National College of Ireland

# Enhanced Scheduling for KubeEdge nodes in Edge Computing using EdgeMesh

MSc Research Project
Cloud Computing


Rahul Dhanapal Narawade
Student ID: 22144943

School of Computing
National College of Ireland


Supervisor: Vikas Sahni

**National College of Ireland**
**Project Submission Sheet**
**School of Computing**

| | |
|---|---|
| **Student Name:** | Rahul Dhanapal Narawade |
| **Student ID:** | 22144943 |
| **Programme:** | Cloud Computing |
| **Year:** | 2023 |
| **Module:** | MSc Research Project |
| **Supervisor:** | Vikas Sahni |
| **Submission Due Date:** | 14/12/2023 |
| **Project Title:** | Enhanced Scheduling for KubeEdge nodes in Edge Computing using EdgeMesh |
| **Word Count:** | 7949 |
| **Page Count:** | 19 |

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

**ALL** internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

| | |
|---|---|
| **Signature:** | Rahul Dhanapal Narawade |
| **Date:** | 11th December 2023 |

## PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

| | |
|---|---|
| Attach a completed copy of this sheet to each project (including multiple copies). | ☐ |
| **Attach a Moodle submission receipt of the online project submission**, to each project (including multiple copies). | ☐ |
| **You must ensure that you retain a HARD COPY of the project**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. | ☐ |

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

| **Office Use Only** | |
|---|---|
| Signature: | |
| Date: | |
| Penalty Applied (if applicable): | |

# Enhanced Scheduling for KubeEdge nodes in Edge Computing using EdgeMesh

Rahul Dhanapal Narawade
22144943

**Abstract**

Edge Computing processes data at the edge to reduce latency and deliver faster responses by reducing data that has to be synced with the cloud. Strategic load balancing plays a crucial role in maximizing IoT efficiency and reliability. KubeEdge commendably orchestrates containerized applications on edge nodes by extending Kubernetes and maintaining cloud-edge node networks. This facilitates the seamless load balancing and discovery of services on edge nodes through EdgeMesh. However, this approach could decrease KubeEdge cluster throughput and delay services due to the need to forward traffic between edge nodes in different locations. Therefore, a better scheduling algorithm is required than the current default algorithem.

This paper presents a solution that uses EdgeMesh to strategically enhance load balancing to improve the KubeEdge cluster throughput. This solution utilizes the best possible nodes in the cluster to serve the user requests efficiently. The outcome of the study shows that the enhanced custom scheduling outperforms the default scheduling algorithms. The average response time was improved by approximately 33% and requests per second increased by approxmatly 94%. Overall, cluster throughput is enhanced by 63.5%.

**Keywords**- Kubernetes, KubeEdge, Cluster throughput, EdgeMesh, Scheduling, Edge Computing, IOT.

## 1 Introduction

The field of computing is experiencing a notable transformation, especially in the world of the Internet of Things. The change in perspective in consideration involves connecting billions of devices, which creates huge amounts of data at speeds that have never been seen before. This ecosystem of devices and apps must work with very little to no latency to meet the need for quick responses as articulated by Pan and McElhannon (2018). Internet of Things (IoT) devices are made up of many different sensors made by different vendors that can do various things. The rise of smart houses, transportation networks, healthcare facilities, and metropolitan areas depends on these electronic devices.

Despite the success and wide adoption of large-scale data centers by many organizations, IoT has new challenges. The large volume, variation in data, and the latency between edge and cloud are major concerns. Local processing is essential for IoT devices, and edge computing can efficiently do that. Processing raw data at the edge and delivering only aggregate data to the cloud addresses IoT devices' large data processing

burden. According to Pan and McElhannon (2018), edge clouds enable IoT devices to transfer data processing tasks. Edge computing optimizes the framework, improving IoT application efficiency while addressing data size, variety, and latency.

Previously, there was no standard approach to limit application resources on a physical server. Virtualization converts physical servers into virtual servers aiming to lessen the expense of managing numerous physical servers. The virtualization proved cost-effective because it allows several virtual instances on a single physical server, distributing resources efficiently.

The introduction of containerization advances over virtualization, containers share the OS kernel on a particular machine. This goal was achieved by implementing containers as distinct and independent processes within the user space. Thus, one physical machine may efficiently run more programs, requiring fewer virtual machines and operating systems. Kubernetes orchestrates containerized microservice deployment and coordination. B et al. (2023) states that Kubernetes effectively manages deployment size, adapts to variation, and autonomously addresses new issues, enabling dynamic application scaling.

KubeEdge acts as an expansion of Kubernetes for edge computing, enabling effective management of edge nodes and deploying applications at the edge. In particular, KubeEdge effortlessly provides this functionality without necessitating any changes to the core Kubernetes API. The system comprises critical components, specifically specialized edge controllers responsible for managing nodes and their workloads, an exclusive network protocol and the distributed metadata storage system. According to Pan and McElhannon (2018) KubeEdge has been intentionally designed with robustness in mind by enabling it to efficiently manage system failures and cloud-edge offline conditions. This is especially important in cases where edge nodes are disconnected from the cloud.

The load balance optimization topic has gotten a lot of attention which has led many academics to study it to find better ways to use resources. Load balancing is still an essential technique for optimizing resources in which the workload is evenly distributed among worker nodes. This distribution helps to decrease task response times and improve the usage of resources. For the KubeEdge advance configurations EdgeMesh is used for service discovery along with load balancing, but this system faces challenges when these worked nodes are spread across multiple locations, potentially causing delays. The current EdgeMesh load balancing mechanism is inadequate for the full adoption of KubeEdge's dynamics.

It is known that load distribution strategies have an effect on the general performance of the KubeEdge cluster. Thus, additional research was needed to find optimal solutions. In their paper from Kim and Kim (2023), they propose a local schedule method that focuses on assigning work to a single node instead of spreading it out among all nodes. This approach helps to reduce the delays that can happen because of nodes that are far away, but it's also possible to overload a single local node, which would make job execution take longer.

In this study, the researcher looked into the feasibility of using a different method of task scheduling. In this research, the prospect of delegating work to a small set of nodes physically adjacent to the local node where the request was made was explored. The goal of this approach was to achieve a balance by reducing the potential for delays caused by remote nodes and preventing the overloading of any one local node. Work was distributed across the nodes based on how close they were to the local node processing the request and how much capacity they had to process the request. At runtime, these parameters are considered, and the decision of scheduling is made.

## 1.1 Research Question

**RQ1:** To what extent does the proposed enhancement in scheduling improve the throughput of the KubeEdge cluster containing geographically distributed edge nodes?
**RQ2:** How can the delays due to traffic distribution on widely separated edge nodes and the computation capabilities of nodes be considered to enhance the performance of a KubeEdge cluster?

