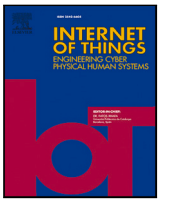
Internet of Things 28 (2024) 101368

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot



EdgeBus: Co-Simulation based resource management for heterogeneous mobile edge computing environments

Babar Ali [a], Muhammed Golec [a,b,*], Sukhpal Singh Gill [a], Huaming Wu [d], Felix Cuadrado [c], Steve Uhlig [a]

[a] School of Electronic Engineering and Computer Science, Queen Mary University of London, United Kingdom
[b] Abdullah Gul University, Kayseri, Turkey
[c] Technical University of Madrid (UPM), Spain
[d] Center for Applied Mathematics, Tianjin University, Tianjin, China

ARTICLE INFO

Keywords:
Edge computing
Internet of Things
Artificial intelligence
Google Kubernetes Engine
Container orchestration

ABSTRACT

Kubernetes has revolutionized traditional monolithic Internet of Things (IoT) applications into lightweight, decentralized, and independent microservices, thus becoming the de facto standard in the realm of container orchestration. Intelligent and efficient container placement in Mobile Edge Computing (MEC) is challenging subjected to user mobility, and surplus but heterogeneous computing resources. One solution to constantly altering user location is to relocate containers closer to the user; however, this leads to additional underutilized active nodes and increases migration's computational overhead. On the contrary, few to no migrations are attributed to higher latency, thus degrading the Quality of Service (QoS). To tackle these challenges, we created a framework named EdgeBus[1], which enables the co-simulation of container resource management in heterogeneous MEC environments based on Kubernetes. It enables the assessment of the impact of container migrations on resource management, energy, and latency. Further, we propose a mobility and migration cost-aware (MANGO) lightweight scheduler for efficient container management by incorporating migration cost, CPU cores, and memory usage for container scheduling. For user mobility, the Cabspotting dataset is employed, which contains real-world traces of taxi mobility in San Francisco. In the EdgeBus framework, we have created a simulated environment aided with a real-world testbed using Google Kubernetes Engine (GKE) to measure the performance of the MANGO scheduler in comparison to baseline schedulers such as IMPALA-based MobileKube, Latency Greedy, and Binpacking. Finally, extensive experiments have been conducted, which demonstrate the effectiveness of the MANGO in terms of latency and number of migrations.

1. Introduction

The Internet of Things (IoT) integrates all intermediary devices along the edge-to-cloud continuum, from miniaturized sensors to computationally intensive servers [1]. Recent advancements in software and hardware have been accountable for the development of intelligent IoT applications envisioning to enhance the quality of life by reaping the benefits of forecasting in all aspects of life,

* Corresponding author at: School of Electronic Engineering and Computer Science, Queen Mary University of London, United Kingdom.
E-mail addresses: b.ali@qmul.ac.uk (B. Ali), m.golec@qmul.ac.uk (M. Golec), s.s.gill@qmul.ac.uk (S.S. Gill), whming@tju.edu.cn (H. Wu), felix.cuadrado@upm.es (F. Cuadrado), steve.uhlig@qmul.ac.uk (S. Uhlig).

[1] The source code is available at https://github.com/BabarAli93/EdgeBus.

https://doi.org/10.1016/j.iot.2024.101368
Received 20 May 2024; Received in revised form 1 August 2024; Accepted 8 September 2024
Available online 12 September 2024

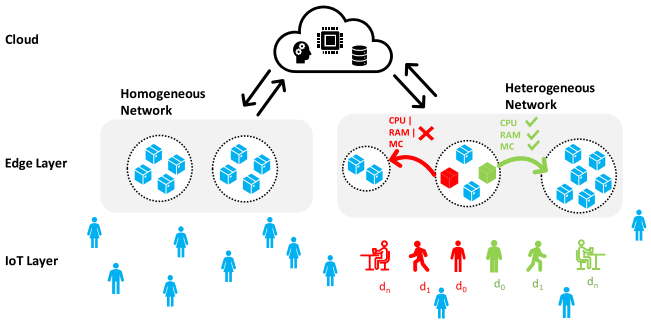


Fig. 1. Challenges of mobility and heterogeneity in resource-constrained MEC environments.

preparing humans for unforeseen events in fields such as health care, smart cities, transportation, and agriculture [2–4]. Mobile devices employing these applications produce an enormous amount of data which requires processing in a timely manner; however, these gadgets with limited power and resources are incapable of executing these tasks on premises [5]. The revolution of IoT applications architecture redesigning from monolithic to microservices remedy these challenges enabling to leveraging of computing resources from IoT to cloud continuum [6].

Microservices are loosely coupled components hosting atomic and independent assigned tasks sharing insights over the network [7]. These services are encapsulated in containers and managed by container orchestration frameworks like Kubernetes and Docker.[2] To handle fault tolerance in overload conditions and conserve resources in underload scenarios, containers hosting application modules are created destroyed, and migrated in a real-time manner. Hosting microservices-enabled IoT applications at cloud [8] can lead to hurt latency-sensitive applications due to bandwidth limitation. Thus, requiring to utilize resources in closer proximity at the highly available, distributed, and heterogeneous Mobile Edge Computing (MEC) paradigm [9]. Additionally, the microservices architecture aligns with the distributed nature of edge where limited available resources can be utilized by deploying microservices, which may be wasted by the failed deployment of a monolithic one.

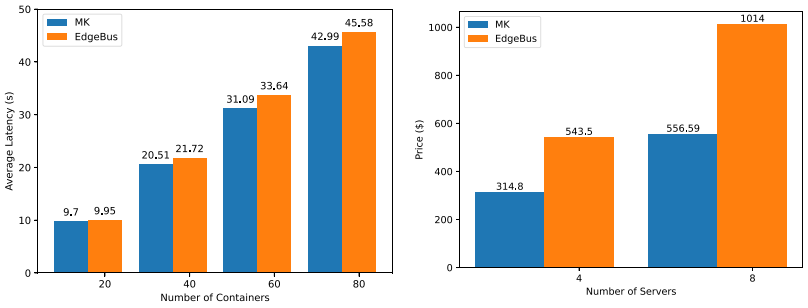
Given that edge devices provisions extensive distribution and supply of heterogeneous computational CPU and memory resources, they require effective container management [10]. To meet user requirements in a timely manner for better Quality of Service (QoS) and efficiently utilize available resources, containers need optimized scheduling and necessary migrations in the edge paradigm. One strategy is to relocate the containers in closer proximity to the user whenever the user's location changes, but it should be noted that container migration incurs cost in terms of latency to the newly selected destination in addition to computational cycle waste on packing and unpacking of containers [11]. Although this overhead and cost are lesser in comparison to Virtual Machines (VMs) (a pioneer in virtualization enablers but mainly replaced by lightweight containers), the cumulative impact is higher for frequent migrations [12]. It causes more time to be spent on migration compared to job execution. On the contrary, few to no migrations result in higher latency. Fig. 1 shows homogeneous (left) and heterogeneous (right) edge computing environments demonstrating container migrations for mobile users. It can be seen that one container migration is successful because of the surplus resource availability. However, the second migration fails as the edge server closer to the user's new location is resource-constrained in heterogeneous settings.

The default Kubernetes scheduler performs balanced container scheduling with no consideration for migrations and rather auto-scales the computational resources. Nodes in the cluster are scored based on the available CPU and memory with the higher score assigned to ones offering balanced load [13]. However, the default Kubernetes scheduler and recent studies explored container migration in the cloud and edge paradigm without considering migration cost and user mobility [14–16]. Some studies employed evolutionary algorithms in the heterogeneous settings for container migrations [17–19], however, these approaches are prone to slow convergence leading to higher overhead [20]. Few researchers modeled the container migration in the edge and cloud paradigm with objectives of energy conservation and employed reinforcement learning algorithms but these works failed to explore heterogeneity and migration cost [21,22].

1.1. Motivating example

As stated by recent studies [23,24], a wide range of devices are situated in the continuum of the edge paradigm, offering varying processing, computational, and storage capabilities. Often, these devices are replaced or upgraded for performance issues, thus constituting a comprehensive network of heterogeneous devices. To meet the strict latency requirements of mobile IoT users, Ghafouri et al. [25] proposed a Kubernetes container scheduling approach by employing Deep Reinforcement Learning (DRL) based IMPALA for homogeneous settings called MobileKube (MK). Fig. 2(a) shows the average network latency comparison of MK and EdgeBus for IMPALA training. EdgeBus implements IMPALA in a heterogeneous environment for a variable number of containers.

[2] https://medium.com/swlh/what-exactly-is-docker-1dd62e1fde38



(a) Average training latency (b) GKE cluster monthly cost

Fig. 2. A motivating example.

It can be seen that IMPALA performs well for homogeneous MK for each scenario, and EdgeBus is lagging in heterogeneous environments depicting the impact of heterogeneity.

