, a fully functional 5G and Edge computing network was constructed. Following the notion of Edge Computing, all the LTE virtualized functions are located on the same server rack as the Kubernetes cluster. They demonstrate that the system can optimize workload distribution among nodes, taking into account memory and CPU utilization and a wide range of network and infrastructure factors. The approach speeds up application launches and cuts down on scheduling times without sacrificing quality. The authors do not consider the diversity edge nodes and their work is particular to deploying a single replica.

A topology-aware Kubernetes is required to enhance its popular feature set concerning network latency since most delay-sensitive applications are to be deployed on edge. Thus, Haja et al. [6] take the initiative to mould Kubernetes into a tool that can be used with edge infrastructure. Furthermore, self-healing capabilities need more attention than in the default Kubernetes since edge infrastructure is vulnerable to failures and is thought to be costly to create and maintain. Considering the need to meet both application delay restrictions and edge resilience, they developed a customized Kubernetes scheduler. Their product is an add-on to Kubernetes that monitors latency between individual nodes. Each node will release a measurement pod, ping each other at regular intervals, and keep track of the round-trip timings. The scheduler takes the collected data and creates a delay matrix, using the matrix to assign node labels. Whereas this study aims to improve latency, it does not consider optimization of resource utilization. Also, they do not consider the heterogeneity of edge nodes.

To schedule pods in Kubernetes using the latest network measurements, [26] create and deploy a tool called NetMARKS. This scheduler extender makes use of data gathered by Service Mesh. In particular, they noted that there was room for development in pod collocation. Findings showed that application response times might be cut by as much as 30%, even for a modest pipeline consisting of just a few services. Using Istio Service Mesh, they showed a new way to inform the Kubernetes scheduler of dependencies between pods. The experimental findings demonstrate that the Service Mesh may enhance pod placement by utilizing the data it collects, leading to a decrease in application response time and conservation of inter-node bandwidth. In contrast, this work is designed only for minimizing response time, yet it is important to make the best use of resources. This is because edge devices are limited in resources, and it is important to examine the heterogeneity of edge nodes.

A novel distributed system and orchestration called Geolocate is implemented for hybrid Cloud and Edge computing in [24]. To achieve this, the scheduler must be able to determine, for a particular data-processing task, which node is best suited to handle the data coming from a specific data-producing system situated in a specified geographic region. In addition to enhancing network latency, data-processing delay, and service response time, the chosen nodes should reduce the physical distance between data producers and consumers. Computational times, service response

times, and network performance all improve with Geolocate, and bandwidth utilization is diminished, leading to higher throughput for applications under typical cluster settings. Response times for all services is cut by up to 62% with Geolocate's algorithm which help simply by minimizing the lag time between the services that generate and analyse data. Nevertheless, the paper does not study the variety of edge environments and placing a set of replicas as well.

According to [5], Hona is a replica scheduler for applications that consider tail latency, which is proposed as an addition to the Kubernetes container orchestration platform. Hona relies on Kubernetes to keep a watch on the state of the system's resources, Vivaldi to calculate the ping times between nodes, and Proxymity to check up on where the traffic is coming from and where it is going. After placing the first duplicate, Hona employs several heuristics to effectively filter through all the available choices for where to put the next one. Last but not least, it performs proactive replacement tasks automatically when end-user demands are incredibly different. However, the paper does not consider the execution time, which might vary between the heterogenous cluster nodes, and it is designed for fog computing. Also, the authors do not take into account the elastic provision resources.