## 1.2 Structure of the Paper

This study report is structured as follows. Section 1 briefly introduces the research area and topic background, and section 2 uncovers the related work along with the research niche. Section 3 explains the investigation methods utilized to address the research objectives. Section 4 discusses the design specifications with a quick summary of KubeEdge and EdgeMesh, existing default scheduling, and solutions to the study objectives. Section 5 uncovers the implementation approach of the solution, and section 6 reveals the study outcome by comparing previous and newly implemented solutions along with the discussion on the overall outcome. The report concludes with section 7, which presents findings and suggestions for future work.

# 2 Related Work

This section explores a range of studies dedicated to Kubernetes, KubeEdge, and the implementation of load balancing techniques aimed at enhancing the throughput of clusters within edge computing.

## 2.1 Kubernetes - Dynamic Load Balancing

The paper Liu et al. (2020) proposes load balancing algorithm for Kubernetes that dynamically distributes load based on real-time data. The MLB algorithm considers including CPU utilization, memory usage, and network IO, to calculate the load of each service. Iptables are utilized for resource management, and the iptables size increases gradually as the number of rules grows which degrades efficiency. The Khalel et al. (2022) proposes the method for deploying load-balanced Kubernetes cluster using Minikube. Service was created for deployment and to distribute requests to the pods in the deployment using a round-robin algorithm. System was evaluated by deploying the MongoDB and Mongo Express service. The paper does not cover with other load balancing technique than the default round-robin in detail.

High-performance cloud-native software load balancer to replace commercial physical load balancers was proposed by Lee et al. (2021). The load balancer implemented as container on Linux server that utilizes eBPF/XDP to manage packets at the data path layer. This allows the load balancer to achieve high performance and scalability. The system evaluated using the RFC2544 performance standard and IMIX traffic. Proposed load balancer results in achieving throughput close to the loopback interface and outperforms Iptables DNAT by up to 27 times. As this was on simulated environment it will be interesting to see in actual environment.

The study Nguyen and Kim (2020) examines the leader election algorithm and its effects on stateful applications with the aim of scalable load balancing. The Kuber-

netes leader election process fails to achieve optimal leader distribution, and a significant number of leaders are assigned to a node. The study demonstrated that Kubernetes' leader election mechanism could maximize leader distribution and increase performance for stateful applications with numerous replicas. The drawback of this method is potential data inconsistency in stateful applications.

The study Takahashi et al. (2018) proposes a portable load balancer for Kubernetes cluster systems. Using the Linux kernel's IPVS, a containerized software-based load balancer was built and set up to run as part of a container cluster. The efficiency of load balancer compared to existing iptables DNAT and Nginx load balancers. The proposed container based IPVS load balancer improves Kubernetes cluster system portability while maintaining performance levels like the iptables DNAT load balancer. Weight-based load balancing approach for canary deployments in Kubernetes using AWS was proposed in B et al. (2023). The authors use Kubernetes, Linkerd, Flagger, load generator, and AWS to measure success rate, latency, and throughput to evaluate the proposal. The proposed approach achieved a 100% success rate, with a P99 latency of 4 ms for the primary app and 9 ms for the canary app. Study lack in comparison of the proposed approach to other load balancing.

## 2.2 Kubernetes Load balancing - Optimization for Edge Computing

The Phuc, Phan and Kim (2022) aims to improve the performance of IoT applications with Traffic Aware Horizontal Pod Autoscaler. The THPA assigns pods to worker nodes using pod scheduling method which considers the network traffic. The Kubernetes cluster with three worker nodes along with simple web application used to test the proposed THPA. THPA succeeded in improving the throughput and response time of IoT apps by 150% than default autoscaler. A two-stage scheduling hybrid workflows on edge cloud computing systems presented in Alsurdeh et al. (2021). Resource estimate begins with gradient descent search and second cluster-based hybrid workflow provisioning and scheduling on heterogeneous edge cloud resources. The solution optimises execution time and cost within deadline and throughput limitations. It would be interesting to test solution with a bigger cluster with more complicated programs like in production environment.

The authors of Ogbuachi et al. (2020) proposes scheduler which considers the physical, operational, and network characteristics of edge devices while assigning the pods. The scheduler uses weighted score to evenly distribute the workload among the nodes,tested on Raspberry Pi cluster. The outcome shows that custom scheduler outperformed than default K8 scheduler in terms of speed and efficiency in scheduling pods. The paper Nguyen et al. (2022) proposes a Resource Adaptive Proxy (RAP) algorithm for Load balancing in K8-based edge computing. The algorithm prioritizes local request and forwards to the optimal remote node when local nodes are overloaded. The authors assessed RAP with a synthetic benchmark under varied conditions. The results shows that RAP improved throughput by up to 400% and reduced latency by approximately 90% compared to the default kube-proxy. It would be interesting to see how RAP performs with real-world traffic.

The paper Wang et al. (2022) proposes A Dynamic algorithm DWRR in Kubernetes clusters with heterogeneous CPUs. DWRR calculates the weight of each pod based on the CPU usage of the node. The algorithm then updates the traffic routing rules, considering

their weights. DWRR significantly outperforms the default load balancing algorithms in Kubernetes in terms of throughput, average response time, and CPU utilization. But this method only works well above a certain level. It might be helpful to use this method in KubeEdge. The author of Furusawa et al. (2022) proposes a co-op load balancing for adjacent edge resources using a service mesh. Service mesh controller used to dynamically set the load balancing destination and weight values for each edge server. The controller monitors the rps of requests to apps on each edge server and calculates weight to forward excess requests to neighbouring nodes. Proposed method significantly reduces response time and prevent edge servers from being overloaded. This algorithm may not be suitable for environments with very low latency requirements.

Authors of Yang et al. (2022) emphasize the necessity of integrating AI and network considerations to deal with edge computing challenges. The objective of the proposed architecture was to optimize the allocation of resources and migration of services. Significant contributions have been made through the integrating AI into network design and for the prioritization of dynamic service provisioning. Shen et al. (2023) proposes learning based scheduling for k8s oriented edge-cloud networks called KaiS. This scheduler uses synchronized multi-agent and actor critic algorithm to distribute request and another GNN-based policy gradient algorithm for service orchestration. KaiS was evaluated on edge-cloud network with 10 k3s primary nodes and 100 k3s edge nodes. The results shown that KaiS achieved 14.3% higher throughput rate and 34.7% lower scheduling cost. The proposed solution was complex, it may not work for all edge-cloud networks.

## 2.3 KubeEdge Integration for Edge Computing

The study Kim and Kim (2023) proposes a local scheduling approach to deal the performance drop caused by traffic routing between edge nodes. Authors created KubeEdge setup on a cluster of three edge nodes and examined its throughput and response time under varied traffic loads and pods. The authors found that round-robin load-balancing reduces system performance and response time due to edge node traffic. But the newly proposed system, regardless of traffic patterns, the local scheduling method exceeds the performance than the round-robin load-balancing. The research could have explored dynamic load distribution instead of overwhelming a single local node.