In addition, the cost to create a Kubernetes cluster in the Google Kubernetes Engine (GKE) is presented in Fig. 2(b), where the price is in the United States Dollar ($). This price is calculated for four and eight edge server nodes in GKE, and the configurations are given in Table 3. For a homogeneous environment, all four and eight nodes were e2-standard-4. The heterogeneous settings for the four series had one node from each category of e2-small, e2-medium, e2-standard-2, and e2-standard-4, and the eight series had two nodes from each of the aforementioned categories. The cost of a homogeneous environment is twice that of a resource-constrained heterogeneous one. Taking these facts into account, there is a need to investigate container management utilizing a lightweight scheduler for a heterogeneous scenario.

1.2. Our contributions

To address these challenges of mobility and heterogeneity in the resource-constrained MEC paradigm, we developed a coupled-simulation (co-simulation) based framework for Kubernetes-based heterogeneous edge computing environments called **EdgeBus**. Co-simulation enables the production and evaluation of a diverse range of algorithms and models inconsistent settings. Further, we proposed a new **M**obility **AN**d mi**G**ration c**O**st-aware **(MANGO)** scheduler to optimize QoS parameters for effective container resource management. The **main contributions** of this paper are:

- We design a novel framework called EdgeBus, which incorporates heterogeneity in Kubernetes-based edge computing environments to address network latency, container migrations, energy consumption, and resource utilization of nodes. Our focus is on network latency considering the challenges offered by mobile end users, where communication delays between IoT devices and servers can significantly impact overall system performance subjected to locations.
- We propose a lightweight MANGO scheduler for efficient container management by incorporating container migration cost, CPU cores, and memory usage for container scheduling for mobile end users.
- We leverage real-world user activity traces for mobility and integrated this test environment into the Google Cloud Platform.
- A comparison of the MANGO scheduler to other state-of-the-art schedulers is shown, revealing that the MANGO has better performance in terms of latency and number of migrations than baseline schedulers such as IMPALA-based MobileKube [25], Latency Greedy [26], and Binpacking [27]. The result highlights that the proposed EdgeBus performs better than MK for latency and reduces container migrations mainly by the consideration of migration cost along with CPU and memory.

The rest of the paper is organized as follows. Section 2 shows the studies found in the literature. Problem formulation is described in Section 3. The proposed EdgeBus framework is explained in detail in Section 4 highlighting the container placement technique. The performance evaluation against state-of-the-art works is presented in Section 5. Section 6 concludes the paper with future work.

2. Related works

2.1. Heuristic-based approaches

To conserve energy in the containerized data center, Khan et al. proposed a consolidation technique for container migration [15] in the cloud paradigm. The authors formulated container migration to reduce energy consumption and cost by leveraging fewer migrations. Simulation results in the heterogeneous settings showed containers running for extended time periods and recovering their cost. Chhikara et al. [28] presented a best-fit heuristic scheduler for container management to conserve energy in the edge paradigm. Edge server nodes available in the system were clustered into balanced, over-, and under-loaded categories based on CPU and memory insights to relocate containers from overloaded nodes to under-utilized ones. Simulation results showed improved energy consumption with fewer active nodes.

Table 1
Comparison of EdgeBus with existing works.

Work	Edge/ Cloud	Mobility	Heterogeneity	Method	Coupled simulation	Migration cost	Performance parameters		
							Energy	Latency	Migrations
[15]	Cloud		✓	Heuristic			✓		✓
[26]	Hybrid		✓	Heuristic			✓	✓	
[27]	Cloud		✓	Heuristic		✓	✓		✓
[28]	Edge			Heuristic			✓		
[21]	Edge			RL				✓	
[29]	Cloud			RL		✓		✓	
[13]	Edge		✓	DRL				✓	
[30]	Edge	✓		DRL		✓			✓
[31]	Edge	✓		DRL					
[32]	Edge	✓		DRL		✓		✓	
[25]	Edge	✓		DRL	✓		✓	✓	
[17]	Cloud		✓	Evolutionary/ AC					
[19]	Cloud		✓	Evolutionary/ GA				✓	
EdgeBus	Edge	✓	✓	Heuristic	✓	✓	✓	✓	✓

Tang et al. [26] proposed a task scheduling response time greedy technique for collaborative edge and cloud environment. Provisioning high priority to smaller tasks, this work aimed to minimize response time, cost, and energy. Simulation results showed improved response time, cost, and energy to counterparts. To conserve energy in the heterogeneous cloud servers, Farahnakian et al. [27] proposed an energy greedy consolidation algorithm. It leverages current and predicted usage of VMs to initiate container migrations aiming to lessen the number of active nodes. Simulation results showed improved SLA and energy conservation to counterparts.

2.2. Reinforcement learning (RL)-based approaches

To reduce latency in the Edge paradigm, Garg et al. [21] proposed a RL-based strategy. A container was either held on the current node or duplicated on the closest node by evaluating the computed latency of a user to the closest node against a threshold. The proposed work improved latency at the price of scheduling overhead. Taking into consideration the cost associated with each service in terms of delay, Cao et al. [29] proposed a Q-Learning-based service migration scheme for the cloud paradigm. RL agent employed historical data of service load to train and the services were migrated to nodes with higher Q-values. Results from experiments conducted on a small scale showed reduced delay.

2.3. Deep reinforcement learning (DRL)-based approaches

Jian et al. proposed a Deep Q Network [13] based technique for pod management to enhance cluster resource utilization. Resources such as CPU, memory, disk usage, and network constituted a system's state, whereas a set of nodes that could satisfy pod requirements constituted an action. Kubernetes testbed experiments in heterogeneous settings showed balanced resource utilization at the expense of increased scheduling overhead. Brandherm et al. [30] proposed BigMEC to improve service placements and lessen container migrations. By exploiting migration cost in container migration decision-making, authors anticipated user mobility patterns and implemented decentralized Double Q-learning to produce efficient service migration. BigMEC reduced the number of container migrations in the simulated environment with increased scheduling overhead.

Yamansavascilar et al. [31] proposed a Double DQN-based task offloading technique to reduce the failure rate in the context of mobile IoT users in the edge paradigm. DRL states were comprised of IoT applications with distinctive characteristics and dynamic network conditions while the action was task assignment to the edge server node, ensuring task completion. Experimental results for different IoT applications in homogeneous settings demonstrated improved task failure rate and utilization.

In the Fog/Edge computing paradigm, Tang et al. [32] proposed a container migration technique leveraging the deep Q-Learning algorithm. They took into account user mobility and migration costs for state creation while the container was relocated to nodes offering delay reduction. The proposed work improved the overall delay of the system; however, it lacked exploration in a heterogeneous environment.

2.4. Evolutionary algorithms (EA)-based approaches

Several studies have employed EAs to represent the container management problem, particularly using gradient-free methods. The container placement challenges were modeled by Ant Colony (AC) [17] and Genetic (GA) [19] optimization algorithms with a range of objectives, such as cluster load, failure rate, network overhead, power consumption, etc. From the initial container placements in the cloud-based devices, populations were created, and these populations improved over time. In comparison to counterpart EAs, these techniques performed well for the desired objective functions in the simulated heterogeneous environments. When looking for a global solution to high dimensional issues, EAs, especially the gradient-free ones, offered significant computing costs and they were prone to slow convergence [33] but the diversity of population led to a global solution.

2.5. Existing frameworks

iFogSim2 [34] is the latest and advanced version of the fog-cloud simulation toolkit enabling container migrations and provisioning mobility support for IoT applications. It is a Java-based simulator allowing researchers to deploy and evaluate resource management policies. Despite its resource-rich capabilities, it lacks support for real testbed implementation. Tuli et al. proposed Python-based COSCO [20] for container orchestration. COSCO provisions resource management techniques development and evaluation in both simulations and Microsoft Azure real testbed implementation. However, this work does not provide user mobility support.

2.6. Critical analysis

Table 1 compares EdgeBus with existing works belonging to different families of schedulers. All the research works investigated container scheduling with differing objectives, mainly measuring the performance by simulations in cloud and edge. None of the container management solutions for the cloud considered user mobility, and these migrations were principally influenced by workload conditions. The evolutionary algorithms [17,19] primarily focused on the heterogeneity in the cloud to improve latency or conserve energy but fell short of addressing user mobility. Research work [30] incorporated migration cost into decision-making for mobile scenarios in simulation setups but failed to explore heterogeneity and migrations, which cannot be overlooked because of significant overhead. Only [13] explored heterogeneity at the edge layer but failed to consider mobility along with migration cost. Moreover, iFogSim2 [34] provisions user mobility, heterogeneity, and container migration but failed real testbed experimentation. COSCO [20] is closest in characteristics to EdgeBus offering simulated and real testbed experimentation. However, it does not offer user mobility.