3 System Design

This study aims to significantly minimize user-experienced latency by dynamically selecting the location of edge application replicas in a heterogeneous edge computing architecture. Also, we aim to optimize resource utilization by integrating Kubernetes Event-Driven Autoscaling (KEDA) tool within our proposed scheduler. As Fig. 1 shows, developers or system managers submit the deployment file to the proposed scheduler, where it is placed in the master node. Then, the scheduler assigns the first pod to a node with the least response time. The following section explains in detail how the proposed scheduler calculates the response time and places the initial replica. After deploying the application, the system managers expose the deployment to be accessed externally by configuring the Kubernetes service (Node Port). After that, the number of maximum replicas, the number of target requests, and the service name are specified in KEDA's file. When the clients send requests to the system, KEDA intercepts them for scale-up/down replicas based on the configuration file. If KEDA increased replicas, our schedulers placed pods in different nodes without violating threshold latency. However, when no request comes to the service, KEDA decreases the replicas to the minimum number. Thus, the proposed scheduler deletes the last pods created and keeps only the first pod live.

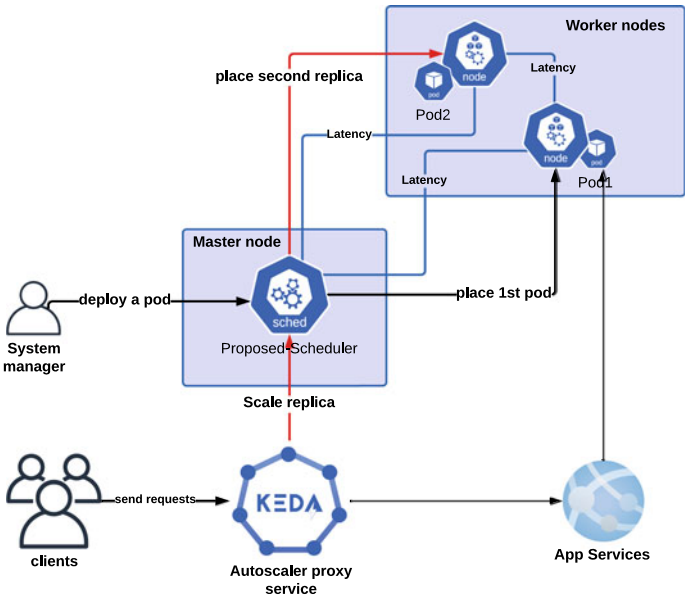

Fig. 1 System's Architecture Design

3.1 Replica Placement

Our scheduling method considers the latency between the nodes and execution time to improve the response time and meet the user requirements. When deploying an application, the scheduler ensures placing it in a node with the least response time. Algorithm 1 presents that there are three inputs to the scheduler. *Nodes:* Kubernetes manages everything about them, from the pods and their resources to the node they run. Kubernetes's etcd service allows us to access node's data easily. *Response time threshold:* The client provides the threshold latency for an application that needs to be met when deploying the application. *Profiling:* The execution time of an application for all the nodes in the cluster should be provided to the scheduler. This is because the execution time of a service varies between nodes as the nodes are different in their abilities.

The scheduler goes over all the nodes that have been provided as inputs and measures the distance between the node and the master node where the scheduler is placed. This happened by pinging the node and getting the value, and then the scheduler calculates the predicted response time for that node by summing the distance value with execution time. In addition, our algorithm filters nodes that meet the delay constraints and excludes nodes when the response time prediction does not meet the latency threshold. After that, we label the nodes with key-value pairs of the prediction response time that has been calculated. Before placing the first pod of replicas, the scheduler sorts all the nodes based on the label and then binds the

pod to the node with the lowest predicted response time value. When the scheduler receives the first pod, it assigns it to a node with a negligible response time value.