Authors of Vilaça et al. (n.d.) presents Geolocate scheduler for kubeedge which considers producer and consumer nodes geographic locations. The proposed scheduler first filters the available nodes to find those that fulfil workload resource requirements and are in a suitable geographic region based on latency. Geolocate scheduler was tested using a POC application. The Geolocate scheduler increased response times by 62% when nodes were in the same region. According to the report Xiong et al. (2018) KubeEdge provides unified architecture that seamlessly connects and coordinates cloud-edge resources to enhance the performance. KubeEdge supports RPC between the edge and cloud which creates a unified containers runtime and the means to sync and store metadata. This study uses the Raspberry Pi 3 Model B, and AWS cloud. Evolution measured network latency between the edge and the cloud, but study could have been more comprehensive by exploring a range of scenarios.

The research Ermolenko et al. (2021) explores utilizing microservices to implement distributed edge services for IOT systems. The method uses Kubernetes for orchestration and applications of edge computing. Docker containers along the Container Network Interface (CNI) for networking are used for implementation. The study compares the

latency Apache server as a standalone instance and as part of a Kubernetes cluster. This study revealed that load balancing in the Kubernetes cluster improves response time. The study focuses on response time but lacks scalability, resource usage, and fault tolerance.

The research Wang et al. (2021) uses the KubeEdge for of spectrum recognition, which allows collaborative recognition of objects by effectively managing the integration of cloud servers and edge spectrum recognition nodes. Algorithm undergoes training on the cloud, at the edge node spectrum recognition and categorization takes place. KubeEdge uses Docker to encapsulate the spectrum intelligent detection algorithm in container images which improves distribution efficiency. The study addresses spectrum recognition terminal edge node storage and computational resource restrictions. The scalability and performance implications of this system on edge nodes vs the cloud have not been examined.

Heterogeneity, complicated ecosystems, and inconsistent standards of edge computing are addressed by study Tong et al. (2022). Study proposes KubeEdge for Cloud-Edge device integration and management which unifies resource management. The KubeEdge adds node-level metadata persistence to recover from faults and maintain service readiness in the case of a node offline. Research evaluates performance, resource utilization, and comparison with other edge computing environment. However, the study lacks to address the challenges in large-scale node management and distributed computing. The Le and Yoo (2021) addresses the difficulties associated with the deployment of KubeEdge. Paper presents a practicable lightweight tool solution. KubeEdge installation process and architecture overview highlighted by authors. The paper lacks comprehensive information regarding research design.

The Phuc, Kundroo, Park, Kim and Kim (2022) suggests a node-oriented horizontal pod autoscaler (NHPA) for improving KubeEdge application performance. NHPA was beneficial in edge computing contexts having unstable communication links. NHPA adjusts edge node pod counts based on workload. Pods are scaled up or down based on node CPU usage. The article compares NHPA's throughput and response time to KubeEdge's built-in load balancer. NHPA surpasses KubeEdge's built-in load balancer in throughput and response time, especially in imbalanced traffic. Exploring the impact of delays due to edge node distribution and weak cloud-edge connections would be interesting.

## 2.4 Diverse Approaches using EdgeMesh and ServiceMesh

Sahni et al. (2017) proposes Edge Mesh that focuses on distributed intellect in IoT. EdgeMesh uses genetic algorithm and has three levels corresponding to End devices, Cloud and Edge Mesh. A random task graph and varying network devices were used to test their job allocation technique. The suggested genetic algorithm minimized energy usage significantly. Apart from energy consumption objective, the latency was not considered. Also, the authors noted that more extensive experiments are needed to test in real-world scenarios.

Shitole (2022) study addresses load balancing, and latency issues across microservices. This solution offers integrating ISTIO with service mesh to include side car proxies to services. These proxies direct tasks to appropriate services via control plane. Employing mTLS encryption for security was considered. The response rate exhibited consistency and stability using ISTIO compared to default. Although the request increases, fewer resources are consumed. This suggests an approach to latency of applications that might be considered for the ongoing study.

| Author | Platform | Research | Basis of Solution |
|---|---|---|---|
| Liu et al. (2020) | Kubernetes | Multimetric Load balancing for Kubernetes using real-time data | CPU utilization, memory usage, network IO |
| Khalel et al. (2022) | Kubernetes (Minikube) | Load-balanced Kubernetes cluster by Minikube | Minikube, round-robin algorithm |
| Lee et al. (2021) | Linux-Kubernetes container | Cloud-native software load balancer | Software balancer with iptables DNAT |
| Takahashi et al. (2018) | Kubernetes | Portable load balancer for Kubernetes cluster | Linux kernel's IPVS |
| B et al. (2023) | Kubernetes, Linkerd, Flagger | Weight-based load balancing for canary deployments | Weight assigned to service, traffic split |
| Nguyen et al. (2022) | K8-based edge computing | Resource Adaptive load balancing | Resource (CPU, RAM) and network status of pods |
| Wang et al. (2022) | K8-based edge computing | Dynamic adjustable algorithm in Kubernetes clusters with CPU | The CPU usage of all pods and nodes |
| Furusawa et al. (2022) | Edge Computing service mesh | Co-op load balancing using a service mesh | Based on monitored RPS, requests forwarded to neighboring servers |
| Kim and Kim (2023) | KubeEdge | Local scheduling approach for traffic routing between edge nodes | All requests to local only |
| Vilaça et al. (n.d.) | KubeEdge | Geolocate scheduler for KubeEdge considering geographic locations | Preferred geographic locations considered |

Table 1: Summary of previous research and the basis for solution

## 2.5 Research Niche

Most of the research on Kubernetes and edge computing focuses on the utilization of resources, load balancing, and scheduling. Despite being a crucial edge computing platform, KubeEdge has yet to be extensively investigated for performance. There are few research articles on KubeEdge for load balancing as shown in Table 1. The lack of research on scheduling to enhance cluster performance for distributed edge nodes in KubeEdge utilizing EdgeMesh appears in most studies discussing wider concerns. Previous approaches include restricted alternatives like configuring request processing locally or given location preferences. The goal of this research was to improve the throughput of the cluster using EdgeMesh and a customized algorithm. Therefore, workload could be assigned to nodes while considering latency due to node remote locations and weights considering resources.

# 3 Methodology

This section outlines the research approach utilized to investigate the enhanced scheduling of user requests to geographically scattered KubeEdge nodes.

Preliminary research on the enhancements of KubeEdge cluster has emphasized the importance of scheduling all requests locally to avoid the delay due to traffic forwarding to remote edge nodes Kim and Kim (2023). Though the local scheduling approach may be beneficial in reducing delay due to traffic however may lead to underutilization of remote nodes and overloading of local nodes. It is therefore crucial to consider more than just local node and less than all remote nodes for processing to get efficient cluster performance.