Therefore, there is a need for a lightweight scheduler to investigate Kubernetes container management in the inherent mobile and heterogeneous edge paradigm. It should be responsible for finding the right balance between migrations and latency. Lesser migrations lead to lesser overhead and increased fruitful computation time but it can pose higher latency. To conduct quality migrations while maintaining the QoS, the scheduler needs to incorporate migration cost as a key decision-maker. It should measure the effectiveness of latency, energy consumption, and migrations. Last but not least, simulations can never face the challenges and uncertainty of real-world machines, therefore, it should provision support for validation in the real-world testbed. EdgeBus is a framework that enables both simulation and real-world implementation and offers schedulers to measure the effectiveness of the mentioned QoS parameters.

3. Problem formulation

This section presents problem formulation of container management in the heterogeneous edge computing environment for mobile end users. Let $N = \{n_1(\omega_1^C, \mu_1^C), n_2(\omega_2^C, \mu_2^C), \dots, n_k(\omega_k^C, \mu_k^C)\}$ be the list of heterogeneous edge server nodes where ω^C and μ^C represents CPU cores and memory capacities, respectively. $S = \{s_1(\omega^r, \mu^r), s_2(\omega^r, \mu^r), \dots, s_k(\omega^r, \mu^r)\}$ is the list of services or containers hosting user application modules where ω^r and μ^r represents the amount of CPU cores and memory requests made by the containers to run their tasks, respectively. $U = \{u_1, u_2, u_3, \dots, u_k\}$ be the list of mobile users in the network system. Every user is associated with exactly one service $u_k \rightarrow s_k$. Each service can be deployed at any node and all the services are distinct. A container $s_k(\omega_k^r, \mu_k^r)$ migration is subjected to data rate R, associated user location $u_k^l(t)$ at time t, new destination node $n_k(\omega_k, \mu_k)$ current CPU and memory utilization. Table 2 lists the descriptions of notations used in this work.

3.1. Objective function

The primary objective is to reduce the overall cost of the system by reducing the latency, energy consumption and number of container migrations which ultimately lead to improved QoS performance. The objective function $C(t)$ can be formulated as:

$$\min \quad C(t) = w_1 D_{total}(t) + w_2 P_{total}(t) + w_3 M_{total}(t) \tag{1}$$

$$s.t \quad \sum_{s=1 \in n_k}^{S} s(\omega^r) \leq n_k(\omega_k^C) \quad \forall k, \tag{2}$$

$$\sum_{s=1 \in n_k}^{S} s(\mu^r) \leq n_k(\mu_k^C) \quad \forall k \tag{3}$$

where $C(t)$ is the cost of the system at time t to be minimized calculated by total latency $D_{tota}(t)$, power $P_{total}(t)$ and total migration cost of containers $M_{total}(t)$. w_1, w_2, and w_3 are the relative weights to set the importance of sub-goals. The objective is constrained to the CPU and memory capacities of edge servers. CPU $s(\omega^r)$ and memory $s(\mu^r)$ resources occupied by all the containers in a given node n_k must not exceed the maximum capacity of node $n_k(\omega_k^C, \mu_k^C)$.

The system delay/ latency can be calculated from the network D_{net} and queuing delays D_q. Queuing delay drops to zero since the objective is not to overload any node and leave any services in a pending state. Thus, a service is either migrated or keeps running on the current node.

$$D_{total}(t) = D_{net}(t) + D_q(t) \tag{4}$$

Table 2
Symbols and definitions.

Symbol	Definition
N	Set of edge server nodes
S	Set of container and/or services
U	List of mobile users
E	Total episodes
T	Total time interval in an episode
$t\ \varepsilon\ T$	One time step of T
$C(t)$	The system cost at time t
$D_{total}(t)$	Total delay of the system at time t
$D_{net}(t)$	Network delay of the system at time t
$d_k(t)$	Delay of user k at time t
$P_{total}(t)$	Total power of the system at time t
$M_{total}(t)$	Total migration cost in the system at time t
λ_k	Migration time of container k
K	Total number of migrations to perform at time t
δ	Container turning on and off overhead
ω^C	CPU cores capacity of node
μ^C	Memory capacity of node
$s(\omega_k^r)$	CPU cores requested by container k
$s(\mu_k^r)$	Memory requested by container k or container size
l	Location
n^l	Edge server node location
$u^l(t)$	Location of a user at time t
$U^l(t)$	Location of all users at time t
P_n^{idle}	Idle power of node n
P_n^{max}	Max power of node n
P_n^{min}	Min power of node n
$util_k(t)$	Utilization of a node k at time t
O	Array containing current containers placementand user connectivity details
R	Data rate
τ_{prop}	Propagation delay of container migration
τ_{proc}	Processing delay of container migration
p	Current container placements on nodes
p'	New container placement decision produced by scheduler
d_k	Delay of user k to its connected service

$$D_{net}(t) = \frac{\sum_{u=1}^{U} d_u}{|U|} \tag{5}$$

At any time step t, $D_{net}(t)$ can be calculated from the network latency of all the mobile users represented by $|U|$ to their connected edge server node. The power of the system can be calculated by [32]:

$$P_{total}(t) = \int_{t-1}^{t} (\sum_{n=1}^{N} (P_n^{idle} + (P_n^{max} - P_n^{min}) \times util_n(t))) \mathrm{d}t \tag{6}$$

This equation calculates the total power consumption of the system between $t-1$ to t, it comes from the idle and dynamic power subjected to the utilization of a node. This equation further integrates the power of all the nodes in the given interval. The higher the utilization, the higher will be the power consumption. While utilization of a node in our system can be calculated by:

$$util_k(t) = \sum_{s=1 \in n_k}^{S} s(\omega^r, \mu^r) \in n_k(\omega_k^C, \mu_k^C) \tag{7}$$

Utilization is related to the total resources requested by the containers residing on a given node n_k at time t. Migration cost associated with a container is measured in terms of the time taken by a container to migrate and it depends upon multiple factors including container size in terms of memory $s(\mu^r)$, data rate R, propagation, and processing delays. Total migration cost $M_{total}(t)$ at any time t can be calculated by [32]:

$$M_{total}(t) = \int \sum_{s=1}^{S} \lambda_s(t) dt \quad \textbf{if } u^l(t-1) \neq u^l(t), \tag{8}$$

where λ is the amount of time taken by a container to migrate if the location of the user $u^l(t)$ connected to subjected container s_k is different at the current time step t in comparison to previous time step $t-1$ and the new node is ready to host it. If the user does not move or the new node does not have the capacity, then the container stays at the same node and λ_s is zero for this container. The higher the number of migrations, the higher the cost. The cost associated with a container can be calculated by considering container transmission, propagation, and processing times following:

$$\lambda = \frac{s(\mu^r)}{R} + \tau_{prop} + \tau_{proc} + \delta \tag{9}$$

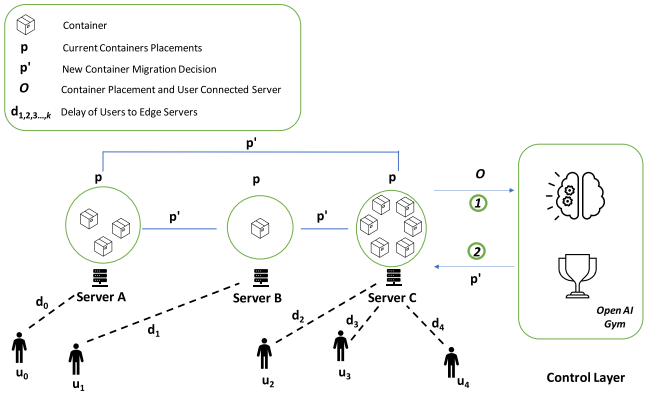


Fig. 3. The overall architecture of EdgeBus.

where first term is the transmission, followed by propagation τ_{prop}, processing τ_{proc} times and migration overhead δ. τ_{prop} and τ_{proc} are zero due to lack of support for live/ physical container migration in GKE [25]. Therefore, a container is turned off at the source and turned on at a new location in case of migration. Finally, δ is the overhead of a container to shut down at the source and turn on at the new destination.

4. The EdgeBus framework

4.1. System architecture

The EdgeBus system architecture is illustrated in Fig. 3. To create a complete experimental setup, a series of steps must be executed. Initially, edge server nodes will be created to host user applications. It is feasible to supply customized CPU and memory configurations to build a heterogeneous or homogeneous cluster, depending on the requirements. Subsequently, services and users are created. We can define the number of services/users as per the desired network size. The next step involves the creation of workload, which is explained in Section 5.1.5. There are two approaches for edge server nodes and users deployment. In the training mode of DRL algorithms, edge server nodes, and users are deployed at random locations. Meanwhile, real-world traces are employed for testing.

User mobility traces are produced for specified experiment time steps for each mode. At last, each user is associated with a service in closer proximity, and each service is deployed at a random node at the start of experimentation followed by service migrations during the experimentation. Initialization can also be seen in Algorithm 1.