3.2 Autoscaling Replicas

We use KEDA to optimize resource utilization, which sometimes means when there is no traffic coming to the service, there is no need to have multiple replicas. Thus, it enables the cluster to have more available resources and can be utilized from another deployment. KEDA works based on the events as it collects metrics from the database and intercepts them, then scales the application when it reaches the target event. We utilize KEDA to scale the replicas based on the number of HTTP requests. Therefore, the KEDA uses Prometheus to monitor the Traffic and then scale-up or scale-down the replicas. When the target number of requests is reached, our algorithm increases the number of pods. When KEDA notifies the proposed scheduler to increase the number of replicas, it places the pod in a node that is close to the first pod. This is because we ensure a balance between nodes in the cluster and, simultaneously, do not violate the response time constraints as Algorithm 1 depicts. As the number of replicas increases, the scheduler assigns the pods to nodes in our cluster until all the nodes have replicas (first round). Then goes again over the nodes and places pods in the nodes that have available space (second round). The maximum number of replicas should not exceed the number of nodes. This is to ensure our algorithm works probably. When the target events decline, the autoscaler decreases the number of replicas. As Algorithm 1 shows, our scheduler investigates the nodes and tries to find a node with a pod with a minimum value of time creation. Then it unbinds that pod which also has the maximum response time value, and the scheduler behaves like this every time when KEDA scales down the replicas. When no requests are coming to the service, KEDA scales the number of replicas down to a minimum of one replica in our system's design. This is because when putting the minimum number of replicas to zero, the service would provide some latency to the first incoming requests due to the starting-up time of the first pod [19]. Therefore, our scheduler terminates the pods created lately and keeps only the first pod with the least response time.

4 Evaluation Environment and Experimental Setup

4.1 Testbed Configuration

We utilize Kind to build our Kubernetes cluster, because it enables us to run Kubernetes cluster on a local machine without needing a large infrastructure. Consequently, we can emulate real cluster environment. According to [12], using Docker containers as "nodes", Kind facilitates the operation of local Kubernetes clusters. The primary

Algorithm 1 Pseudo code for proposed scheduler

Require: $nodes, responsetimethreshold, profiling$
for each node in nodes **do**
  $distance[nodes] = ping(node)$
  $predictionresponse[node] = distance[node] + profiling[node]$
  if prediction response [node] $\leq$ response time threshold **then**
    $group1 = add(node : predictionresponse[node])$
  end if
end for
if place 1st pod **then**
  Function bind initial pod(group1):
  $score = \min(group1)$
  Return score
else if increase replicas **then**
  Function bind replicas(group1):
  if all nodes in group1 contain same pod **then**
    $score = \min(group1)$
  else
    $score = min(group1 - ignorenodesthathavesamepod)$
  end if
  Return score
else if decrease replicas **then**
  Function unbind replicas(group1):
  $remove = terminateslastpodwascreated$
  Return remove
end if

intent of kind is to test Kubernetes, although it may also be used for local development and continuous integration. We build our configuration in Kind to create five nodes: one master node and four worker nodes with their capacity, as Fig. 2 depicts. The heterogeneous edge devices have been emulated in our setup regarding the size of the CPU, memory [11]. We emulate the latency between nodes and the nodes distributed over different places by using the control traffic tool (tc) [25]. We use MacBook Pro laptop with 8 GB memory and 8 CPUs as our local machine and install Docker Desktop v4.12.0 as well as kind v0.14.0.

4.2 Benchmark Applications and Workloads

We test our scheduler over two types of workloads to ensure that our proposed algorithm is not workload specific and works perfectly with any service. The requests are sent using Locust to stress the services. Locust is an open-source tool for load testing [7]. The first workload is web service, and we used Nginx image as an open source. The Locust file of the web service was written to send five parallel requests to the target service for each user. We ran various tests and changed the number of users every time to invoke the autoscaler for scaling up/down replicas, as Fig. 3 shows. As