This research focuses on the significance of EdgeMesh in the data plane of the KubeEdge cluster. EdgeMesh enables service discovery and traffic proxy functionalities and assists in the management of intricate networks in edge computing. Sahni et al. (2017) states that edge devices utilizing EdgeMesh coordinate with other nodes at the edge to distribute the workload across all the nodes in the network. As depicted in Figure 1, an EdgeMesh agent will be coupled to every node in the cluster, resulting in the formation of a mesh network composed of geographically dispersed nodes.

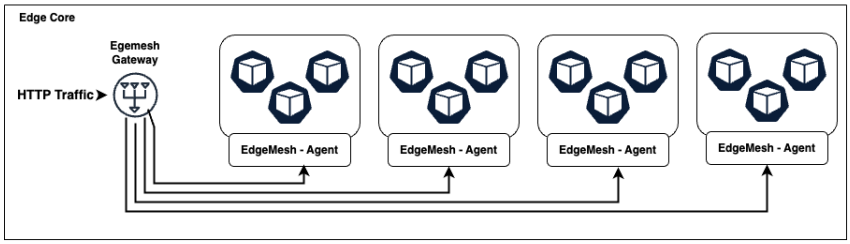


Figure 1: EdgeMesh Agents on KubeEdge nodes

KubeEdge facilitates load balancing via EdgeMesh Kim and Kim (2023). Requests are directed to the EdgeMesh agent followed by agent distributes the load to various nodes by following the load balancing policies that are configured. Under the standard default policy, the requests are evenly distributed among nodes that exist in the cluster. In the case of random policy assigns requests to all nodes randomly. As the KubeEdge cluster could contain the nodes at multiple locations, the inbuilt load balance policy neglects the delays arising from traffic to these long-distance nodes, which negatively impacts the cluster's overall performance.

To address the potential drawbacks associated with default load balancing policies in a KubeEdge cluster, it becomes necessary to improve and deploy a new load balancing policy that considers factors such as latency and the computational capabilities of edge nodes. The enhancement in load distribution can be achieved by using the available metrics related to delay due to location and node utilization. To determine the appropriate destination for the user request, the load balancer necessitated the inclusion of these required supplementary parametersNguyen et al. (2022). In this research, static values of latency and weight are assigned to each available pod. The values mentioned are utilized in the development of novel load-balancing policies. The rationale behind assigning static values has its foundation in the understanding that latency increases for nodes that are located further away, while nodes with higher processing capacity carry

more weight. The proposed algorithm in this study uses these values for decision making of load distribution.

The cluster performance of the experimental setup was evaluated using two key metrics: throughput and response time. Throughput refers to the rate at which a cluster can process a certain number of tasks or requests processed within a given time frame. Throughput provides insights into the cluster's overall processing capacity and efficiency. On the other hand, response time quantifies the time taken by the cluster to respond to a specific task or request. It reflects the measurement of throughput involved in determining the quantity of requests that were successfully processed within a one-second timeframe. Meanwhile, the response time was assessed by calculating the average duration it took for individual requests executed by the edge node, considering the latency incurred during forwarding.

HTTP traffic was generated by using the HEY tool, a widely recognized traffic generator and benchmarking tool [1]. The proposed solution was evaluated by comparing the throughput and response time metrics of the default architecture of KubeEdge and EdgeMesh with the throughput and response time metrics of the enhanced solution.

# 4 Design Specification

The specifications and underlying architecture of the implementation are covered in this section.

## 4.1 KubeEdge and EdgeMesh Architecture

KubeEdge has firmly established its place as a Kubernetes Native Edge Computing Framework inside the CNCF ecosystem by achieving an Incubating maturity level.[2] KubeEdge architecture enables efficient edge computing by reducing transmission costs, minimizing latency, and improving quality of service. Developers could construct applications using either HTTP or MQTT protocols and package them into containers for easy deployment in both Edge and Cloud environments. KubeEdge seamlessly integrates with Kubernetes. It allows users to orchestrate applications, manage devices, and monitor the status of Edge nodes, like how they would do with traditional Cloud-based Kubernetes clusters. KubeEdge enables seamless access to broad array of applications, which includes complex machine learning, image recognition, and event processing. These applications can be easily deployed and operated at the Edge.[3]

The two main modules of KubeEdge Edge and Cloud Core vary from the traditional master-worker architecture of Kubernetes. This architecture manages network connections between edge nodes and the cloud and maintains sessions during breakdown periods. Figure 2 illustrates the architectural framework of KubeEdge.

In addition to standard Kubernetes master node at cloud core, the Cloud Hub a web socket server that communicates with EdgeHub, caches data and monitors changes in the cloud. The Edge Controller, a Kubernetes controller that oversees the management of edge nodes and the metadata of pods. This makes it possible to send data to specific edge nodes. IoT devices data gets synchronised with the cloud via the device controller.

[1]Hey GitHub Repository: `https://github.com/rakyll/hey`

[2]CNCF KubeEdge Project: `https://www.cncf.io/projects/kubeedge/`

[3]Kubeedge: `https://kubeedge.io/`

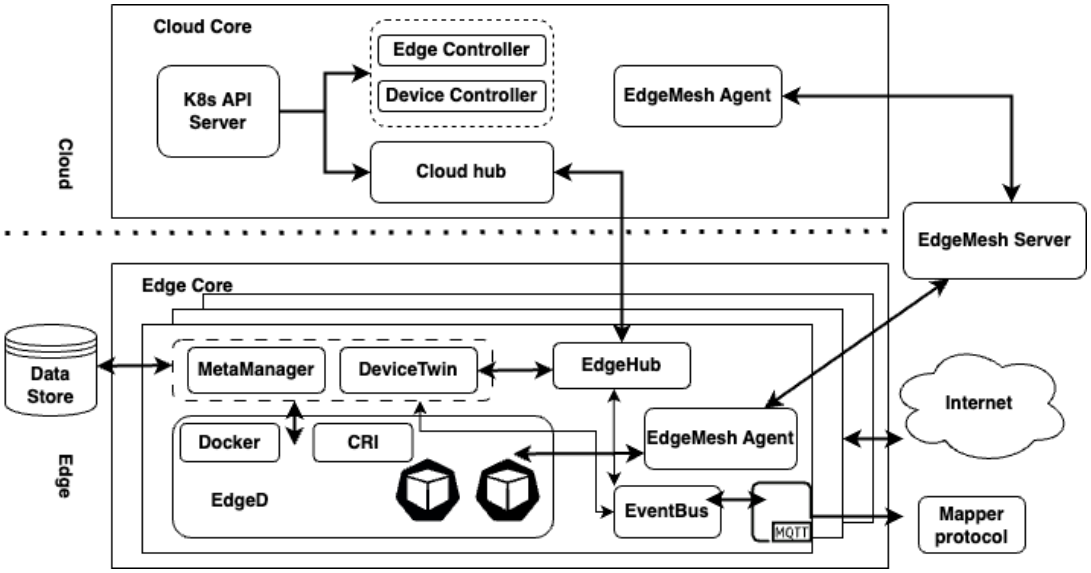


Figure 2: KubeEdge with Edgemesh Kim and Kim (2023)

EdgeD runs containerized applications on edge nodes and supports pod and event lifecycle management, authentication, CRI-container runtime support, and seamless workload deployment. EdgeD works with EdgeHub and manages Docker, containerd, and CRI-O through the Container Runtime Interface on edge nodes. EdgeD supports several container runtimes and allows resource-constrained edge nodes to install containerized workloads for simple telemetry data processing to complicated analytics and ML. The kubectl command line interface lets cloud users easily launch workloads. The edge hub facilitates communication between the cloud and the edge, with a specific emphasis on ensuring that updates made on the cloud side are synchronized with the edge. Additionally, it serves to notify the cloud of changes that occur in the state of hosts as well as devices located at the edge.