Fig. 3 shows heterogeneous edge servers nodes A, B, and C containing several services based on their capacities. O is a concatenated list showing the current placement of containers on the edge server nodes and the connectivity of each user to the associated server (explained in Section 4.2). p is the current container placements while p' is the new container placement decision received from the scheduler. This newly received action is implemented and the containers are moved to new locations. The container will stay in these nodes until the next time step. d_k is the latency/ delay experienced by user k subjected to user location and associated service placement. It varies with user mobility and container migration decisions. The cumulative latency of all the users is calculated by Eq. (5).

4.2. Gym environment

This section explains the implementation of a custom gymnasium environment[3] used in this work to model the nodes, services, and users. There are states and actions for each time step. In this work, the state is defined as the current placement of containers on the nodes and the association of each user to the node containing the service. $U = [u_1, u_2, \dots, u_k]$ is an array of users where the index represents each user ID and the corresponding value is the connected server ID. $S = [s_1, s_2, \dots, s_k]$ is an array of services where index is the service ID and the value is the node ID where this service is currently placed. The concatenation of these arrays $O = U \bigcup S$ makes a state of our scenario, which can be called an observation.

At the end of time step $t-1$, the location of all the users $U^l(t)$ is updated. The scheduler receives these updated locations along with the state explained above. These values are employed to make a decision for each container location subjected to user mobility, edge

[3] https://zenodo.org/record/8127025

Algorithm 1 Network Generation

```
Output: Edge Server and User Placement
Begin
for i ← 1 to n servers do
    CreateServer(CPU, RAM)                          ▷ Heterogeneous or homogeneous server nodes
    Populate servers list
end for
for i ← 1 to n services do
    CreateService(CPU, RAM)                         ▷ Container creation
    Populate containers list
end for
Generate constant workload
if training then
    Deploy Edge servers at random locations
else
    Position servers based on dataset locations
end if
if training then
    Place users at random
    for i ← 1 to n users do
        Generate random user mobility traces
    end for
else
    Position users on dataset location
    for i ← 1 to n users do
        Generate cab spotting-based user mobility traces
    end for
end if
Associate users to servers
Initial placement of containers on nodes
End
```

node resources, and migration cost for the next interval t. This decision is known to be an action as per the gymnasium definition represented by p' in Fig. 3. This action is an array of container size lists, where the index is the container ID and the value represents the updated edge server node. This is an updated list of S.

Every gymnasium environment needs to define a mechanism to judge an action as legal or illegal. It sets action limits and boundaries. As stated earlier, that action is a container placement decision, thus an illegal action is an attempt to assign containers more than the capacity of the edge server, i.e. $n_k(\omega_k^C, \mu_k^C)$. For each illegal action, EdgeBus gives a -1 penalty. The terminal state in the gymnasium is the state when the current episode finishes, and in this custom environment, there is only one way of terminating, which is to run for all the time steps. At the end of an episode, evaluation metrics including average latency, number of empty servers, and migrations are calculated, which are explained in Section 5.2.

4.3. Network and mobility

EdgeBus simulates edge servers, end users' deployment, and users' mobility. As per details in Algorithm 1, these actions are either random or dataset oriented where dataset details are given in Section 5.1.4. EdgeBus employed *NetworkX* [35] Python package to create a graph network followed by edge servers deployment and user association to the closest server. We generate user mobility traces at this stage of the total time steps T for all the users. These traces hold each user's longitude and latitude values at each time step. These values are employed during the experimentation to update locations and container migrations. Considering the mobility dataset, user locations extracted at 5-minute intervals define user mobility. There are no external factors affecting locations or user speed. On the contrary, user mobility is random for the training phase and these randomly produced longitude and latitude values are influenced by a few factors including user speed and time steps T. Thus, at each time step t, we have the location coordinates of each user for the experiment lifespan.

4.4. Kubernetes application

Kubernetes works on a request and limit model in which a container $s_k(\omega^r, \mu^r)$ requests for minimum resources required for the hosted applications inside the container from the edge server node. The limit should be equal to or higher than the requests for the reason that the subjected container may acquire dynamic resources to execute heavy workloads. Kubernetes can host any kind of application and in this research work, we are using a stateless application. This application module does not retain any state at the

time of migration. To do so, EdgeBus schedules the Linux tool stress-ng to generate artificial CPU and memory load in the container after creation.

Algorithm 2 MANGO Scheduler

1: **Input**: Network, Workload, Observation, Migration Cost
2: **Output**: Migration Decision (μ)
3:
4: **procedure** CONTAINER –MIGRATION
5: **for** $t = 1$, t++, while $t < T$ **do**
6: $U^l(t)$, O, $s(\omega^r)$, $s(\mu^r)$, $util_N(t-1)$, $S(t-1)$
7: **for** $u = 1$, u++, while $u < U$ **do**
8: **for** $n = 1$, n++, while $n < N$ **do**
9: $\Delta d[n]$ = latency($u^l(t)$, n^l)
10: **end for**
11: **for** $j = 1$, j++, while $j < |\Delta d|$ **do**
12: **if** $s(\omega^r, \mu^r) + util_j(\omega_j, \mu_j)(t-1) \leq n_j(\omega_j^C, \mu_j^C)$ and $\lambda_s < \Delta d[j]$ **then**
13: update node in list S
14: break
15: **end if**
16: **end for**
17: **end for**
18: migrate containers to new nodes in $S(t)$
19: **end for**
20: **end procedure**

4.5. MANGO scheduler

Algorithm 2 shows the mechanism for container migration decision-making of MANGO. The scheduler ingests network information, workloads, observation O and migration cost λ calculated from Eq. (9). At the end of the time step $t-1$, each user location $U^l(t)$ is updated and the scheduler has to decide for each service $s_k(\omega^r, \mu^r)$. It receives previous decision O from the environment of the time step $t-1$ containing container placement and user connection information. MANGO calculates latency Δd of a user u_k with all the edge server nodes N shown in step 9 of Algorithm 2.

To improve system performance and reduce latency, MANGO aims to select a node offering the lowest delay for each user. However, the migration decision is not only subjected to CPU and memory resources mentioned in Eq. (2), (3) but it also takes into consideration the migration cost. Therefore, the selected node is evaluated in terms of CPU and memory resources in the first place. If the selected node has enough CPU and memory capacity to accommodate the subjected container, the next job is to evaluate the migration cost posed by the container. If the container $s_k(\omega^r, \mu^r)$ migration cost λ by Eq. (9) is less than $\Delta d[j]$ which is the delay of the user to server j, the container can be migrated to new server j closer to the user and the node ID for this container is updated in the S container list. On the contrary, when the migration cost to server j is higher than the user delay $\Delta d[j]$ to server j, the container is not migrated because this leads to service unavailability for longer time intervals thus degrading user experience

Underlined resource-constrained heterogeneous environments and uncertain user mobility patterns can lead to overloading edge server nodes in crowded situations. MANGO foresees this challenge and it explores other nodes offering delays lower than the migration cost. It should be noted that excessive migrations can lead to higher overhead in real-world implementations, therefore, to find the right balance between migrations count and latency, MANGO chooses a new destination for each user from the two closest nodes in the range. If the user location at t is the same as $t-1$, the container will stay in the same node as it is already on the closest node. After evaluating each user, the migration decision p' is implemented and the containers are migrated to new nodes given in the list S.

4.6. Complexity analysis

In the MANGO scheduler, the factors contributing to the computational complexity include the total time steps of experimentation T, the number of end-user/services U in the vicinity, and the total number of edge server nodes N provisioning services to end users. In such a case, the computational cost can be expressed as $T \times U \times (N+N)$. For each user, MANGO calculates latency to the available edge server node and sorts this latency array of size N in ascending order. Later, this array is employed for migration decision-making leaving with the complexity of $T \times U \times 2N$. At step 18 of Algorithm 2, the number of migrations is estimated from the difference of $S(t-1)$ and $S(t)$. Let K be the number of migrations and δ is the overhead for one migration, the total migration overhead at t becomes $K \times \delta$. The best case is when there are no migrations at all in the system with a complexity of $T \times U \times 2N$. The worst case happens when the users are highly mobile and all the services are migrated at every time step. The worst case complexity is $T \times (K \times \delta) \times (U \times 2N)$.

Table 3
Computational nodes and service configurations.