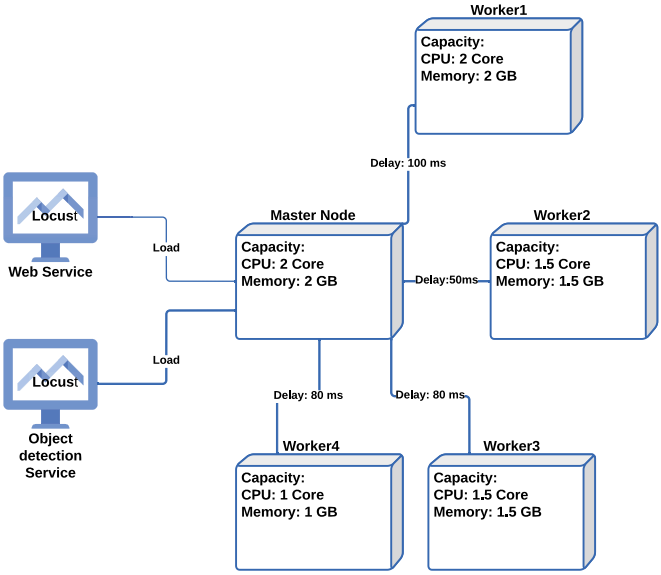

Fig. 2 Experimental setup diagram

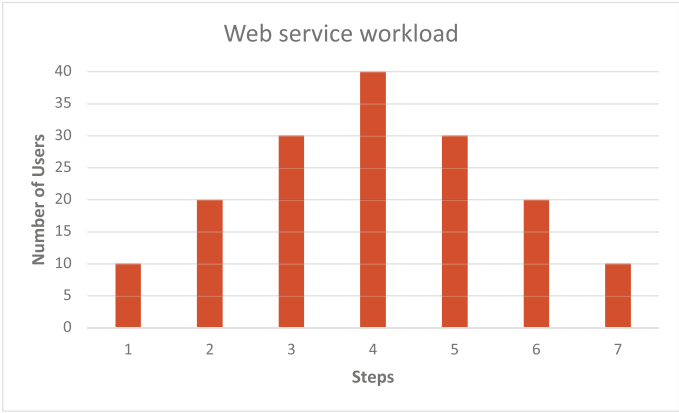

Fig. 3 Test workload for web service

the target event of autoscaling is 50 requests, we set ten users for first step. Then, we increase the number of users to 20 to invoke the autoscaler and have two replicas. Likewise for step 3 and step 4, we add ten users per step incrementally. After that, the number of users is decreased by ten users for each step in decremental.

Moreover, we created an object detection service written in Python language and using Flask as a web application framework. OpenCV (Open-Source Computer Vision Library) is used to execute necessary image operations/transformations, and

Fig. 4 Test workload for Object Detection service

the YOLO (you only look once) library is used since it is a cutting-edge real-time object recognition system. The YOLO and OpenCV libraries are open-source and popular computer vision and machine learning technologies written in Python. The function of this application is to return detected objects and a box around them in a JSON format when the client posts an image to the service. We created a Locust file to invoke the object detection application and simulate that every user post ten images, and the size of each image is 201 KB. We run the test and change the workload for object detection service same as web service workload manner (increment, decrement) as Fig. 4 depicts. However, since the target event for autoscaling is 20 requests, we initially set the number of users to two each sending 10 parallel requests in order to trigger the autoscaler. Similarly, for steps 2, 3, and 4, we incrementally added two users per step. Subsequently, we decreased the number of users by two for each decremental step.

4.3 Metrics

We evaluate our work using response time and throughput to demonstrate the effectiveness of the proposed method. Response time is the total elapsed time from when a request is made to when it is completed, and the client receives the response. Response time is automatically provided by Locust results when sending requests. We report the average and standard deviation of response time for all requests submitted in each step. In addition, throughput is the number of requests processed successfully per second. To ensure our evaluation results are more reliable, we run each experiment five times and report the average of collected results and add these values as error bar in figures.