DeviceTwin facilitates the storage and synchronization of device status with the cloud, while also offering query functionalities. EventBus functions as a MQTT client, providing components and devices with the ability to publish and subscribe to events. MetaManager acts as a mediator between edged and edgehub, facilitating the storage and retrieval of metadata using a local lightweight database (SQLite). ServiceBus functions as a client for HTTP that facilitates the connection between edge servers and the cloud.

EdgeMesh, an essential component of the cluster, acts as the data plane. It simplifies service identification and enables traffic proxy functions for deployed applications. The main purpose of edge mesh involves protecting applications from the complicated network structures found in edge computing. In the KubeEdge cluster, edge nodes in different areas often disconnect, and seamless communication between applications expected for critical business requirement. EdgeMesh provides a solution to this challenge with lightweight, high availability & reliability capabilities Sahni et al. (2017).

EdgeMesh component proxier configures kernel iptables rules and intercepts request to the EdgeMesh process.Edgemesh agent has built-in DNS resolver that translates DNS requests on the node into service cluster IP. The LoadBalancer forwards requests to backend instances using default load-balancing strategies. Metadata (e.g., Service, Endpoints, Pod) obtained by accessing the Kubernetes or KubeEdge apiserver through con-

troller. Tunnel helps utilization of automatic relay, MDNS, and hole punching to enable communication across subnets.[4]

## 4.2 EdgeMesh Gateway

The Edge Gateway enables access to the internal services within the cluster via the gateway. It acts as an ingress gateway a traffic entry point that indirect incoming request to the appropriate services within the cluster. The gateway handles traffic routing, load balancing, and SSL termination ensuring seamless and secure access to applications deployed on the KubeEdge cluster. [5]

## 4.3 Default Load balancing in KubeEdge

Load balancing evenly distributes the workload throughout the available resources. Precisely, it ensures uninterrupted operation in case of a resource downtime by efficiently allocating resources and service instances. Load distribution reduces costs and directly enhances system performance by decreasing task response time and increasing resource utilization.

KubeEdge balances the load through EdgeMesh by allocating user requests to available pods. The node receives the request and forwards it to the edge mesh agent, that, in accordance with the configured load balancing policy, eventually transmits the information to the remote edge mesh agent. Round Robin and Random distribution are the two policies are used in EdgeMesh as shown in below Figure 3.

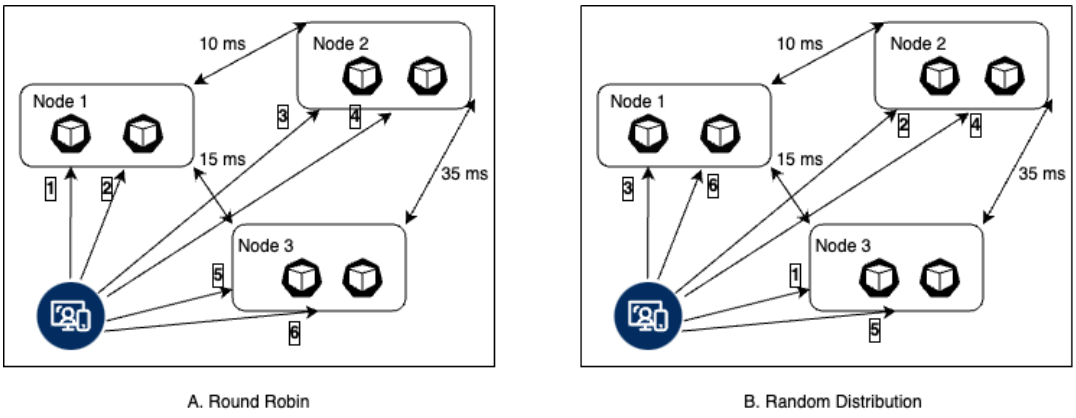


Figure 3: KubeEdge default scheduling policies

Round-robin: In this policy user requests are distributed evenly among all pods. As shown in Figure 3 - A, six pods of applications are deployed among three nodes 1, 2, and 3. Considering six user requests, all six requests distributed equally to each pod. First and second to node 1, third and fourth to node 2 and fifth and sixth transmitted to node 3.

Random Distribution: The user requests are randomly distributed to any pod. As presented in Figure 3 - B, the requests are transmitted to pods in random sequence. This

[4]EdgeMesh Docs: `http://github.com/kubeedge/edgemesh`
[5]EdgeMesh Gateway Docs: `https://edgemesh.netlify.app/guide/edge-gateway.html`

algorithm generates a random number among a total available number of pods and the requests get assigned to that pod.

## 4.4 Enhancement to default scheduling

The default load distribution policies reduce KubeEdge cluster performance due to the delay due to forwarding user requests to geographically distributed edge nodes Kim and Kim (2023). This delay exists due to latency in between the nodes. Also, the resources may vary for the nodes. Therefore, enhanced scheduling necessary to improve the KubeEdge cluster.

In this study, the researcher proposed a technique in which the latency, weight, and number of previously assigned request count of nodes are considered. Also, the proposed solution does not consider the all the pods available in cluster instead it selects the few best pods in terms of latency and capacity to process the user requests.

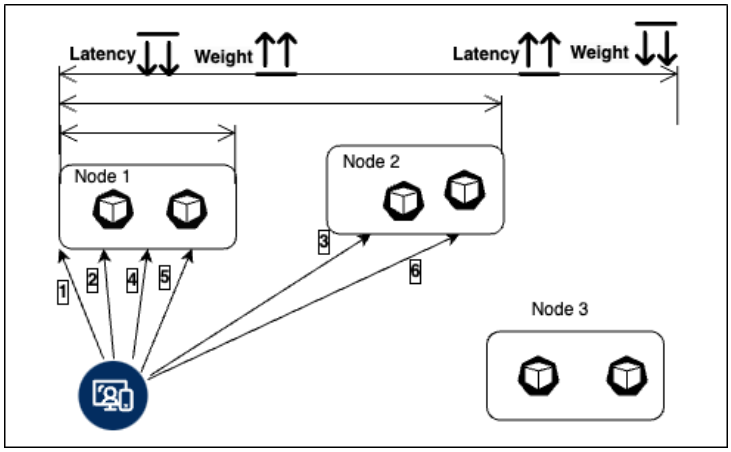


Figure 4: selection of most suitable nodes

The latency of a node signifies the duration required to communicate with each endpoint, whereas weights indicate the relative capability of each endpoint to process requests. In determining which pods are optimal, the solution considers a few factors. Initially, it determines top three pods that has the lowest latency, this ensures faster communication. In situations where multiple pods are available with low latency, it considers the weights and then lower request counters to determine the final endpoint for routing the incoming request. The Figure 4 demonstrates the proposed solution.