Computational modules	Type	Memory	CPU	Quantity
Nodes	e2-small	2 GB	1	2
	e2-medium	4 GB	1.5	2
	e2-standard-2	8 GB	2	2
	e2-standard-4	16 GB	4	2
Services	type-1	250 MB	0.125	20
				40
				60
				80

5. Performance evaluation

5.1. Experimental setup

In this section, we explain the experimental setup, workloads, baseline schemes, and results. We devise a hybrid experimental setup to validate effectiveness in both simulation and real-world GKE testbed environments. To incorporate heterogeneity, four different types of Kubernetes nodes were deployed. These nodes offer heterogeneous computational resources in terms of memory and CPU cores. Details about the node type, provisioned RAM, number of CPU cores, and number of nodes for each category are given in Table 3. All the compared schemes are trained and evaluated for eight edge server nodes in both the simulated and GKE testbed. GKE provisions CPU in millicores, and one core equals 1000 millicores. Subsequently, as per the request and limit model explained in Section 4.4, each Kubernetes container requests 250 MB of memory and 125 millicores of CPU. Both the request and limit configurations are kept the same to avoid uncertainty in decision-making. Considering the single-user association to one service, 20, 40, 60, and 80 containers have 20, 40, 60, and 80 users deployed in the network, respectively.

5.1.1. Implementation of EdgeBus

This section presents EdgeBus framework implementation details. We use Python programming language, utilizing an object-oriented design approach. Fig. 4 illustrates the UML class diagram of the EdgeBus architecture. The Network Generator module is used to create a complete network, including edge server nodes, users, and mobility traces, as explained in Algorithm 1. Open AI Gymnasium enables the modeling of real-world container migration problems, and we created a custom gym environment for this purpose. Sim-Mango and Kube-Mango are inherited schedulers for simulation and the GKE testbed, respectively. The migration decision issued by the scheduler is executed using the $\_migrate()$ method. Finally, the Mobility Simulator module is responsible for the continuous alteration of user location at each time step. Both Sim-Mango and Kube-Mango invoke $mango\_action()$ of the Mobility Simulator, and it generates container migration action for the next time step utilizing the defined migration cost.

5.1.2. Simulated environment

For the training of IMAPALA-based homogeneous MK, a simulation-based network, is generated based on [36], where the edge servers and users are deployed at random locations. The simulation environment is built to mimic the GKE testbed to train DRL algorithms, and this model is later used for testing because training in the GKE imposes a significant cost, as described in Section 1.1. To measure the impact of heterogeneity in scalable settings, MK and EdgeBus (our proposed heterogeneous framework) are trained for 20, 40, 60, and 80 containers. MK has eight edge server nodes of type e2-standard-4, while the EdgeBus simulation environment makes use of two nodes from each category of four given types in Table 3. For each user, 100,000 steps are simulated. Testing of both the simulation and GKE testbeds is conducted on real-world traces, as explained in Section 5.1.4. Simulation results are compiled from 10 episodes on average, and every episode consisted of 3453 steps.

5.1.3. Real testbed

The real testbed is developed using Google Kubernetes Engine (GKE) and we have created a cluster of eight heterogeneous nodes because of the limitation of cost budget. Edge server node configurations are the same as the simulated environment which is shown in Table 3. Due to higher migration overhead, GKE-based experiments are performed for a single episode of 30 time steps. These user movements and tower locations were based on Cabspotting and San Francisco Tower locations. These experiments are conducted for 20 users in this research work.

5.1.4. Datasets

The user movement and node locations were simulated by the mobility simulator module. Edge server nodes were deployed on the tower locations collected from the San Francisco area.[4] Towers' information includes their locations, number of antennas

[4] Antennasearch, https://www.antennasearch.com.

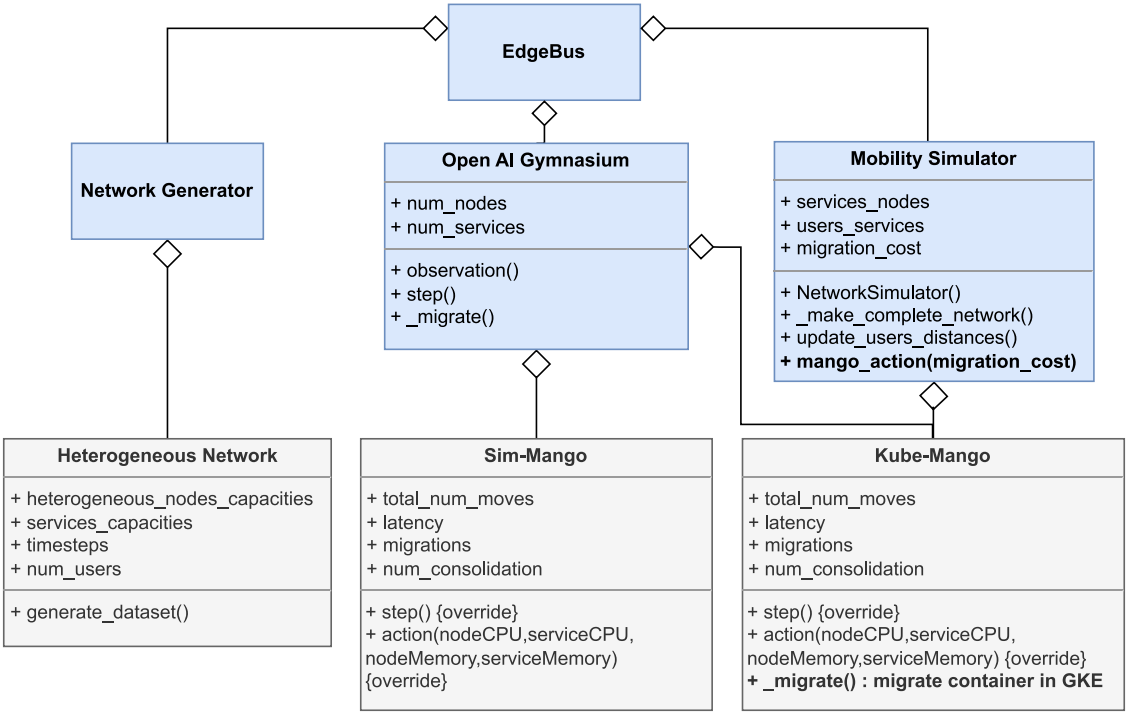


Fig. 4. A class diagram of EdgeBus.

deployed, and frequencies. We extracted their locations for edge server deployments. In this work, edge server nodes and the stations are co-located. All the nodes were connected using the Minimum Spanning Tree (MST).

To simulate user mobility, the cab spotting dataset[5] was employed. This dataset consists of approximately 500 taxicabs' mobility traces in the San Francisco area. These traces were collected from the GPS locations of each taxi over 10 s for research purposes in 2008. Each record contains information in the format of (longitude, latitude, occupancy, timestamp). However, there were missing values in the dataset, which were interpolated using the Euclidean distance. Despite the interpolation, the number of records was 3453 in total; therefore, we simulated user movements for training purposes. For both the training and dataset-oriented user movements, each user moves in −122.45 to −122.38 longitude and 37.72 to 37.78 latitude. Each user in the simulated network was connected to the closest edge server.

5.1.5. Workloads

Firstly, this work focuses on migration decision-making based on the amount of resources requested by each container rather than the workload inside the container. Secondly, keeping in view the complexity of the service migration problem in the presence of heterogeneity, the workload inside the containers is kept constant using stress-ng (details in Section 4.4). Thirdly, each container requests for same resources from underlying heterogeneous servers where the requested resources are given in Table 3. However, we played with the variable number of users and/or containers. The number of users and containers is kept the same for each case. As mentioned, there is a one-to-one mapping between users and containers; therefore, 20, 40, 60, and 80 containers are deployed in the network and these containers remain the same in the lifespan of the experiment.

5.2. Evaluation metrics

5.2.1. Average latency (L_{avg})

EdgeBus considers network latency, and it can be described as the amount of time taken by a request from a user to the associated container. Latency is dependent on user mobility, and average latency (L_{avg}) can be calculated by:

$$L_{avg} = \frac{\sum_{e=1}^{E}\sum_{t=1}^{T}\sum_{u=1}^{U} Latency}{E \times T \times U} \tag{10}$$

where E shows the number of episodes, T is the number of time steps, and U is the total number of users.

[5] CRAWDAD epfl/mobility, https://doi.org/10.15783/C7J010.

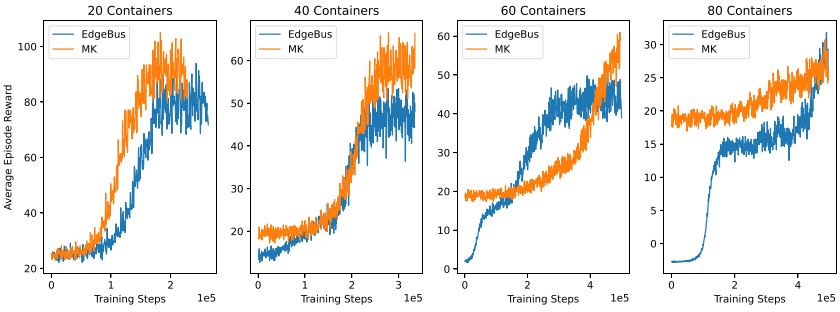


Fig. 5. Average episode reward for training.