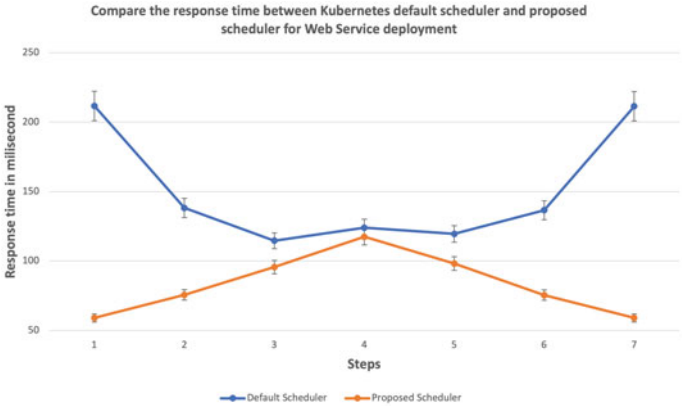

Fig. 5 Response time of Web service deployment

5 Results and Analysis

5.1 Web Service Results

We evaluate our scheduler's algorithm and compare it with the Kubernetes default scheduler using web service deployment. Firstly, our scheduler places the first replica in a node considering response time. Therefore, as Fig. 5 presents that it improves the response time significantly. On the other hand, the default scheduler assigns the initial replica to a node with the most available resources (CPU, memory) and does not consider the latency between nodes. Thus, it violates the response time constraints and results in a significant delay. After that, we increase the workloads to invoke the system to scale up the replicas. Then, the proposed scheduler increases the pods and places them in nodes in the cluster except the nodes whose response time would reduce the excess of the threshold. As a result, the response time rises while the replicas increase. However, the default scheduler declines the response time when scaling up the replicas. This is because it spreads the replicas over all the nodes, so the probability of getting the nodes with higher latency goes up. There is more similarity of response time and throughputs between our scheduler and default scheduler when the system has the maximum number of replicas, whereas step 4 in Fig. 5 shows that.

Moreover, we decrease the workloads to scale down the number of replicas. The pattern of response time when decreasing the workload is the same as increasing. This is because the default scheduler deletes the pod in a node that has the least available resources. In contrast, our scheduler terminates the pod in a node with a high response time. Regarding the throughput, our scheduler has better results while increasing the replicas until it has three pods; then, we have stable throughputs. This is because placing more than one replica in the same node would result in traffic bottlenecks. Nevertheless, the default scheduler has identical throughput to

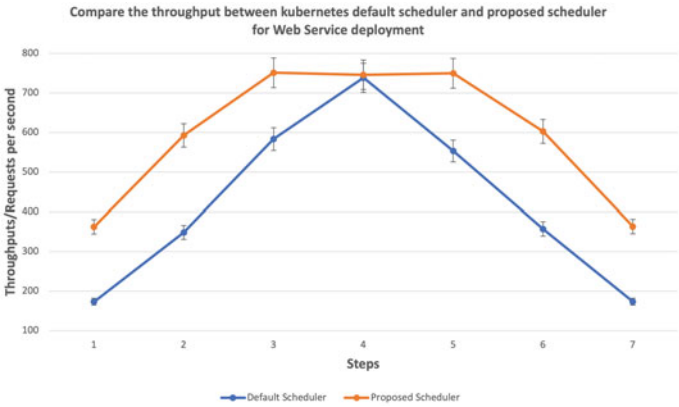

Fig. 6 Throughput of Web service deployment

the proposed scheduler in step 4 in Fig. 6. This is because it places the pods in different nodes.

5.2 Object Detection Service Results

In this section, we explore the results of object detection service deployment and analyse the comparison between the default scheduler and our proposed scheduler. The experiment demonstrates that our scheduler substantially enhances the response time when placing the initial replica. However, after the workloads are increased, the default scheduler reduces the latency and the response time between both schedulers becomes slightly different. Yet, the service delay in our scheduler shows better results when it reaches the maximum number of replicas, as Fig. 7 in step 4 depicts. This is because of assigning the pod to the same node as the first replica. However, the default scheduler tries to place the replicas on different nodes as there are available resources.

Regarding the throughput, the proposed scheduler has higher results than the default one. However, the throughputs of both schedulers have the same outputs when the service has three replicas. This is because default improves the throughputs by placing the pods to different nodes, which increases the opportunity of finding the nodes that can process a request quickly. After that, our scheduler has better throughputs in step 4, as Fig. 8 shows, because it assigns the replica to the node with the initial replica. Moreover, we decrease the workloads to let the system scale down the number of replicas. Thus, the results demonstrate that our scheduler has provided better throughput compared to the default scheduler.