The algorithm in Figure 5 aims to proportionally distribute requests in consideration of latency, weights, and historical request counts. This mechanism assures a balanced distribution of requests among the accessible endpoints. This mitigates the risk of individual nodes becoming overloaded and excludes those with higher latency that hinder performance.

# 5 Implementation

This section provides a detailed description of the implementation process for the proposed solution, which utilises EdgeMesh to improve the cluster efficacy of KubeEdge.

**Algorithm 1** Latency and Weight-Based Enhanced Scheduling Algorithm

```
function PICK(endpoints []string, srcAddr net.Addr, cliReq *http.Request)
        selectedEndpoints := rd.selectLowestLatencyEndpoints(endpoints)
    if len(endpoints) > 1 then
            weightedEndpoint := rd.selectWeightedEndpoint(selectedEndpoints)
            rd.requestCount[weightedEndpoint]++
            return weightedEndpoint, cliReq, nil
    end if
        selectedEndpoint := selectedEndpoints[0]
        rd.requestCount[selectedEndpoint]++
        return selectedEndpoint, cliReq, nil
end function
function SELECTLOWESTLATENCYENDPOINTS(endpoints []string)
        Sort endpoints based on latency
        Return the first 3 (half of total) endpoints with the lowest latency
end function
function SELECTWEIGHTEDENDPOINT(endpoints []string)
        weightedRequests := make(map[string]float64)
    for endpoint in endpoints do
            calculate weighted requests (Number of requests/weight of endpoint)
    end for
        Sort endpoints based on weighted requests
        Return the endpoint with the lowest weighted request
end function
```

Figure 5: Enhanced Algorithm

Starting with the setting up of a KubeEdge and EdgeMesh cluster was undertaken. Subsequently, modules that require modification are identified and the necessary modifications are implemented to the EdgeMesh Agent and EdgeMesh Gateway. At the very end, the performance evaluation was conducted by sending HTTP requests.

## 5.1 Experimental cluster

The implementation started with setting up the KubeEdge cluster to study the default behavior and identify how it works. The researcher established the starting point for this study by creating a KubeEdge cluster comprising three edge nodes and one master node on a local system using virtual instances. Master node was deployed with Kubernetes (K8) ecosystem that includes docker (v24.0.6), kubelet(v1.21.1-00), kubeadm (v1.21.1-00 ) and kubectl (v1.21.1-00). K8 initialized at master node using kubeadm init and then the latest KubeEdge (v1.11.3) was installed [6]. The keadm (KubeEdge) initialized with advertising local IP. The token was generated at master node which was used to connect the edge nodes.

At edge nodes, like master node, docker (v24.0.6) and latest KubeEdge (v1.11.3) installed. The edge node was connected to master node by using keadm join along with the advertised master node IP and token to connect generated at master node.

For EdgeMesh, cloned the latest repo of EdgeMesh [7] and applied the crds and resources from build for EdgeMesh agent and gateway both. The sample web application was installed on master with the six instances. The configurations were made to make

[6]KubeEdge: `https://kubeedge.io/`

[7]EdgeMesh: `https://edgemesh.netlify.app/`

the sample application URL accessible outside of VMs. For the purpose of evolution, the hey tool was installed on the local system.

## 5.2 Enhanced Scheduling

EdgeMesh performs load balancing, Utilising an Istio DestinationRule within the service. Kim and Kim (2023) The default load balancer implementation and implemented policies exist in edgmesh/pkg. For this research, the implementation of a random load balancer was modified. The method called 'pick' from 'policy.go', which gives the endpoint from the available endpoint was modified with enhanced code with two additional methods that gives the lowest latency endpoints, and out of those lowest latency endpoints, the highest weight with the lowest request counter endpoint was returned.

The MakeFile was updated with the personal docker repo name to make a docker build. The docker build was made once the load balancer code was modified. The docker file of build/agent/docker file and build/gateway/docker file are updated to include the pkg/loadbalancer changes. The deployment.yaml of agent and daemonset.yaml of gateway updated with an image of personal docker hub locations. After these changes the researcher made the images locally and pushed to the docker hub.

To apply these custom changes at master node, from the cloned repo of EdgeMesh deployment.yaml of agent and daemonset.yaml of gateway at build/resources are updated with the personal docker hub locations and so that the newly modified changes are deployed to master node. Once these changes are modified the yaml files build/agent/resources/ and build/gateway/resources/ are applied. The sample http application was installed with the six pods, and the response to curl was the pod from where the request was served. The test application URL was used with the Hey tool for evaluation.

# 6 Evaluation

This section evaluates the KubeEdge experimental setup and solution described above. The performance of KubeEdge was measured by the average response time and the requests per second processed with increasing concurrent requests. The system was evaluated by comparing the default (random) and enhanced load-balancing strategies.

The researcher evaluated the setup containing one master node and three edge nodes with six pods deployed between three nodes. The sample application URL was given to HeyTool [8], which sends HTTP requests to the URL and provides results. The result contains a response time histogram, latency distribution, and summary. The average response time and Request/sec are the only two results considered for the evaluation from the summary section.

The requests are gradually increased, from 100 requests with 50 concurrent to 800 requests with 400 concurrent. Below are the details of the default and enhanced scheduling evaluation.

## 6.1 Default (Random) Load Balancing Strategy

The random distribution Figure 6 shows the positive relation between the number of requests and average response time. As the number of requests increases, the average

[8]Hey Tool: `https://github.com/rakyll/hey`

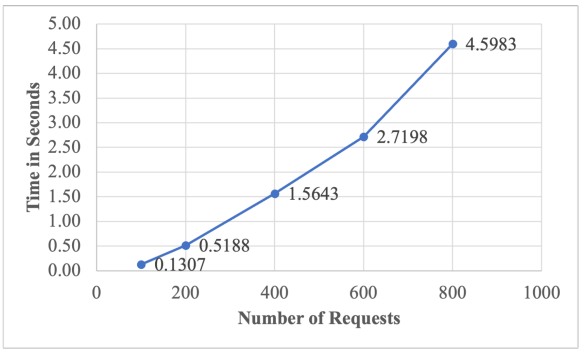


Figure 6: Random-Avg Response time

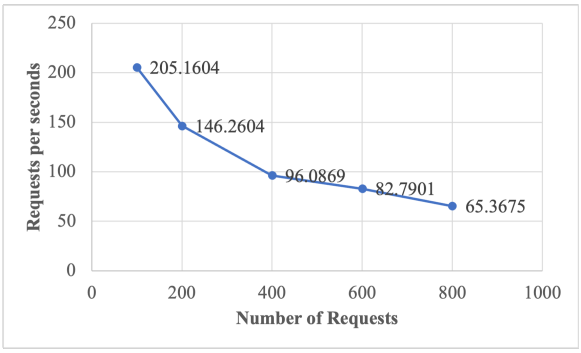


Figure 7: Random-Requests Per Second

response time increases, which results in more time to finish the tasks when the system receives more requests. The system's performance may degrade under a higher load, possibly due to various reasons, such as resource limitations and traffic distribution between all nodes, resulting in network congestion. The random distribution Figure 7 shows the negative relation between the number of requests and requests processed per second. As the number of requests increases, the throughput decreases, which results in more time to finish the requests. The system's performance slowly degrades under a higher load. A downward trend in the number of requests processed per second indicates that the system could not effectively handle large requests.