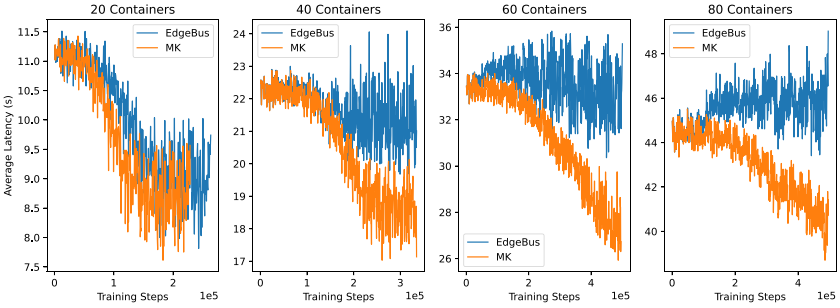


Fig. 6. Average training latency.

5.2.2. Average number of empty servers (ES_{avg})

It is defined as the number of servers that are free and ready to shut down leading to energy conservation. The higher the number of empty servers, the higher be amount of energy conservation. The average empty servers over the given number of episodes can be found by:

$$ES_{avg} = \frac{\sum_{e=1}^{E} \sum_{t=1}^{T} EmptyServer}{E \times T} \tag{11}$$

5.2.3. Average number of migrations (M_{avg})

The number of container migrations over all the episodes is given by:

$$M_{avg} = \frac{\sum_{e=1}^{E} \sum_{t=1}^{T} Migrations}{E \times T} \tag{12}$$

5.3. Experimental results: Validation of EdgeBus

In this section, we have validated the proposed heterogeneous EdgeBus framework using an existing baseline MK [25].

5.3.1. Baseline model: MobileKube (MK)

MK employed the latest and promising deep reinforcement learning algorithm IMPALA for decision-making. It takes into account the number of CPU cores and the memory. It aims to achieve a balance between latency and energy consumption. For energy conservation, MK attempts to reduce the number of active edge servers. MK tuned batch size, gamma, and learning rate hyperparameters for IMPALA. These parameter values were 1000, 0.99, and 0.0003, respectively. EdgeBus uses the same values for fair comparison.

5.3.2. Performance comparison of EdgeBus and MK

Fig. 5 shows the average training reward for each set of users for 500,000 time steps. For 20 and 40 containers, the reward surges to around 200,000 steps for the reason that there are not many illegal migration decisions. MK reward paces up at 250,000 time steps for both 40 and 60 users, and a gradual increment can be seen for 80 users. It shows that the higher the number of users/containers, the higher the illegal actions, thus leading to lesser rewards. On parallel lines, EdgeBus performs poorly, and the reward for 80 users is negative in the beginning. However, it improves as the training goes on by heavily penalizing agents for illegal actions. EdgeBus suffers from illegal actions due to the resource-constrained edge nodes, but it adapts to the settings later on.

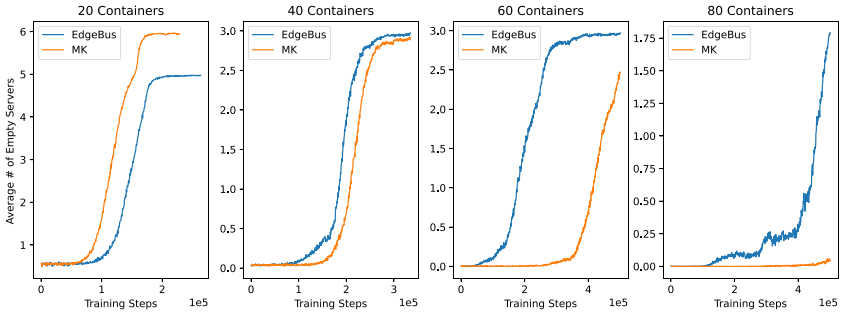


Fig. 7. Average number of empty servers for training.

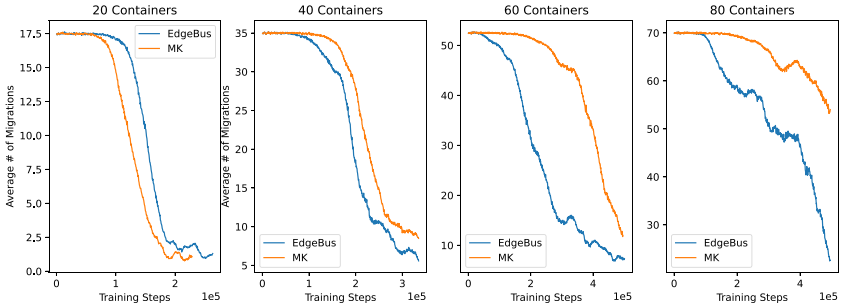


Fig. 8. Average number of migrations for training.

Average training latency can be seen in Fig. 6. Firstly, MK latency decreases as the number of steps increases for each experiment. However, EdgeBus converges only for 20 users. Secondly, comparing the starting latency for each experiment confirms that the number of users in the network influences MK and EdgeBus latency. Fig. 7 shows the average number of empty servers. EdgeBus achieves 5, 3, 3, and 1.75 empty servers for 20, 40, 60, and 80 users, respectively. In comparison, MK performs better for 20 and 40 containers, while it lags for 60 and 80 containers. The comparison of 60 and 80 users in Figs. 5 and 7 shows that EdgeBus penalizes agents early by courtesy of illegal actions, which pays off in terms of more empty servers. The reward for the 80-user scenario shows that MK is not overloading the servers because of the excessive resource availability, thus not quickly learning and resulting in near-zero empty servers.

Fig. 8 shows the average number of migrations. A similar pattern can be observed for migrations and empty servers. MK and EdgeBus perform closer for 20 and 40 containers, while there is a significant difference for scalable settings. The impact of reward for 60 and 80 containers is reflected as EdgeBus migrations drop quickly. By paying close attention to migrations for each scenario, it can be deduced that the higher the containers, the higher the migrations. Overall training performance shows that EdgeBus performs poorly in the beginning, but it adapts to the environment and outperforms MK for empty servers and a number of migrations. It also shows that heterogeneity can impact, and algorithms trained for homogeneous environments cannot be deployed in heterogeneous settings.

Our experimental findings show that EdgeBus outperforms the baseline MK. As a result, we have adopted the EdgeBus framework to conduct a further set of experiments.

5.4. Experimental results using EdgeBus

In this section, we are comparing the performance of our proposed scheduler (MANGO) with baseline schedulers such as MKS [25], Latency Greedy (LG) [26], and Binpacking (BP) [27].

5.4.1. Baseline schedulers

The description of baseline schedulers is given below:

- **MobileKube Scheduler (MKS)** [25]**:** MKS used the deep reinforcement learning algorithm IMPALA to schedule containers aiming to optimize energy consumption, and more details are given in Section 5.3.1.
- **Latency Greedy (LG)** [26]**:** It is a heuristic algorithm and its goal is to reduce the latency and improve QoS by serving the end users in the minimal possible time. To do so, every time it migrates the container to the closest available server to the user where all the containers are of the same size having equal priority. For migration decision-making, it also evaluates the destination server node for the CPU and memory.

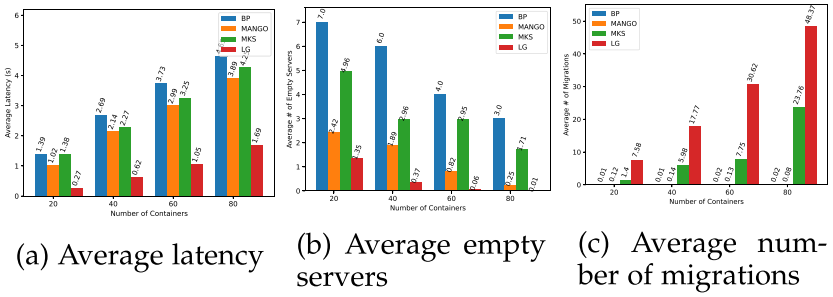


Fig. 9. Simulation results for variable numbers of containers.

Table 4
Migration cost for variable data rates.

Container size (MB)	Data rate (Gbps)	Migration cost (s)
250	1	0.25
	1.5	0.167
	2	0.125

- **Binpacking (BP)** [27]**:** It is the energy conservation scheduling algorithm. It aims to increase the number of empty servers so that the free servers are shut down. BP migrates containers to nodes with available CPU and memory ignoring over-utilization and impact on other QoS parameters including latency.

5.4.2. Simulated environment-based results

From Fig. 9(a), the latency increases as the number of containers and/or users increases for each scheduler. LG performs better among all as it does not compromise on latency and migrates containers to the closest node of the user. It is followed by the proposed MANGO, as it performs better than MK for each set of results. The highest latency is offered by BP because it aims to conserve energy and pays the price in terms of high latency. MK offers higher latency in comparison to the proposed MANGO and LG, mainly due to the consideration of the decision-making parameters and the training objective. MK is trained to reduce the number of active servers by looking into CPU and memory. However, the proposed MANGO not only considers these aforementioned parameters but also incorporates migration costs. This leads to improved performance in heterogeneous settings by a heuristics MANGO scheduler against deep reinforcement learning IMPALA.