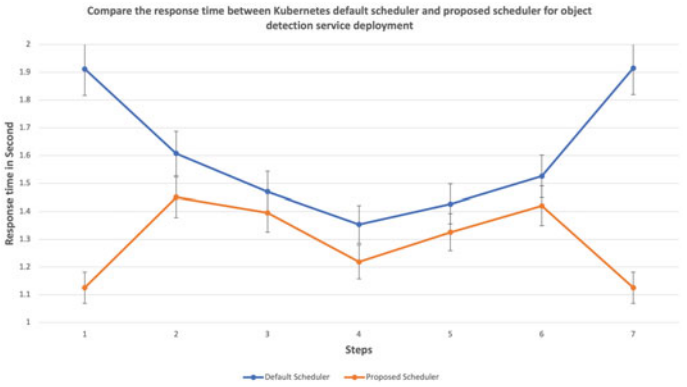

Fig. 7 Response time of Object Detection service deployment

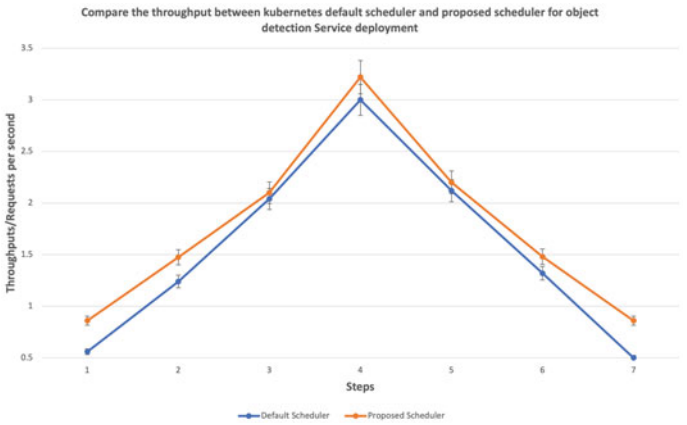

Fig. 8 Throughput of Object Detection service deployment

6 Discussions and Limitations

It is essential to consider the delay between nodes and the execution time of services in nodes to improve the response time in the heterogeneous edge computing environment. Also, maximizing resource utilization can be achieved by activating the autoscaler in the system for scale-up and scale-down replicas. The findings support that our proposed scheduler algorithm is able to minimize the latency and have better throughput than the default scheduler when placing initial replicas. Whereas using KEDA to scale down the replicas shows excellent performance regarding resource utilization, scaling up presents similar results for both schedulers. We evaluated our proposed method in a small cluster which consists of four nodes as we used real emulator that could be ran over a single machine. We set the maximum number of replicas to four as the number of nodes, the default scheduler using spread policy

when placing replicas. Therefore, the possibility of getting the nodes with less delay is high. It is worth testing the proposed scheduler in a larger cluster and comparing it with the default scheduler, which might affect the response time results.

7 Conclusions and Future Work

The rise of edge computing has created a demand for optimizing Kubernetes schedulers. Therefore, many studies have addressed latency by considering delays between nodes, and they evaluate their work using a homogeneous cluster. Yet, the nature of edge nodes is diverse in capacity (CPU and memory), and heterogeneity is an inseparable part of every edge domain. Thus, we proposed a scheduler to consider improving the quality of service and optimizing resource utilization in heterogeneous edge computing environments. We addressed these objectives by considering the delay between nodes, the execution time of different nodes, and adapting an autoscaler such as KEDA and resource schedulers. Our algorithm tries to prioritize the placement of replicas on nodes with the least response time to maintain latency constraints. When decreasing replicas, the proposed scheduler terminates in the reverse order of creation and placement. Moreover, we utilize Kind to emulate heterogeneous clusters for evaluation purposes. The results demonstrate that the proposed scheduler significantly reduces response time and has better throughputs for placing initial replicas. However, scaling up the replicas increases latency slightly and shrinks the difference between the default scheduler and our scheduler. In future, we will evaluate our proposed scheduler in a larger cluster and write a scheduler plugin for filtering and sorting steps.