## 6.2 Enhanced Load Balancing Strategy

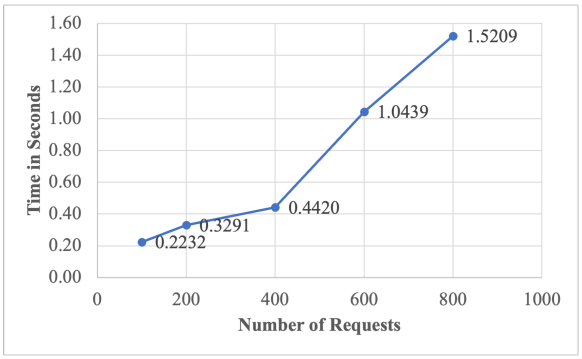


Figure 8: Enhanced-Avg Response time

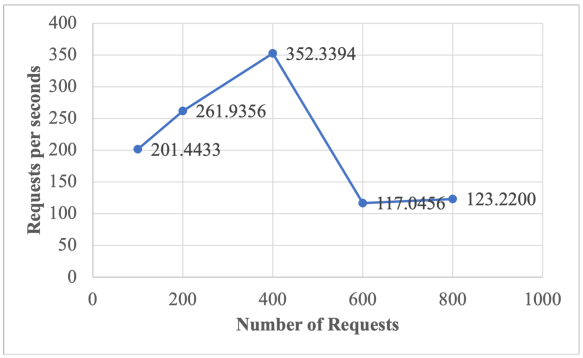


Figure 9: Enhanced-Requests Per Second

Though Figure 8 demonstrates the positive relation between the number of requests and average response time, but it shows the sudden rise in average response time for requests after 400. The average response time gradually increased for 100 to 400 requests. After 400 requests, the average response time starts to increase more rapidly. Similarly, for Figure 9, demonstrates that after a gradual increase in requests processed per second for 100 to 400, the sudden drop in number of requests per second was noticed for 400 to 800.

The possible reason for the sudden increase in response time and sudden decrease in requests per second was that only the top three (50% of total) low-latency nodes were shortlisted for transferring the request. This threshold could be used to adapt the performance based on system requirements.

## 6.3 Default VS Enhanced

As indicated in Figure 10, the average response time for both the default and enhanced systems increases with the number of requests. The enhanced system displays a reduced average response time in all request counts compared to the default system. The results indicate that the enhanced system surpassed the default regarding request processing speed. The difference in average response time between the two systems becomes more noticeable as the volume of requests escalates, indicating that the scalability of the new system excels that of the default system.

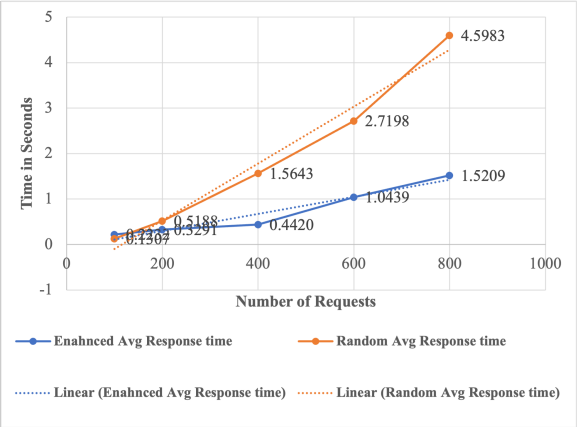


Figure 10: Random v/s Enhanced Avg Response Time

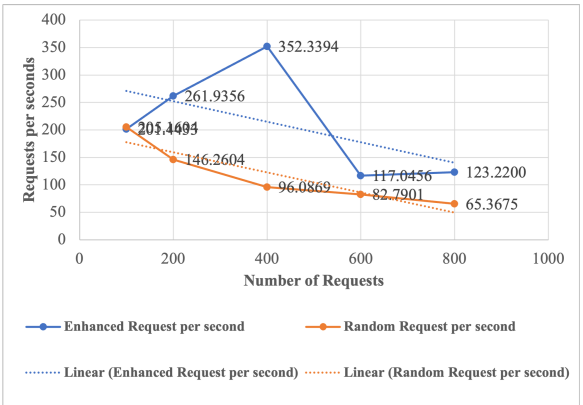


Figure 11: Random v/s Enhanced Request Per Second

Additionally, the enhanced system outperformed the default system in terms of efficiency by utilizing fewer pods to process requests. The default system could only process 800 requests in an average of 4.5983 seconds of response time, whereas the enhanced system could process 800 requests in 1.5209 seconds of average response time. Results indicate that even though requests are processed on limited pods, the enhanced system could process requests three times faster than the default system.

As shown in Figure 11, as the number of requests increases and the response time for both systems increase. However, the enhanced system consistently outperforms the default system regarding requests per second (RPS). At the beginning, for 100 requests, the enhanced system processes 201.4433 requests per second, while the default system processes 205.1604 requests per second, with a difference of 1.84% improvement. The RPS gap becomes even more evident when more requests are processed. The requests per second for the enhanced system was 352.3394 when handling 400 requests, whereas the default system processed 96.0869 requests per second, resulting in a 267.98% enhancement.

However, the RPS of the enhanced system stands at 123.2200 seconds when 800 requests are processed, whereas the response time of the previous system is 65.3675 seconds. The results showed 88.46% better performance. The result of 800 requests was decreased as compared to 400 requests. This drop might be because the requests are not processed on each pod on the cluster; those are processed on limited pods that are the most suitable as expected.

### 6.4 Discussion

The cluster throughput has an impact on geographically distributed edge nodes. Utilizing all the nodes to complete all the requests results in a delay in processing all requests. The scheduling of requests can be adjusted to improve this delay. The existing system uses all the nodes by default, and the previous studies show that using all the nodes downgrades cluster performance.

Utilizing a single node was an option to execute all the requests, but this could result in over-utilizing a single node and under-utilizing other nodes. The waiting for task execution may be more than the actual latency of executing the tasks on remote nodes. This study has attempted to utilize the most suitable nodes based on certain factors, resulting in better performance than the default mechanisms. Certain factors are used as static values; however, these values could be dynamically assessed in the future.