Empty servers can be seen in Fig. 9(b). As expected, BP has the highest number of empty servers, followed by MK, the proposed MANGO is in third, and LG is in last position. The number of empty servers decreases as the number of containers increases. To entertain a large number of users, more servers need to be active and serving. MK has an advantage over the proposed MANGO because it is designed to increase the number of empty servers, which can be confirmed from Fig. 7. MANnodesGO is not directly aiming to reduce the active servers, but the objective is to achieve a balanced performance between two extremes of latency and energy, i.e., LG and BP. This leads to a slightly poor performance of MANGO for empty servers. LG outperforms every scheduler in terms of latency at the cost of almost zero empty servers for 60 and 80 containers.

Fig. 9(c) shows the average number of migrations. BP and MANGO offer the lowest number of migrations for each experiment. BP migrates all the containers to the minimum possible nodes, which stay there for a lifetime. On the contrary, the number of migrations is increasing for MK and LG. The highest number of migrations for LG resulted in the lowest latency, as shown in Fig. 9(a). MK has scored the second-highest number of migrations, but it still has a higher latency than MANGO because of MK's training objective. Under the light of Fig. 6, MK is not converging for latency, thus making costly migration decisions.

As shown in Fig. 9(c), MANGO has 0.12, 0.14, 0.13, and 0.08 migrations for 20, 40, 60, and 80 containers/users, respectively. Careful observation can reveal that the number of migrations decreases as the number of containers increases for MANGO, but it performs better than MK in terms of latency. Secondly, migrations for 60 and 80 containers are either influenced by the resource scarcity offered by the heterogeneous edge nodes or by a higher migration cost. Therefore, MANGO is experiencing fewer migrations. Overall, it can be stated that MANGO is influenced by user mobility and migration costs for the less crowded scenarios of 20 and 40 containers. While the overcrowded 60 and 80 containers are overloading the resource-constrained servers, and the migration decision-makers are the CPU and memory limitations.

5.4.3. MANGO for variable migration cost

We evaluate MANGO for latency, migrations, and empty servers by conducting experiments for variable migration costs under different data rates and number of containers. The lower the migration cost, the higher the migration probability. As explained in Section 3, the migration cost comes from the transmission time of the container that depends on the data rate. Data rates and respective migration costs are shown in Table 4, where migration cost is in seconds. Each of the 20, 40, 60, and 80 containers is

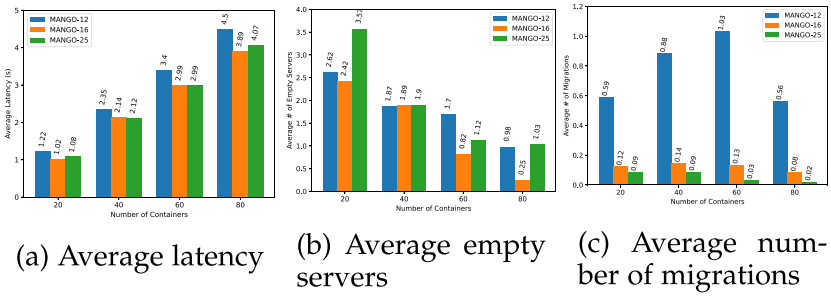


Fig. 10. Simulation results for variable migrations costs.

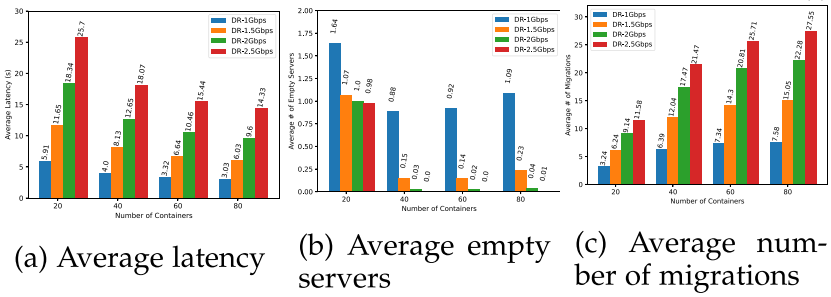


Fig. 11. Sensitivity analysis for variable data rates and random user deployment.

evaluated for the given data rates. MANGO-25 incurs 0.25 s of migration cost, MANG0-16 offers 0.16 and MANGO-12 has the lowest cost of 0.12 s.

Fig. 10(a) shows that latency increases as the number of contains/users increases irrespective of the data rates, while Fig. 10(c) shows that the number of migrations increases as the migration cost decreases. However, this pattern contradicts 80 containers. For lesser users, edge servers have enough available resources, thus the migration decisions are mainly dependent on cost and user mobility. As the number of containers increases, the major decision-maker is resource availability. Therefore, we are experiencing fewer migrations for higher containers. Fig. 10(b) shows the number of empty servers. As we are experiencing migrations, these migrations are leading to more active nodes but they are performing closer in terms of both evaluation metrics. MANGO-0.25 offers the highest number of empty servers for 20 users despite the lowest number of migrations because the containers mainly stay at the same edge server nodes.

5.4.4. Sensitivity analysis of MANGO

Fig. 11 shows the variations of latency, migrations, and empty servers for random edge nodes and user deployment for variable data rates. Considered data rates are 1, 1.5, 2, and 2.5 Gbps. Latency is decreasing as the number of containers increases. We are observing higher latency for 20 users in Fig. 11(a) due to the sensitivity of user location in the network. A higher data rate implies lower migration costs, which leads to higher migrations in each scenario. Similar results can be seen in 11(b), where the higher data rate and high number of containers are leading to fewer empty servers.

5.4.5. Real testbed based results

The experimental results depicted in Fig. 12 are based on the GKE platform, conducted with a single episode lasting 30-time steps due to the significant migration overhead. In order not to lose the system state at any time in the experiment, the subjected container is deleted from the previous node after its successful creation on a new destination. Therefore, many migrations directly impacted the total experiment time on GKE. From Fig. 9(c), the LG scheduler had the highest number of migrations in simulations for each series of experiments. Keeping in view the posed overhead and cost budget, LG is not validated in GKE.

MANGO offers lowest latency in simulations (Fig. 9(a)) among MKS and BP. GKE results confirm this and it can be seen in Fig. 12a that MANGO is performing better than its counterparts. It is mainly coming from migration cost-based informed decision-making. Fig. 12b shows the number of empty servers with BP performing better by posing high latency. However, MKS has a lower number of empty servers than MANGO contradicting simulation results for 30 time steps. From Fig. 12c, MKS offers higher migrations than MANGO and BP. Despite higher migrations, it offers higher latency confirming the simulation results. In parallel, BP has 20

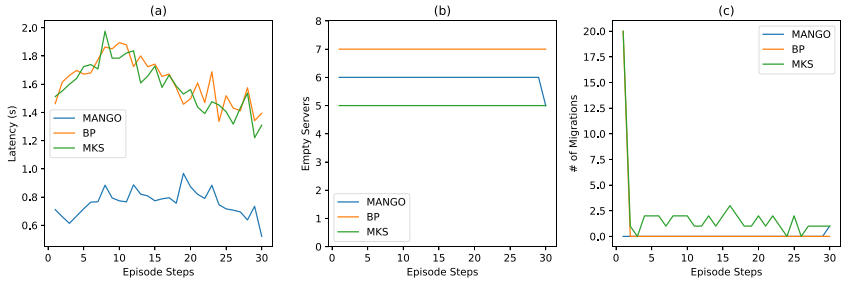


Fig. 12. Real testbed results (a) Latency, (b) Energy, and (c) Number of migrations for a variable number of containers.

migrations in the first time step and the containers stay at the same nodes for the rest of time steps to achieve the highest number of empty servers. MANGO has one migration at the 29th time step which leads to lesser delay (Fig. 12a) but we resulted in one lesser empty server. Overall, these schedulers showed similar results as simulations in the real-world GKE testbed.

6. Conclusions and future work

In this paper, we presented a coupled simulation framework called EdgeBus to investigate Kubernetes container resource management in MEC environments to improve QoS parameters. EdgeBus offers support for simulations and GKE-based real-world testbed experimental setups to explore scheduling policies in the presence of mobile end users in homogeneous and heterogeneous edge computing paradigms. We further implemented a lightweight MANGO scheduler for container management in heterogeneous settings for effective resource management. MANGO scheduler reduces unnecessary migrations achieving better latency by accounting for the migration cost, CPU cores, and memory usage for container migration, ultimately lessening the number of migrations which conserves the overall experiment time. However, MANGO offered more active nodes in comparison to state-of-the-art schedulers.