References

1. Casquero, O., Armentia, A., Sarachaga, I., Pérez, F., Orive, D., Marcos, M.: Distributed scheduling in kubernetes based on mas for fog-in-the-loop applications. In: 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1213–1217. IEEE (2019)
2. Chima Ogbuachi, M., Reale, A., Suskovics, P., Kovács, B.: Context-aware kubernetes scheduler for edge-native applications on 5g. J. Commun. Softw. Syst. **16**(1), 85–94 (2020)
3. Docker: What is a container? (2023). https://www.docker.com/resources/what-container/
4. Eidenbenz, R., Pignolet, Y.A., Ryser, A.: Latency-aware industrial fog application orchestration with kubernetes. In: 2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC), pp. 164–171. IEEE (2020)
5. Fahs, A.J., Pierre, G.: Tail-latency-aware fog application replica placement. In: Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings 18, pp. 508–524. Springer (2020)
6. 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, pp. 136–137 (2019)

7. Jonatan, H., Hugo, H., Carl, B., Joakim, H.: A modern load testing framework (2023). https://locust.io/
8. Katenbrink, F., Seitz, A., Mittermeier, L., Mueller, H., Bruegge, B.: Dynamic scheduling for seamless computing. In: 2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2), pp. 41–48. IEEE (2018)
9. Kayal, P.: Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope. In: 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), pp. 1–6. IEEE (2020)
10. Keda: Kubernetes event-driven autoscaling (2023). https://keda.sh/
11. Kind: Configuration (2023). https://kind.sigs.k8s.io/docs/user/configuration/
12. Kind: Home (2023). https://kind.sigs.k8s.io/
13. Kubernetes: Horizontal pod autoscaling (2023). https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
14. Kubernetes: Kubernetes scheduler (2023). https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
15. Kubernetes: Overview of kubernetes (2023). https://kubernetes.io/docs/concepts/overview/
16. Menouer, T.: Kcss: kubernetes container scheduling strategy. J. Supercomput. **77**(5), 4267–4293 (2021)
17. Meulen, R.: What edge computing means for infrastructure and operations leaders (2018). https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders
18. Nastic, S., Pusztai, T., Morichetta, A., Pujol, V.C., Dustdar, S., Vii, D., Xiong, Y.: Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pp. 206–216. IEEE (2021)
19. Polencic, D.: Scaling kubernetes to zero (and back) (2022). https://www.linode.com/blog/kubernetes/scaling-kubernetes-to-zero-and-back/
20. Pusztai, T., Rossi, F., Dustdar, S.: Pogonip: Scheduling asynchronous applications on the edge. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pp. 660–670. IEEE (2021)
21. Rausch, T., Rashed, A., Dustdar, S.: Optimized container scheduling for data-intensive serverless edge computing. Futur. Gener. Comput. Syst. **114**, 259–271 (2021)
22. Rocha, I., Göttel, C., Felber, P., Pasin, M., Rouvoy, R., Schiavoni, V.: Heats: Heterogeneity- and energy-aware task-based scheduling. In: 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 400–405. IEEE (2019)
23. 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. IEEE (2019)
24. Vilaça, J.P.M.: Orchestration and distribution of services in hybrid cloud/edge environments. Ph.D. thesis, Universidade do Minho (Portugal) (2021)
25. Vouzis, P.: How to use the linux traffic control (2017). https://netbeez.net/blog/how-to-use-the-linux-traffic-control/
26. Wojciechowski, Ł., Opasiak, K., Latusek, J., Wereski, M., Morales, V., Kim, T., Hong, M.: Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In: IEEE INFOCOM 2021-IEEE Conference on Computer Communications, pp. 1–9. IEEE (2021)
27. Yang, S., Ren, Y., Zhang, J., Guan, J., Li, B.: Kubehice: Performance-aware container orchestration on heterogeneous-isa architectures in cloud-edge platforms. In: 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), pp. 81–91. IEEE (2021)