## 7 Conclusion and Future Work

This study investigated the effects of improved scheduling on cluster throughput in the KubeEdge cluster that are affected because of traffic distribution across geographically distributed edge nodes. Scheduling algorithm requires prioritizing low-latency nodes and higher weights for optimal performance. The main objective was to evaluate how new scheduling improves cluster performance compared to default. This study suggests an enhanced scheduling strategy for KubeEdge which prioritizes low-latency nodes and optimizes request distribution with weights. The results of the evaluation demonstrated that enhanced scheduling surpassed default scheduling in throughput based on response time, and requests per second. The average response time had an improvement of approximately 33%, while the number of requests processed per second increased by approximately 94%. As a result, the overall cluster throughput improved by nearly 63.5%. This improvement was achieved by strategically selecting nodes based on factors such as latency, weight, and historical request counts.

The study uncovered that selecting a small number of optimal nodes can improve performance up to a certain threshold. However, for subsequent requests, the performance may be slightly degraded. Nevertheless, this approach was still better than using the default scheduling method. The proposed solution has potential usefulness in IoT device management solutions that employ custom scheduling to efficiently distribute requests among edge nodes, thereby ensuring the reliability and responsiveness of IoT applications.

A potential future research includes the use of dynamic parameters that consider the changing network conditions to determine the latency of nodes and adjust weights to nodes based on their previous performance. Traffic patterns can be incorporated into the scheduling algorithms to optimize the distribution of requests and reduce the congestion in specific nodes.

## References

Alsurdeh, R., Calheiros, R. N., Matawie, K. M. and Javadi, B. (2021). Hybrid workflow scheduling on edge cloud computing systems, *IEEE Access* **9**: 134783–134799.

B, S., T, K., G, N. R., S, N. and Noronha, J. (2023). Weight based Load Balancing in

Kubernetes using AWS, *2023 International Conference on Intelligent Data Communication Technologies and Internet of Things (IDCIoT)* .

Ermolenko, D., Kilicheva, C., Muthanna, A. and Khakimov, A. (2021). Internet of Things Services Orchestration Framework Based on Kubernetes and Edge Computing, *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)* .

Furusawa, T., Abe, H., Okada, K. and Nakao, A. (2022). Service Mesh Controller for Cooperative Load Balancing among Neighboring Edge Servers, *2022 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)* .

Khalel, M. M., Arul Pugazhendhi, M. and Raj, G. R. (2022). Enhanced Load Balancing in Kubernetes Cluster By Minikube, *2022 International Conference on Smart Technologies and Systems for Next Generation Computing (ICSTSN)* .

Kim, S. H. and Kim, T. (2023). Local Scheduling in KubeEdge-Based Edge Computing Environment, *Sensors* **23**(3): 1522.

Le, V. C. and Yoo, M. (2021). Lightweight KubeEdge Tool for Edge Computing Environments, *The Journal of Korean Institute of Communications and Information Sciences* **46**(9): 1507–1514.

Lee, J. B., Yoo, T. H., Lee, E. H., Hwang, B. H., Ahn, S. W. and Cho, C. H. (2021). High-Performance Software Load Balancer for Cloud-Native Architecture, *IEEE Access* **9**: 123704–123716.

Liu, Q., Haihong, E. and Song, M. (2020). The Design of Multi-Metric Load Balancer for Kubernetes, *2020 International Conference on Inventive Computation Technologies (ICICT)* .

Nguyen, N. and Kim, T. (2020). Toward Highly Scalable Load Balancing in Kubernetes Clusters, *IEEE Communications Magazine* **58**(7): 78–83.

Nguyen, Q. M., Phan, L. A. and Kim, T. (2022). Load-Balancing of Kubernetes-Based Edge Computing Infrastructure Using Resource Adaptive Proxy, *Sensors* **22**(8): 2869.

Ogbuachi, M. C., Reale, A., Suskovics, P. and Kovács, B. (2020). Context-Aware Kubernetes Scheduler for Edge-native Applications on 5g, *Journal of communications software and systems* **16**(1): 85–94.

Pan, J. and McElhannon, J. (2018). Future Edge Cloud and Edge Computing for Internet of Things Applications, *IEEE Internet of Things Journal* **5**(1): 439–449.

Phuc, L. H., Kundroo, M., Park, D. H., Kim, S. and Kim, T. (2022). Node-Based Horizontal Pod Autoscaler in KubeEdge-Based Edge Computing Infrastructure, *IEEE Access* **10**: 134417–134426.

Phuc, L. H., Phan, L. A. and Kim, T. (2022). Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure, *IEEE Access* **10**: 18966–18977.

Sahni, Y., Cao, J., Zhang, S. and Yang, L. (2017). Edge Mesh: A New Paradigm to Enable Distributed Intelligence in Internet of Things, *IEEE Access* **5**: 16441–16458.

Shen, S., Han, Y., Wang, X., Wang, S. and Leung, V. C. M. (2023). Collaborative Learning-Based Scheduling for Kubernetes-Oriented Edge-Cloud Network, *IEEE/ACM Transactions on Networking* pp. 1–15.

Shitole, A. (2022). Dynamic load balancing of microservices in kubernetes clusters using service mesh, *Norma* .

Takahashi, K., Aida, K., Tanjo, T. and Sun, J. (2018). A Portable Load Balancer for Kubernetes Cluster, *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region* .

Tong, Y., Jia, X., Qiao, H. and Chen, X. (2022). Cloud-Edge-Device Integration Based On Edge Computing, *2022 3rd International Conference on Computer Science and Management Technology (ICCSMT)* .

Vilaça, J., Paulo, J. and Vilaça, R. (n.d.). Geolocate: A geolocation-aware scheduling system for edge computing, *HASLab - High-Assurance Software Lab, INESC TEC U. Minho, Portugal.* .

Wang, D., Zhou, Y., Chen, J. and Zhang, C. (2021). Research on Spectrum Intelligent Recognition Technology Based on an Edge Processing Framework KubeEdge, *2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP)* .

Wang, Q., Ren, Y., Yang, S., Guan, J., Li, B., Zhang, J. and Tan, Y. (2022). Proxydwrr: A Dynamic Load Balancing Approach for Heterogeneous-CPU Kubernetes Clusters, *2022 IEEE International Conference on Joint Cloud Computing (JCC)* .

Xiong, Y., Sun, Y., Xing, L. and Huang, Y. (2018). Extend Cloud to Edge with KubeEdge, *2018 IEEE/ACM Symposium on Edge Computing (SEC)* .

Yang, T., Ning, J., Lan, D., Zhang, J., Yang, Y., Wang, X. and Taherkordi, A. (2022). Kubeedge Wireless for Integrated Communication and Computing Services Everywhere, *IEEE Wireless Communications* **29**(2): 140–145.