In the future, we will investigate the CPU utilization of edge server nodes to improve resource utilization by offering a balanced resource utilization technique. Meanwhile, we will explore variable workloads to assess the performance in stress conditions. We aim to deploy real-world IoT applications in Kubernetes containers to incorporate computation, processing, and propagation delays of containers for migration decision-making and integrate stateful container migration for these applications.

Software availability

It has been released as open-source software. The implementation code with experiment scripts and results can be found at the GitHub repository: https://github.com/BabarAli93/EdgeBus

CRediT authorship contribution statement

Babar Ali: Conceptualization, Data curation, Investigation, Methodology, Software, Visualization, Validation, Formal analysis, Writing - original draft. **Muhammed Golec:** Visualization, Validation, Formal analysis, Writing - original draft. Priyanshukumar Choudhary: Conceptualization, Data curation, Investigation, Methodology, Writing - original draft.. **Sukhpal Singh Gill:** Conceptualization, Data curation, Investigation, Methodology, Validation, Formal analysis, Writing - original draft and Supervision. **Huaming Wu:** Conceptualization, Visualization, Validation, Formal analysis, Writing - original draft and Supervision. **Felix Cuadrado:** Conceptualization, Data curation, Investigation, Methodology, Validation, Formal analysis, Writing - original draft and Supervision. **Steve Uhlig:** Conceptualization, Data curation, Investigation, Methodology, Validation, Formal analysis, Writing - original draft and Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

B. Ali is supported by the Ph.D. Scholarship at the Queen Mary University of London, United Kingdom. M. Golec is supported by the Ministry of Education of the Turkish Republic. H. Wu is supported by the National Natural Science Foundation of China (No. 62071327) and Tianjin Science and Technology Planning Project, China (No. 22ZYYYJC00020). F. Cuadrado has been supported by the HE ACES project, Spain (Grant No. 101093126).

References

[1] G.K. Walia, M. Kumar, S.S. Gill, AI-empowered fog/edge resource management for IoT applications: A comprehensive review, research challenges, and future perspectives, IEEE Commun. Surv. Tutor. 26 (1) (2024) 619–669.
[2] M. Hartmann, et al., Edge computing in smart health care systems: Review, challenges, and research directions, Trans. Emerg. Telecommun. Technol. 33 (3) (2022) e3710.
[3] L.U. Khan, et al., Edge-computing-enabled smart cities: A comprehensive survey, IEEE Internet Things J. 7 (10) (2020) 10200–10232.
[4] E. Ahmed, et al., Internet-of-things-based smart environments: state of the art, taxonomy, and open research challenges, IEEE Wirel. Commun. 23 (5) (2016) 10–16.
[5] M. Aazam, S. ul Islam, S.T. Lone, A. Abbas, Cloud of things (CoT): cloud-fog-IoT task offloading for sustainable internet of things, IEEE Trans. Sustain. Comput. 7 (1) (2020) 87–98.
[6] S. Iftikhar, et al., AI-based fog and edge computing: A systematic review, taxonomy and future directions, Internet Things 21 (2023) 100674.
[7] Z. Ding, S. Wang, C. Jiang, Kubernetes-oriented microservice placement with dynamic resource allocation, IEEE Trans. Cloud Comput. 11 (2) (2023) 1777–1793.
[8] P. Mach, Z. Becvar, Mobile edge computing: A survey on architecture and computation offloading, IEEE Commun. Surv. Tutor. 19 (3) (2017) 1628–1656.
[9] Y. Mao, et al., A survey on mobile edge computing: The communication perspective, IEEE Commun. Surv. Tutor. 19 (4) (2017) 2322–2358.
[10] T. Zhang, W. Chen, Computation offloading in heterogeneous mobile edge computing with energy harvesting, IEEE Trans. Green Commun. Netw. 5 (1) (2021) 552–565.
[11] S.S. Gill, M. Golec, J. Hu, M. Xu, J. Du, H. Wu, G.K. Walia, S.S. Murugesan, B. Ali, M. Kumar, et al., Edge AI: A taxonomy, systematic review and future directions, 2024, arXiv preprint arXiv:2407.04053.
[12] W. Zhang, L. Chen, J. Luo, J. Liu, A two-stage container management in the cloud for optimizing the load balancing and migration cost, Future Gener. Comput. Syst. 135 (2022) 303–314.
[13] Z. Jian, X. Xie, Y. Fang, Y. Jiang, Y. Lu, A. Dash, T. Li, G. Wang, DRS: A deep reinforcement learning enhanced Kubernetes scheduler for microservice-based system, Softw. - Pract. Exp. n/a (n/a) (2023).
[14] A. Marchese, O. Tomarchio, Network-aware container placement in cloud-edge kubernetes clusters, in: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid, IEEE, 2022, pp. 859–865.
[15] A.A. Khan, et al., An energy and performance aware consolidation technique for containerized datacenters, IEEE Trans. Cloud Comput. 9 (4) (2019) 1305–1322.
[16] Y. Hu, H. Zhou, C. de Laat, Z. Zhao, Concurrent container scheduling on heterogeneous clusters with multi-resource constraints, Future Gener. Comput. Syst. 102 (2020) 562–573.
[17] M. Lin, J. Xi, W. Bai, J. Wu, Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud, IEEE Access 7 (2019) 83088–83100.
[18] M. Imdoukh, et al., Optimizing scheduling decisions of container management tool using many-objective genetic algorithm, Concurr. Comput.: Pract. Exper. 32 (5) (2020) e5536.
[19] C. Guerrero, I. Lera, C. Juiz, Genetic algorithm for multi-objective optimization of container allocation in cloud architecture, J. Grid Comput. 16 (2018) 113–135.
[20] S. Tuli, et al., COSCO: Container orchestration using co-simulation and gradient based optimization for fog computing environments, Trans. Parallel Distrib. Syst. 33 (1) (2021) 101–116.
[21] D. Garg, N.C. Narendra, S. Tesfatsion, Heuristic and reinforcement learning algorithms for dynamic service placement on mobile edge cloud, 2021, arXiv preprint arXiv:2111.00240.
[22] F.o. Brandherm, A learning-based framework for optimizing service migration in mobile edge clouds, in: 2nd International Workshop on Edge Systems, Analytics and Networking, 2019, pp. 12–17.
[23] Y. Zhang, et al., Hetmec: Heterogeneous multi-layer mobile edge computing in the 6 G era, Trans. Veh. Technol. 69 (4) (2020).
[24] K. Zhang, et al., Energy-efficient offloading for mobile edge computing in 5G heterogeneous networks, IEEE Access 4 (2016) 5896–5907.
[25] S. Ghafouri, et al., Mobile-Kube: Mobility-aware and energy-efficient service orchestration on kubernetes edge servers, in: 2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing, UCC, IEEE, 2022, pp. 82–91.
[26] B. Tang, J. Luo, M.S. Obaidat, P. Vijayakumar, Container-based task scheduling in cloud-edge collaborative environment using priority-aware greedy strategy, Cluster Comput. 26 (6) (2023) 3689–3705.
[27] F. Farahnakian, T. Pahikkala, P. Liljeberg, J. Plosila, N.T. Hieu, H. Tenhunen, Energy-aware VM consolidation in cloud data centers using utilization prediction model, IEEE Trans. Cloud Comput. 7 (2) (2016) 524–536.
[28] P. Chhikara, et al., An efficient container management scheme for resource-constrained intelligent IoT devices, IEEE Internet Things J. 8 (16) (2020) 12597–12609.
[29] S. Cao, et al., Service migrations in the cloud for mobile accesses: A reinforcement learning approach, in: 2017 International Conference on Networking, Architecture, and Storage, NAS, IEEE, 2017, pp. 1–10.
[30] F. Brandherm, et al., BigMEC: Scalable service migration for mobile edge computing, in: IEEE Symposium on Edge Computing, SEC, IEEE, 2022, pp. 136–148.
[31] B. Yamansavascilar, et al., Deepedge: A deep reinforcement learning based task orchestrator for edge computing, IEEE Trans. Netw. Sci. Eng. 10 (1) (2022) 538–552.
[32] Z. Tang, et al., Migration modeling and learning algorithms for containers in fog computing, Trans. Serv. Comput. 12 (5) (2018) 712–725.
[33] L. Bogolubsky, et al., Learning supervised pagerank with gradient-based and gradient-free optimization methods, Adv. Neural Inf. Process. Syst. 29 (2016).
[34] R. Mahmud, S. Pallewatta, M. Goudarzi, R. Buyya, Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments, J. Syst. Softw. 190 (2022) 111351.
[35] A. Hagberg, P.J. Swart, D.A. Schult, Exploring network structure, dynamics, and function using networkx, Tech. Rep., Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.
[36] S. Ghafouri, A.A. Saleh-Bigdeli, J. Doyle, Consolidation of services in mobile edge clouds using a learning-based framework, in: 2020 IEEE World Congress on Services, SERVICES, IEEE, 2020, pp. 116–121.